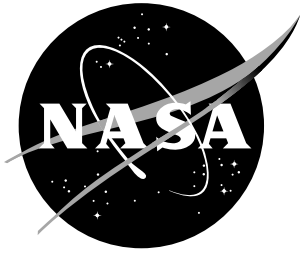# *Lfm*2000: Fifth NASA Langley Formal Methods Workshop

C. Michael Holloway, Compiler
Langley Research Center, Hampton, Virginia

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320
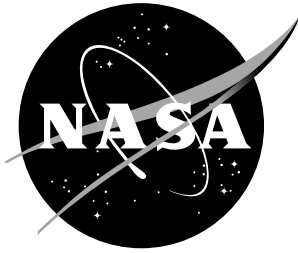
*Lfm*2000: Fifth NASA Langley Formal Methods Workshop

C. Michael Holloway, Compiler
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

June 2000

General Chairman's Message

On behalf of the NASA Langley Formal Method's Team, I welcome you to Lfm2000, the Fifth NASA Langley Formal Methods Workshop. When the series began in 1990, attendees and presenters were limited to people directly involved in NASA Langley's nascent formal methods program. Subsequent workshops in 1992 and 1995 also restricted attendance to invited people. With the 1997 workshop, we removed attendance restrictions, and also issued an international call for papers. We continued this approach for Lfm2000.

We believe that the program has something to offer just about everyone, from those interested in the theoretical aspects of formal methods to those interested in the practical application of formal methods to help solve industrial problems. We hope that you agree, and that your time at the workshop is both interesting and useful to you.

The paper copy of the proceedings contains the papers selected by the program committee for presentation. The CD contains Portable Document Format (PDF) and (in many cases) PostScript versions of the papers, supplementary information from some authors, tutorial slides and supplementary material, and information about the NASA Langley formal methods program. Much of this material will also be available on the world-wide web at the Lfm2000 web site at <http://shemesh.larc.nasa.gov/fm/Lfm2000>.

Once again, welcome! I look forward to meeting you during the workshop. Please let me know if there is anything that I can do to help you while you are here.

C. Michael Holloway, Lfm2000 General Chairman
email: <c.m.holloway@larc.nasa.gov>
postal address: Mail Stop 130, NASA Langley Research Center, Hampton VA 23681-2199

Program Committee Chairman's Message

Welcome to Lfm2000! We are pleased to be able to bring you a strong program of research papers and experience reports. This year we added a tutorial track to complement the research presentations. We were fortunate to receive several excellent tutorial proposals from some rather accomplished presenters, so we hope you find this a valuable addition to the workshop format.

Following the organization we adopted at Lfm97, our previous workshop, we drew the bulk of the Lfm2000 program from refereed submissions. We received 37 paper submissions, from which 17 papers were selected for presentation at the workshop and publication in the proceedings. Each paper received at least three reviews, either by members of the Program Committee or by outside referees. In addition to selected papers, we invited several speakers to give talks on trends and perspectives, including presentations on ongoing NASA activities and interests.

Submissions to Lfm2000 showed a continued strong interest in the area of applied formal methods. The diversity of submissions increased somewhat over our previous workshop in 1997. Also evident in the accepted papers was a decided shift toward lighter weight methods and the algorithmic analysis techniques typified by model checking. This trend reflects the growing interest in finite state analysis that has been seen at other research meetings. It is too soon to tell, however, whether this growth comes at the expense of interest in the deductive analysis methods. Perhaps by the time of our next workshop we can gauge the community's directions more definitively.

I would like to thank members of the Program Committee for all their hard work in reviewing and selecting papers for this year's program. Thanks are also due to the auxiliary referees who contributed their time. Finally, let me thank the Organizing Committee for helping to give shape to the finished product.

I hope you find this a rewarding meeting. We welcome any feedback you might wish to provide so that our next offering will be better still.

Ben Di Vito, Lfm2000 Program Chair
email: <b.l.divito@larc.nasa.gov>
postal address: Mail Stop 130, NASA Langley Research Center, Hampton VA 23681-2199

$\mathcal{Lfm}$ 2000 Organization

Program Committee
Ben Di Vito, NASA Langley Research Center, Chairman
Michael Holloway, NASA Langley Research Center, Executive Secretary
Ricky Butler, NASA Langley Research Center
Victor Carreño, NASA Langley Research Center
David Dill, Stanford University
Eric Engstrom, Honeywell Technology Center
David Guaspari, Odyssey Research Associates
Klaus Havelund, NASA Ames Research Center
Kelly Hayhurst, NASA Langley Research Center
Mats Heimdahl, University of Minnesota
Thierry Jéron, IRISA/INRIA
Chris Johnson, University of Glasgow
Steve Johnson, Indiana University
John Kelly, Jet Propulsion Laboratory
John Knight, University of Virginia
Mike Lowry, NASA Ames Research Center
Gerald Lüttgen, Institute for Computer Applications in Science and Engineering
Paul Miner, NASA Langley Research Center
César Muñoz, Institute for Computer Applications in Science and Engineering
C. R. Ramakrishnan, SUNY, Stony Brook
John Rushby, SRI International
Mark Saaltink, ORA Canada
Matt Wilding, Rockwell Collins
Secondary Reviewers: Mark Bickford, David Greve, Alain Le Guennec, Peter Habermehl,
          Loïc Hélouët, Dimitri Naydich, Nicolas Roquette, Mahadevan Subramaniam

Tutorial Selection Committee
Ben Di Vito, NASA Langley Research Center, Chairman
Michael Holloway, NASA Langley Research Center
Ricky Butler, NASA Langley Research Center
Victor Carreño, NASA Langley Research Center
Kelly Hayhurst, NASA Langley Research Center
Gerald Lüttgen, Institute for Computer Applications in Science and Engineering
Paul Miner, NASA Langley Research Center
César Muñoz, Institute for Computer Applications in Science and Engineering

Organizing Committee
Michael Holloway, NASA Langley Research Center, Workshop General Chairman
Lisa Peckham, NASA Langley Research Center
Kelly Hayhurst, NASA Langley Research Center
Andrea Carden, Science and Technology Corporation

Table of Contents

On Tableau Constructions for Timing Diagrams

Kathi Fisler

Department of Computer Science

Rice University

6100 S. Main, MS 132

Houston, TX 77005-1892

kfisler@cs.rice.edu

Abstract

Designers often cite unfamiliar notation as one obstacle to wider acceptance of formal methods. Formalizations of design notations, such as timing diagrams, promise to bridge the gap between design practice and formal methods. How to use such formalizations effectively, however, remains an open question. Developing new tools around design notations might provide better support for reasoning at the level of the preferred notations. On the other hand, translating the formalizations into established notations enables leveraging off of existing tools. This decision of whether to treat design notations as interfaces depends largely on computational tradeoffs. This paper explores this issue in the context of specifying properties for automata-theoretic verification using timing diagrams. Automata-theoretic algorithms perform a tableau construction to convert properties into Büchi automata. We contrast direct compilation of timing diagrams into Büchi automata with an approach that uses linear-time temporal logic (LTL) as an intermediate language during translation. Direct compilation generally produces much smaller automata and scales significantly better with variations in key timing diagram parameters. We attribute this to combination of a correspondence between timing diagrams and weak automata and certain shortcomings in current LTL-to-Büchi algorithms.

1 Introduction

Computer-aided verification uses techniques from logic and mathematics to prove whether design models satisfy certain properties. Although these techniques have been used successfully on several sizable examples, many designers are reluctant to adopt them. One frequently cited problem is the notation that verification tools employ [9]. Verification technologies are grounded in formal logic. Accordingly, most tools use their underlying logics as property specification languages. For example, model checkers employ temporal logics, while theorem provers use various flavors of higher-order logic. In contrast, designers use a wide array of notations, including circuit diagrams, timing diagrams, state machines, VHDL and Verilog. This rich array of representations, some of them diagrammatic, stands in stark contrast to the monolithic textual logics of verification tools.

Bridging this gap requires verification tools that support notations that are more familiar to designers. One approach is to develop new tools and algorithms which support design notations directly [3]. Another is to create interfaces from design notations to existing languages [1, 8]; this leverages off existing tool development efforts.[1] Which approach yields more efficient algorithms is an open question. There may exist algorithms for model checking timing diagrams, for example, that outperform those for the temporal

---

[1] Many efforts (other than those cited) are ad-hoc, however, because they do not formalize the design notations.

logics into which we might translate timing diagrams. Understanding these tradeoffs requires studies of the logical nature of design notations and their role in verification algorithms.

This paper explores these tradeoffs in the context of compiling timing diagrams to Büchi automata. Automata-theoretic verification tools, which support linear-time logics such as LTL, operate at the level of automata. Using such tools on timing diagrams requires algorithms for compiling timing diagrams to Büchi automata. We compare two compilation methods, one which compiles timing diagrams directly into Büchi automata and one which translates timing diagrams into LTL and then uses existing algorithms for compiling LTL into Büchi automata. Our results show that the direct approach produces far smaller machines even on simple examples. This appears due to a combination of structural properties of the automata that capture timing diagrams and shortcomings in existing LTL-to-Büchi translation algorithms.

Section 2 presents an overview of automata-theoretic verification. Section 3 describes timing diagrams and linear-time temporal logic, the two notations used in this paper. Section 4 presents our algorithms for compiling timing diagrams into LTL and Büchi automata. Section 5 presents an experimental comparison of the two approaches to obtaining Büchi automata from timing diagrams. Section 6 discusses the experimental results and their implications for verification research.

2   Automata-Based Verification

Automata-theoretic verification views both systems and properties as finite-state automata [12, 14]. Verifying whether a system satisfies a property is analogous to asking whether the property automaton accepts the language generated by the system. In other words, for a system S and a property P, verification reduces to a language containment question of the form $\mathcal{L}(S) \subseteq \mathcal{L}(P)$, where $\mathcal{L}$ denotes the language of an automaton. This is equivalent to asking whether $\mathcal{L}(S) \cap \mathcal{L}(\overline{P}) = \emptyset$. In practice, automata-theoretic verification tools implement the latter; they intersect the automaton for the negation of the property with the automaton for the system and check whether the language of the product automaton is empty.

Many other verification problems can be expressed in terms of operations on languages. Property decomposition is one example. Properties often prove intractable to verify because they require too many computational resources, such as time or memory. One can approach such cases by decomposing the original property into a set of simpler properties, each of which is tractable to verify. If the simpler properties collectively imply the original property, then verifying each simple property independently is sufficient to verify the original property. To support decomposition, verification tools must check whether one set of properties implies another. If a property P is decomposed into properties $P_1, \ldots, P_k$, this check reduces to $\mathcal{L}(P) \subseteq \mathcal{L}(P_1) \cap \ldots \cap \mathcal{L}(P_k)$.

Both of these checks are decidable for a large class of verification problems. Implementing them requires procedures to obtain two kinds of automata: those that accept the language of a given property and those that accept the language of the negation of a given property. This project investigates both problems in the context of timing diagrams.

3   Timing Diagrams and LTL

3.1   Timing Diagrams

Timing diagrams express patterns of value changes on signals. In addition, they express precedence and synchronization relationships between changes, and timing constraints between changes. As part of our overall research program, we have developed a logic of timing diagrams [5]. This section describes the portion of the logic that is relevant to this paper.

Figure 1 provides a sample timing diagram that will serve as our running example. Variables a, b, and c name boolean-valued signals. To the right of each name is a *waveform* depicting how the variable's value changes over time. For example, b transitions from low to high, then later returns to low. We interpret low as logical false and high as logical true. Arrows indicate temporal ordering between transitions; for this paper, we assume that timing diagrams spec-
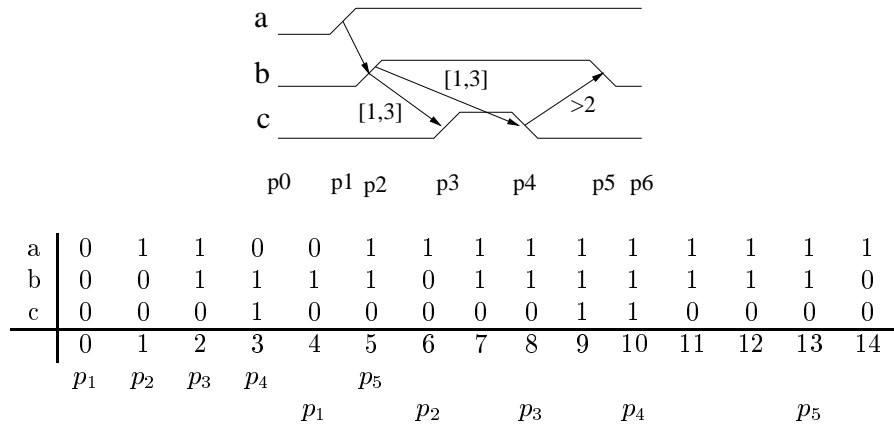
Figure 1: A timing diagram and an illustration of its semantics.

	p0	p1	p2	p3		p4		p5	p6						
a	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
b	0	0	1	1	1	1	0	1	1	1	1	1	1	1	0
c	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	p_1	p_2	p_3	p_4	p_5										
						p_1		p_2		p_3		p_4		p_5	

ify a total ordering on the transitions through arrows and ordering within waveforms. Annotations of the form $[l, u]$ on the arrows indicate lower and upper bounds on the time between the related transitions; l is a natural number and u is a natural number or the symbol ∞.[2] The labels at the bottom, referred to as *time points*, are for explanatory purposes and are not part of the timing diagram; intuitively, there is one time point for each transition in the diagram, plus one for each of the endpoints of the diagram. The portion of the diagram between each pair of time points is an *interval*; interval I_j spans from time point p_j to p_{j+1}.

Since timing diagrams express sequences of values of variables over time, an appropriate semantic model for them must do the same. Formal languages, which are sets of sequences over a given alphabet, suggest such a model. Our semantics considers finite or infinite words over an alphabet consisting of all possible assignments of boolean values to the names labeling waveforms. Intuitively, a word models a timing diagram when the transition patterns in the diagram reflect the changes in values assigned to names in the word. A *timing diagram language* is any set of words such that every word in the set models the timing diagram. This paper provides an intuitive description of the semantics; the full details appear elsewhere [5].

Consider the timing diagram and word in Figure 1.

The word appears in tabular form: the waveform names label the rows and the indices into the word label the columns. Each cell in the table indicates the value on the corresponding signal at the corresponding index. Symbols 0 and 1 denote false and true, respectively. The two lines directly beneath the table indicate two separate assignments of indices to time points, as explained shortly.

Intuitively, the semantics walks along a word looking for indices that satisfy each time point. An index satisfies a time point if the values assigned to each variable correspond to those required by the transitions at the time point; satisfaction relies on both the current index and its immediate successor. For example, in Figure 1, time point p_1 contains a rising transition on signal a; index d satisfies p_1 if d assigns value 0 to a and index $d + 1$ assigns value 1 to a.

For the word and timing diagram in Figure 1, index 0 satisfies the rising transition on a. The walk now searches for an index containing a rising transition on b; index 1 meets this criterion. When the walk locates the rising transition on c in index 2, the semantics checks whether the located indices respect the timing constraint between the transitions on b and c. The two transitions occurred one index apart, which is valid. Continuing the walk locates time point p_4 at index 3 and time point p_5 at index 5. The first row below the table shows this assignment of time

---

[2] The full logic supports richer bounds with variables [5].

points to indices. The second row shows another assignment, starting from index 4. This walk fails, because the distance between the indices satisfying p_2 and p_4 is larger than 3, the maximum allowed by the time bound on the arrow from the rising transition on b to the falling transition on c. The semantics always checks the first occurrence of a transition that it finds once it begins searching for it. The formal semantics [5] defines this precisely.

Three other aspects of our semantics are relevant:

- Timing diagrams express assume-guarantee relationships; we specify some prefix of the time points as the *assume portion*, and only check the entire diagram when we locate indices satisfying the assume portion. In our example, taking the assume portion to be time points p_0 and p_1, we would search for the entire diagram only if an index reflects a rising transition on a.

- We view timing diagrams as invariants, meaning that we attempt to satisfy the timing diagram from every index which satisfies the assume portion. In our example, we would search from every index containing a rising transition on a, namely indices 0 and 4, as in our demonstration.

- A parameter over the timing diagram indicates which segments of waveforms should be matched exactly within words; the rest are treated as don't-cares. Segments to be matched exactly are called *fixed-level constraints*. For example, we could require a to remain high until the rising transition on c by putting a fixed-level constraint on a between time points p_1 and p_4.

Index satisfaction and fixed-level constraints are simply constraints on the values of particular variables; each constraint is a conjunction of literals capturing the values required on each variable. A fixed-level constraint requiring a to be low and c to be high would be the conjunction $\neg a \wedge c$. The actual conjunctions are irrelevant to the algorithms in the rest of the paper. We therefore describe our algorithms in terms of the following symbols:

- A_i: the fixed-level constraint in interval I_i.

- $AP_{i\text{init}}$: the first index required to satisfy the transition at time point i.

- AP_i: the second index required to satisfy the transition at time point i.

- T_i: The conjunction $AP_{i\text{init}} \wedge \mathsf{X} AP_i$, which uses the temporal logic next-time operator to capture the requirements for satisfying a transition.

3.2 Linear-time Temporal Logic

Like timing diagrams, linear-time temporal logic describes patterns of changes in variables over sequences of assignments. LTL is a propositional temporal logic [13], defined relative to a finite set of propositions $\mathcal{P}$. The formulas of LTL include $\mathcal{P}$ and are closed under unary operators $\neg$ and X (next), and binary operators $\vee$ and U (until). Intuitively, $\mathsf{X}\varphi$ says that φ holds in the next state, while $\varphi\mathsf{U}\psi$ says that φ holds in every state until ψ holds, and ψ eventually holds. Other temporal operators, such as G (something holds in all states) are defined in terms of U. Formally, LTL formulas are given semantics relative to sequences of assignments to $\mathcal{P}$. An infinite word $\xi = x_0 x_1 \ldots$ is a sequence of elements of $2^{\mathcal{P}}$. ξ_i denotes the suffix of ξ starting at x_i. A word ξ models formulas according to the following definition:

- $\xi \models q$ iff $q \in x_0$, for $q \in \mathcal{P}$,
- $\xi \models \neg\varphi$ iff not $\xi \models \varphi$,
- $\xi \models \varphi \vee \psi$ iff $\xi \models \varphi$ or $\xi \models \psi$,
- $\xi \models \mathsf{X}\varphi$ iff $\xi_1 \models \varphi$,
- $\xi \models \varphi\mathsf{U}\psi$ iff there is an $i \geq 0$ such that $\xi_1 \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$.

A language models a formula iff every word in the language models the formula.

4 Tableau Constructions

As discussed in Section 2, automata-theoretic verification tools compile formulas into Büchi automata. As LTL model checking uses the automata-theoretic framework, several algorithms exist for compiling

LTL formulas into Büchi automata [2, 7]; these algo-
rithms use a technique called tableau construction.
The timing diagram semantics effectively define a
Büchi automaton accepting a timing diagram lan-
guage. Thus, we have two possible routes to compil-
ing timing diagrams into Büchi automata, as shown
in the diagram below: compile the timing diagram
directly to a Büchi automaton which corresponds to
the semantics, or translate the timing diagram into
LTL and use existing LTL-to-Büchi algorithms. The
second approach reflects the view of timing diagrams
as visual interfaces for temporal logics [1].
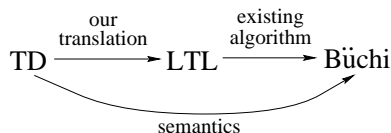
We would like to compare the Büchi automata aris-
ing from these two approaches. Is one substantially
larger than the other? Size is important because this
form of verification computes the cross-product of
the automata representing the design and the prop-
erty. Does one approach yield a Büchi automaton
that is more amenable to verification than the other?
Some verification heuristics work only on property
automata with particular structural features. An-
swers to these questions help determine whether ver-
ification tools can safely treat timing diagrams as in-
terfaces to LTL expressions without having an ad-
verse effect on the verification process.

Our translations from timing diagrams to each of
LTL and automata rely on the same intermediate rep-
resentation, a form of abstract state machine. States
in this machine record which interval they correspond
to, their transitions to other abstract states, and a
set of labels which provide information to the back-
end tools. The abstract machine captures one pass
or walk of the timing diagram semantics, leaving the
backend tools to support repetitions as necessary.

4.1 Generating the Abstract Machine

Generating an abstract machine from a given tim-
ing diagram proceeds in two steps. First, we need to

I_1
1+

I_2	I_3
1	1
1	2
2	1

I_4
2+

Figure 2: Step distribution tables for the example
timing diagram.

calculate the possible numbers of steps that a valid
word can spend in each interval. We partition the
time points into cells such that time points i and j
are in the same cell iff there is an arrow spanning
intervals i and j: for our example timing diagram,
the cells are {0}, {1}, {2, 3}, {4}, and {5}. For each
cell, we generate a table showing the possible combi-
nations of steps allowed in each interval. Each row
of the table provides one distribution of the time al-
lowed by the bounds across the corresponding inter-
vals; if the total amount of time is a lower bound,
the value in the last column of the table is marked
with a +. Figure 2 shows the tables for our example
diagram. They say that a valid word must contain at
least one letter in the first interval (1+ in the first ta-
ble), some combination of 2 or 3 letters in the interval
between time points 2 and 4 (the middle table), and
at least two letters in interval I_4. We generate the
tables using a straightforward procedure for calculat-
ing distributions across variables. We then eliminate
distributions that violate some timing constraint; the
example diagram, for example, allows the arrow from
the rising transition on b to the rising transition on
c to last 3 steps, but doing so would violate the con-
straints of the edge from the rising transition on b
to the falling transition on c. The tables in Figure 2
contain no row allowing 3 steps in interval I_2.

Next, we generate abstract states from the cells
and tables. Each abstract state contains the time
point it corresponds to, a set of transitions to other
abstract states, and a set of labels (which we de-
scribe shortly). We generate a final state (labeled
final) with a self-transition; this corresponds to the
maximal time point. We also generate two abstract
states with transition to the final state for each time
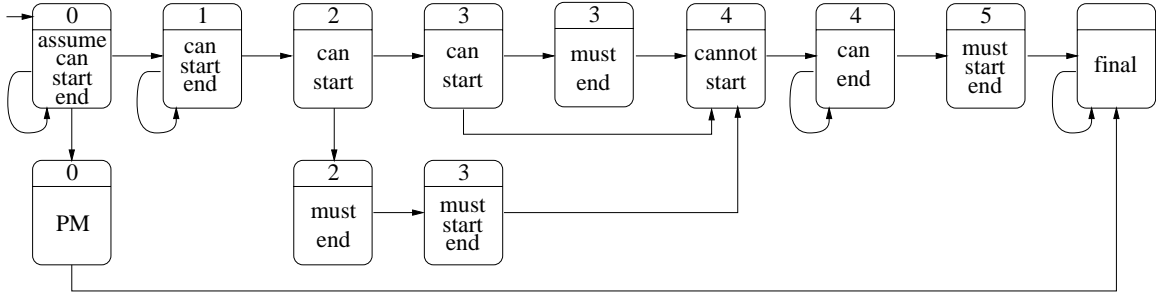point in the assume portion: one labeled PM for pat-

Figure 3: The abstract machine for the example timing diagram.

tern mismatches and one labeled CV for constraint violations; these capture violations of the timing diagram patterns in the assume portion. The generation method processes the cells in reverse order. For each cell, we generate a set of states, designating one as the initial state for the cell, as follows. If there is no table for the cell, we generate one abstract state with two transitions: one to itself and one to the initial state for the cell containing the next time point. If the time point is in the assume portion, the abstract state also contains a transition to the pattern-mismatch state for the corresponding time point.

If there is a table for a cell, we must generate sequences of states that count steps in the intervals as indicated in the tables. Rather than generate these sequences independently, however, we share states at the prefixes of the sequences when possible. All sequences will share at least one common prefix state; this is the initial state for the cell. For the example timing diagram, all rows for cell $\{2, 3\}$ require at least one state in interval 2. Each state contains a transition to the next state in the sequence; states in common prefixes may have transitions to multiple suffixes. In addition, if the last entry in a row is annotated with +, the final state in the sequence for the row contains a self-loop. If the cell is in the assume portion, each state also contains transitions to the pattern-mismatch and constraint-violation states for the corresponding time points. Figure 3 shows the abstract machine corresponding to our example timing diagram. We have explained the structure of this machine; we now describe the labels.

Each state corresponding to a time point in the assume portion receives the label assume. For each state other than the final, pattern-mismatch, and constraint-violation states, we add all labels from the following list for which the state satisfies the indicated constraints relative to the structure of the transition system; let B be a state at time point p_i:

- start: no other state for time point p_i reaches B;
- end: B reaches no other state for time point p_i;
- can: B has successors for time points p_i and p_{i+1};
- cannot: all successors are for time point p_i;
- must: no successor is for time point p_i.

The labels start and end indicate the first and last states for each corresponding time point; can, cannot, and must indicate whether a word can, cannot, or must advance to the next time point from this state. While some of these labels have overlapping meaning (all must states are end states, for example), no two labels are equivalent.

4.2 Generating LTL

This section generates an LTL formula corresponding to one pass of the timing diagram semantics. Wrapping the formula in LTL operator G yields the invariant formula. The procedure follows the structure of the abstract machine. There are two steps in generating the LTL for a given abstract state: generating the propositional expression that captures the fixed-level constraints for the state and connecting this expression with those for other states using temporal

$$([(A_0 \wedge \neg T_0) \; \mathsf{U} \; (A_0 \wedge T_0)] \rightarrow$$
$$[(A_0 \wedge \neg T_0) \; \mathsf{U}$$
$$(A_0 \wedge T_0 \wedge$$
$$\mathsf{X}[(A_1 \wedge \neg T_1) \; \mathsf{U}$$
$$(A_1 \wedge T_1 \wedge \mathsf{X}((A_2 \wedge T_2 \wedge \mathsf{X}((A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \; \mathsf{U} \; (A_4 \wedge T_4)])) \vee$$
$$(A_3 \wedge \neg T_3 \wedge \mathsf{X}(A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \; \mathsf{U} \; (A_4 \wedge T_4)]))))) \vee$$
$$(A_2 \wedge \neg T_2 \wedge \mathsf{X}(A_2 \wedge T_2 \wedge \mathsf{X}(A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \; \mathsf{U} \; (A_4 \wedge T_4)]))))))])])$$

Figure 4: LTL generated for example timing diagram

operators. The expression for a state is the fixed-level constraint A_i; if the state is the first or last in a time point, we conjoin A_i with AP_i or $AP_{i+1\text{init}}$, respectively. The temporal operators are based on the transition structure of the abstract machine.

Formally, procedure GenLTL(B) produces the LTL for abstract state B as follows, where R is the transition relation of the abstract machine. For abstract states B without self-loops, GenLTL(B) produces

$$A_i \wedge T_i \wedge \bigvee_{B' \in R(B)} \mathsf{X}(\text{GenLTL}(B')).$$

For abstract states B with self loops, GenLTL(B) is

$$[(A_i \wedge \neg T_i) \; \mathsf{U} \; (A_i \wedge T_i \wedge \bigvee_{B' \in R(B)} \mathsf{X}(\text{GenLTL}(B')))].$$

The T_i's require the expression to match the first available transition to the next time point. To handle the assume portion, the algorithm generates LTL for the restriction of the abstract machine to the assume portion and forms an implication from this formula to the LTL for the entire diagram. This follows the intuitive semantics of timing diagrams. Figure 4 shows the resulting LTL for our running example. The contrast between the formula and the original timing diagram motivates designers' frustrations with common verification notations.

4.3 Generating Büchi Automata

A Büchi automaton is a tuple $\langle Q, q_0, R, L, \mathcal{F} \rangle$ where Q is a set of states, q_0 is the initial state, $R \subseteq Q \times Q$ is the transition relation, L indicates propositions that are true in each state, and $\mathcal{F} \subseteq Q$ is a set of fair states. The abstract machine resembles a Büchi automaton; however, it does not capture a timing diagram because it does not enforce matching the first occurrences of transitions. The Büchi automaton states enforce this by examining propositions $AP_{i+1\text{init}}$ and AP_{i+1}, which indicate when transitions should occur. These states also refer to the fixed-level constraint A_i.

Monitoring $AP_{i+1\text{init}}$ and AP_{i+1} implies that an abstract state can expand into four Büchi states (A_i must hold in each; the pattern-mismatch states account for when A_i does not hold). The number may be more or less depending on the abstract state's labels. Regardless of the labels, only a few combinations of propositions arise in practice. Table 1 (left) lists templates of the generated Büchi states. For each state, we list the propositions that are true in that state and a set of labels. These labels are not part of the Büchi automaton; the algorithm uses them to create transitions between states. The labels can be divided into two sets, depending upon whether they contain this; we explain the distinction shortly.

The Büchi automaton generator converts abstract state B into Büchi automaton states $b_1, \ldots, b_m$ in two steps. First, it creates the template states indicated in Table 1 (right). Second, it adds the outgoing transitions for each b_k. These outgoing transitions depend on B's labels and whether b_k outputs proposition $AP_{i+1\text{init}}$. This proposition matters because it indicates that b_k could recognize the start of the next time-point. Any transitions from b_k to states outputting proposition AP_{i+1} must be to states corresponding to the next time-point.

	Propositions	Incoming Labels
S_1	A_i, $AP_{i+1\text{init}}$, AP_{i+1}	this-tp, this-tp-trans
S_2	A_i, $\neg AP_{i+1\text{init}}$, AP_{i+1}	this-tp, this-tp-trans
S_3	A_i, $AP_{i+1\text{init}}$, $\neg AP_{i+1}$	this-tp, this-tp-no-trans
S_4	A_i, $\neg AP_{i+1\text{init}}$, $\neg AP_{i+1}$	this-tp, this-tp-no-trans
S_5	A_i, $AP_{i+1\text{init}}$, AP_i	tp-start
S_6	A_i, $\neg AP_{i+1\text{init}}$, AP_i	tp-start
S_7	$\neg AP_i$	cv-no-trans
S_8	A_i, AP_i	cv-trans
S_9	$\neg A_i$	pv-this
S_{10}	$\neg A_i$, AP_i	pv-on-trans
S_{11}	$\neg A_i$, $\neg AP_i$	pv-this-no-trans
S_{12}		final

Type	Start?	States
cannot	yes	S_5, S_6
cannot	no	S_1, S_2, S_3, S_4
must	yes	S_5 plus this-tp label
must	no	S_1, S_3
can	yes (ex. p_0)	S_1, S_2, S_3, S_4, S_5, S_6
can	no or p_0	S_1, S_2, S_3, S_4
CV		S_7, S_8
PM		S_9, S_{10}, S_{11}
final		S_{12}

Table 1: Tables defining translation of abstract states to Büchi states.

Type	Next Init?	Outgoing Labels
can	yes	tp-start, this-tp-no-trans pv-this-no-trans, pv-on-trans
can	no	this-tp pv-this
cannot	yes	this-tp-no-trans, pv-this, cv-trans
cannot	no	this-tp, pv-this, cv-trans
must		tp-start, pv-on-trans, cv-no-trans

Table 2: Determining transitions between states.

More specifically, we connect the transitions for b_k, generated from abstract state B, according to the following algorithm: Let $c_1, \ldots, c_n$ be the states that expand all successors of B in the abstract machine. Let h_k be the set of labels for b_k according to Table 2. For each c_j, add a transition from b_k to c_j iff c_j comes from the same (resp. a different) time point as b_k and the incoming labels for c_j contain some this (resp. non-this label) label from h_k. The fair states consist of the state labeled final and all states expanding abstract states labeled assume.

As an example, let B be the rightmost abstract state for time point 4 from Figure 3. The following diagram shows the expansion. The four states in the dashed box correspond to B. Table 1 (right) tells us to create these states because B matches the second can line. State S_5 expands the abstract state for

time point 5; we include it to illustrate the transition connection procedure.
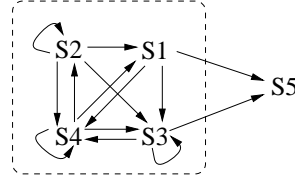
Tables 1 and 2 determine the outgoing transitions for each state in the dashed box. For example, S_3 matches the first row in Table 2 because B has label can and S_3 outputs $AP_{i+1\text{init}}$. Thus, it needs a transition to each state in the dashed box with incoming label this-tp-no-trans (states S_3 and S_4 by Table 1 (left)) and each state outside the box with label tp-start (state S_5). We ignore the pv labels since there are no PM states for time points 4 or 5. A similar process yields the transitions for the remaining states.

Having presented algorithms for translating timing diagrams to both LTL formulas and Büchi automata, we need to check whether the derived formulas and automata correspond on a logical level. Given a timing diagram D, let D_{LTL} and D_{BA} be the formula and automaton derived for D, respectively. We have proven that $\mathcal{L}(D_{\text{BA}})$ models D_{LTL} according to LTL's semantics. As a sanity check on this result, we constructed an LTL formula capturing the structure of D_{BA} and compared it to D_{LTL} using an LTL equivalence checker [10]. These formulas are equivalent

for a large test suite of timing diagrams, including those used in our experiments. Thus, we have high confidence in the correctness of our translations.

5   Experimental Results

This section compares our D_{BA} automata to those derived from D_{LTL} using an existing LTL-to-Büchi translation algorithm [2] with respect to their numbers of states. We do not report running times because the algorithms have been implemented in different paradigms, which reduces the value of such figures; in practice, the direct translations were substantially faster than the LTL-based translations. We report two groups of experiments. In the first, we generate automata for one pass of the timing diagram semantics. In the second, we generate automata for the negation of timing diagrams when treated as an invariant. The latter is required to model check timing diagrams using an automata-theoretic approach.

When comparing how each approach scales with respect to a given timing diagram, there are two classes of parameters to consider: the values of the lower and upper time bounds on the edges and the size of the assume portion. While the bounds certainly affect the size of the resulting automata, we conjecture that the size of the assume portion will be more significant. Consider the structure of D_{LTL}. As Figure 4 shows, the subexpression for the assume portion appears on both sides of the implication in the LTL formula. LTL-to-Büchi algorithms normalize formulas before translation: the normalization process will destroy the similarities between the two copies of the assume portion. Our timing diagram to automaton algorithm, in contrast, translates the assume portion only once. Our experiments use Daniele, Giunchiglia, and Vardi's LTL-to-Büchi algorithm, which yields more compact automata than other algorithms [2].

5.1   Accepting Timing Diagrams

As an initial experiment, consider a very simple diagram with an empty (trivial) assume portion. The table shows the number of states in the D_{BA} automaton (column "D_{BA}") and the number of states

obtained compiling D_{LTL} to an automaton (column "via D_{LTL}"). The first two columns vary the bounds. Each automaton sees constant growth with respect to increases in the time bounds. This supports our hypothesis that the magnitude of the bounds does not yield significant differences between the two translation algorithms. Similar experiments on diagrams with more transitions show similar results: while the magnitude of the constant difference between the two machines increases slightly on these examples, the differences are still small constants when the assume portion is empty.
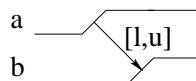
l	u	D_{BA}	via D_{LTL}
1	1	7	9
2	2	10	12
3	3	14	16
4	4	18	20
1	∞	12	17
2	∞	12	16
3	∞	16	20
4	∞	20	24

The picture changes dramatically as the assume portion grows beyond one transition. Consider a diagram with four transitions, as shown below. Each group of three experiments uses the same bounds and varies the assume portion size. The difference between assume portion sizes of one and two is substantial in each group. Furthermore, as the bounds in the assume portion grow, this difference appears to grow exponentially. Growth of each automaton still appears constant across experiments with the same assume portion size and varying bounds. This supports our hypothesis that the size of the assume portion is more important than the size of the bounds. The size of the bounds appear to matter more in the assume portion than in the non-assume portion. This makes sense, as the LTL-to-Büchi algorithm negates the assume portion to construct the automaton. This negation creates many disjunctions, which lead to branching and extra states in the LTL-to-Büchi translation. The larger the bounds, the more disjunctions result from the assume portion.
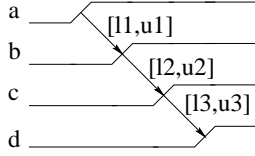
l_1	u_1	l_2	u_2	l_3	u_3	Split	D_{BA}	D_{LTL}
1	1	1	1	1	1	0	9	9
1	1	1	1	1	1	1	11	25
1	1	1	1	1	1	2	12	119
1	1	2	2	2	2	0	15	13
1	1	2	2	2	2	1	17	29
1	1	2	2	2	2	2	18	123
1	1	3	3	3	3	0	23	19
1	1	3	3	3	3	1	25	35
1	1	3	3	3	3	2	26	129
2	2	1	1	1	1	0	12	11
2	2	1	1	1	1	1	14	27
2	2	1	1	1	1	2	16	319
2	2	2	2	2	2	0	18	15
2	2	2	2	2	2	1	20	31
2	2	2	2	2	2	2	22	323
3	3	1	1	1	1	0	16	14
3	3	1	1	1	1	1	18	30
3	3	1	1	1	1	2	20	666

The LTL-to-Büchi approach produces smaller automata than our approach in some cases when the assume portion is empty. We believe this is due to a slight difference in how we handle relationships between the symbolic propositions (A_i, etc) in the two algorithms that would favor the LTL-based approach.

5.2 Rejecting Timing Diagrams

Model checkers require an automaton accepting the negation of a property. Even though we cannot draw the negation of a timing diagram as a timing diagram, we can still produce an automaton that accepts all words that fail to satisfy the timing diagram. This section compares these automata to those obtained for the expression $\neg \mathsf{G} D_{\mathrm{LTL}}$. We present two tables: the first summarizes experiments on the single transition diagram from the previous section and the second summarizes experiments on the two tran-

sition diagram. As an experiment in how the placement of temporal operators affects the construction of automata from LTL formulas, the first table includes an additional column, "Distrib", for which we distributed all X operations in D_{LTL} formula over boolean operators before compiling to an automaton.

l	u	Split	D_{BA}	via D_{LTL}	Distrib
1	1	0	7	112	199
2	2	0	10	310	588
3	3	0	14	654	1506
4	4	0	18	1307	3077
5	5	0	22	2613	6153
1	∞	0	12	295	295
2	∞	0	12	382	772
3	∞	0	16	705	1596
1	8	0	34	14599	24926
2	8	0	34	14632	25055
3	8	0	34	14728	25461
1	1	1	9	117	210
2	2	1	12	315	599
3	3	1	16	659	1519
1	∞	1	14	300	300
2	∞	1	14	387	781

l_1	u_1	l_2	u_2	Split	D_{BA}	via D_{LTL}
1	1	1	1	0	8	650
2	2	2	2	0	14	5372
3	3	3	3	0	22	24174
1	∞	1	∞	0	18	4999
2	∞	1	∞	0	18	6369
2	∞	2	∞	0	18	8286
1	1	1	1	1	10	655
1	1	1	1	2	11	658
1	∞	1	∞	1	20	5004

In these tables, the difference between the two algorithms is striking. The direct translation still shows linear growth as we vary the bounds under a trivial assume portion. For the first section of the first table, the LTL-based algorithm shows exponential growth. The difference between zero and one transitions in the assume portion is not significant for either algorithm in the first table. Unfortunately, we were unable to

generate the LTL-based automata for larger configurations than those shown within a reasonable amount of time (several hours per construction). However, the existing results are sufficient to demonstrate the drawbacks of the LTL approach to compiling timing diagrams into automata.

6   Discussion

The data in Section 5 suggest clear differences between our two approaches for compiling timing diagrams into Büchi automata. These differences could be due to the LTL-to-Büchi automaton translation, to our timing diagram to LTL translation, or to some property of timing diagrams that provides an inherent advantage over LTL.

LTL-to-Büchi algorithms are not canonical, in that they may produce different automata for logically equivalent LTL formulas; the Distrib experiments in the previous section show this. The Daniele *et al.* algorithm produces smaller automata than other algorithms because it uses some simple syntactic optimization techniques on propositional formulas [2]. More work should be done in this area; our timing diagrams research has yielded several formulas where simple manual transformations yielded much smaller automata from the Daniele *et al.* algorithm. Algorithms which perform optimizations across temporal operators are also needed, as our experiments show.

Currently, no known metrics indicate when one LTL formula will yield a smaller automaton than another. Therefore, it is possible that a different translation from timing diagrams to LTL would yield smaller automata. For several timing diagrams, we have tried to manually construct LTL formulas that yield our D_{BA} automata. We have been successful on occasion by translating the structure of D_{BA} into LTL. We are still working on such a translation procedure that acts as a fixpoint over Büchi automata, as a means of understanding the LTL-to-Büchi algorithms better. However, this approach is clearly redundant in practice, as it requires the construction of D_{BA}. We continue to experiment with other timing diagram to LTL translation algorithms, particularly ones which enable sharing of the assume portion.

This project is part of a larger investigation into whether timing diagrams offer any computational benefits over existing logics (including LTL) in verification contexts [4]. We have identified several differences between the two notations. Full timing diagrams and LTL have incomparable expressive powers [5] (this paper uses only a subset of timing diagrams). Timing diagrams enable sharing of common subexpressions to a greater extent than LTL. The LTL formula in Figure 4, for example, duplicates subexpressions across its disjuncts; these expressions correspond to entire suffixes of the timing diagram. LTL does not appear to provide a way to avoid this duplication. However, it is not yet clear whether these duplicated expressions contribute to the explosion in the generated Büchi automata.

The most interesting distinction that we've discovered between timing diagrams and LTL arises from the structure of the Büchi automata corresponding to each notation. Our timing diagram to Büchi translation always produces a particular structure of automaton known as a *weak* automaton [11]. An automaton with states Q and fair set $\mathcal{F}$ is weak if there exists a partition of Q into disjoint sets $Q_1, \ldots, Q_n$ such that (1) each Q_i is either contained in $\mathcal{F}$ or is disjoint from it, and (2) the Q_i's are partially ordered so that there is no transition from Q_i to Q_j unless $Q_i \leq Q_j$. Weak automata have several attractive features in the context of verification [11]; for example, symbolic cycle detection is effectively linear in weak automata, whereas existing algorithms for the general case are quadratic [6].

Another feature of weak automata is important to our study of timing diagrams: complementation of weak automata requires only complementation of the fair set $\mathcal{F}$; the structure of an automaton and its complement are otherwise identical. In Section 5, we explored translations of timing diagrams and their negations to Büchi automata. Our direct translation produces the same size automaton for a given timing diagram under each experiment because we exploit this feature of weak automata.[3] LTL-to-Büchi algorithms do not currently consider weak automata; this is an open problem as many LTL formulas do not cor-

---

[3]We require one extra transition to handle the invariant.

respond to weak automata. When we use LTL as an intermediate language, the Büchi automata for the negated timing diagrams are much larger than in the non-negated case. This is partly due to the structure of the LTL formulas corresponding to timing diagrams. As Figure 4 shows, LTL formulas corresponding to timing diagrams involve disjunctions of long sequences of conjunctions and temporal operators. The negation of such a formula contains many more disjunctions than the original formula. Disjunctions force branching and extra states in Büchi automata. It is therefore not surprising that the automata for the negated timing diagrams are so much larger than those for the one-pass timing diagrams.

In summary, many factors influence the size of the automata obtained when treating timing diagrams as an interface to LTL. These factors suggest a host of research problems in verification. We fully expect that improved LTL-to-Büchi algorithms would reduce the sizes of automata generated in our experiments. Until researchers develop such algorithms, however, direct compilation of timing diagrams to Büchi automata appears a better approach for verification applications.

References

[1] Damm, W., B. Josko and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In Egon Börger, editor, *Specification and Validation Methods*, pages 331–409. Oxford Science Publications, 1995.

[2] Daniele, M., F. Giunchiglia and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11th International Conference on Computer-Aided Verification*. 1999.

[3] Dillon, L., G. Kutty, L. Moser, P. Melliar-Smith and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. Technical report, UCSB, 1993.

[4] Fisler, K. Diagrams and computational efficacy. In review, Proc. of the CSLI Workshop on Logic, Language, and Information, October 1999.

[5] Fisler, K. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8:323–361, 1999.

[6] Fisler, K., R. Fraer, G. Kahmi, M. Y. Vardi and Z. Yang. A new symbolic cycle detection algorithm. In preparation, March 2000.

[7] Gerth, R., D. Peled, M. Y. Vardi and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of Protocol Specification, Testing, and Verification*, pages 3–18, August 1995.

[8] Grass, W. et al. Transformation of timing diagram specifications into VHDL code. In *Proceedings of Computer Hardware Description Languages and Their Applications*, pages 659–668, August 1995.

[9] Heitmeyer, C. On the need for 'practical' formal methods. In *Proc. 5th Intl. Symposium on Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems*, pages 18–26. Springer-Verlag, 1998.

[10] Janssen, G. PTL: A propositional logic tautology checker. Available online from http://www.ics.ele.tue.nl/es/research/fv/ research/research_index.shtml.

[11] Kupferman, O. and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *International Conference on Logic in Computer Science*, 1998.

[12] Kurshan, R. P. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[13] Pnueli, A. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[14] Vardi, M. Y. and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic and Computer Science*, 1986.

Abstraction Relationships for Real-Time Specifications

Monica Brockmeyer

Computer Science Department

Wayne State University

Detroit, MI 48202

mab@cs.wayne.edu

Abstract

This paper introduces the use of abstraction relationships for timed automata. Abstraction relations make it possible to determine when one specification implements another, i.e. when they have the same set of computations. The approach taken here permits the hiding of internal events and takes into account the timed behavior of the specification. A new representation of the semantics of a specification is introduced. This representation, min-max automata is more compact than other types of finite state automata typically used to represent real-time systems, and can be used to define a variety of abstraction relationships.

1 Introduction

This paper describes the use of *min-max* automata to specify the behavior of real-time systems compactly. Originally developed [2] as an alternative representation of timed behavior for the Modechart language[12], in order to support the evaluation of abstraction relationships between Modechart specifications, min-max automata are a general construct for representing the behavior of timed systems. Min-max automata are a more general form of automata than the computation graphs originally developed for Modechart[27], but are more compact than other types of automata which represent the passage of each unit of time as a distinct edge. Thus, min-max automata are more suitable for model-checking and automated evaluation of abstraction relationships between automata.

Abstraction and *refinement* relationships permit the evaluation of whether one specification can replace another. When can one specification replace another? What does it mean for two specifications to have the same behavior or for one specification to have more general behavior? Abstraction permits the substitution of module with a simpler implementation for one that is more complex. In abstraction, modules can be simplified by hiding internal details or by simplifying timing constraints.

There are several important uses for abstraction relations. This work was primarily motivated by the desire to ameliorate the state-space explosion problem which arises in mechanical model-checking. If one specification is an abstraction of another (i.e. it has more general behavior), then all behav-

iors of original are behaviors of the abstraction. Therefore, it may be advantageous to mechanically verify the abstraction rather than the original specification, should it have a more compact representation. Frequently, abstractions are created in an *ad hoc* manner in order to perform model checking. This research provides a formal basis for creating and using abstractions for real-time specifications.

Two other scenarios for using abstraction relations merit discussion. First, abstractions may be applied as part of a "top-down" development procedure. First, a very general specification of a real-time system may be defined. Then, a series of refinements may add increasing detail, resulting in specifications which are more operational. If this sequence of refinements is designed while maintaining an abstraction relation at each step, then properties which have been verified at

In particular, for real-time systems, the process of refinement might include the specification of tighter and tighter timing bounds as assumptions about the environment of a system are refined. the previous step will hold for each refinement step.

The last scenario involves showing an abstraction relationship between two specifications where one represents an implementation and the other represents the properties which must hold. In this case, instead of performing model-checking, one shows that a property, described as a specification, holds for the implementation.

Because of the timed behavior of Modechart specification, it is not possible to use the standard notion of program equivalence used to relate untimed concurrent programs [23]. The usual approach relies on the representation of the system as a *labeled transition system*. The original behavior representation of a Modechart specification [27], a computation graph, is a type of labeled transition system which captures the untimed behavior of a Modechart specification. Timing information is described in associated separation graphs. As a consequence it is not possible to define abstraction relationships directly for computation graphs.

The approach taken here is to represent all timing constraints explicitly in the labeled transition system. Then, the simulation relationships described in the literature can be directly applied. A new type of labeled transition system, *min-max timed automata*, are introduced. Each edge in the automata represents either the passage of time or a discrete system event which takes no time. Min-max automata represent elapsed time with time-passage edges which specify the minimum and maximum amount of time which can elapse between two discrete events.

The rest of this paper is organized as follows: Section 2 introduces both discrete-timed automata and min-max automata. Section 3 describes the extensions to the usual definitions for a *move* in an automata necessary to define abstraction and simulation relationships. Section 4 defines bisimulation and trace inclusion relationships for min-max automata. Conclusions and future work are found in Section 5.

2 Definition of Min-Max Automata

This research is motivated by two goals. First is the ability to mechanically evaluate abstraction relationships between automata representing timed systems. Second is achieving a compact representation of timed systems. These goals are achieved by the use of

min-max automata in which each time passage edge denotes a *range* of possible times elapsed. This results in a more compact representation than other approaches because multiple paths can be collapsed into one.

However, because each path in the min-max automata can potentially represent more than one timed execution of a system, the usual notions of bisimulation and abstraction relations cannot be directly applied.

Definition 2.1. A min-max automata, A is defined as the tuple
$< states(A), initial(A), actions(A), next(A) >$,
where

- The initial states, $initial(A) \subseteq states(A)$,

- The actions of A, $actions(A)$, is the union of the the sets $external(A)$ and $\{\tau\}$ and $times(A) = \{(min, max) : min, max \in \{\mathbf{Z}^+ \cup \infty\} \text{ and } min \leq max\}$ where τ is called the internal action and $times(A)$ are time-passage actions, and

- The next-state relation, $next(A)$ is a subset of $states(A) \times actions \times states(A)$.

Min-max automata, like discrete timed automata, are examples of Lynch's [22] untimed automata. And like discrete-timed automata, the time-passage actions can be used to assign occurrence times to external events in a trace to form a computation.

τ is distinguished as the *internal* action of A. It is considered to be invisible outside of A. If σ is a sequence of actions in $actions(A)$, then $\hat{\sigma}$ is the same sequence with all τ actions removed, and $\bar{\sigma}$ is the sequence with the time actions (elements of $times(A)$) removed.

If $(s, a, s') \subset next(A)$, then the notation $s \xrightarrow{a} s'$ may be used to indicate this. If there is a sequence, σ for which there are states

$s_0, s_1, s_2, \ldots s_n$, such that for all i, $s_i \xrightarrow{\sigma_i} s_{i+1}$, σ is called a finite execution fragment of A, and one can write $s_0 \xrightarrow{\sigma} s_n$. For an infinite sequence, the notation $s_0 \xrightarrow{\sigma}$ is used. A *move* of A, indicated by $s \xRightarrow{\gamma} s'$, occurs if $s \xrightarrow{\sigma} s'$ and $\gamma = \hat{\sigma}$. Thus, a move ignores internal actions.

If s_0 is an initial state of A, then σ is an execution of A. The sets, $execs^*(A), execs^\omega$, and $execs(A)$ indicate the sets of finite, infinite, and executions of A. If the time passage actions are removed, ($\bar{\sigma}$), the resulting sets are the untimed finite, untimed infinite, and untimed executions of A, denoted $execs_U^*(A), execs_U^\omega$, and $execs_U(A)$. If the internal action is removed from an execution of A, $\gamma = \hat{\sigma}$, the resulting sequence is called a trace of A. The sets of traces of A are $traces^*(A), traces^\omega(A)$, and $traces(A)$ for the finite, infinite, and all traces of A. The corresponding untimed traces, $traces_U^*(A), traces_U^\omega(A)$, and $traces_U(A)$ are also defined, for the corresponding $\bar{\gamma}$.

The actions in the set $external(A)$ represent the discrete, externally visible actions of the system. In the context of Modechart, these could represent mode entry, mode exit, and mode transition events which are visible on the interface of a Modechart module. The symbol τ is used to represent internal events which can not be observed externally. Both $external(A)$ and τ events occur instantaneously. The set $external(A) \cup \{\tau\}$ is called $discrete(A)$.

The time passage actions represent the passage of an amount of time between the values of min and max. When they occur in an execution, they represent time elapsing between the instantaneous external and τ events. The values of a time-passage edge, e, are indicated by $min(e)$ and $max(e)$. A *timed event sequence* is a se-

quence $\delta = d_0, d_1, d_2, \ldots$ with $d_i = (a_i, t_i) \in discrete(A) \times \mathbf{Z}^+$ and t_i increasing. If the timed event sequence corresponds to some execution σ of A such that for every $d_i \in \delta$, if $a_i = \sigma_k$ then $t_i \geq \sum\limits_{\substack{0 \leq j < k \\ \sigma_j \in times(a)}} min(\sigma_j)$ and $t_i \leq \sum\limits_{\substack{0 \leq j < k \\ \sigma_j \in times(a)}} max(\sigma_j)$ if $max_j \neq \infty$ for all j. If $max_j = \infty$ for any j, then only the lower bound restriction holds. then δ is called a *timed execution* of A. It can be observed that δ assigns times to the discrete events in σ in a way that is consistent with the time passage events in σ. If the timed event sequence corresponds to a trace of A it is called a *timed computation. comps(A)* indicates the set of timed computations of A.

Figure 1: Complications in matching time passage edges in a min-max automata

3 Issues in Defining Abstraction Relations for Min-Max Automata

Direct application of the definitions for abstraction relations described in the literature is problematic, since each path through a min-max automata represents more than one (timed) computation. As a consequence, soundness and completeness results which hold for the ordinary definitions of abstraction relationships (e.g. bisimulation) will hold for traces of min-max automata, but not necessarily for computations.

Moreover, time-passage edges have some properties which cause unexpected results when the ordinary abstraction relations are applied directly using the usual definition of a move. The definition of a move is relaxed, leading to more powerful abstraction relations.

Example 3.1. Consider the min-max automata P and Q, depicted in Figure 1.

The ordinary definition of a move will not permit the sequence $0 \longrightarrow_P 1, 1 \longrightarrow_P 2$, to be matched to $0 \Longrightarrow 4$ in Q, for any of the abstraction relations described. Yet, the two systems describe the same set of timed computations and have very similar structure.

It should be possible to extend the definition of a move to permit a single time-passage edge to be matched with an appropriate sequence of time-passage edges in the abstraction such that a time passage edge on (m, n) could be matched by a sequence of time passage edges whose minimums sum to m and whose maximums sum to n. However, the definitions of abstraction relationships described in the literature match a single edge to a move. That is, if a min-max automata has an edge with action $(1, 2)$ followed by $(2, 3)$, while it can be said that the automata moves on $(3, 5)$, what move should each of $(1, 2)$ and $(3, 5)$ be matched to in the abstraction automata?

Other approaches (discrete-timed automata[3] and [22] for example) ad-

dress this problem by filling in all the possible time passage edges. In this case, if there were an edge $(3, 5)$ in a min-max automata between points s and s', then there would have to be every possible sequence of edges between s and s' such that the sum of the minimum times was 3 and the sum of the maximum times was 5. However, this defeats the purpose of min-max automata which is to provide a finite and more compact representation of a system, by using min-max time passage edges.

Instead, the problem is addressed by defining a canonical representation for a system. The canonical representation combines all sequences of time-passage edges and replaces them with new edges corresponding to a move. In the example, the sequence $(1, 2), (2, 3)$ would be replaced by a single time-passage edge $(3, 5)$. The abstraction relations are then defined on the canonical representation. A *canonical* representation of a min-max automata, A, denoted $can(A)$, is defined by computing the closure of a min-max automata with regard to the time-passage edges and deleting all but the maximal length edges.

A consequence of computing the canonical representation of a min-max automata is that some points are left unreachable. Since the canonical representation represents the same set of timed computations as the original min-max automata, this is of no consequence. However, the definitions of the simulation relationships must be adjusted to take this into account. The unreachable points are not required to be included in the simulation relations.

Definition 3.1. A point s in a min-max automata is *reachable* if there is a sequence σ such that $s_0 \xrightarrow{\sigma} s$, where s_0 is an initial point of the automata. The set of reachable points

of an automata A is denoted $reachable(A)$.

The abstraction relations will be defined almost identically as in the literature. However, only reachable points will be included and the canonical representation of the min-max automata will be used. This will address the anomaly from Example 3.1.

A second issue is described in Example 3.2.
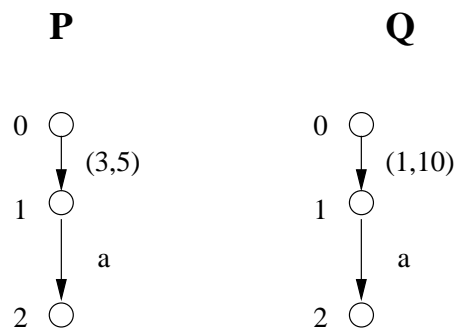
Figure 2: Rationale for a relaxed-time move in a min-max automata

Example 3.2. Consider min-max automata P and Q, depicted in Figure 2.

Then $comps(P) \subset comps(Q)$, but there is no abstraction relationship between P and Q. If the individual computations were represented on separate paths as they are for discrete-timed automata, then an abstraction relation would exist.

This problem is avoided by extending the definition of a move, to permit time-passage edges to be matched to time-passage edges which are inclusive of the times represented by the original edge. That is, a time-passage edge (m, n) will be matched to a time-passage edge (m', n') if $m' \leq m$ and $n \leq n'$.

First, a *time-relaxed step*, relaxes the timing requirements of a time-passage edge.

Definition 3.2. If $s \xrightarrow{(m,n)} s'$, and $m' \leq m$ and $n \leq n'$, then $s \xmapsto{(m',n')} s'$

Next, the definition of a move is expanded to accommodate time-relaxed steps.

Definition 3.3. A *time-relaxed move* of A, indicated by $s_0 \overset{\gamma}{\leadsto} s_n$, occurs if $\gamma = \hat{\sigma}$ where σ is a sequence of states, $s_0, s_1, s_2, \ldots s_n$, such that for all i, $s_i \overset{\sigma_i}{\longrightarrow} s_{i+1}$ or $s_i \overset{\sigma_i}{\mapsto} s_{i+1}$.

By substituting time-relaxed moves for ordinary moves in the definitions of the abstraction relations, the anomaly described in Example 3.2 is avoided. It is now possible to define abstraction relations for min-max automata.

4   Abstraction Relations for Min-Max Automata

This paper now considers the issues of when one specification is an abstraction (or implements) another specification. Trace inclusion or trace equivalence has been widely used to describe when one system implements another [21, 22]. The terms simulation [21], homomorphism [18], and refinement mapping [1] have all been used to reduce the problem of showing trace inclusion to proving something about transitions in some kind of automata. Thus, only a local property needs to be demonstrated. All of these techniques relate systems in terms of the timed behavior of visible events. In each case, the behavior of internal events is hidden. This section describes several such relationships in the context of discrete-timed automata.

4.1   Bisimulation and Forward Simulation

One common technique for showing that two systems are observationally equivalent is called *bisimulation* [25]. Bisimulation involves finding a relation on the states of two systems such that two states being bisimilar means that each state has an edge to a state so that the resulting states are bisimilar. This approach can be relaxed (called *weak* bisimulation) so that an edge in each system is matched by a *move* (including internal events) so that the resulting states are bisimilar. Bisimulation is a rather conservative notion of system equivalence, as it is sound but not complete, but it is widely used especially in process algebras [24].

In order to hide internal events, a sequence of steps, or a *move* is more relevant to the question of whether two automata similarly. A move, as defined above, is a subpath between two points where no intervening events are externally visible. A *weak bisimulation* [25] relaxes the requirement that the two systems proceed in lockstep. Rather, it is only necessary that an edge between two points correspond to a move between two points.

Definition 4.1. For min-max automata, P and Q, $r \subset reachable(can(P)) \times reachable(can(Q))$, is a *weak bisimulation*, if

- for all $p \in initial(P)$, there is some q, such that $(p, q) \in r$ and $q \in initial(Q)$,

- for all $q \in initial(Q)$, there is some p, such that $(p, q) \in r$ and $p \in initial(P)$,

- if $\forall (p, q) \in r$:

  - if $p \overset{e}{\longrightarrow} p'$ then $\exists q' : q \overset{e}{\leadsto} q'$ and $(p', q') \in r$, and

  - if $q \overset{e}{\longrightarrow} q'$ then $\exists p' : p \overset{e}{\leadsto} p'$ and $(p', q') \in r$.

Informally, this states that two points are bisimilar if any edge from one of the points

can be matched by the other point making a move on the same event and reaching a point that is weakly bisimilar to the point reached from the first point. Since weak bisimulations are closed under union, it can be shown that there is a largest weak bisimulation, denoted $\approx$, for any pair of computation graphs for a given set of observable events.

The following theorem establishes the soundness of bisimulation.

Definition 4.2. The notation $comps(P) \equiv comps(Q)$ indicates $comps(P) \subseteq comps(Q)$ and $comps(Q) \subseteq comps(P)$.

Theorem 4.1. $P \approx Q \implies comps(P) \equiv comps(Q)$.

Proof. Similar to the proof for ordinary timed automata found in the literature [22]. The proof is in [2], which shows that the extensions to the definition of a move do not violate the conditions of the usual proof. $\square$

Bisimulation is not complete. That is, there are systems which have the same set of timed traces, but which are not bisimilar. This is because bisimulation captures some aspects of system structure. Each point must be bisimilar to a point in the other system which permits actions which move to points which are bisimilar to those which can occur in the original specification. As a consequence, bisimulation distinguishes with regard to the state of the system as well as the sequence of actions or events.

4.2 Forward Simulations

If the definition of bisimulation is modified to apply in only one direction, the result is called a *forward simulation* [21]. Forward simulations are also related to simulations [28, 13], history measures [17], downward simulations

[9, 11, 15], and possibilities mappings [20]. Because the restriction is in one direction, a forward simulation shows trace inclusion rather than trace equivalence.

In practice, this approach is desirable. Often a general purpose specification will be designed as well as an implementation or operational specification which has a narrower set of behaviors. It is not necessary for the implementation to have the full set of behaviors as the specifications. Alternatively, perhaps a simplification can be made to a specification which reduces the size of the computation graph, but which admits a larger set of behaviors. If the a trace inclusion relationship holds between the two systems, then it may be possible to model-check the simpler system and apply the results to the more complicated system.

Definition 4.3. For min-max automata, *forward* simulation from P to Q is a relation f over $reachable(can(P))$ and $reachable(can(Q))$ a *forward* simulation if:

- for all $p \in initial(P)$, there is some q, such that $(p, q) \in f$ and $q \in initial(Q)$,

- if $\forall (p, q) \in f$ and all $e \in actions(P)$, $p \xrightarrow{e} p'$ then $\exists q' : q \overset{e}{\rightsquigarrow} q'$ and $(p', q') \in f$.

Lynch [21] shows that forward simulations are a pre-order (i.e. they are reflexive and transitive). Soundness follows from the soundness of bisimulations.

4.3 Forward-Backward Simulations

Forward-Backward simulations were also described by Lynch and are similar to the invariants and ND-measures of [16, 17] as well

as subset simulations [14], and simple failure simulations [7]. They are less restrictive than forward simulations. Perhaps, most noteworthy is that they are complete for trace inclusion. However, since a single trace of a min-max automata can represent more than one timed computation, forward-backward simulations are not complete for timed computations.

Definition 4.4. For min-max automata, *forward-backward* simulation from P to Q is a relation fb over $reachable(can(A))$ and $\mathbf{N}(reachable(can(B)))$ [1] such that:

- for all $p \in initial(P)$, there is some set A, such that $(p, A) \in fb$ and $A \subseteq initial(Q)$,

- if $p \stackrel{e}{\longrightarrow} p'$ and $(p, A) \in fb$, then there exists a set A' such that $(p', A') \in fb$ such that for every $q' \in A'$ there is some $q \in A$ such that $q \stackrel{e}{\rightsquigarrow} q'$.

P   Q
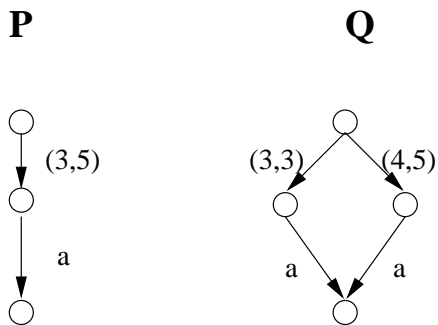
(3,5)   (3,3)   (4,5)

a   a   a

Figure 3: Completeness Problem for Min-Max Automata

Example 4.1. To understand why forward-backward simulations are not complete for min-max automata, consider min-max automata, P and Q, depicted in Figure 3.

---

[1]For a set X, $\mathbf{N}(X)$ indicates the set of non-empty subsets of X.

Then, $comps(P) \equiv comps(Q)$ but it is not the case that $P \leq_{FB} Q$, because there is no match for the time-passage edge, $(3, 5)$.

Therefore, further research is required to find an abstraction relation which is complete for computations of min-max automata.

4.4 Homomorphisms and Refinements

Homomorphisms [8, 18] and refinement mappings [1, 19, 21], are more restrictive than forward simulations, because they require a *function* from $states(P)$ to $states(Q)$ rather than a relation.

Definition 4.5. For min-max automata, P and Q, a function f between $reachable(can(P))$ and $reachable(can(Q))$, is a *refinement* if:

- for all $p \in initial(P)$, $f(p) \in initial(Q)$,

- if for all $e \in actions(P)$ $p \stackrel{e}{\longrightarrow} p'$ then $f(p) \stackrel{e}{\rightsquigarrow} f(p')$

The proof of soundness for forward simulations, forward-backward simulations, and refinements is similar to that for bisimulations.

Another interesting type of relationship between two automata is failures inclusion or equivalence, developed by Hoare [4, 10]. An alternative characterization, given by Hennessy and de Nicola [6], is called testing equivalence in which equivalent automata pass or fail the same set of tests. Testing and failures relationships cannot be characterized by matching an edge in one automata with some kind of move in another automata and so are not discussed in this paper.

5 Conclusions and Future Work

This paper has introduced *min-max automata* which are a compact form of timed automata suitable for mechanical evaluation of simulation and abstraction relationships. Extensions to the definition of a move necessary to support simulation and abstraction relationships were defined and several types of equivalence and abstraction/simulation relationships were described in the context of min-max automata. Related research efforts extend these ideas by describing automatic generation of abstractions [2].

Future work involves integration of min-max automata into existing software tools to automatically generate min-max automata for Modechart specifications and to automatically check for the simulation and abstraction relationships defined in this paper. The Modechart Toolset [5, 26] provides a graphical interface for editing, consistency-checking, simulation, and verification of real-time specifications in the Modechart Language. This will permit evaluation of the techniques on real-world examples. Future work is also required to define an abstraction relationship which is complete for trace inclusion of min-max automata.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–281, 1991.

[2] M. Brockmeyer. *Monitoring, Testing, and Abstractions of Real-Time Specifications*. PhD thesis, The Department of Electrical Engineering and Computer Science, The University of Michigan, 1999.

[3] M. Brockmeyer. Using modechart modules for testing formal specifications. In *Proceedings of the High Assurance Systems Engineering Workshop*. IEEE, 1999.

[4] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, pages 560–599, 1984.

[5] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[6] R. de Nicola and M. C. Hennessy. Testing equivalences for processes. *Journal of Theoretical Computer Science*, pages 83–133, 1983.

[7] R. Gerth. Foundations of compositional program refinement. In *Proceedings REX Workshop on stepwise refinement in distributed systems: Models, Formalism, Correctness, Lecture Notes in Computer Science*, volume 430, pages 777–808, 1987.

[8] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press, 1968.

[9] J. He. Process simulation and refinement. *Journal of Formal Aspects of Computing Science*, 1:229–241, 1989.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[11] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

[12] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Software Engineering*, 20(10), 1994.

[13] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Upsaala University, 1987.

[14] B. Jonsson. Simulations between specifications of distributed systems. In *Proceedings Concur '91, Lecture Notes in Computer Science*, volume 527, pages 347–360. Springer-Verlag, 1991.

[15] M. B. Josephs. A state-based approach to distributed processing. *Distributed Computing*, 3:9–18, 1988.

[16] N. Klarlund and F. Schneider. Verifying safety properties using infinite state automata. Technical Report 89-1039, Department of Computer Science, Cornell University, 1987.

[17] N. Klarlund and F. Schneider. Proving non-deterministically specified safety properties using progress measures. *Information and Computation*, 171(1):151–170, November 1993.

[18] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Princton University Press, 1994.

[19] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages*, 5:190–222, 1983.

[20] N. Lynch. Multivalued possibilities mappings. In *Proceedings REX Workshop on stepwise refinement in distributed systems: Models, Formalism, Correctness, Lecture Notes in Computer Science*, volume 430, pages 519–543, 1987.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations – part i: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations – part ii: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[24] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[25] D. Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science*, 104, 1980.

[26] A. Rose, M. Perez, and P. Clements. Modechart toolset user's guide. Technical Report NRL/MRL/5540-94-7427, Center for Computer High Assurance Systems, Naval Research Laboratory, Washington, D.C., February 1994.

[27] D. Stuart. Implementing a verifier for real-time systems. In *Real-Time Systems Symposium*, pages 62–71, Orlando, FL, December 1990.

[28] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University, The Netherlands, 1990.

Algebra of Behavior Tables[*]

Steven D. Johnson and Alex Tsow

Indiana University Computer Science Department
sjohnson@cs.indiana.edu

Abstract

A design formalization based on behavior tables *was presented at Lfm97. This paper describes ongoing work on a supporting tool, now in development. The goal is to make* design derivation*, the interactive construction of correct implementations, more natural and visually palatable while preserving the benefits of formal manipulation. We review the syntax and semantics of behavior tables, introducing some new syntactic elements. We present a core algebra for architectural refinement, including new notational conventions for expressing such rules.*
KEYWORDS: behavior table, design derivation, formal synthesis.

1. Introduction

Behavior table notation emerged out of case studies in formal *design derivation* between 1985 and 1995. The *DDD transformation system* [7] is based on functional algebra. Behavioral expressions at the level of *algorithmic state machines* [1] are represented by recursive systems of function definitions, and architecture oriented implementations are represented by recursive systems of stream expressions. In DDD, these representations are manipulated as transformations on Scheme programs, so the expressions are also executable.

The primary goal in our early case studies was to interactively impose hardware architectures on algorithmic specifications. As these studies became larg-er, a practice emerged of printing DDD expressions in a tabular form, reminiscent of register transfer tables. The tables helped design teams visualize their architectural goals so they could strategize about how to accomplish them in the DDD algebra.

We began to contemplate using the tables more directly as formal objects, retargetting the DDD algebra to operate on tabular representations. We believe the tables are more perspicuous to practicing professionals who, it has been claimed, are put off by the notation used in formal reasoning systems.

The rising visibility of tabular specification languages such as *Tablewise* [3], *SCR** [2], and *and-or* transitions in *RSML* [8], helped convince us to look at behavior tables more seriously as a formalism rather than merely as a visual aid. Subsequently, we have undertaken to develop a tool for interactive design derivation using them.

In this paper, we develop a core algebra for architectural manipulation. In the main, this algebra correlates to the "structural" algebra of *sequential systems*, presented in [5]. Although the main purpose is to lay the groundwork for tool implementation, one ancillary contribution of this paper is its notational conventions for stating the rules of the algebra, which use *table schemes* to simplify quantification.

The conclusion lists additional topics and issues entailed in the implementation effort. We extend the term-level syntax presented at Lfm97 [6] to include provisions for *bounded indirection*, additional algebra for a simple kind of *data refinement*, and possible extensions for *verification*.

2. Terms

Behavior tables are arrays of *terms* in a ground vocabulary of constants and operations. We very briefly review the terminology of first order structures then introduce the extensions that are assumed in behavior tables.

A *first order structure* describes a family of value sets, A_1, ..., A_n, together with a collection of total functions, f_1, ..., f_m, on these sets. With each set A_i is associated a type symbol, τ_i,. There are *constant* and *operator* symbols representing the functions f_i, and a distinct set of *variable* symbols. The notation $v \colon \tau_i$ asserts that the variable v ranges over values in A_i. The *signature* of an operator specifies its domain and range, which in general are nested products. The formula $f \colon (\tau_1, (\tau_2, \tau_3)) \to (\tau_4, \tau_5, \tau_6)$ asserts that the operation f maps the product $A_1 \times (A_2 \times A_3)$ to the product $A_4 \times A_5 \times A_6$. We shall allow for *multioutput* operations, as suggested here, whose output signatures are n-tuples.

A *term* is a variable, constant, or application, $f(T_1, \ldots, T_n)$, of an operation f to the terms T_i according to the f's signature.

A structure becomes an equational algebra when it is provided with a set E of equational *identities* among terms (over a distinguished set of *logical variables*). E induces an equivalence relation; and we write $\models_E s \equiv t$ to express the fact that s and t are provably equivalent under E.

Certain additional features are assumed of all structures used in behavior tables and are thus absorbed at the metalinguistic level.

- A sort Bool is assumed with constants true and false and the identities of *boolean algebra*. Operations with range Bool are called *tests*.

- A *don't care* constant is designated by '♮'.

- Finite product (tupling) and projection operations of each type are assumed Projections are denoted by sans-serif adjectives, 1st, 2nd, 3rd, 4th, 5th ..., ith, ... An n-tuple is expressed as a parenthesized series of n terms, $(T_1, \ldots, T_n)$. Projections applied to n-tuples can be simplified

at the syntactic level; for instance,

$$\models \mathsf{2nd}(T_1, T_2, T_2) \equiv T_2$$

- It is assumed that arbitrary finite sets of *tokens* can be represented (e.g. by n-tuples over Bool). We shall extend this idea to what Hoover calls a *finite logic* [3], with which we associate a specific selection operation, written

$$
\begin{aligned}
&\mathsf{case}\ s\ \mathsf{of} \\
&\quad \mathsf{a}_1 \quad : \quad t_1 \\
&\qquad\qquad \vdots \\
&\quad \mathsf{a}_k \quad : \quad t_1
\end{aligned}
$$

The usual treatment of terms is extended for explicit multioutput operations. The definition of substitution on terms is adapted for multioutput operations by allowing nested lists of variables to serve as substitution patterns. Such a list is called an *identifier*.

Definition 1 *An* identifier *is either a variable or a nested list,* $(X_1, \ldots, X_n)$, *of* distinct *identifiers, meaning that they share no common variables.*

Definition 2 *The formula* $T[R/X]$ *denotes a substitution of the term R for the identifier X in the term T. The formula* $T[R_1/X_1, \ldots, R_n/X_n]$ *denotes the simultaneous and respective substitutions of terms R_i for identifiers X_i, $i \in \{1 \mathinner{\ldotp\ldotp} n\}$. Substitution is defined by induction on the language of terms. In the base cases, constants are unchanged and for a variable symbol u,*

$$u[R/X] = \begin{cases} R & \text{if } X = u \\ u & \text{if } X \neq u \end{cases}$$

For applications and n-tuples,

$$f(T_1, \ldots, T_n)[R/X] = f(T_1[R/X], \ldots, T_n[R/X])$$

For nested identifiers, a simultaneous substitution is done on the constituents:

$$T[R/(X_1, \ldots, X_n)] = T[\mathsf{1st}(R)/X_1, \ldots, \mathsf{nth}(R)/X_n]$$

In the last case, substitution of an n-tuple for an n-element identifier simplifies to

$$\begin{aligned} &T[(R_1, \ldots, R_n)/(X_1, \ldots, X_n)] \\ &\quad = T[R_1/X_1, \ldots, R_n/X_n] \end{aligned}$$

3. Syntax of behavior tables

Behavior tables are closed expressions whose terms contain variables from three disjoint sets: I (inputs), S (sequential signals, or data state), and C (combinational signals). Fix these sets for the remainder of this section. We will write ISC for $I \cup S \cup C$ and SC for $S \cup C$. We use the term "register" for an element of S, but this is a euphemism that should be interpreted very abstractly. There is no assumption that these variables denote finite values, nor are tables intended only for register-transfer specification. The form of a behavior table is:

| $Name$: $Inputs \rightarrow Outputs$	
$Conditions$	$Registers\ and\ Signals$
$\vdots$	$\vdots$
$Guard$	$Computation\ Step$
$\vdots$	$\vdots$

$Inputs$ is a list of input variables and $Outputs$ is a set of terms over ISC, but without loss of generality, assume $O \subseteq SC$. $Conditions$ is a set P of predicates over ISC, that is, finitely typed terms ranging over finite types, such as truth values, token sets, etc.

The notion of term evaluation used here is standard. The value of a term, t, is written $\sigma[\![t]\!]$, where σ is an *assignment* or association of values to variables.

Definition 3 *A* guard *is a set of constants indexed by a condition set P: $g = \{c_p\}_{p \in P}$. A decision table* $\mathbf{D} = [P, G]$*, consists of a condition set and a an associated list of guards. We say g holds for an assignment σ to ISC when, for each $p \in P$, either $c_p = \natural$ or $\sigma[\![p]\!] = c_p$.*

Following [3], we say a decision table is *functional* when G describes a proper partitioning of the possible assignments to ISC. In other words, the guards are "consistent" and "complete".

Definition 4 *A* computation step *or* action *is a set of terms, one for each register and signal: $a = \{t_v\}_{v \in SC}$. An* action table *is a set of actions typically indexed by the guards of a corresponding decision table.*
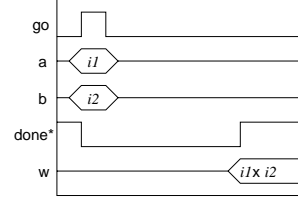
MULT:(go, a, b) $\rightarrow$ (done*, w)						
go	P	(even? u)	u	v	w	done*
1	$\natural$	$\natural$	a	b	0	$P \wedge \neg$go
0	1	$\natural$	$\natural$	$\natural$	w	$P \wedge \neg$go
0	0	1	u÷2	v×2	w	$P \wedge \neg$go
0	0	0	u÷2	v×2	w+v	$P \wedge \neg$go

where $P \equiv$ (zero? u) $\vee$ (zero? v)

Figure 1: Example of a behavior table

Definition 5 *A behavior table for $I \rightarrow O$ consists of a decision table, $\mathbf{D}$, with guards $G = \{g_1, \ldots g_n\}$, and an action table indexed by G, $\mathbf{A} = \{t_{v,k} \mid v \in SC \text{ and } g_k \in G\}$.*

Figure 1 shows a shift-and-add multiplier, expressed as a behavior table. The timing diagram is provided to explain the interface, with multiplication performed within a full handshake.

4. Synchronous semantics

A behavior table $[\mathbf{D}, \mathbf{A}]$ for $O \subseteq SC$ denotes a relation between infinite input and output sequences. We call these sequences *streams* because in prior work we obtain a semantics by interpreting a table as a (co)recursive system of stream-defining equations [7]. More directly, suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I. Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \ldots \rangle$ for ISC as follows:

(a) $\sigma_n(i)$ is given for all $i \in I$ and all n.

(b) For each $s \in S$, $\sigma_0(s) = x_s$.

(c) $\sigma_{n+1}(s) = \sigma_n[\![t_{s,k}]\!]$ if guard g_k holds for σ_n.

(d) For each $c \in C$, $\sigma_n(c) = \sigma_n[\![t_{c,k}]\!]$ if guard g_k holds for σ_n.

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \ldots \rangle$. This semantic relation is well defined if there are no circular dependencies among the combinational actions $\{t_{c,k} \mid c \in C,\ g_k \in G\}$. The relation is a function (i.e. deterministic) if decision table **D** is functional. We shall restrict our attention to behavior tables that are *well formed* in these respects. In essence, well formedness reflects the usual properties required of synchronous finite state machines.

To achieve well formedness, we constrain behavior tables in two ways. First, we prohibit "combinational feedback" in the actions. Given row k in the action table $\{t_{v,k} \mid v \in SC\}$, there is a natural dependence graph with vertices corresponding to the signal names and edges given by the relation: $a \to b$ iff a is a subterm of $t_{b,k}$. Checking for combinational cycles is a straightforward depth-first search.

Even if the actions themselves do not contain combinational loops, the decision table can still induce race conditions or metastable behavior. Consider the following table fragment where r and c are registered and combinational boolean signals:

$B : I \to O$					
r	$c*$	$\cdots$	r	$c*$	$\cdots$
0	0		0	1	
0	1		0	0	
1	0		1	1	
1	1		1	1	

Intuitively, if the system makes a transition into a state where $\sigma_n(r) = 0$, then combinational signal a will oscillate. Our semantics is not well defined in this case: if $c_r = 0$ and $c_c = 0$ in some guard $g_k = \{c_p\}_{p \in P}$ at timeslice n, then $\sigma_n(c) = 1$ by (d). Since g_k no longer holds at σ_n, some other guard $g_j = \{d_p\}_{p \in P}$ in which $d_r = 0$ and $d_c = 1$ hold changes $\sigma_n(c)$ back to 0.

The race condition occurs in our example when $\sigma_n(r) = 1$ and $\sigma_n(c) = 0$. Although one could argue that σ_n is well defined, we shall prohibit this mode of expression anyway, as it reflects a kind of transition race.

To eliminate these scenarios, we constrain the predicates of the decision table to use only registered variables and input signals. This way, no action can directly change the guard g_k since the values of registered signals persist for the duration of the present action (c).

In addition, we shall require a functional set of guards, as noted earlier. This results in deterministic and total behavior, for which the algebra presented here is intended.

We think of behavior tables as denoting persistent, communicating processes, rather than subprocedures. In other words, behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Composition is specified by giving a connection map that is faithful to each component's arity. In our function-oriented modeling methodology, such compositions are expressed as recursive systems of equations,

$$\lambda(U_1, \ldots, U_n).(V_1, \ldots, V_m) \ where$$

$$
\begin{aligned}
(X_{11}, \ldots, X_{1q_1}) &= \mathcal{T}_1(W_{11}, \ldots, W_{1\ell_1}) \\
&\vdots \\
(X_{p1}, \ldots, X_{pq_p}) &= \mathcal{T}_p(W_{p1}, \ldots, W_{p\ell_p})
\end{aligned}
$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

Valid systems must preserve I/O directionality, excluding both combinational cycles and output conflicts. Checking validity has two stages and is again a graph problem:

1. For each behavior table let its inputs and outputs be vertices, and let $i \to o$ when output signal o combinationally depends on input signal i.

2. Add the following edges to the disjoint union of the behavior table I/O graphs: $o \to i$ when $\mathcal{T}_j(\ldots, o, \ldots)$ is the right hand side of an equation where $\mathcal{T}_j$'s I/O signature is $\mathcal{T}_j : (\ldots, i, \ldots) \to O$.

A legitimate connection network exists when this graph has no cycles.

Provided they are well formed, deterministic systems are readily animated in modeling languages that allow recursive stream networks to be expressed [4]. As long as each register has an initial value, the streams are constructed head-first as a fixed-point computation. Translation to both cycle-based and event-based simulation languages is also relatively straightforward, as long as the systems are expressed over the concrete data types these tools recognize.

A synchronous semantics is simple and suited to the clocked implementation models most high-level synthesizers use. In fact, behavior tables will acquire a range of semantics, depending on their applications, just as HDLs and programming languages do. Even with a variety of interpretations, their inherent structure helps reduce the mathematical bookkeeping that often obscures semantic definitions.

5. Behavior Table Algebra

The collection of transformation rules presented in this section applies to architectural refinement. This set is not claimed to be complete nor is minimal in any mathematical sense. At this stage, our principal object is to build a set of rules that is robust enough to serve as a core rule set for tool implementation. mathematical efficiency is a secondary concern, for the moment.

5.1. Notational conventions

Defining these rules has led to some stimulating notational issues. In attempting to present the rules in a clear way, we have been led to consider some novel conventions for expressing features, particularly for quantification. For reasons of both typography and clarity, we want to reduce use of ellipses, columns,

and subscripts to describe a table as, for example,

$b \colon (I_1, \ldots, I_k) \to (O_1, \ldots, O_\ell)$					
P_1	$\cdots$	P_m	S_1	$\cdots$	S_p
1 $\quad g_{11}$	$\cdots$	g_{1m}	t_{11}	$\cdots$	t_{1p}
$\vdots$	$\ddots$		$\vdots$	$\ddots$	
n $\quad g_{n1}$		g_{nm}	t_{n1}		t_{np}

Our *table scheme* notation uses the table itself as a quantifier, and uses set elements as indexes rather than number ranges. Uppercase italic variables denote sets; and differently named sets are always assumed to be finite and disjoint. Lowercase *italic* variables denote indices ranging over sets of the same name. The form

$$R \boxed{\begin{matrix} S \\ x_{rs} \end{matrix}}$$

represents a two-dimensional array (table) of items, $\{x_{rs} \mid r \in R \text{ and } s \in S\}$. A san seriff[1] identifier denotes a fixed (throughout the scope of the rule) element from the set of the same name. Thus, the form

$$R \boxed{\begin{matrix} \mathsf{s} \\ x_{r\mathsf{s}} \end{matrix}}$$

represents a column, $\{x_{r\mathsf{s}} \mid r \in R\}$, and similarly for rows.

Under these conventions, the table scheme from Section 3 looks like

$b \colon I \to O$	
P	S
1 $\quad g_{n,p}$	$t_{n,s}$
$\vdots$	
N	

The use of ellipses $1 \cdots N$ on the left is not necessary, but serves as an reminder that the rows are typically numbered. That is, we usually take the set N to be the first "N" numbers.

5.2. The rules

Some structural rules subsumed by the semantics, must be implemented in the tool. For example, interchanging rows and columns is allowed since indices

---

[1]Where possible, we display these identifiers in red.

range over sets, not sequences. The underlying semantics remain well defined because the order of equations in a system is irrelevant. Similarly, *renaming* variables is allowed under the usual rules of α substitution[2].

The rules fall into three groups, the first involving both the decision and action table parts, the second being operations on the action table part, and the third being operations that affect the decision table part.

Replacement

$b\colon I \twoheadrightarrow O$		
	P	S
n	g_{np}	t_{ns}

$$\models t_{ns} \equiv u_{ns} \qquad \Downarrow$$

$b\colon I \twoheadrightarrow O$		
	P	S
n	g_{np}	u_{ns}

One term can be replaced by another term that is (proven to be) equivalent in the underlying structure (or theory). Recall that $\models t \equiv u$ is a provable equivalence in the underlying structure. In practice, establishing equivalence would be done with a rewriting tool or proof assistant.

Decomposition

$b\colon I \twoheadrightarrow O$		
	P	S	T
1 $\vdots$ N	g_{np}	t_{ns}	t_{nt}

$$\Downarrow$$

$b_1\colon I \cup T \twoheadrightarrow O \cap S$		$b_2\colon I \cup S \twoheadrightarrow T \cap O$	
P	S	P	T
g_{np}	t_{ns}	g_{np}	t_{nt}

Decomposition splits one table into two, both inheriting the same decision table. The *compose* operator connects the two tables to maintain the original dependence among the signals. Interpreting the tables as functions on streams—and reading '$\cup$ and '$\cap$ as list operations—$\mathcal{B}_1 \circ \mathcal{B}_2$ yields the system

$$\mathcal{B}(I) \stackrel{\mathrm{def}}{=} O \ \ where$$
$$(O \cap S) \ = \ \mathcal{B}_1(I \cup T)$$
$$(O \cap T) \ = \ \mathcal{B}_2(I \cup S)$$

It is a background job of the table editor to maintain the connection hierarchy as a byproduct of decomposition. An upward *composition* transformation ($\Uparrow$), if formulated, would require conditions to exclude name clashes and preserve well formedness. In using tables for design derivation, one would typically decompose tables rather than compose them.

This is by no means all there is to say about composition. This strong (in the sense of not being very general) form of the Decomposition rule is essentially a partitioning rule, allowing one to to impose hierarchy on designs.

---

[2]Actually, behavior tables do not have free variables, so α conversion is even simpler.

Conversion

$b\colon I \rightharpoonup O$		
p	**q**	**s**
J $g_{j\mathsf{p}}$	$v_{j\mathsf{q}}$	$t_{j\mathsf{s}}$

$$\Updownarrow \quad \begin{array}{l} \forall j, j' \in J\colon g_{j\mathsf{p}} \equiv g_{j'\mathsf{p}} \\ \bigcup_{j \in J} v_{j\mathsf{q}} = \mathrm{dom}(\mathsf{q}) \end{array}$$

$b\colon I \rightharpoonup O$		
p	**q**	**s**
g_{p}	♮	case **q** $\{\, v_{j\mathsf{q}} : t_{j\mathsf{s}} \,\}_J$

This rule, allowing function to be moved between the decision and action parts of a table, provides the means to change the boundary between control and architecture. The side conditions say that, within the range indicated by J, the guards outside column **q** agree, and the guards within column **q** are exhaustive.

Action collation

$b\colon I \rightharpoonup O$		
P	**s**	**r**
$1 \cdots N$ g_{np}	$t_{n\mathsf{s}}$	$t_{n\mathsf{r}}$

$$\Downarrow \quad \begin{array}{l} (\textit{defined}) \\ (\textit{compatible}) \\ (\textit{well formed}) \end{array}$$

$b\colon I \rightharpoonup O$		
P	**s**	**r**
$1 \cdots N$ g_{np}	$t_{n\mathsf{s}} \diamondsuit t_{n\mathsf{r}}$	**s**

The idea behind collation is that two, or several, compatible signals can be merged into one by instantiating don't-cares. The '$\diamondsuit$' operator denotes term-level

instantiation,

$$t \diamond t' = \begin{cases} t & \text{if } t' = \natural \\ t' & \text{if } t = \natural \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Compatible means that both variables must be combinational or both must be sequential. If both signals are combinational, an audit is required to assure that the resulting system remains well formed, that is, that instantiation does not introduce feedback.

Action identification

$b\colon I \rightharpoonup O$	
P	S
$1 \cdots N$ g_{np}	t_{ns}

$$\textit{combinational} \quad \Updownarrow$$

$b\colon I \rightharpoonup O$		
P	S	**y**
$1 \cdots N$ g_{np}	$t_{ns}\ [\,\mathsf{y}\,/\,r_{n\mathsf{y}}\,]$	$r_{n\mathsf{y}}$

In terms of systems, this is the recursion rule, stating that **y** is equal, in a logical sense, to its defining equation, and hence that one can be replaced for the other. In fact, this rule can be applied on a row-by-row basis, but we give the full-column version to reflect the more typical case when a common subterm is being identified. If **y** were a sequential variable, it would acquire the value $r_{n\mathsf{y}}$ in the next step and so the replacement is invalid.

Action introduction

$b\colon I \rightharpoonup O$	
P	S
g_{np}	t_{ns}

(rows $1 \ldots N$)

y *fresh*
 well formed $\Updownarrow$ y *unused*

$b\colon I \rightharpoonup O$		
P	S	y
g_{np}	t_{ns}	r_{ny}

(rows $1 \ldots N$)

A new action column can be added ($\Downarrow$) as long as the signal name is not redundant and, in the case of combinational signals, the action terms do not refer to the signal being introduced.

Action grouping

$b\colon I \rightharpoonup O$	
P	s
g_{np}	t_{ns}

(rows $1 \ldots N$)

(*both comb. or both seq.*) $\Updownarrow$

$b\colon I \rightharpoonup O$		
P	″1(s)″	″2(s)″
g_{np}	$1(t_{ns})$	$2(t_{ns})$

(rows $1 \ldots N$)

Columns can be grouped and ungrouped as long as the resulting columns are purely sequential or purely combinational. Thus, one canonical form for action tables has just two columns. Recall that signals names are nested identifiers; the notation '″1(s)″' means that ungrouping transformations require *explicit* tuples in the header fields, and destructure them in the obvious way. For instance, if s $\equiv$ (a, b) the ungrouped columns will be headed with a and b.

Action table entries need not be explicit tuples, although they can be, because 1, 2, etc. are legitimate operators.

Decision grouping

$b\colon I \rightharpoonup O$	
p	S
(d_{np}, e_{np})	t_{ns}

(rows $1 \ldots N$)

compatible $\Updownarrow$

$b\colon I \rightharpoonup O$		
1(p)	2(p)	S
d_{np}	e_{np}	t_{ns}

(rows $1 \ldots N$)

As with action tables, decision table columns can be grouped into tuples. In contrast, the entries are values and the headers are terms, so explicit use of detupling projectors is allowed in both.
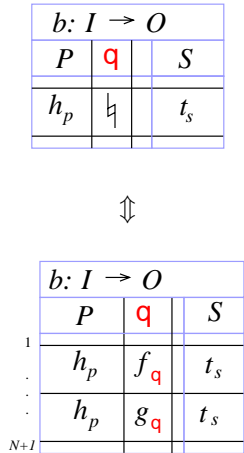
Decision introduction

$b\colon I \rightharpoonup O$	
P	S
g_{np}	t_{ns}

(rows $1 \ldots N$)

Q finite $\Updownarrow$

$b\colon I \rightharpoonup O$		
P	Q	S
g_{np}	$\natural_{nq}$	t_{ns}

(rows $1 \ldots N$)

One can introduce a new test with don't-care criteria. The underlying intent of this rule is its use in ad hoc table constructions. A possible well formedness restriction on this rule is that the resulting table be safe from race conditions. Such a restriction can, in principle, be applied when decisions are instantiated (see just below), yielding a more general algebra.

Decision instantiation

$b\colon I \to O$		
P	q	S
h_p	♮	t_s

$\Updownarrow$

	$b\colon I \to O$		
	P	q	S
1	h_p	f_{q}	t_s
.	h_p	g_{q}	t_s
$N{+}1$			

Having introduced a new test to a behavior table, instantiation is used to do case splitting. In the simplest case, suppose that a ♮ appears in a decision table entry. Then this rule provides for expanding that row into enough duplicates to account for all the possible values of the test. In the upward direction, the rule gives us a way to combine rows whose actions are identitical. The notation $f_{\mathsf{q}} \cup g_{\mathsf{q}}$ anticipates allowing for decision table entries to be sets of values, as is seen in requirements specification languages.

I/O restriction

Input and output signals may be added to behavior tables without concern so long as the inputs and outputs of the encapsulating system remain the same. Such additions cannot introduce combinational feedback until they are used, and the decision/action introduction rules check for well formedness.

Conversely, an unused I/O signal qualifies for removal. We can remove input i to a behavior table if no action or predicate contains i as a subterm. A behavior table output may be removed when is unused in the surrounding interconnect expression.

6. Other aspects

This paper has developed a core algebra of behavior table manipulation for architectural refinement. In practice, the product of such manipulation is a decomposition of the specification into subsystems for synthesis into hardware or compilation into embedded software components. This section briefly describes a number of other immediate issues and aspects entailed in the development of a design tool.

Figure 2 shows a derivation decomposing a behavior table into two components, one allocating two arithmetic operations to a single device. This is an example of a *system factorization*, a fundamental transformation in the DDD algebra [5], and the instance in the figure comes from an illustration in Johnson's *Lfm97* presentation of behavior tables. The example shows that the algebraic rules presented in this paper are much more finely grained than the transformations that typically would be used in an interactive setting, but would instead serve as a core set of rules from which larger-scale ones are composed.

6.1. Stream semantics

Given a behavior table, one can construct an equivalent sequential system by repeated applications of the Decomposition and Conversion rules. Use Decomposition to separate every column of the action table, then Conversion to reduce each of the resulting tables to a single row. The resulting nested system description can be flattened and simplified. Alternatively, Decomposition can be generalized to simultaneously split tables into several components. To complete the transformation, we must make initialization of the sequential signals explicit. The resulting system is

$$\mathcal{B}(I) \overset{\mathrm{def}}{=} O \ where$$
$$\left\{ \begin{array}{rcl} X_{\mathsf{s}} &=& x_{\mathsf{s}} \ !\ \mathsf{select}(tests,\, alternatives) \\ Y_{\mathsf{c}} &=& \mathsf{select}(tests,\, alternatives) \end{array} \right\}_{\mathsf{s}\in S,\, \mathsf{c}\in C}$$

where the expression $v \ ! \ S$ denotes an initialized stream [5]. In DDD, this construction is reversed. An initial behavior table is built from a system of stream equations, each with a common selection combination [7].

6.2. Bounded indirection

We have found an extension to the term-level syntax called *indirection* [11] which is highly useful for hardware applications and appears to be equally useful in incremental specification development. If v is a signal name, the term $^\vee v$ stands for a "reference" to signal v; concretely, it is actually a token which can later be used to select v. The term $_\wedge w$ denotes that selection. As an illustration, consider the table:

$I \to O$					
P	s	t	u	v	S
1	$^\vee$t	f_1	h_1	♮	
2	$^\vee$u	f_2	h_2	♮	
3	♮	f_3	h_3	$_\wedge$s	

In essence, the term $_\wedge$s in the third row stands for the term:

```
case s
  ∨t::t
  ∨u::u
```

Uses of indirection include the description of bidirectional buses, other forms of implied selection, and control branching. Of course, such use also necessitates consistency audits over the whole table; for instance, to verify that selected signals are compatible and uniformly typed.

6.3. Data refinement

Another important set of rules for *data refinement*, will be presented in a future paper. Data refinement involves the translation between levels of data abstraction. In our approach, the foundation for data refinement lies in algebraic specification and equational logic. Consequently, the initial connection to the architectural rules presented here will lie in a more general version of the Replacement rule.

However, replacement is only adequate for the straightforward, combinational expansion of simple representations; it does not address implementations that involve sequential behavior. Our research on *sequential decomposition* [10] has not yet been reflected in behavior tables.

6.4. Verification

We are also interested in integrating the derivational formalism with *property verification*. One way to approach this is to augment behavior tables with *assertions* in a suitable temporal logic. Since we are primarily interested in higher levels of specification, "model checking" [12] these assertions would likely require interaction. Considered as an algorithmic state machine, the table would provide contextual information making the proof process more agreeable.

6.5. Animation

Finally, animation, particularly symbolic execution, would be an important feature of any practical behavior table tool. Consequently, we want to integrate our tool with proof assistants—particularly term rewriters—not only to support replacement rules, verification and type inference, but to provide interactive simplification of terms in the fashion of Moore's *symbolic spread sheets* [9].

References

[1] Christopher R. Clare. *Designing Logic Systems Using State Machines*. McGraw-Hill, 1973.

[2] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[3] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, National Aeronautics and Space Administration Langley Research Center (NASA/LRC), Hampton VA 23681-0001, November 1994.

[4] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, Cambridge, 1984.

[5] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.

[6] Steven D. Johnson. A tabular language for system design. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Lfm97: Fourth NASA Langley Formal Methods Workshop*, September 1997. NASA Conference Publication 3356, in press.

[7] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

[8] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specifiation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[9] J Strother Moore. Symbolic simulation: an ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD'98)*, pages 334–350. Springer LNCS 1522, 1998.

[10] Kamlesh Rath, Venkatesh Choppella, and Steven D. Johnson. Decomposition of sequential behavior using interface specification and complementation. *VLSI Design Journal*, 3(3-4):347–358, 1995.

[11] M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI95)*, pages 86–89. IEEE, March 1995.

[12] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In *CAV'98*. Springer, 1998.

now	u=0 || now	done*	u	v	w
		FIB: (go, in) → (done*, v)				
1	♮ || done*	¬go	in	0	1
0	1 || "	u=0	♮	v	♮
0	0 || 2	false	u-1	v	w
2	♮ || done*	u=0	u	w	v+w

action introduction ⇒

now	u=0 || now	done*	u	v	w	x*	y*	z*	ao*
		FIB: (go, in) → (done*, v)								
1	♮ || done*	¬go	in	0	1	♮	♮	♮	P
0	1 || "	u=0	♮	v	♮	♮	♮	♮	P
0	0 || 2	false	ao	v	w	sub	u	1	P
2	♮ || done*	u=0	u	w	ao	add	v	w	P

where P =(case x* y*+z* y*-z*)

decomposition ⇒

FIB: (go, in, ao) → (done*, v, now, u, w, x*, y*, z*)

now	u=0 || now	done*	u	v	w	x*	y*	z*
1	♮ || done*	¬go	in	0	1	♮	♮	♮
0	1 || "	u=0	♮	v	♮	♮	♮	♮
0	0 || 2	false	ao	v	w	sub	u	1
2	♮ || done*	u=0	u	w	ao	add	v	w

ALU: (go, done*, v, u, now, w, x*, y*, z*) → (ao*)

now	u=0 || ao*
1	♮ || (case x y+z y-z)
0	1 || (case x y+z y-z)
0	0 || (case x y+z y-z)
2	♮ || (case x y+z y-z)

output restriction ⇒   input restriction ⇒

FIB: (go, in, ao) → (done*, v, x*, y*, z*)

now	u=0 || now	done*	u	v	w	x*	y*	z*
1	♮ || done*	¬go	in	0	1	♮	♮	♮
0	1 || "	u=0	♮	v	♮	♮	♮	♮
0	0 || 2	false	ao	v	w	sub	u	1
2	♮ || done*	u=0	u	w	ao	add	v	w

ALU: (u, now, x, y, z) → (ao*)

now	u=0 || ao*
1	♮ || (case x y+z y-z)
0	1 || (case x y+z y-z)
0	0 || (case x y+z y-z)
2	♮ || (case x y+z y-z)

decision generalization ⇒

ALU: (u, now, x, y, z) → (ao*)

now	u=0 || ao*
1	♮ || (case x y+z y-z)
0	♮ || (case x y+z y-z)
2	♮ || (case x y+z y-z)

decision generalization ⇒

ALU: (u, now, x, y, z) → (ao*)

now	u=0 || ao*
♮	♮ || (case x y+z y-z)

decision introduction ⇒

ALU: (u, now, x, y, z) → (ao*)

x	now	u=0 || ao*
♮	♮	♮ || (case x y+z y-z)

conversion ⇒

ALU: (u, now, x, y, z) → (ao*)

x	now	u=0 || ao*
add	♮	♮ || y+z
sub	♮	♮ || y-z

decision elimination ⇒

ALU: (u, now, x, y, z) → (ao*)

x || ao*
add || y+z
sub || y-z

input restriction ⇒

ALU: (x, y, z) → (ao*)

x || ao*
add || y+z
sub || y-z

Figure 2: A factorization from [6]

Modeling and Validating Hybrid Systems Using VDM and Mathematica

Bernhard K. Aichernig and Reinhold Kainhofer
Institute for Software Technology (IST),
Technical University Graz, Münzgrabenstr. 11/II, 8010 Graz, Austria

Abstract

Hybrid systems are characterized by the hybrid evolution of their state: A part of the state changes discretely, the other part changes continuously over time. Typically, modern control applications belong to this class of systems, where a digital controller interacts with a physical environment. In this article we illustrate how a combination of the formal method VDM and the computer algebra system Mathematica can be used to model and simulate both aspects: the control logic and the physics involved. A new Mathematica package emulating VDM-SL has been developed that allows the integration of differential equation systems into formal specifications. The SAFER example from [11] serves to demonstrate the new simulation capabilities Mathematica adds: After the thruster selection process, the astronaut's actual position and velocity is calculated by numerically solving Euler's and Newton's equations for rotation and translation. Furthermore, interactive validation is supported by a graphical user interface and data animation.

1 Introduction

Modern control applications are realized through microcontrollers executing rather complex control logics. This complexity is increased by the fact that control software interacts with a physical environment through actors and sensors. Such systems are called *hybrid systems* due to the hybrid evolution of their state: One part of the state (variables) changes discretely, the other part changes continuously over time.

Hybrid systems are excellent examples for motivating the use of formal software development methods. First, their complexity calls for a real software engineering discipline applying both, a process model as well as a mathematical method. Second, these kinds of systems are often safety-critical which justifies formal validation and verification techniques. Third, engineers in the control domain are educated in the use of mathematical models for designing dynamic systems.[1] In our experience, the offer of a formal method for software development is more often appreciated by control engineers, than by software developers used to produce short cycle products in 'Internet time'.

In [11] the hybrid system SAFER has been chosen by NASA in order to introduce to formal specification and verification techniques. SAFER is an acronym for "Simplified Aid For EVA (Extravehicular Activity) Rescue". It is a small, lightweight propulsive backpack system designed to provide self-rescue capabilities to a NASA space crewmember separated during an EVA. In this NASA guidebook[11], SAFER is specified formally in the PVS notation and properties are formally proved using the PVS theorem prover [12]. In the guidebook the dynamic aspects are used to compare the continuous domain model from spacecraft attitude control with the discrete PVS model of SAFER's control logic. It demonstrates that the two models have the same goals: rigorous description and prediction of behavior but that the needed mathematics and calculation techniques are different.

In [1, 2] Agerholm & Larsen have proposed a cheaper testing based validation approach to the SAFER example using an executable VDM-SL model and the IFAD VDM-SL Toolbox [10, 7, 6]. They recommend the use of a specification executor and animator for raising the confidence in a formal model prior to formal proving.

We agree with Agerholm & Larsen's arguments for such a "light-weight" approach to formal meth-

---

[1]The same holds for software developers coming from classical engineering disciplines.

ods in order to facilitate the technology transfer. Since in several industrial projects performed at our institute a similar experience has been made [9, 15, 5], one of our research areas has become the support of testing through formal methods [4].

However, neither the PVS nor the VDM-SL model of SAFER did take the continuous physical models into account. The reason is that, in general, today's formal method tools are not well suited for supporting continuous mathematics. This paper shows a solution the problem.

In the following it is demonstrated how an explicit discrete model can be combined with the continuous physical model for validation and animation. With the right tool there is no reason why a physical model should not be included in the validation process of a hybrid system. Just the opposite is the case: [1] detected several cases where the interface to a cut out automatic attitude hold (AAH) control unit needed further clarification.

In this work the commercial computer algebra system Mathematica [16] has been used to overcome the gap between discrete and continuous mathematics. A VDM-SL package has been implemented that allows to specify in the style of the Vienna Development Method (VDM) inside Mathematica. Thus, explicit discrete models can be tested in combination with differential equation systems modeling physical behavior by solving the equations on the fly. Even pre- and post-condition checking is possible. Again, NASA's SAFER system serves as the demonstrating example. The VDM-SL specification of [2] has been taken and extended with the physics involved in SAFER, expressed through differential equations. More precisely, the physical behavior is movement in space, modeled by the laws for translation and rotation — Newton's and Euler's equations for three dimensional space.

Beside the execution (testing) of hybrid models, Mathematica's front-end supports the visual validation of such systems. The graphical user-interface for SAFER's hand grip is implemented inside the computer algebra system as well as a scientific graph representing the movement of a crew-member using SAFER. After each control cycle, actual physical vectors like angular velocity or acceleration can be inspected together with the logical status, e.g. the thrusters firing. Finally, it is even possible to animate a sequence of performed control-cycles as a movie showing the SAFER representation flying.

The structure of the rest of the paper is as follows. First in Section 2 an overview of the SAFER system is given, which will serve as the demonstrat-
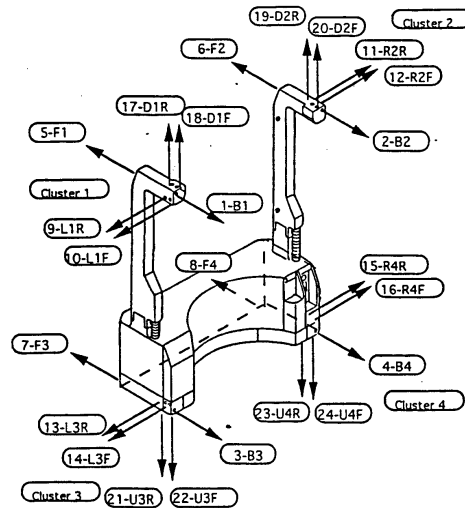

Figure 1. SAFER thrusters.

ing example throughout the paper. This is followed by a discussion of VDM-SL and its realization inside Mathematica in Section 3. Then, a description of the discrete SAFER model is given in Section 4. Section 5 explains the differential equation systems modeling SAFER's physics and the coordinate transformations needed. Then, Section 6 introduces to the hybrid model and demonstrates the integration of VDM-SL and differential equation systems. Next, the validation capabilities of our approach are discussed in Section 7 and Section 8. In the final Section 9 we draw some conclusion regarding the presented work in particular, as well as possible future approaches in general.

2 The SAFER System

The following overview of the SAFER system is based on, and partly copied from, the NASA guidebook [11], which describes a cut-down version of a real SAFER system.

The Simplified Aid for EVA Rescue (SAFER) is a small, self-contained, backpack propulsion system enabling free-flying mobility for a NASA crewmember engaged in extravehicular activity (EVA). It is intended for self-rescuing on Space Shuttle missions, as well as during Space Station construction and operation, in case a crewmember got separated from the shuttle or station during an EVA. This type of contingency can arise if a safety tether breaks, or if it is not correctly fastened. SAFER attaches to the underside of the Extravehicular Mobility Unit

(EMU) primary life support subsystem backpack and is controlled by a single hand controller that is attached to the EMU display and control module. Figure 1 shows the backpack propulsion system with the 24 gaseous-nitrogen (GN_2) thrusters, four in each of the positive and negative X, Y and Z directions. For example, the thrusters denoted by 5-F1, 6-F2, 7-F3 and 8-F4 are firing backwards (indicated by the arrows) resulting in a forward motion.

The main focus of the discrete specification is on the thruster selection logic, which is rather complex due to a required priorization of hand controller commands. Various display units and switches which are not directly related to the selection of the thrusters have been ignored in our model. However, in contrast to [11] and [1] the calculation of the control output in the Automatic Attitude Hold (AAH) is not ignored, but simulated based on a dynamic model of the physics discussed in Section 5.
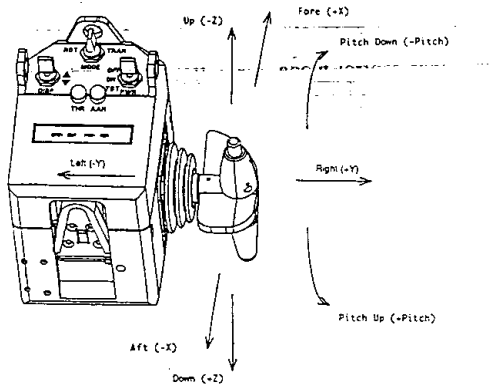

Figure 2. Hand controller module of SAFER.

The hand controller, shown in Figure 2, is a four-axis mechanism with three rotary axes and one transverse axis using a certain hand controller grip. A command is generated by moving the grip from the center null position to mechanical hard-stops on the hand controller axes. Commands are terminated by returning the grip to the center position. The hand controller can operate in two modes, selected via a switch, either in translation mode, where X (forward-backwards), Y (left-right), Z (up-down) and pitch commands are available, or in rotation mode, where roll, pitch, yaw and X commands are available. The arrows in Figure 2 show the rotation mode commands. Note that X and pitch commands are available in both modes.

Pitch commands are issued by twisting the hand grip around its transverse axis, while the other commands are obtained around the rotary axis.

A push-button switch on top of the grip initiates and terminates AAH according to a certain protocol. If the button is pushed down once the AAH is initiated, while the AAH is deactivated if the button is pushed twice within 0.5 seconds.

As mentioned above there are various priorities among commands that make the thruster selection logic rather complicated. Translational commands issued from the hand controller are prioritized, providing acceleration along a single translational axis, with the priority X first, Y second, and Z third. When rotation and translation commands are present simultaneously from the hand controller, rotations take higher priority and translations are suppressed. Moreover, rotational commands from the hand grip take priority over control output from the AAH, and the corresponding rotation axes of the AAH remain off until the AAH is reinitialized. However, if hand grip rotations are present at the time when the AAH is initiated, the corresponding hand controller axes are subsequently ignored, until the AAH is deactivated.

In [1] it is explained how a specification interpreter tool facilitates the validation of the requirements listed in the appendix of the NASA guidebook. Moreover, it is demonstrated that formal validation techniques uncover open issues in informal requirements even if they seem to be straightforward and clear.

The same validation techniques as discussed in [1] can be applied in our Mathematica based framework — and more. However, before we discuss the value added through a hybrid model, in the following section, the realization of our VDM-SL package is discussed.

3 VDM-SL in Mathematica

VDM-SL is the specification language of the Vienna Development Method (VDM) [10, 7]. VDM is a widely used formal method, and it can be applied to the construction of a large variety of systems. It is a model-oriented method, i.e. its formal descriptions (specifications) consist of an explicit model of the system being constructed. More precisely mathematical objects like sets, sequences and finite mappings (maps) are used to model a system's global state. Additional logic constraints, called data-invariants, allow one to model informal requirements by further restricting specified data-

types. For validation purposes the functionality may be specified explicitly in an executable subset of VDM-SL. In addition, pre- and post-conditions state *what* must hold before and after the evaluation of a system's operation. Although VDM-SL is called a general purpose specification language it does not support the specification of dynamic systems. The language's ISO-standard [13] does not even include standard functions like sine or cosine.

Here, as the name indicates, Mathematica's strengths supplement our combined approach. Mathematica is a symbolic algebra system that offers the opportunity of solving arbitrary non-linear as well as linear systems of equations. Mathematica's language interpreter is in fact a rewriting system providing an untyped functional programming language. For an introduction to functional programming in Mathematica see [3]. This programming language has been used in order to define a package emulating the specification language VDM-SL. By emulating we express the fact that the package does not allow one to write specifications in VDM-SL's concrete syntax, but in its abstract syntax with some pretty printing for VDM-SL output.

Mathematica's user interface are so called notebooks, fancy editors structured in cells for input, output or plain text. Entering a Mathematica expression in an input cell, the system tries to evaluate this input through a rewriting procedure based on pattern matching.

The following language constructs have been added to the standard language in order to import the VDM-SL model from [2]:

- abstract datatypes for composite types, sets, sequences and maps

- comprehension expressions for sets, sequences and maps

- let and cases expressions

- operators for propositional and predicate logic

- types optionally restricted by data-invariants

- value and global state definitions

- typed function/operation definitions with pre- and post-conditions

Some of the items above deserve a more detailed discussion.

Comprehensions

A powerful feature of a specification language like VDM-SL is its ability to construct collection types like sets, sequences and maps through comprehensions. For example, a set-comprehension defines a set through an arbitrary expression describing the set-elements with its free variables ranging over a set of values, such that an optional condition holds. The following example demonstrates the value added through a computer algebra system. The set-comprehension

$$\mathtt{set}[\mathtt{x}|\{\mathtt{x} \in \mathbb{Z}\} \cdot \{\mathtt{x}^6 - 44\mathtt{x}^5 + 318\mathtt{x}^4 + 4102\mathtt{x}^3 \\ -4461\mathtt{x}^2 + 550\mathtt{x} + 8750 == 0\}]$$

represents a set of elements x, where x is an integer number such that the equation holds.

The resulting set[2]

$$\mathtt{set}[-7, -1, 25]$$

demonstrates that, unlike IFAD's VDM-SL interpreter, comprehensions ranging over infinite sets may be evaluated.

Types

As already mentioned, in contrast to VDM, Mathematica has an untyped language. Consequently, no type checking mechanism is available. However, types are an important tool for specifying a data-model in VDM. Therefore, type declarations of the form $\mathtt{Type}[\mathtt{name}, \mathtt{type}]$ have been included, where $\mathtt{type}$ is one of the predefined VDM-SL types, like basic types, composite types, sets ... For example, a type ISet representing a set of natural numbers might be declared by $\mathtt{Type}[\mathtt{ISet}, \mathtt{set}[\mathbb{N}]]$.

Optionally, a type can be further constrained by a data-invariant condition. Such invariant types are defined by $\mathtt{Type}[\mathtt{name}, \mathtt{type}, \mathtt{Invariant} - >$ $\mathtt{predicate}]$. The $\mathtt{predicate}$ is defined by a lambda expression mapping $\mathtt{type}$ to a Boolean value. All the invariants are globally stored in the system for invariant checking, before and after the evaluation of a VDM function.

Internally, a type is translated to a Mathematica pattern, matching those values the type denotes. Invariant types are supported by the possibility of defining patterns with arbitrary predicates. These patterns restrict the argument range in the definition of typed VDM functions.

---

[2] The six solutions including double and complex solutions are: $-7, -1, 1 - I, 1 + I, 25, 25$.

Functions

Using the VDM-SL package, typed functions with pre- and post-conditions can be defined using the constructor

```
VDMFunction[id, sig, id[vars] := body, pre, post]
```

with the following parameters:

id the name of the function,

sig the signature of the function,

id[vars] := body the function definition,

pre an optional pre-condition stating what must hold before the evaluation such that the post-condition holds,

post an optional post-condition stating what must hold after the evaluation.

VDMFunction realizes a complex call to Mathematica's internal Function call and emulates the checks for

- the signature types,
- pre- and post-condition,
- data-invariants.

4 Discrete Model

In order to demonstrate the Mathematica package the same functions for the thruster selection logic as in [1] are presented in this section. The six degree-of-freedom of the translation and rotation commands is modeled using a composite type:

```
Type[SixDofCommand, Composite[{"tran", TranCommand},
                              {"rot",  RotCommand }]]
```

whose two fields are finite maps from translation and rotation axis respectively to axis commands. For example the type of translation commands is defined as follows:

```
Type[TranCommand, TranAxis -> Axiscommand,
     Invariant -> (dom[#] == set[X,Y,Z]&)]
```

where the invariant ensures that command maps are total. Here, the invariant predicate is defined by a lambda expression in Mathematica's notation of pure functions. The type of rotation commands is defined similarly. Enumerated types are used for axis commands and translation and rotation axes:

```
VDMFunction[
  SelectedThrusters,
    AUX`SixDofCommand × AUX`RotCommand ×
    set[AUX`RotAxis] × set[AUX`RotAxis]
      -> ThrusterSet,
  SelectedThrusters[hcm, aah, actAxes, ignHcm] :=
  let[{tran, rot, bfMandatory,bfOptional,
    lrudMandatory,lrudOptional,bfThr,lrudThr},
    {tran, rot} =
            (IntegratedCommands[hcm,aah,actAxes,ignHcm]
                     /. SixDofCommand[tr_,ro_]:->tr,ro);
    {bfMandatory, bfOptional} = BFThrusters[tran[X],
                                            rot[PITCH],
                                            rot[YAW]];
    {lrudMandatory, lrudOptional} =
                          LRUDThrusters[tran[Y],
                                        tran[Z],
                                        rot[ROLL]];
    bfThr = If[(rot[ROLL] === ZERO),
               bfOptional ∪ bfMandatory,
               bfMandatory ];
    lrudThr = If[(rot[PITCH] === ZERO) and
                 (rot[YAW] === ZERO),
                 lrudOptional ∪ lrudMandatory,
                 lrudMandatory];
  set @@ (bfThr ∪ lrudThr)
  ]
];
```

Figure 3. The SelectedThrusters function.

```
Type[AxisCommand, NEG | ZERO | POS];

Type[TranAxis, X | Y | Z];

Type[RotAxis, ROLL | PITCH | YAW]
```

In the SelectedThrusters function in Figure 3 grip commands from the hand controller (with six-degree-of freedom commands) are integrated with the AAH control output. The IntegratedCommands function prioritizes hand controller and AAH commands.

Based on these commands, thrusters for back and forward accelerations and left, right, up and down accelerations are calculated by two separate functions. Figure 4 presents cut-down versions of these functions. These represent a kind of look-up tables, modeled using cases expressions. Note that they return two sets of thruster names, representing mandatory and optional settings respectively.

5 Physics Involved in SAFER

This section presents the continuous model of the physics involved in our hybrid model. For the SAFER example, translation and rotation equations from mechanics are sufficient for modeling the motion of a crewmember using the propulsion system. The purpose of this model is twofold: First, we need to calculate the sensor inputs of angular velocity for simulating the AAH. Second, in order to visualize

```
VDMFunction[
  BFThrusters,
    AUX`AxisCommand × AUX`AxisCommand × AUX`AxisCommand
      -> ThrusterSet × ThrusterSet,
  BFThrusters[A, B, C] :=
    cases[{A, B, C},
      {NEG, ZERO, ZERO} -> {{B4}, {B2,B3}},
      {ZERO, ZERO, ZERO} -> {{}, {}},
      {POS, NEG, ZERO} -> {{F1,F2}, {}},
      ...
    ]
];

VDMFunction[
  LRUDThrusters,
    AUX`AxisCommand × AUX`AxisCommand × AUX`AxisCommand
      -> ThrusterSet × ThrusterSet,
  LRUDThrusters[A, B, C] :=
    cases[{A, B, C},
      {NEG, NEG, ZERO} -> {{}, {}},
      {NEG, ZERO, ZERO} -> {{L1R,L3R}, {L1F,L3F}},
      {POS, ZERO, POS} -> {{R2R}, {R2F,R4F}},
      ...
    ]
];
```

Figure 4. Extracts from BFThrusters and LRUDThrusters.

the SAFER movement, absolute coordinates have to be determined. The mathematics needed can be found in the standard literature of mechanics, like [8].

Translation

The translation of a crewmember wearing SAFER is described by Newton's second law of motion expressed by

$$F = m\dot{v} = \dot{p} \tag{1}$$

where F, m, v and p denote force vector, mass, velocity vector and impulse vector. It states that "The time rate of change of the momentum of a particle is proportional to the force applied to the particle and in the direction of the force."

Rotation

The rotation is modeled by three equations known as the *Euler's equations* of motion for the rotation of a rigid body.

Denote by Ω the angular velocity defined with respect to the center of mass, and by I the moments of inertia. The equations describing the body rota-

tions are then given by

$$I_1\dot{\Omega}_1 + (I_3 - I_2)\Omega_2\Omega_3 = Q_1 \tag{2}$$

$$I_2\dot{\Omega}_2 + (I_1 - I_3)\Omega_3\Omega_1 = Q_2 \tag{3}$$

$$I_3\dot{\Omega}_3 + (I_2 - I_1)\Omega_1\Omega_2 = Q_3 \tag{4}$$

or as a vector equation where I is a diagonal matrix:

$$I \cdot \dot{\Omega} + \Omega \times I \cdot \Omega = Q \tag{5}$$

Q_i denotes a torque causing a rotation around the i-axis, in the body's own coordinate system. Here, the torque is given by the sum over the thrusters firing. Actually, a component Q^{th} is calculated by the cross product of a thruster's position vector relative to the center of mass and its force. SAFER does not use proportional gas jets, but thrusters whose valves are open or not, which simplifies the calculation.

Motion

In order to combine translation and rotation in a single model of motion, suitable for our purposes, coordinate transformations are necessary. More precisely, the fixed coordinate system values for visualization (position and velocity) have to be related to SAFER's coordinate system values (angular velocity).

As Ω is calculated in the body's own coordinate system, they have to be transformed back to the fixed coordinate system. Given the Euler angles φ, θ and ψ that denote the deviation of the fixed x, y and z axis, the angular velocities can be calculated according to the following formula.

$$\Omega_1 = \dot{\varphi}\sin\theta\sin\psi + \dot{\theta}\cos\psi \tag{6}$$

$$\Omega_2 = \dot{\varphi}\sin\theta\cos\psi - \dot{\theta}\sin\psi \tag{7}$$

$$\Omega_3 = \dot{\varphi}\cos\theta + \dot{\psi} \tag{8}$$

The derivation of these equations can be found in [8]. Using vector notation we get the equation:

$$\Omega = D_3(\psi) \cdot D_1(\theta) \cdot (\dot{\theta}, 0, \dot{\varphi})^T + (0, 0, \dot{\psi})^T \tag{9}$$

$$D_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix} \tag{10}$$

$$D_3 = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{11}$$

where D_1 and D_3 are rotation matrices that turn the coordinate system by a given angle.

D_1 and D_3 are used to transform a vector from our fixed coordinate system to a turned coordinate system. For translation motion, the thruster's force vector F has to be transformed from SAFER's coordinate system to the fixed one using the transposed rotation matrices:

$$(D_3(\psi) \cdot D_1(\theta) \cdot D_3(\varphi))^T$$

Summarizing, these four vector differential equations are sufficient for modeling SAFER's motion over time:

$$v = \dot{x} \tag{12}$$

$$m \cdot v = (D_3(\psi) \cdot D_1(\theta) \cdot D_3(\varphi))^T F \tag{13}$$

$$I \cdot \dot{\Omega} + \Omega \times I \cdot \Omega = Q \tag{14}$$

$$\Omega = D_3(\psi) \cdot D_1(\theta) \cdot (\dot{\theta}, 0, \dot{\varphi})^T + (0, 0, \dot{\psi})^T \tag{15}$$

Solving these equations with given thruster forces results in SAFER's position vector $x(t)$ and the angular velocity $\Omega(t)$ used for AAH.

Alternatives to the Euler's equations model are possible. For example, an aproach could have involved the less computationally intensive quaternions. However, for validation purposes the model should be as intuitive as possible, here efficiency plays a minor role.

6 A Hybrid Model

The hybrid model of SAFER consists of the hand controller and the Automatic Attitude Hold as its discrete parts on one side and the equations of motion as the continuous part on the other side. Both are modeled in Mathematica, the first in the form of the VDM-SL specification using our VDM-SL emulation package, the later in the form of ordinary differential equations in Mathematica notation.

The combination of the discrete control system and the continuous physical model during the testing phase carries certain advantages:

Not only can the system specification be tested in an (idealized) physical simulation, but also the system parameters like the force of the thrusters and the moments of inertia of the backpack can easily be adjusted until the system responds in a way suitable for practical use.

This is not a very rigorous approach, and it is not intended to replace other testing tools and methods. Rather it can serve as a valuable supplementary tool.

```
VDMFunction[
  ControlCycle,
  SwitchPositions × HandGripPosition ×
    RotCommand × InertialRefSensors -> ThrusterSet,

  ControlCycle[SwitchPositions[mode_, aah_], rawGrip,
             aahCmd, IRUSensors]:=
    let[{
      gripCmd=HCM`GripCommand[rawGrip, mode],
      thrusters=SelectedThrusters[gripCmd, aahCmd,
        AAH`ActiveAxes[], AAH`IgnoreHcm[]]
    },
    AAH`Transition[IRUSensors, aah, gripCmd, SAFER`clock];
    SAFER`clock=SAFER`clock+1;
    PosData=CalcNewPosition[thrusters];
    thrusters
  ],
  True,
  card[RESULT] ≤ 4 ∧ ThrusterConsistency[RESULT]
];

VDMFunction[
  SensorControlCycle,
  SwitchPositions × HandGripPosition -> ThrusterSet,

  SensorControlCycle[SwitchPositions[mode_, aah_],
             rawGrip]:=
    ControlCycle[SwitchPositions[mode,aah],rawGrip,
      AAHControlOut[Sensors], Sensors ]
];
```

Figure 5. The ControlCycle function.

The Control Cycle

The `ControlCycle` function (Figure 5) integrates the discrete model of hand control, thruster selection and Automatic Attitude Hold (AAH) with the continuous physical model of motion presented above.

The Control Cycle is implemented in two different functions. `ControlCycle` takes the state of the hand control (switches and hand grip) as well as the already calculated or manually entered AAH commands and the sensor values. `SensorControlCycle` takes the values of the sensors (here simulated by the solutions of the equations of motion of the previous control cycle) and determines which thrusters are invoked by the AAH. These are then passed on to `ControlCycle`.

After determining the active thrusters and the AAH state, the differential equations are solved numerically in the `CalcNewPosition` function and the current position is updated. These results simulate the values measured by the sensors (with exception of the heat sensors, which are left out in our model) providing data for AAH. This part of the control system is completely left out in [1] and only included in the form of two unspecified functions in the PVS model [11].

Here the SAFER state is not as trivial as in [1] where it holds only a clock variable.

```
VDMFunction[
  AAHControlOut,
  InertialRefSensors->RotCommand,

  AAHControlOut[IRUSensors]:=
    let[{rr=IRUSensors."RollRate",
         pr=IRUSensors."PitchRate",
         yr=IRUSensors."YawRate"},
      map[
        ROLL->Which[
             rr ≤ -EpsRoll,POS,
             rr ≥ EpsRoll, NEG,
             True, ZERO],
        ...
      ]]
];
```

Figure 6. The Bang Bang algorithm for AAH.

```
State[SAFER,
  Type[clock, ℕ],
  Type[PosData, PositionData],
  Type[Sensors, InertialRefSensors],
  Type[step, ℝpos],
  Type[PosDataList, List[PositionData]],

  init[SAFER] := SAFER[0,
     PositionData[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     InertialRefSensors[0, 0, 0, 0, 0, 0, 0, 0, 0],
     1/4, {{{0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}}]
];
```

The state above also includes the current position, Euler angles and velocities stored in a variable of type `PositionData`.

Even the past position data is stored for providing full information about SAFER's trajectory. For simulation this data will be used to display the history as a Mathematica "movie" showing the astronaut flying around in the coordinate system.

Automatic Attitude Hold (AAH)

Simulating the measured sensor values by the results of the equations of motion provides the opportunity of including the Automatic Attitude Hold mechanism by a simple Bang Bang [11] algorithm: If the angular velocity for an axis where AAH is turned on exceeds a certain threshold, selected thrusters are fired in order to slow down this rotation (Figure 6). AAH is limited to this mechanism because SAFER is only based on simple thrusters with two states: on and off.

The Differential Equations

The equations of motion used to determine the new position of the astronaut are Newton's and Euler's equations described above. Although this model neglects any gravitational forces and other disturbing influences, they could easily be added by an additional acceleration in the equations or random fluctuations applied to the results of the differential equations.

The new position is obtained by numerically solving the equations rather than algebraically which is less time-efficient, beside the fact that the algebraic solution is not necessary as only the result at time `step` is needed for simulation.

Since the equations are only slightly coupled, they can be solved in four steps, which is numerically more stable than solving them all at once. This functionality is provided by Mathematica's `NDSolve` function, which takes the differential equations and the initial conditions and returns numeric functions that approximate the exact solutions of the equations. In this case the trajectory is calculated piecewise: in every control cycle the trajectory only for that cycle is solved using the position before the cycle as the initial conditions and the force and torque applied by the thrusters as parameters. These can easily be calculated, the force by a simple vector addition of the forces applied by every single thruster, and the torque by adding up the cross products of the thruster positions with the force applied by that thruster.

First, Euler's equation in the astronaut's coordinate system is solved giving the angular velocity. This needs the forces and the torque applied by the fired thrusters as parameters. The result is then transformed back to the fixed coordinate system and used to solve the differential equation for the Euler angles. In a third step Newton's equation can be solved using the results from the previous equations. Finally, a simple integration of the velocities gives the position of the astronaut.

These numerical solutions to the differential equations can also be used to investigate stability. In the simplified case without any external forces like gravitation, this might not be so interesting, but as soon as external forces are modeled into the differential equations, stability is a crucial concern. What happens if the hand controller keeps in the same position over a long period of time? Such questions can easily be answered by solving the differential equations for a time period longer than just the control cycle.

7 Simulating SAFER

Mathematica does not only provide algebraic and numeric functionality, but also an extensive reper-

toire of plotting functions. Thus Mathematica has also been used to visualize SAFER's current position together with other state information.
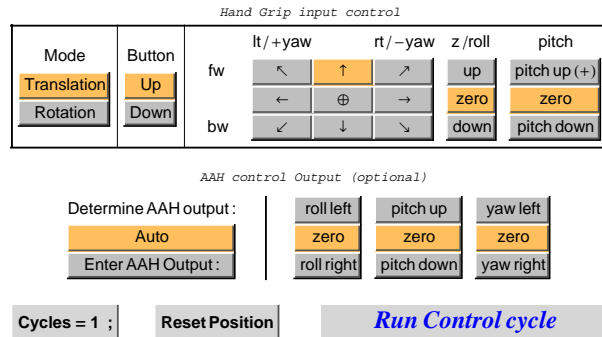
Figure 7. The GUI for the hand controller.

Figure 8. A sample trajectory of the SAFER.

An interface to the hand controller similar to that in [2] is provided in Mathematica (Figure 7). It contains buttons for all the hand controller states as well as for manual input of the AAH output for overriding the simulated AAH in the model.

Pressing one of the buttons sets a global variable that is used to determine the parameters passed to the `ControlCycle` function. Additionally, the "Cycles=1" button determines how many control cycles should be evaluated when the "Run Control Cycle" button is pressed.

Pressing "Run Control Cycle" initiates the control cycle and after calculating the new position prints out a plot of the astronaut's path so far together with his orientation indicated by the axes of his own coordinate system (Figure 8). Additionally, his velocity and angular velocity are shown as vectors. Optionally a table with the list of the fired thrusters as well as the axes where AAH is turned on is printed.

Since all the previous position data is stored, Mathematica can even animate this graph so that one can inspect the SAFER moving through space.

A graphical interface to the simulation like in Figure 7 is interesting when testing the system's behavior in general. However, when adjusting parameters or testing specific cases, it's more convenient to run the control cycles directly using Mathematica input commands. Figure 9 shows the input to create Figure 8.

In [1] the visualization is done outside the toolbox using dynamic link modules, which are programmed specifically for this one application. In Mathematica, changing only the differentia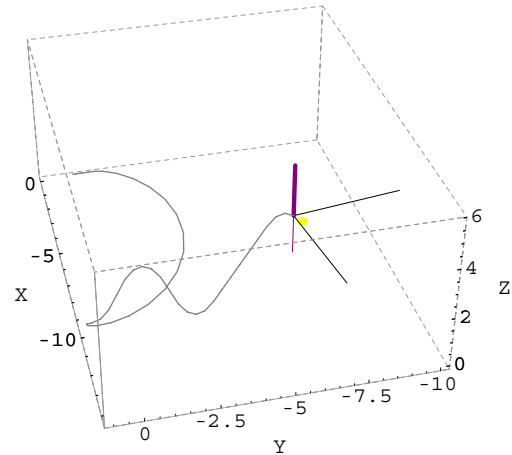l equations suffices to include other influences like gravity, as Mathematica chooses the algorithm to solve the equations.

However, testing in Mathematica is not restricted to graphical simulation. Like in [1], the output of the thruster selection logic can be validated by enumerating all possible states of the Hand controller, or in an extended version enumerating all possible states of the hand controller and the AAH. Figure 10 shows these functions formulated in Mathematica's VDM-SL notation. On every possible state, ControlCycle is applied to calculate the fired thrusters. The result of this large map comprehension then has to be investigated manually.

Another important part in the process of verifying software would be coverage testing, which is unfortunately not possible in Mathematica.

8 Enhanced Analysis of the System

The simulation possibilities described in the last section can be exploited for risk and safety analysis of the system. A very simple application is the case when one of the thrusters fails due to a mechanical defect or an iced valve. The most important questions in this scenario are whether the astronaut will still be able to navigate the system, and whether it is possible to return before the air or the nitrogen for the thrusters is used up.

We investigated the functionality of AAH in the case where one thruster (6-F2) fails. Figure 11 shows the angular velocity of the system, with the

```
ResetSAFERPosition[ ];
(* 1 right *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[ZERO,ZERO,POS,ZERO]],{1}];
(* 3 yaw *)
Do[SensorControlCycle[SwitchPositions[ROT,UP],
    HandGripPosition[ZERO,ZERO,POS,ZERO]],{3}];
(* 15 "right" *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[ZERO,ZERO,POS,ZERO]],{15}];
(* wait *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[ZERO,ZERO,POS,ZERO]],{2}];
(* 3 up *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[POS,ZERO,ZERO,ZERO]],{3}];
(* 6 down *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[NEG,ZERO,ZERO,ZERO]],{6}];
(* 5 up *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[POS,ZERO,ZERO,ZERO]],{5}];
(* nothing, just keep floating in space *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[ZERO,ZERO,ZERO,ZERO]],{6}];
(* finally, 2 down *)
Do[SensorControlCycle[SwitchPositions[TRAN,UP],
    HandGripPosition[NEG,ZERO,ZERO,ZERO]],{2}];
```

Figure 9. The commands to create the sample trajectory.

```
VDMFunction[ ControlCycleTest,
  SwitchPositions × HandGripPosition × RotCommand ->
    ThrusterSet,
  ControlCycleTest[SwitchPositions[mode_, aah_], rawGrip,
      aahCmd]:=
    SelectedThrusters[HCM`GripCommand[rawGrip, mode],
      aahCmd, AAH`ActiveAxes[], AAH`IgnoreHcm[]],
  True,
  card[RESULT]≤ 4 ∧ ThrusterConsistency[RESULT]
];

VDMFunction[ BigTest,
  {}->(HCM`SwitchPositions × HCM`HandGripPosition ×
       AUXIL`RotCommand -> ThrusterSet),
  BigTest[]:= map[({switch, grip, aahLaw}->
    ControlCycleTest[switch, grip, aahLaw])|
    {switch∈switchPositions, grip∈gripPositions,
    aahLaw∈allRotCommands }]
]

VDMFunction[ HugeTest,
  {}->(HCM`SwitchPositions × HCM`HandGripPosition ×
       AUXIL`RotCommand -> ThrusterSet),
  HugeTest[]:= map[({switch, grip, aahLaw}->
    ControlCycleTest[switch, grip, aahLaw])|
    {switch∈switchPositions, grip∈allGripPositions,
    aahLaw∈allRotCommands}]
];
```

Figure 10. The testing functions.

hand grip set to forward acceleration. Just before cycle 4 is initiated, thruster 6-F2 breaks, which would be used in this acceleration. This leaves thruster 7-F3 applying an additional torque to the system, which results in an increasing angular velocity. In cycles 9 and 10 the astronaut initiates AAH, but keeps the forward acceleration (cycles 10 to 17 and 20 to 25). AAH is now only able to compensate the additional torque, but not to reduce the angular velocity. Only when the forward acceleration is turned off (cycles 17 to 20 and 25 to 30), AAH shows effect.

The functionality of AAH could be improved by immediately excluding thruster 7-F3 from the translational commands when thruster 6-F2 fails (and thus allowing thruster 3-B3 to be used by AAH instead of 6-F2). This would require a slightly modified and more complex thruster selection logic, providing a higher level of safety for the astronaut.

9 Concluding Remarks

In this article a hybrid model of NASA's SAFER system has been presented using the specification language VDM-SL inside the computer algebra system Mathematica. We demonstrated that the implementation of a VDM-SL package for Mathematica provides both, VDM-SL's powerful language features, like comprehensions, as well as the mathematical power of Mathematica, e.g. solving differential equation systems.

The SAFER example shows the validation possibilities of such a combined tool. Like in [1] the complex discrete model of the control logic can be validated through testing. This is a cheap technique for raising the confidence that the right model has been specified prior to the application of more expensive formal proof techniques.

However, with the right tool, there is no reason why the continuous models of a hybrid system should be excluded from validation. Such a hybrid validation is more suitable for finding unjustified domain assumptions made in the discrete model. We strongly propose such validations, due to the fact that making wrong assumptions is the weak point of formal verification techniques, possibly leading to correct proofs of the wrong model.

Furthermore, we demonstrated that the visualization features of Mathematica provide a convenient way to communicate a model to a customer. Moreover, in contrast to [1], our visualization is a functional graph that facilitates the communication to control experts as well as to customers with a technical expertise.

In the Irish school of VDM, Mathematica has been used to explore explicit VDM specifications [14], but to our present knowledge not for modeling hybrid systems.
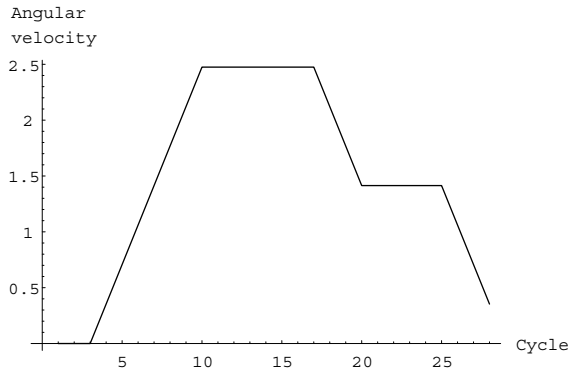
Figure 11. Angular velocity with a broken thruster, AAH initiated in cycle 9.

Note that the conclusion of our work is not that Mathematica is the best tool for validating hybrid system specifications. Our Mathematica approach has its disadvantages, too: Our VDM-SL representation is not as readable as the notation of standard VDM-SL and a typed language would be more suitable for specification purposes. Rather than proposing a certain tool, our work points out the features a powerful toolset should provide for validating hybrid systems.

Another future approach would be the integration of a classic formal method tool with a computer algebra system. For example a combination of Mathematica with the IFAD VDM-SL Toolbox used in [1] would be a possibility. This could be realized with the lately developed CORBA API of this tool, that enables access to the toolbox as a CORBA object and thus calling its VDM-SL interpreter from programs implemented in C or Java. Mathematica provides an interface through its MathLink facility.

Summarizing, we feel that our approach of hybrid validation is a valuable technique for producing systems of higher reliability and hope that it will stimulate further research in this area.

Acknowledgment

References

[1] Sten Agerholm and Peter Gorm Larsen. Modeling and validating SAFER in VDM-SL. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop (Lfm97)*. NASA, September 1997. `http://shemesh.larc.nasa.gov/fm/Lfm97/proceedings/`.

[2] Sten Agerholm and Peter Gorm Larsen. SAFER specification in VDM-SL. Technical report, IFAD, September 1997. VDM Examples Repository: `http://www.ifad.dk/Products/VDMTools/vdmsl-examples.htm`.

[3] Bernhard K. Aichernig. Teaching programming to the uninitiated using Mathematica. Technical Report IST-TEC-98-03, Institute for Software Technology, TU-Graz, Austria, May 1998.

[4] Bernhard K. Aichernig. Automated black-box testing with abstract VDM oracles. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Computer Safety, Reliability and Security: proceedings of the 18th International Conference, SAFECOMP'99, Toulouse, France, September 1999*, volume 1698 of *Lecture Notes in Computer Science*, pages 250–259. Springer, 1999.

[5] Georg Droschl. Events and scenarios in VDM and PVS. In *3rd Irish Workshop in Formal Methods, Galway*, Electronic Workshops in Computing. Springer-Verlag, July 1999.

[6] John Fitzgerald. Information on VDM. VDM: `http://www.csr.newcastle.ac.uk/vdm/`.

[7] John Fitzgerald and Peter Gorm Larsen. *Modelling Sytems, Practical Tools and Techniques*. Cambridge University Press, 1998.

[8] Walter Hauser. *Introduction to the Principles of Mechanics*. Addison-Wesley, 1965.

[9] Johann Hörl and Bernhard K. Aichernig. Formal specification of a voice communication system used in air traffic control, an industrial application of light-weight formal methods using VDM++ (abstract). In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999*, volume

1709 of *Lecture Notes in Computer Science*, page 1868. Springer, 1999. Full report at `ftp://ftp.ist.tu-graz.ac.at/pub/publications/IST-TEC-99-03.ps.gz`.

[10] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

[11] John C. Kelly and Kathryn Kemp. Formal methods, specification and verification guidebook for software and computer systems, volume II: A practitioner's companion, planning and technology insertion. Technical Report NASA-GB-001-97, NASA, Washington, DC 20546, May 1997.

[12] SRI Computer Science Laboratory. The PVS specification and verification system. PVS: `http://pvs.csl.sri.com/`.

[13] P. G. Larsen, B. S. Hansen, H. Bruun, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996. International Standard ISO/IEC 13817-1.

[14] Colman Reilly. Exploring specifications with Mathematica. In *Proceedings of the Z User Workshop*, Department of Computer Science, Trinity College, Dublin, 1995.

[15] Rudi Schlatte and Bernhard K. Aichernig. Database development of a work-flow planning and tracking system using VDM-SL. In John Fitzgerald and Peter Gorm Larsen, editors, *Workshop Materials: VDM in Practice!, Part of the FM'99 World Congress on Formal Methods, Toulouse*, September 1999.

[16] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 3rd edition, 1996.

Modeling the Fault Tolerant Capability of a Flight Control System: An Exercise in SCR Specification[*]

Chris Alexander	Vittorio Cortellessa	Diego Del Gobbo
Azimuth Inc.	Comp. Sc. and Electr. Eng.	Mech. and Aerosp. Eng.
1000 Technology Drive	West Virginia University	West Virginia University
Fairmont, WV 26554	Morgantown, WV 26506-6109	Morgantown, WV 26506
chrisa@azimuthwv.com	vittorio@csee.wvu.edu	delgobbo@cemr.wvu.edu

Ali Mili [†]	Marcello Napolitano
Comp. Sc. and Electr. Eng.	Mech. and Aerosp. Eng.
West Virginia University	West Virginia University
Morgantown, WV 26506-6109	Morgantown, WV 26506
amili@csee.wvu.edu	mnapolit@cemr.wvu.edu

Abstract

In life-critical and mission-critical applications, it is important to make provisions for a wide range of contingencies, by providing means for fault tolerance. In this paper, we discuss the specification of a flight control system that is fault tolerant with respect to sensor faults. Redundancy is provided by analytical relations that hold between sensor readings; depending on the conditions, this redundancy can be used to detect, identify and accommodate sensor faults.

Keywords

Flight Control Systems; Fault Tolerance; Flight Dynamics; Sensor Failure Detection, Identification, and Accommodation.

1 Introduction

Providing the fault tolerant capability (FTC) to control systems is a major issue in domains where system fault occurrences may give rise to unrecoverable damages to people and/or to very expensive devices (e.g., nuclear plants, space missions, aircrafts). In this paper we discuss modeling and specifying the fault tolerant capability of a flight control system (FCS), with respect to sensor faults. Relevant issues include: achieving fault tolerance for FCS's, defining fault domain (i.e., specifying fault hypotheses), adequate modeling, and adequate representation of the model.

A typical approach to introduce fault tolerance in a control system is physical redundancy of components. The detection/identification of a fault is achieved by comparing the behavior of replicated components accomplishing the same task and having the same features. Cost and complexity considerations led recently an increasing interest in alternative approaches, mostly based on analytical redundancy. Outputs of components measuring different but related items are observed in order to detect/identify the faulty component. We go towards the specification of the fault tolerant capability (based on analytical redundancy) for a FCS, bounded to sensors faults.

We only focus on *critical* sensors, i.e. sensors measuring modes of the aircraft that change too quickly to be controlled by the pilot. We neglect multiple, transient and simultaneous faults. Therefore the goal of this work is not producing specifications for a complete fault tolerant capability of a real world FCS, but conceptually addressing most of issues that also persist in large scale systems. The space of sensor readings is partitioned, under fault hypotheses, and for each partition analytical relations among system variables are introduced in order to characterize the partition and to express constraints that must be satisfied when a fault occurs while system conditions fall in the partition, in order to guarantee stability and maneuverability of the aircraft.

The formulation of such relations using the Software Cost Reduction (SCR) notation is based on the tabular representation of variable behavior in SCR: it is straightforward to introduce the expression whose result is the

value that a variable must assume, under given conditions. Functional dependency among tables is exploited to catch out indirect relations. On the other hand, several representation/execution issues are raised here on the usage of SCR for such a domain (e.g., modeling time).

Section 2 gives an overview of a FCS, in terms of hardware/software components and input/output variables. In Section 3 analytical relations are introduced that describe how analytical redundancy provides fault tolerant capabilities to a FCS; domain partition is also provided. In Section 4 major issues related to the SCR modeling are dealt, and the specification refinement process, as part of validation, is also sketched. In Section 5 a wider perspective of the problem is given, as part of an ongoing project, where current and future possible directions are outlined. Conclusions are reported in Section 6.

2 A Fault Tolerant Flight Control System

2.1 Structure of a Flight Control System

Figure 1 shows the basic architecture of a Fly-By-Wire Flight Control System (FBW-FCS). In FBW technology conventional mechanical controls are replaced by electronic devices coupled to a digital computer. The net result is a more efficient, easier to maneuver aircraft. Four subsystems form the core of such FCS's. The *Measurement Subsystem* (MS) consists of the *Sensors* and the *Conditioning Electronics*. It measures quantities that allow observation of the state of the aircraft. *Primary* sensors are those sensors whose correct operation is required to maintain a safe flight condition. The *Actuator Subsystem* (AS) consists of the *Control Surfaces*, the *Power Control Units* (PCU's), and the *Engines*. It produces aerodynamic and thrust forces and moments by means of which the FCS controls the state of the aircraft. The *Control Panel Subsystem* contains all control devices and displays through which the pilot maneuvers the aircraft. The *Flight Control Software subsystem* (FCSw) includes all software components of the FCS. It interfaces to the hardware of the FCS through A/D and D/A cards (not shown in the figure). Current measurements, pilot inputs, and commands to the actuators are processed according to the Flight Control Law (FCL) to obtain the commands to the actuators at the next time step. Dash blocks and arrows represent the system providing Analytical Redundancy based Fault Tolerant Capability (AR-FTC) to the FCS and will be described in the next section.

2.2 Deploying Redundancy for Fault Tolerance

Any hardware or software fault within the FCS can compromise the safety of the aircraft. For this reason FBW-FCS's must meet strict Fault Tolerance (FT) requirements. The standard solution adopted to achieve fault tolerance is physical redundancy. A typical multichannel architecture for the FCS consists of three intercommunicating FCS's, that are equivalent —yet able to work independently. A voting mechanism checks for consistency and can, under some conditions, identify faulty components. Brute force physical redundancy is no panacea, however: product redundancy (duplicating copies of the same product) does not protect against design faults, and design redundancy (independent designs) has two major drawbacks, which are cost and complexity. Complexity, in turn, adversely affects overall reliability, and defeats the whole purpose of the fault tolerant scheme. This additional complexity affects not only the development costs, but also the maintenance costs.

These factors have, in recent years, led to an increased interest in alternative approaches for enhancing FCS's reliability. In the past two decades a variety of techniques based on *Analytical* Redundancy (AR) have been suggested for fault detection purposes in a number of applications [12]. The AR approach is based on the idea that the output of sensors measuring different but functionally related variables can be analyzed in order to detect a fault and identify the faulty component. Furthermore, preserved observability allows estimating the measurement of an isolated (allegedly faulty) sensor, while preserved controllability allows controlling the system with an isolated (allegedly faulty) actuator. Fault tolerance is achieved by means of software routines that process sensor outputs and actuator inputs to check for consistency with respect to the analytical model of the system. If an inconsistency is detected, the faulty component is isolated and the flight control law is reconfigured accordingly. By introducing AR it is possible to take off redundant sensors, electronics, mechanical linkages, hydraulic lines, PCU's, etc., thus cutting costs and weight, and reducing overall complexity of the FCS. Physical redundancy would be required only where either post-failure system observability and controllability are not preserved or detection of the fault by means of AR is not feasible in the first place.

Application of AR in FCS's is not new. The very same airplane used to conduct research on FBW technology was also used as testbed for an AR based fault detection algorithm [14]. The algorithm showed adequate performance during flight tests. However, poor robustness to modeling errors and the amount of required modeling hampered further development. Since then, a number of results have been obtained in the area of robust fault de-
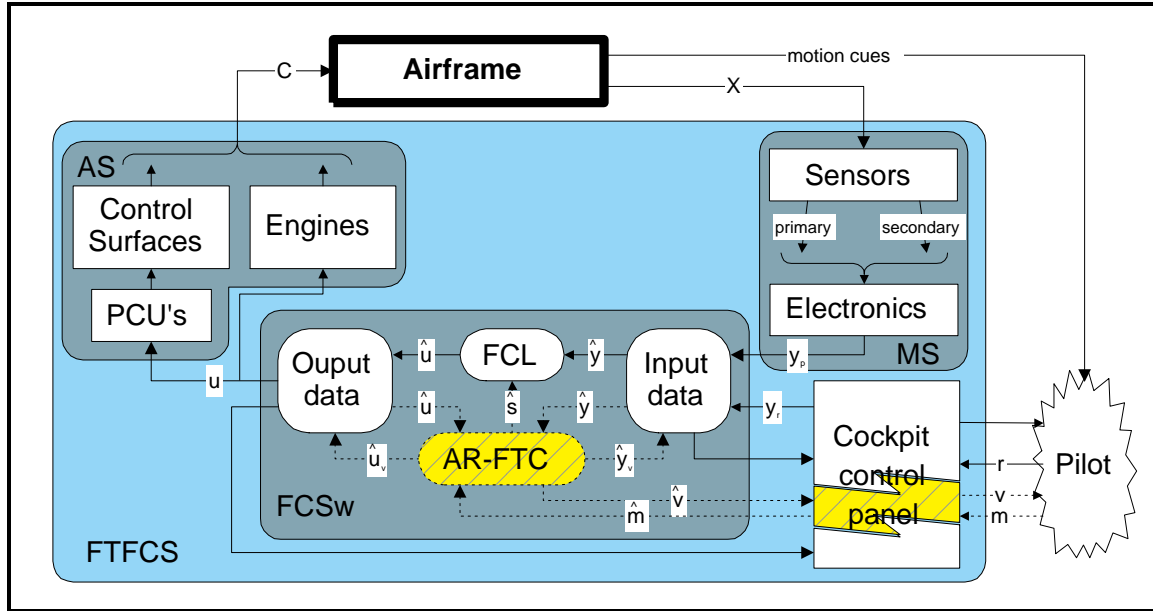
Figure 1: Fly by Wire Flight Control System.

tection [11]. *Unknown-input observers, robust parity relations, adaptive modeling*, and H_∞ *optimization* are a few examples. While recent research has enabled us to gain new insights into modeling analytical redundancy, it has fallen short of an integrated design methodology involving feasibility analysis, requirements specification, and certification of AR based fault tolerant control systems. Exploring strengths, weaknesses, related degree of reduction of physical redundancy, and overall reliability is a fundamental step in the engineering process of such systems.

3 Analytical Redundancy for Fault Tolerance

3.1 The Flight Control System and Its Environment

The airplane we adopt in this study is an F16. A detailed non-linear model of the dynamics of this airplane is presented in [13]. The analytical redundancy of a fault tolerant flight control system depends on an analytical model of the system and its environment. The dynamics of many systems can be described in terms of a set of relations among its inputs, outputs, states, and state derivatives. These relations represent constrains imposed by laws of mechanics, electronics, and thermodynamics upon system inputs, outputs, and their derivatives. Because of neglected dynamics, disturbances, and measurement errors

the analytical model of a system is not necessarily a truthful representation of the real system. Control system designers are mindful of this discrepancy and adopt design techniques that are robust with respect to such uncertainties.

In describing how analytical redundancy provides fault tolerance capabilities to a flight control system, we adopt the following notation:

$$R_c(state, state\ derivative, input, output,$$
$$process\ uncertainty, measurement\ error)\ (1)$$
$$R_d(current\ state, new\ state, input, output,$$
$$process\ uncertainty, measurement\ error)\ (2)$$

Parameters involved in these relations are vectors. The relations are deterministic with respect to the first four parameters, and stochastic with respect to the last two. Whenever a parameter is not involved in a relation, we write the symbol '-' in its place. Relation (1) is used for time *continuous* models, where all parameters are evaluated at the same instant t. Relation (2) is used for time *discrete* models, where all parameters are evaluated at the discrete time t_k, except the *new state*, which is evaluated at t_{k+1}. The difference $(t_{k+1} - t_k)$ is the sampling time of the discrete system. For the sake simplicity we describe only the most relevant parameters of the relations that we introduce in the remainder of the paper. Accordingly, we will often skip the description of *state, state-derivative, process-uncertainty,* and *measurement-error* parameters unless they play a prominent role in the analytical redun-

dancy framework.

The following relations represent the analytical models of the hardware systems in Fig. 1:

$$P\left(x(t), \dot{x}(t), c(t), x(t), \xi(t), -\right) \tag{3}$$

$$A\left(x_a(t), \dot{x}_a(t), u(t), c(t), \xi_a(t), \eta_a(t)\right) \tag{4}$$

$$M_p\left(x_p(t), \dot{x}_p(t), x(t), y_p(t), \xi_p(t), \eta_p(t)\right) \tag{5}$$

$$M_r\left(x_r(t), \dot{x}_r(t), r(t), y_r(t), \xi_r(t), \eta_r(t)\right) \tag{6}$$

Relation (3) describes the dynamics of the aircraft, i.e. the *process* to be controlled by the FCS. It involves force, moment, kinematics, and navigation equations. The state vector includes the flight variables used in the above equations. A typical state vector is:

$$x = [U, V, W, P, Q, R, \Phi, \Theta, \Psi, p_N, p_E, h]^T \tag{7}$$

where the elements are the three linear velocities, the three angular velocities, the three attitude angles, and North, East, and altitude position of the aircraft. Forces and moments applied to the airframe by the control surfaces and the engines are included in the input vector $c(t)$. The output coincides with the aircraft state and represents the *actual* value of the flight variables in (7). Since it is a set of actual values, there is no output uncertainty. The process error vector ξ is related to the uncertainty of the relation with respect to neglected dynamics and unknown inputs.

Relation (4) describes the dynamics of the actuator subsystem. The input vector $u(t)$ includes command signals to the actuators, while the output vector includes thrust and aerodynamic forces and moments. Relation (5) describes the dynamics between aircraft state $x(t)$ and process measurements $y_p(t)$. A typical set of sensors provides the following measurements:

$$y_p = \left[P_s, P_t, \alpha, \beta, A_x, A_y, A_z, \tilde{P}, \tilde{Q}, \tilde{R}\right]^T \tag{8}$$

These are static pressure, total pressure, angle of attack, sideslip angle, body accelerations, and body angular rates respectively. To differentiate *measured* from *actual* body rates we adopt the 'tilde' notation. Relation (6) describes the dynamics between the *actual* position of pilot controls $r(t)$ and their measurements $y_r(t)$. The two measurement vectors will be referred to in the sequel as:

$$y(t) = [y_p(t), y_r(t)]^T \tag{9}$$

The following relations represent the analytical models of the software systems in Fig. 1:

$$I\left(-, -, y(t_k), \hat{y}(t_k), -, \eta_I(t_k)\right) \tag{10}$$

$$L\left(x_L(t_k), x_L(t_{k+1}), [\hat{y}(t_k), \hat{u}(t_k)], \hat{u}(t_{k+1}), \right.$$
$$\left. -, \eta_L(t_k)\right) \tag{11}$$

$$O\left(-, -, \hat{u}(t_k), u(t_k), -, \eta_O(t_k)\right) \tag{12}$$

The FCSw closes the control loop between sensors and actuators subsystems. To distinguish software variables from related electrical signals we adopt the 'hat' notation. Relation (10) represents the relationship between measurement samples $y(t_k)$ and the corresponding software variables $\hat{y}(t_k)$. Since this is an algebraic relation, there is no need to introduce state variables. $\eta(t_k)$ takes into account quantization error. Relation (11) describes the dynamics of the the flight control law. Current value of sensor measurements and actuator commands are processed to produce the actuator commands at the next time step $\hat{x}(t_{k+1})$. Relation (12) describes the relationship between software commands $\hat{u}(t_k)$ and electrical commands $u(t_k)$.

In order to complete the set of relationships needed to illustrate the principles at the basis of AR-FTFCS's we introduce two relationships capturing the FT requirements:

$$R_h\left(x(t), \dot{x}(t), r(t), \dot{r}(t)\right) \tag{13}$$

$$R_l\left(x(t), \dot{x}(t), r(t), \dot{r}(t)\right) \tag{14}$$

Relation (13) describes the high priority responsiveness requirements of the aircraft to pilot commands in terms of the true state of the airplane, the input commands, and their derivatives. Relation (14) describes the low priority requirements. For an airplane to be safe it is mandatory that the high priority requirements are preserved even in case of fault.

3.2 The AR-FTFCS

After having introduced the analytical model of the FCS, its environment, and its fault tolerant requirements it is possible to illustrate how an AR-FTFCS works.

At the instant t_k new measurements are available as software data $\hat{y}(t_k)$. The elements of this vector are not independent; they are correlated by means of relations (3), (5), (4), (10), and (12). Furthermore, they are correlated to $\hat{u}(t_k)$ by virtue of the same relations. By analyzing sensor measurement and actuator command histories it is possible to check whether the above relations are satisfied. If a fault within the hardware loop produces an inconsistency with respect to the analytical model the system is said to hold AR properties allowing *detection* of the fault. After detection of the fault it is necessary to identify which component has failed. Each component of the FCS plays a different role within relations (5) and (4). Hence, the distortion affecting these relations at the occurrence of a failure depends on the component failed and on the fault mode. By processing sensor measurement and actuator command histories it is possible to locate the source of distortion. If a fault within the hardware loop produces a distinct signature in terms of commands/measurements

correlation the system is said to hold AR properties allowing *identification* of the fault. Once the faulty component component is identified, the FCS needs to be accommodated in order to preserve responsiveness requirements. Accommodation can be carried out at the software level because the flight control algorithm is not unique. Given relations (3), (5), (4), (10), and (12) describing the dynamics of the aircraft, sensors, actuators, and interfaces with the FCL, there can be a number of different control algorithms satisfying responsiveness requirements. Some of these algorithms do not use all of the sensors and/or actuators available. Hence, if a hardware component of the FCS fails, responsiveness requirements can be maintained by switching to a control algorithm that does not employ that component. If such an algorithm exists the system is said to hold AR allowing *accommodation* of the fault.

Figure 1 shows how a FCS is enhanced to an AR-FTFCS. The dash blocks and arrows represent the subsystem providing Fault Tolerant Capability (FTC) to the FCS. The core of this subsystem is the AR-FTC software module, while the dash section within the *Control Panel* block represents the hardware interface to the pilot. Following the notation adopted in the previous section we describe the FTC subsystem by means of the following two relations:

$$AR_{Sw}(x_{ar}(t_k), x_{ar}(t_{k+1}), [\hat{y}(t_k), \hat{u}(t_k), \hat{m}(t_k)],$$
$$[\hat{y}_v(t_k), \hat{u}_v(t_{k+1}), \hat{v}(t_k), \hat{s}(t_k)], -, \eta_{ar}(t_k)) \quad (15)$$
$$AR_{Hw}(-, -, [\hat{v}(t_k), m(t_k)], [\hat{m}(t_k), v(t_k)]$$
$$-, \eta_{ar}(t_k)) \quad (16)$$

Relation (15) describes the dynamics of the software module. It processes current measurements $\hat{y}(t_k)$ and commands $\hat{u}(t_k)$ to validate $\hat{y}(t_k)$ against the analytical model of the system. If no inconsistencies are detected the FCL module takes over and produces the new command $\hat{u}(t_{k+1})$. If an inconsistency is detected the AR-FTC module further processes incoming data to identify the faulty component. Then, it either produces a *virtual* set of validated measurements $\hat{y}_v(t_k)$ for the FCL, or it bypasses the FCL and produces a new set of validated commands $\hat{u}_v(t_{k+1})$ according to a *safe* control law that does not use the faulty component. The two options are typically adopted for sensor and actuator faults respectively. If a component of the actuator subsystem fails then it is often necessary to reconfigure the control law to take into account the control deficiency. If a sensor fails its output can be estimated and the estimation substituted into the measurement vector $\hat{y}(t_k)$ to produce $\hat{y}_v(t_k)$. However, the solution can be adopted where the FCL is bypassed and an alternative control law is used in its place. The $\hat{s}(t_k)$ signal is used to synchronize execution of the FCL and the AR-FTC modules. $\hat{m}(t_k)$ and $\hat{v}(t_k)$ are the operational mode selected by the pilot and the diagnos-

Fault mode	y_i (Volts)	$\hat{y}_i$ (deg/sec)
Loss of signal	$[2.0, 2.5]$	$[-22, 0]$
Loss of power	0	-90
Loss of ground	12	90

Table 1: Fault modes

tic information respectively. Relation (16) represents the relationship between pilot's controls $m(t_k)$ and displays $v(t_k)$ and related software variables $\hat{m}(t_k)$ and $\hat{v}(t_k)$.

3.3 Fault Hypotheses

In the previous section we have shown how a system featuring AR properties can be made fault tolerant with respect to sensor faults at the software level. However, software along with its supporting hardware (computers, data buses, etc.) and hardware systems other than sensors and actuators can fail as well. AR cannot be adopted to provide fault tolerance with respect to failure of such components; a different approach must be adopted for these types of faults.

In this phase of the study we focus our attention on sensor faults only. More specifically, we require fault tolerance with respect to failure of the *roll, pitch*, and *yaw* rate gyros. The sensors used are a solid state rate gyro where a vibrating element is used to measure rotational velocity by employing the Coriolis principle. The output range of the sensor is ± 90deg/sec and its bandwidth is 18Hz. The output of the sensor has been recorded while simulating the failures. Three different fault modes have been considered: *loss of signal, loss of power, loss of ground reference*. The fault modes along with outputs of the sensor and the values of the correspondent software variable are listed in Table 1.

3.4 Fault Modeling

We have to point out that even though analytical redundancy does enable us to achieve some level of fault tolerance, it does not guarantee arbitrary levels of precision in detecting, identifying, and accommodating sensor faults. An exhaustive feasibility analysis covering components subject to failure, fault modes, and possible state evolution for actuator, aircraft, and sensor systems is required. To explain how detectability and identifiability problems arise we will refer once again to relations (3), (5), (4), (10), and (12). These relations can be assembled to form one single relation that captures the system AR at software level in terms of sensor measurement and actuator command histories:

$$R_g([\hat{u}(t_k), \hat{u}(t_{k-1}), \ldots, \hat{u}(t_{k-m})],$$

$$[\hat{y}(t_k), \hat{y}(t_{k-1}), \ldots, \hat{y}(t_{k-n})],$$
$$[\hat{\nu}(t_k), \hat{\nu}(t_{k-1}), \ldots, \hat{\nu}(t_{k-p})]]) \qquad (17)$$

Here ν represents a global uncertainty term collecting all process uncertainty and measurement error terms in relations (3), (5), (4), (10), and (12). Variables n, m, and p represent the depths of the input, output, and uncertainty sequences respectively. If we consider the space whose points have coordinates given by the elements involved in relation (17), we can distinguish those regions of the space where relation (17) is satisfied, from those where it is not. Furthermore, we can characterize those regions related to distortions of relation (17) caused by the given fault modes.

Following Mili et al. [1, 7], we model a fault tolerant scheme by means of a partition of the relevant state space into a hierarchy of classes that represent degrees of *correctness*, degrees of *maskability*, and degrees of *recoverability*. For a program, the relevant state space is the set defined in terms of all the values taken by all the state variables of the program; for a set of sensors, the relevant state space is the set defined in terms of all the values taken by all the sensor readings. The partition that we derive for our purposes is given in figure 2. Process uncertainties (disturbances, simplifying hypotheses, modeling shortcuts, etc) make the actual partition more complex than the original model [1, 7]. In its current form, this partition is incomplete, and is being refined.

The inner ellipsis of figure 2 represents states for which the deterministic relationship within relation (17) holds. The outer ellipsis contains the points for which the stochastic relation (17) holds. Points outside this region imply an inconsistency with the analytical model of the system. The three triangles marked *F1, F2,* and *F3* contain points related to three different fault modes. The intersection between the region '*F1 ∪ F2 ∪ F3*' and the outer ellipsis contains those points that are related to a fault mode, but that preserve relation (17). Hence, this region represents those states where a fault mode is not detectable by means of AR. This region is marked *Detection non Feasible* in the figure. The region marked *Identification non Feasible* contains those points for which the identification of a fault cannot be achieved either because that fault mode has not been considered within the specifications, or because failure effects do not allow us to distinguish between the fault modes.

To describe accommodation feasibility in analogous terms we need to consider the space whose points represent the aircraft states. If after the failure of a component the FCL can be reconfigured to satisfy the safety requirements then accommodation is feasible within the whole space. If instead there are states that cannot be reached without violating the safety requirements the space will be partitioned in regions where accommodation is feasible and regions where it is not. As a limit case, accommodation is not feasible in the whole space if there is no FCL that would satisfy the safety requirements.

4 SCR Modeling

The derivation of this specification is part of a larger project whose purpose is to validate and certify an adaptive fault tolerant capability (AR-FTC in figure 1) for a flight control system, which is concurrently being implemented using a *radial base function* neural network [8]. Our intent is that the specification will be used as an oracle in the testing task which aims to validate/ certify the fault tolerant capability. Consequently, it is rather imperative that the specification be written in a language that is supported by automated tools; so that in the validation/ certification phase, the neural net and the executable specification can be executed independently to provide a basis for checking the former against the latter. We have chosen to use SCR as the specification vehicle, because it lends itself to this type of application: tabular representations, which form the semantic foundations of SCR, were used in [3] to specify the requirements of the Navy's A7-E aircraft and in [10] to specify nuclear power plants; SCR was used to specify an autopilot [2], to specify a variety of high assurance applications [5, 6], and to specify some functions of the space shuttle software [15].

4.1 Scope of the Specification

Figure 1 shows the structure of the overall aircraft system, including the data flow between the aircraft, the flight control system, the cockpit controls, and the environment. The first issue we must address is to delimit the boundaries of our specification. We have pondered two possible options, which we denote by option 1 and option 2: whereas option 1 focuses on the inputs and outputs of the fault tolerant capability component (re: *AR-FTC*, in figure 1), option 2 (the aggregate of the flight control system, with the aircraft) considers the impact of the outputs of the AR-FTC on the aircraft state. The choice of an option is driven by the following considerations.

- *Generality/ Abstraction*. For a given situation, defined by a set of sensor readings, there are many sequences of actions that a flight control system can follow to achieve/ maintain the maneuverability/ stability of the aircraft. At any instant, these actions may be different, but their combined effect over time is identical. Hence by virtue of abstraction (we do not wish to deal with the detailed mechanics of how the AR-FTC operates) and generality (writing specifications that apply across a wide range of possible
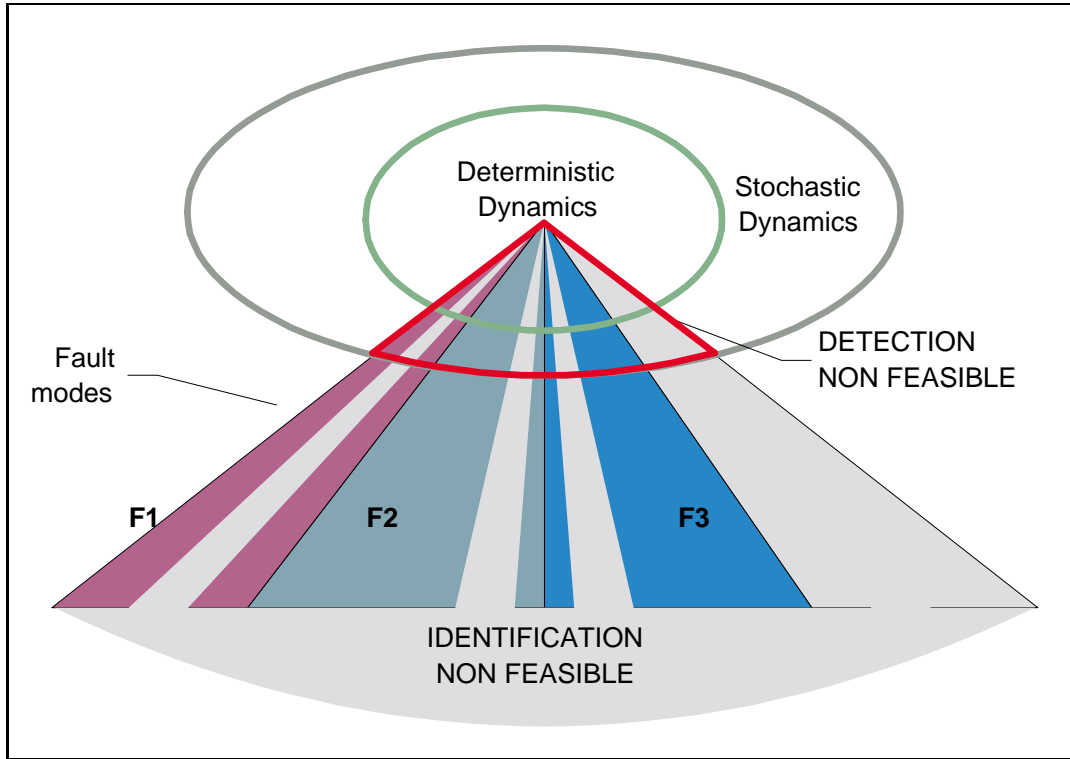
Figure 2: Partition of the Space of Sensor Readings

implementations), the second option is better than the first.

- *Observability/ Controllability*. If we choose option 2, then we cannot judge the outputs of the AR-FTC directly, but we have to observe their effect on the aircraft. This gives us much lower observability of the AR-FTC than option 1. Controllability is the same for both options.

Ultimately, this decision amounts to choosing between observability (option 1), i.e. the ability to observe and monitor the exact values that are produced by the FTFCS implementation, and abstraction (option 2), i.e. the ability to give the implementer some latitude in how to maintain maneuverability. We have ruled in favor of abstraction.

4.2   Representation Issues

By virtue of the choice discussed above, the input variables of the specification are the sensor readings of relevant flight parameters (altitude, speed, acceleration, angle of attack, rate gyros, aileron deflections, elevator deflections, rudder deflection) and the actuator input values; the output variables are the actual values (i.e., the validated vectors) of the same parameters and a fault report.

Hence, for parameter Y, for example, we define variable mY (named after SCR parlance: monitored Y) which represents the sensor reading for Y, and variable cY (controlled Y) which represents the actual value of parameter Y. Because of sensor failures, cY may differ from mY. In addition, because of the latency of the FCS and (especially) of the aircraft (in reacting to adjusted actuator values), the value of cY at time t ($cY(t)$) is not functionally related to the value of mY at the same time t ($mY(t)$), but rather to previous values of mY. In addition to the sensor readings of the flight parameters, the set of input variables also includes the values of the cockpit controls. The second and fourth columns of table given in figure 3 show the structure of the input and output spaces. We now give a mapping of those spaces into the partition of sensor readings in figure 2.

The innermost ellipsis of figure 2 represents the fault-free case, whose relation takes the form

$$R = \{(\langle mY, mU \rangle, \langle cY, cU \rangle)|p(mY, mU, cY, cU)\},$$

where predicate p characterizes the condition of deterministic fault freedom. The outermost ellipsis of figure 2 represents the case in which a measurement error in sensor readings leads to uncertainty in the fault process; the rela-

Family of Variables	Input (monitored) Variable	State (Term) Variable	Output (controlled) Variable
Flight Parameter, Y	mY	tY	cY
Actuator Inputs, U	mU	tU	cU
Pilot Command, MR	mMR		
Fault Report, V			cV

Figure 3: Space Structure

tion for this ellipsis takes the form

$$R' = \{(\langle mY, mU\rangle, \langle cY, cU\rangle) | q(mY, mU, cY, cU, \eta, \xi)\},$$

where η represents the measurement error and ξ the process uncertainty; predicate q characterizes the condition of fault status in the stochastic dynamics.

Each triangle F_i of figure 2 represents a different fault mode, whose relation take the form

$$R_{F_i} = \{(\langle mY, mU\rangle) | f_i(mY, mU)\},$$

where predicate f_i characterize the condition of fault mode i. Different shades and bold lines in figure 2 separate areas with different fault capabilities. In order for a fault i to be detected, it must satisfy

$$R_{F_i} \bigcap R = \emptyset;$$

also, to be identified, it must satisfy

$$R_{F_i} \bigcap R_{F_j} = \emptyset \quad (\forall j \neq i).$$

With regard to accomodability, the specification must reflect the property that the aircraft remains maneuverable despite the presence of sensor faults. In particular, in every triangle F_i maneuverability is a binary function in variables $mMR(t)$ and $cY(t)$ (linking pilot commands to actual flight parameters). Because $mMR(t)$ and $cY(t)$ are not instant variables but rather functions of time, it is conceivable that the value of cY at some time t be a function of the value of mMR at a previous time $t' < t$.

4.3 Modeling Issues

Time is inherent in the specification of the FTFCS. Execution of the FTFCS takes place in the context of a sequence of sensor inputs which, except for faults, represents a physically feasible flight path. The FTFCS is aware of the passage of time through the advent of clock pulses;

at each clock pulse, the FTFCS takes a snapshot of the sensor readings, processes them, computes actuator values, then awaits the next clock pulse. Note that the sensor readings may well remain constant across two or more clock pulses; the FTFCS processes them at each clock pulse all the same. On the other hand, sensor readings may take several distinct values between two successive clock pulses; the FTFCS is only aware of their two values at the successive clock pulses.

Whereas the real-time operation of the FTFCS is driven by the clock pulses, the execution of the SCR specification (for the purposes of validating the specification or verifying implementations against it) is driven by the successive application of the functions defined by SCR's tabular expressions on the input variables and the state variables. SCR specifications are executed in a kind of a *batch* mode, where the real time between two successive function applications need not bear any relation to the actual time between two successive clock pulses.

The concept of time arises naturally in flight dynamics equations, which are differential equations of flight parameters and pilot commands. Let us consider some controlled variable cX, and let us assume that this variable satisfies the following differential equation:

$$\frac{d(cX)}{dt} = F(t),$$

where t is the time variable, F is a function of t that potentially involves monitored and controlled variables (including cX), and $\frac{dX}{dt}$ is the derivative of X with respect to time. If we approximate the derivative by means of finite differences, we find $\frac{(cX(t) - cX(t-\delta t))}{\delta t} = F(t)$. If we let δt be the interval between two successive clock pulses, and let this be the unit of time (i.e., $\delta t = 1$), then $cX(t)$ and $cX(t - \delta t)$ measure the current value and the past value of parameter cX. Solving this equation for $cX(t)$, we find

$$cX(t) = cX(t - \delta t) + F(t). \tag{18}$$

Each application of this transformation (from $cX(t - \delta t)$ to $cX(t)$) represents the effect of the advent of one clock pulse. In order to distinguish between the current and past values of variable cX, we use SCR's concept of *term variable*. To each flight parameter (X) we associate a *term variable*, which we denote by prefixing the variable name with t; the term variable is used to represent the value of the variable at the preceding clock pulse. In order to represent the transformation described in equation (18), we write SCR tables to perform the following transformations *in sequence*.

```
tX := cX;
cX := tX + F;
```

We cannot merely define two tabular expressions that compute variables cX and tX according to these formulas, for they produce a circular reference (and SCR does not recognize the sequence command —the order of execution in SCR is driven merely by functional dependencies).

A tantalizing alternative is, of course, to use SCR's primed variable convention, whereby the primed version of any given (non-primed) variable is the previous value of that variable. This option does not work for our purposes, because of the specific interpretation of *previous value* in SCR. SCR is event-driven, where each change of value of any variable is understood to be an event; by contrast, our model is time-driven, where an event is the advent of a clock pulse. If, between two successive clock pulses, three monitored variables change values, SCR considers that it has witnessed three events, and *previous* refers to the most recent one; by contrast, our model considers that only one event has occurred, and *previous* refers to the state of the system at the previous clock pulse.

5 Assessment

In this section, we review our specification project (although it is still in progress) and assess some of our decisions, with partial hindsight.

5.1 SCR Adequacy

We briefly report on our experience with using SCR for the purposes of our specific situation. We acknowledge that we have very little prior experience with SCR, and our comments must be qualified accordingly.

The general pattern of a table in SCR is to compute a controlled variable in terms of monitored variables and possibly term variables. Many requirements that we encounter are instead best formulated as a relation between monitored variables (to limit the domain of a relation), or a relation between controlled variables (to limit the range of a relation). Also, even if the controlled variable is a deterministic function of the monitored variables, it may be more natural to represent this function by a conjunction of non-oriented, non-deterministic, relations.

We find it unsettling that in a specification that has a large number of tables, the only composition operator between these tables is functional dependency, which is not even explicit. We would find it more powerful to have a wide vocabulary of composition operators which we can use to compose tables together. There is undoubtedly a sound basis for letting functional dependency be the sole criterion that determines the order of evaluation of tabular expressions. But our experience with modeling time would have been more successful if we had the ability to impose an arbitrary sequencing between tables, to break the cycle of circular dependencies. (Note: It is possible to define a refinement-monotonic sequence like operator, using demonic semantics). Furthermore, because tables are combined only with functional dependencies (rather with refinement-monotonic composition operators) we find no natural discipline for stepwise specification generation. Such a discipline would enable us to compose a specification in a stepwise manner, and to know that as we produce more and more tables, the overall specification grows increasingly more refined (until completeness). The structure afforded by such explicit composition operators can be used to control the complexity of subsequent validation and verification tasks.

Because SCR, and the tabular expressions on which it is based [9, 4], support model-based specifications, it elicits more detail from the specifier than a behavioral specification. This excess detail makes the specification more complex, and may lead to inconsistencies.

We find that the data type offerings of SCR are more akin to those of a programming language than to those of a specification language. In an application such as ours, we needed a variety of data types, ranging from angles (degrees) to durations (seconds) to engine speeds (rotations per minute) to positions (meters) to speeds (meters/ second) to accelerations (meters/ second/ second) to angular velocity (degree/ second), etc. We found ourselves mapping all of them into reals, when a language supported typing system that provided a wide range of data types and a corresponding type checking function would have enhanced the readability and reliability of our specification. We also found that it would have been helpful if SCR provided dimension-checking functions, whereby whenever we write an equation, it checks the dimensions of both sides to ensure that they are consistent. Whether this is an extension of the type checking function, a separate function, or actually the same (only more elaborate) function, we do not know.

Many of the issues that we raised here are interrelated (e.g. a single design decision dictates a host of interrelated issues), and many stem from legitimate design tradeoffs (e.g. favoring efficient executability vs expressive power). We assume that as we become more acquainted with the spirit of SCR's specification model, some of these issues will may grow increasingly insignificant.

5.2 Non-determinacy

We faced a dilemma while trying to derive a specification for the fault tolerant capability flight control system, dealing with the determinacy of the specification. We had two options:

Make the specification deterministic. This is more natural, from the standpoint of SCR (which revolves around the pattern of formulating controlled variables as a function of monitored variables), and yields generally simpler specifications. The main drawback of this solution, of course, is that it forces us to second guess the designer of the neural net, because we have to derive a specification for the exact function that the neural net is implementing. This, in turn, has two drawbacks: first, it imposes much coordination between the implementer team and the specifier team, and is counterproductive from a V&V viewpoint (V&V relies primarily on redundancy); second, it imposes early constraints on the designer, prohibiting him from altering design decisions that affect the specifier team.

Make the specification non deterministic. The position here is to let the specification focus on expressing the desired functional properties, without going as far as to uniquely specify which output will satisfy these desired properties. This solution is consistent with traditional guidelines for good specification, but causes some difficulty in SCR, because SCR does not handle non-determinacy naturally. This is the most striking limitation we have encountered using SCR. In our example (and certainly in many other applications as well), we often encounter requirements that are not deterministic; also, many complex requirements are best formulated as the aggregate of a set of simpler, non-deterministic requirements.

We felt very justified in choosing the second option, but have found that it raises an issue which may, with hindsight, cause us to reassess our choice: Under the first option (deterministic specification), the specification of the system does not have to capture the criteria under which differences of output between the specification and the implementation can be considered tolerable; this decision can be made during the verification and validation step, by the V&V team, to take into consideration any special circumstances that may arise at run-time. By contrast, under the second option, the tolerance margins have to be hardcoded into the specification, and cannot be adjusted subsequently by the V&V team to account for special testing/operational conditions. Hence both options force us to make early decisions: The first option imposes on us to agree with the implementer on specific design decisions; the second option imposes on us to agree with the V&V team on specific tolerance margins.

6 Prospects

6.1 A Testing Plan

Our plan calls for using the target specification as an oracle in the test plan of the neural network. Specifically, the neural net feeds its inputs into a certified flight simulator, which plays the role of the aircraft components in the graph of figure 1. This aggregate is placed side by side with the SCR specification, whereby the SCR is used as an oracle to test the neural net. Input data is submitted to the system under test and the SCR oracle, to check for correctness. This input data is the aggregate of sensor readings and pilot controls, which are collected from previously collected flight simulation data. The purpose of the testing plan is to make a ruling on the certifiability of the neural net as an implementation of the fault tolerant capability of the flight control system. The system structure that we have derived for this purpose is presented in figure 4. The fault reports of the neural net and the SCR specification are compared for logical equality, producing the result shown in the lower right corner of the figure. On the other hand, the actual state of the aircraft, produced by the flight simulator, is matched against the pilot controls (by virtue of a law that captures aircraft maneuverability), to return a boolean indicator of whether the aircraft maintains adequate maneuverability (despite the possible presence of faults).

6.2 Interpreting Flight Dynamics Equations

In the process of deriving the SCR specification of the FTFCS system, we are really conducting two activities, namely modeling and representation:

- *Modeling.* This task deals with such matters as deciding which parameters are of interest, how do we represent the fault tolerant capability, how do we represent time, how do we approximate derivatives, how do we enforce sequencing of tabular evaluations/ executions, how do we reflect the dynamic nature of the system, how do we detect, identify and accommodate faults, etc.

- *Representation.* Generally speaking, this matter deals with how do we map our model into SCR terms, and how do we formulate our model in such a way as to take the best advantage of built-in SCR features.

Ideally, we would like to think of these two activities are being strictly sequential; i.e. modeling must be completed before representation can proceed. As attractive as it may
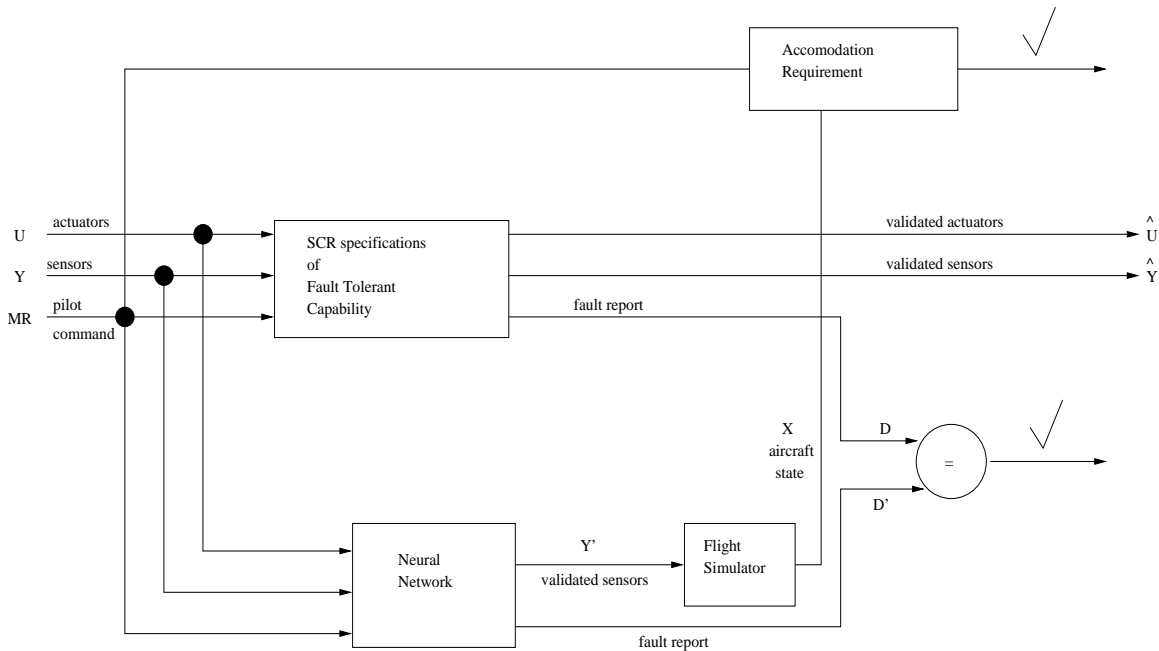
Figure 4: A Testing Plan for the Neural Net

be, this discipline has proven to be a challenge in practice, due to time pressures and to the impact of representation constraints on modeling decisions. This has led us to consider the possibility of producing a syntax directed translation of flight dynamics equations into SCR source code. The main advantage of this solution is that we get to encode all our modeling decisions in the syntax-directed rules; this allows us to keep our modeling options open until very late in the specification lifecycle; most important of all, this solution ensures that our modeling decisions are applied uniformly across all the equations of the specification.

6.3 Analytical Reasoning on Neural Nets

Traditional certification algorithms observe the behavior of a software product under test, and make probabilistic/ statistical inferences on the operational attributes of the product (reliability, availability, etc). The crucial hypothesis on which these probabilistic/ statistical arguments are based is that the software product will reproduce under field usage the behavior that it has exhibited under test. This hypothesis does not hold for adaptive neural nets, because they evolve their behavior (learn) as they practice their function. Of course, one may argue that they evolve their behavior for the better; but *better* in the sense of a neural net (convergence) is not necessarily better in the sense of correctness verification (monotonicity with respect to the refinement ordering). Concretely, a neu-

ral net may very well satisfy the SCR specification in the testing phase, and fail to satisfy it in the field usage phase, even though it converges. See figure 5.

In light of these observations, we envisage to complement the certification testing activity with an analytical method. Such a method would rely on some semantic analysis of the neural net, as well as some hypothesis regarding the data that it receives in the future.

7 Summary

In this paper we have discussed the formal specification, in SCR, of an adaptive fault tolerant flight control system. The specification is due to be used as an oracle in the certification of a radial basis function neural net that implements the adaptive scheme. The fault tolerant properties of the system, the adaptive nature of its implementation, and the specific application for which the specification is intended (certification), contribute to make this a unique experiment in system modeling and representation. In particular, the fact that the system implementation is adaptive (hence does not duplicate its behavior as it evolves) rules out traditional testing techniques. Also, the fact that the system's behavior is dependent on input history precludes the traditional static analysis techniques. The specification generation is under way, and we expect many of the modeling and representation decisions that we have discuss in this paper to remain *in flux*.
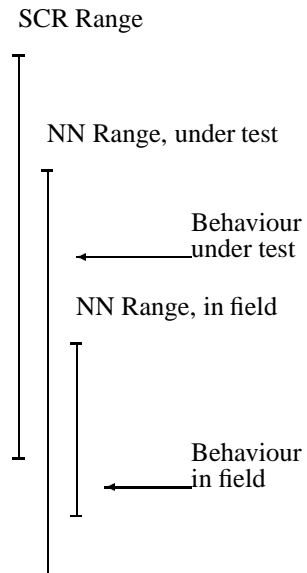
SCR Range

NN Range, under test

Behaviour
under test

NN Range, in field

Behaviour
in field

Figure 5: Convergence does Not Ensure Consistency

Acknowledgments

References

[1] H. Ammar, B. Cukic, C. Fuhrman, and Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10, 2000.

[2] R. Bharadwaj and C. Heitmeyer. Applying the SCR requirements method to a simple autopilot. In *Proceedings, Fourth Langley Formal Methods Workshop*, Hampton, VA, September 1997.

[3] K. L. Heninger, J. Kallander, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report 3876, United States Naval Research Laboratory, Washington D. C., 1978.

[4] R. Janicki, D. L. Parnas, and J. Zucker. Tabular representations in relational documents. In Ch. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, chapter 12, pages 184–196. Springer Verlag, January 1997.

[5] J. Kirby Jr, M. Archer, and C. Heitmeyer. Applying formal methods to a high security device: An experience report. In *Proceedings, IEEE International Symposium on High Assurance Systems Engineering*, pages 81–88. IEEE Computer Society Press, November 1999.

[6] J. Kirby Jr, M. Archer, and C. Heitmeyer. SCR: A practical approach to building high assurance: Comsec system. In *Proceedings, Annual Computer Security Applications Conference*. IEEE Computer Society Press, December 1999.

[7] A. Mili. *An Introduction to Program Fault Tolerance: A Structured Programming Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[8] F.E. Nasuti and M. Napolitano. Sensor failure detection, identification and accomodation using radial basis function networks. Technical report, West Virginia University, Mechanical and Aerospace Engineering, Morgantown, WV, November 1999.

[9] D. L. Parnas. Tabular representation of relations. Technical Report 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, October 1992.

[10] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April-June 1991.

[11] Ron J. Patton. Robust model-based fault diagnosis: The state of the art. In T. Ruokonen, editor, *IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes: SAFEPROCESS '94*, volume 1, pages 1–24, Helsinki Univ. Technol, Espoo, Finland, June 1994. IFAC, Springer Verlag.

[12] Ron J. Patton, Paul Frank, and Robert Clark. *Fault Diagnosis in Dynamic Systems: Theory and Applications*. Prentice Hall, 1989.

[13] Brian L. Stevens and Frank L. Lewis. *Aircraft Control and Simulation*. John Wiley and Sons, New York, N.Y., 1992.

[14] K.J. Szalai, R.R. Larson, and R.D. Glover. Flight experience with flight control redundancy management. In *AGARD Lecture Series No.109. Fault Tolerance Design and Redundancy Management Techniques*, AGARD Lecture Series, pages 8/1–27. AGARD, Neuilly-sur-Seine, France, October 1980.

[15] V. Wiels and S. Easterbrook. Formal modeling of space shuttle change request using SCR. Technical Report NASA-IVV-98-004, NASA IV& V Facility, Fairmont, WV 26554, http://www.ivv.nasa.gov/, 1998.

Towards formal methods for mathematical modeling

Ursula Martin
SRI International, Menlo Park CA
University of St Andrews, Scotland
um@dcs.st-and.ac.uk

Abstract

We survey mathematical modeling, the mathematical and computational technologies upon which it relies, and the potential sources of error. We assess formal methods and computational logic in this light, suggesting that certain well worn paths may have little to offer. We identify as opportunities for the future: analyzing requirements, assumptions and proof obligations for the assessment and confirmation of models, extending such techniques to architectures for heterogeneous distributed models with legacy components, using computational logic to extend the capabilities of computer algebra systems, and techniques for symbolic analysis.

1 Introduction

The purpose of this paper is to assess formal methods and computational logic from the point of view of mathematical modeling. It forms part of a larger research program assessing formal methods and computational logic for mathematics and its applications.

The techniques of mathematical modeling, that is of regarding a physical phenomenon as a dynamical system for the purposes of understanding and prediction, arose in the physical sciences during the twentieth century, were used widely in meteorological and defense applications and later spread to environmental, biological and geological modeling. They were transformed by modern computation, and by increasing reliance on modeling in many aspects of public policy, and have also become the keystone of US undergraduate math curriculum reform [38]. This paper concentrates on the issues arising in bioscience and environmental science, rather than on physical sciences, engineering or control theory: in particular we are considering computational rather than physical models.

In the first part of the paper we survey mathematical modeling, the math and software that it relies upon, and possible sources of error and user concern. We go into some detail, on the grounds that assessing how formal methods might be used in practice requires a general understanding of what the practice of modeling is. In the second part we consider how formal methods and computational logic might address these concerns, and identify some possible new directions.

Section 2 is a methodological aside. Section 3 contains an account of mathematical modeling, which we encapsulate as a "purposeful representation of reality". A modeler devises a "model world" to investigate some "purpose" in the "real world". A mathematical "model" of the model world is constructed using dynamical systems, and the modeler reasons within it. Almost universally today the reasoning is done with the aid of numeric or symbolic computation, so an "implementation" of the mathematical model is built in a computer system: from the "implementation" conclusions are drawn about the "model" or the "model world" and assessed against the hypotheses of the "model world" or against observations of the "real world". We may view this as a pipeline: $\{reality+purposes\} \rightarrow model\ world \rightarrow model \rightarrow implementation$.

Thus modeling relies on two underlying technologies: the mathematical theories of differential equations and dynamical systems, and the computational tools of numeric or symbolic computation.

In Section 4 we give a brief account of the first of these, the mathematical theories. We describe the kind of reasoning that is typically done, and assess the correctness issues. We note in particular that the mathematician developing the theories, the toolsmith using them to devise algorithms and the modeler using those algorithms may have somewhat different perspectives.

In Section 5 we consider the second technology, and describe numeric and symbolic computation and some of the correctness concerns that arise. Numerical systems are widely used because they always give an answer: it is suggested that general software engineering issues rather than bugs in algorithms or floating point arithmetic are the main cause of error. Symbolic computation systems are much less flexible, and further problems arise because of fundamental design issues which mean that continuous math is sometimes handled incorrectly.

Sections 4 and 5 considered the underlying technologies: in Section 6 we return to the modeling process itself and assess correctness concerns. While these can arise anywhere in the pipeline, it is the assessment of a "model" or "model world", against competitors and against purposes that attracts most attention in the modeling community, and in matters such as environmental prediction (for example, querying assumptions about ground water penetration) they can be subject to heated debate. In large or legacy models even tracking built-in assumptions can be hard.

Section 7 addresses how computational logic and formal methods may address some of the correctness concerns raised in the previous sections. The correctness of the mathematical and computational technologies can in principle be addressed using techniques of computational logic: we indicate the main notions for both. In particular we report briefly on our own work using heavy duty theorem proving in PVS to provide convenient embedded reasoning tools for computational mathematics systems. However we assert that in general the modeling community are users rather than creators of mathematics and software, and are not particularly concerned to have formal developments of either the underlying material or its applications in modeling, or to replace them with new foundational approaches: these are all regarded loosely speaking as "solved problems". While in principle techniques based on improved forms of symbolic computation, or on computational logic, would allow richer reasoning about models, it is hard to see them matching the flexibility of numerical systems or overcoming the investment in existing techniques.

Correctness concerns about the modeling pipeline involve, in so far as they can be formalized, tracking of requirements and assumptions, and here we judge there to be much greater potential for formal methods from the user's point of view. We report briefly on our own experience with light formal methods for tracking requirements, assumptions and proof obligations in computational mathematics systems.

In the light of the above Section 8 sets out four main opportunities for the future: analyzing requirements, assumptions and proof obligations for the assessment and confirmation of models, extending such techniques to architectures for heterogeneous distributed models with legacy components, using computational logic to extend the capabilities of computer algebra systems and improved techniques for symbolic analysis.

2 A methodological note

It would be easy enough to tell a rosy story within the contemporary rhetoric of formal methods and computational logic of their potential for mathematical modeling, illustrated with anecdotes of unreliable predictions from unsound models or bugs in numerical code. We might then, with some effort, treat a simple differential equation or verify a numerical algorithm within our formalism of choice, argue with the aid of a large bibliography about how such methods are "growing in importance", "vital for safety critical applications of mathematical modeling", "essential for mathematicians in developing trusted proofs" and so forth, and conclude with an exhortation to the academic and commercial modeling community to take up our ideas forthwith.

We have attempted a somewhat different approach here, by identifying, albeit informally, the practice and concerns of the modeling community and how formal methods techniques might address them.

The identification of "practice" in a discipline involves finding out what people actually do, rather than what they say they do, or what others think they should do. Thus for example in [25] we showed that practice in pure mathematics research does not, as an outsider might suppose, consist in rigorous formal development but rather in the development of "good enough" proofs: this explains why computational logic engines are hardly used by pure mathematicians.

For sociologists such as Latour [22] identifying practice involves detailed observations over many months in laboratories, and careful enquiry as to whether there is any such thing as a universal or context-independent notion of scientific method, rather than "particular courses of action with materials to hand" [24].

For the purposes of this paper we gained an overview from textbooks, university courses, meet-

ings, seminars, newsgroups, bug-reports and discussions with reflective practitioners, who included both developers and users of such software[1]. I am not aware of any thorough study into correctness concerns for modeling and what causes errors, although Mackenzie has touched on such matters in his sociological account of the development of nuclear weaponry [24]. Certainly the matter has not received the attention given to safety-critical systems. This paper can only be regarded as a pilot investigation: I conclude that, while certain individual incidents have been noted and studied, in general correctness is taken for granted, and where it is discussed it is the correspondence of models to reality, rather than the correctness of the underlying mathematics or software, that causes concern.

3 What is a model?

What is a model? A mathematical representation of reality? What is reality? What is a mathematical representation of it? Is it "out there" or "purely formal", or constructed in the minds of scientists with all kinds of motives and purposes, including the quest for truth (whatever that might be)? Questions of this kind have occupied philosophers of science for centuries. For this paper we adopt a work-a-day definition based on the standard student text of Mooney and Swift [28]: a mathematical model is a purposeful representation of reality using the tools and substance of mathematics, including computation.

A classic example is the predator-prey model whose purpose is to understand the long-term behavior of populations of predators (for example lynx) and prey (for example hares) which manifest cyclical behavior: as lynx numbers x rise more hares are eaten, so hare numbers y drop, so lynx numbers drop, so more hares survive, so lynx have more to eat, so lynx numbers increase, and so on. This is modeled by two differential equations, where $\alpha, \beta, \gamma, \delta$ represent parameters which will vary for different populations.

$$\begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{pmatrix} = \begin{pmatrix} \alpha & -\beta x \\ \gamma y & -\delta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \qquad (1)$$

We call this Model PP1. From these equations we may prove that $y^\alpha e^{-\beta y} x^\gamma e^{-\delta y} = K$ and hence deduce that in the model x and y do indeed manifest cyclical behavior over time for certain values of the parameters. Even without this analytic formula we can implement the equations numerically and hence draw graphs of x, y and t to display the cyclical behavior.

A simple account of modeling considers "the real world" (including hare and lynx), a "purpose" (understanding population change in hare and lynx), the "model world" consisting of assumptions we have made or chosen about the real world (for example that lynx die when there are no hares to eat), the mathematical "model" we have built of our model world using dynamical systems,[2] and the "implementation" of that model in a computer system. From the "model" or its "implementation" we can draw conclusions about the "model world" which we can then assess against the hypotheses of the model or against experimental or other understanding of "the real world". We may view this as a pipeline: $\{reality+purposes\} \to model\ world \to model \to implementation$.

The predator-prey model PP1 above is an abstraction, whose purpose is to investigate the apparent cyclical nature of such populations. It tells us that if the hypotheses in the model world about the behavior of hare and lynx are satisfied, and if $\alpha, \beta, \gamma, \delta$ take certain values, then certain consequences ensue in the model, and hence by implication in the "model world". We may then use domain knowledge to give an interpretation of our conclusions for "the real world".

If we wanted to study a particular population of hares and lynx this model would not be of much use. We would need a different "model world" and a more complicated "model", which we denote by PP2. We would take other phenomena into account, for example what hares eat, and consider data, either real or simulated, on weather patterns or grass growth for our particular population. We would probably no longer have an analytical solution, and would have to rely on an "implementation" to obtain numerical, graphical or visual estimates for long term behavior. These estimates would still be contingent upon our assumptions, and the nature and quality of the data we used. PP2 might not manifest cyclical behavior at all: it might not include the equations of PP1. The mathematical relationship between our two models might be complex: it would be unlikely that, in formal method terms, one was a simple refinement of the other for instance. The distinction between these two kinds of model, roughly speaking the first more concerned with abstract principles or putative laws of nature,

---

[2] For the purposes of this paper we ignore stochastic and discrete aspects

the second with simulations and predictions of phenomena, has sometimes been drawn by calling the former "models" and the latter "simulations". However there is no hard and fast distinction.

Both PP1 and PP2 are, in modeling terms, fairly small and straightforward, in contrast to global models of climate or population, refined over many years with complex data sets.

Once we have a model, or several models, we may investigate their solution and other properties, either mathematically or through an implementation. Models are assessed and evaluated against their purposes, or against other models that address the same or related purposes. Of particular concern is the definition and assessment of correctness.

4 Mathematical techniques

The theory

In this section we give a summary of some of the theory of differential equations and dynamical systems from the point of view of mathematical modeling applications.

What do we mean by a differential equation, and a solution? At an elementary level in a modeling text such as Mooney and Swift [28] the notion is often given only by example: for instance suppose we wish to model the motion of a particle in terms of the time and distance from an initial point (y) and the acceleration ($y'' = d^2y/dt^2$). The equation

$$y''(t) + y(t) = 0 \qquad (2)$$

describes the motion at time t, any solution has the form $\phi(t) = A\sin(t) + B\cos(t)$ where A and B are arbitrary constants, and a solution satisfying the initial conditions $y(0) = 1, y'(0) = 2$ is given by $\phi(t) = 2\sin(t) + \cos(t)$. A solution satisfying the initial conditions can be evaluated at any value of t, so that for our solution ϕ at time $t = \pi/2$ the position will be given by $\phi(\pi/2) = 2$. This equation has an explicit mathematical solution (we call this an analytic solution), but for many equations we may know only of the existence of such solutions, and numerical solutions at particular points (subject to the accuracy constraints of numerical analysis) may be all that are available to us.

"Solving" an equation involving an unknown function y and its derivatives, and conditions on the value of y at certain points, involves finding a particular (some possible such y) or a general (all possible such y) analytic solution in terms of known functions. In texts at the level of [28] various standard

"cook-book" techniques are given, accompanied by reassurance and motivation for the reader. There is also particular stress on determining the qualitative or limiting behavior of the solution: does it decay over time for example.

Thus for example [28] contains the following recipe for solving first order linear differential equations of the form

$$\frac{dy}{dx} + a(x)y = b(x) : \qquad (3)$$

the general solution is (sic, including sloppy variable naming)

$$y(x) = \frac{1}{\mu(x)}(\int \mu(x)b(x)dx + C) \qquad (4)$$

where $\mu(x) = \exp(\int a(x)dx)$. This description elides many issues concerned with exactly when functions are defined or differentiable, or solutions exist. The standard approach of an undergraduate course in differential equations makes matters more precise: *Suppose that a and b are continuous functions on an interval I. Let $A(x)$ be a function such that $dA/dx = a(x)$. If C is any constant then the function ϕ given by*

$$\phi(x) = \exp(-A(x))(\int_{x_0}^{x} \exp(A(t))b(t)dt + C) \quad (5)$$

where x_0 is in I, is a solution of (3), and every solution has this form.

The standard treatment continues by considering existence proofs for solutions. A particularly important class is that of linear systems, of the form

$$L(y) = y^{(n)} + a_1(x)y^{(n-1)} + \ldots + a_n(x)y = b(y), \ (6)$$

where under suitable conditions solutions always exist, though they may not have a simple closed form representation.

In the case when all the a_i are constant the solutions to $L(y) = 0$ are found by computing the eigenvalues, or roots of the characteristic equation

$$\lambda^n + a_1\lambda^{n-1} + \ldots + a_n = 0. \qquad (7)$$

Thus for example when $n = 2$ the equation

$$L(y) = y'' + 2by' + cy = 0 \qquad (8)$$

has general solution given by

$$\phi(x) =$$
$$\begin{cases} \exp(-bx)(A + Bx), & \gamma = 0 \\ \exp(-bx)(A\exp(\sqrt{\gamma}x) + B\exp(-\sqrt{\gamma}x)), & \gamma > 0 \\ \exp(-bx)(A\cos(x\sqrt{-\gamma}) + B\sin(x\sqrt{-\gamma})), & \gamma < 0 \end{cases}$$
$$(9)$$

where $\gamma = b^2 - c$. This description of the solution may be further refined to include its qualitative behavior: for example in case $\gamma = 0$, the system oscillates, and if $b > 0$ it tends to zero (is damped), if $b < 0$, it tends to infinity and if $b = 0$ it is stable.

Current mathematical research emphasizes dynamical systems, that is, roughly speaking, solution spaces of systems of differential equations like PP1. Linear systems in n variables can be expressed as a vector equation $\mathbf{X}' = \mathbf{A}\mathbf{X}$, where $\mathbf{A}$ is an $n \times n$ matrix, and the solutions are given in terms of eigenvalues of $\mathbf{A}$. This again allows us to predict the limiting behavior of such a system, and to identify fixed points (equilibrium points) where $\mathbf{X}' = \mathbf{0}$, and behavior near to them: for example does a point near the equilibrium point move towards it (a sink) or away from it (a source). In two dimensions an analysis like (9), called a phase plane analysis, is possible: in dimensions above two chaotic phenomena can occur.

For non-linear systems like the predator-prey model there are extensive theories of existence and uniqueness of solutions. An important practical technique for investigating qualitative behavior near a fixed point is that of taking a linear approximation there and using this to do a phase plane analysis. The full mathematical analysis of such behavior, and of the underlying dynamical systems, possible chaotic behavior and so forth, requires the full apparatus of modern differential geometry.

Applications

In the initial stages the modeler may want to manipulate and transform the model and get a few rough assessments of its behavior. The next stage would be a more detailed investigation, to compare it with alternatives, to calibrate it against data, theory or other models, and to assess its performance. At a more mature stage models may be used for prediction or for reference points against other models, as components in larger systems, or refined as new data or theoretical understanding becomes available.

For example Hammersley's [12] maxims for manipulators at an early stage include: "clean up the notation, choose suitable units, reduce the number of variables, and avoid rigor like the plague as it only leads to rigour mortis", to which one would probably add today "visualize the solution".

A typical more detailed investigation might include:

- *solving a system of differential equations sub-*

ject to initial values or boundary conditions: either analytically or numerically

- *reachability analysis:* determining if there is an analytic or numeric solution satisfying a set of constraints, typically that it starts in one region and passes through another. Thus in example (2) the point $(\pi/2, 2)$ is reachable from $(0, 1)$, but $(r, 3)$ is unreachable for any value of r

- *identification of behavior near a stationary point:* for example by a phase plane analysis

- *limiting behavior over time:* for example by an eigenvalue analysis generalizing (9)

- *perturbation analysis:* to identify behaviors of the model under local variations

- *behavior as some parameter varies:* for example changes in the phase plane as a coefficient varies

Taking a formal methods perspective one might expect to see more general reasoning about properties of the solution, for example using temporal logic. Recent work in the hybrid systems community addresses this for control systems using tools such as HyTech [17], and Dutertre [7] gives examples of reasoning about upper bounds in the requirements of an avionics application, but such work does not seem to be considered at all mainstream in the modeling community. For example searches in Citeseer [3] turn up little of relevance.

Correctness issues

In analyzing correctness issues for modeling we first turn to the correctness of the underlying mathematics.

We note first that applications of modeling are not in practice a particularly rich source of novel mathematics. There is in general [28] little enthusiasm for spending a long time developing new equations for a particular modeling problem. Standard techniques, like linearisation or power-series approximation, for replacing one equation with another that behaves in roughly the same way, may be sufficient when experimenting with a number of models at an early stage. The community tends to work with a smaller number of systems which are reasonably well understood or mathematically well-behaved and which experience or consensus deems sufficient for the domain at hand.

The researcher in dynamical systems, the applied mathematician or numerical analyst 'toolsmith' and

the modeler applying those techniques are doing different things. The researcher is concerned with general theories about the existence of solutions or the behavior of families of systems. The toolsmith is developing effective techniques for solving problems like those above, with the researcher's work to assure correctness. Modelers usually want to take the underlying mathematics for granted, concentrating instead on the modeling issues that arise: their mathematical interest or understanding is perhaps unlikely to go beyond a work-a-day account at the level of [28]. In particular the researcher is doing proofs in the underlying theories, the toolsmith is doing proofs about hand or machine computation techniques, and the modeler is applying those computation techniques.

We have discussed at length elsewhere [25] attitudes to correctness in the mathematical community: we identified current mathematical practice with producing conjectural mathematical knowledge by means of speculation, heuristic arguments, examples and experiments, which may then be confirmed as theorems by producing proofs in accordance with a community standard of rigour, which may be read by the community in a variety of ways. Most of the mathematics used in applications of modeling is not particularly novel, and has been subject to the usual mechanisms of community inspection through courses and text books over many years: there does not seem to be much concern from the mathematician, the toolsmith or the modeler over its correctness. As is usual in contemporary mathematical culture few are much concerned with formal proof or matters of foundation.

When a new technique arises, for example the recent growth of interest in level set methods [35], the focus of the discussion is generally on new applications, or on faster or better (for example with less instability near cusps) performance in old ones, rather than on extended discussions of correctness.

5 Computational techniques

Numerical methods

The standard, and almost universal, approach to computation for modeling, is numerical methods, which have been part of applied mathematics and the physical sciences for almost fifty years. They are widely available through standard commercial libraries such as NAG [29] and MatLab [27], and provide the basis for large software systems, usually written in FORTRAN or C and used in chemical,

physical or astronomical research as well as in practical fields like engineering, meteorology and aeronautics and increasingly today in visualization and animation. Purpose-built implementations, for example, for biosciences, environmental modeling or geology are built on top of general purpose tools such as Simulink [36] which provides a graphical interface to MatLab. For example Simulink may easily be used to run the predator-prey model for different values of the parameters, generating numeric or graphical output, from which various properties of the system may be inferred.

In addition such systems can readily accommodate other inputs, for example from sensors or measuring devices, or other numerical procedures, such as curve fitting. For many problems, for example the investigation of chaotic phenomena, there are no alternative standard techniques.

From the modelers point of view the main advantage of numerical systems is that they will always give an answer, and despite the negative evidence we cite below, with sufficient user expertise are accepted as doing so sufficiently quickly and accurately, with established protocols for testing and error analysis. Numerical methods and software like NAG or Simulink are so standard and so widely used that it is hard to see them being displaced by other techniques. However the output, and properties derived from it, will always be numeric and not analytic, and support for investigating properties of the solution or parameters may be limited.

Numerical methods: correctness issues

The user of such systems can use default settings and work in ignorance of the underlying numerics, or take more detailed control using standard techniques of numerical analysis [18] to ensure results of required accuracy. Indeed, faster and more accurate numerical methods have been the main research thrust in numerical analysis over the past forty years.

A particular issue in numerical work is correctness of floating point implementation (for example the famous Pentium bug): the consistent handling of floating point arithmetic or the translation between machines with different word-lengths are recurring legacy issues. Another is convergence criteria: is the implementation robust enough to produce the same answer again for the same inputs. Kahan [21] maintains a web-site of known problems.

Yet problems persist and even expert users may be unaware of them. The author was told of a

complex bug in the British Met office implementation of the multi-grid finite element method that was worth about 2% accuracy in weather forecasts. Hatton [15] reports on observations of nine independently developed large programs for seismic data processing, and shows that although the programs used the same data and were developed to the same specifications in the same language (FORTRAN), numerical disagreement grows at a rate of 1% in average absolute difference per 4000 lines of implemented code. The programs were used to analyze large scientific datasets where typically results expect around 0.001% accuracy. He concluded that in general problems were caused not by compiler or hardware errors, but by software faults, often off-by-one errors. However the matter has not received much recognition in the modeling community [16].

Symbolic computation

Symbolic computation techniques, such as those embodied in Maple or Mathematica, appear to offer a wide range of additional facilities to the modeler, especially when combined with numerical methods. Thus the *dsolve* command in Maple, or the *DSolve* command in Mathematica, can solve a wide variety of differential equations analytically, and the user can further interact with the system or write their own code, to investigate their properties. As the account of the mathematics above demonstrates, implementations rely on other symbolic computation techniques, such as integration, polynomial solving and computing eigenvalues and eigenvectors.

There is continuing lively debate over the respective merits of symbolic and numeric computation, and active research on the best way to combine the two approaches. The main drawback from the user's point of view is that computer algebra systems are simply unable to solve many of the problems listed above, either because of unsolvability or intractability. Even if there are symbolic solution techniques such systems do not scale, and there are not in general well-developed techniques for combining numeric input or techniques with symbolic ones: hence they lack the flexibility of numerical systems.

Thus for example while symbolic techniques for reachability analysis using quantifier elimination have been investigated [20], they are in general double exponential, and intractable in all but the smallest examples.

There are a few cases where symbolic techniques are better developed than numeric ones, for example the use of model checking in systems like Hytech to reason about hybrid systems, discrete combinations of control systems. There are also a few applications where symbolic systems are used in preference to numerical systems, for example in robotic or satellite motion planning.

Symbolic computation: correctness issues

By contrast with numerical techniques, users often find symbolic computation or computer algebra systems (CAS) like Maple frustrating and hard to use: see Wester [37] for a survey. Even in situations where the user is expecting them to work they may fail to produce an "obvious" answer, or produce unexpected or wrong answers, and their performance can be very unpredictable, varying widely on apparently similar inputs.

One cause of error is failure to check side-conditions: this is not so much an error as a design decision for ease of use, since even small procedures may produce large numbers of side conditions, often intractable or undecidable. This illustrates a more general design issue: there are many examples of processes (for example definite symbolic integration via the Fundamental Theorem of Calculus) where a CAS may be able to compute an answer, sometimes correct, on a large class of inputs, be provably sound on only a subclass of those inputs (where the function is continuous) and be able to check soundness easily on a smaller subclass still (for example, since continuity is undecidable, systems use a simpler check for functions with no potential poles or discontinuities). Some CAS are cautious, only giving an answer when pre-conditions are satisfied: however this means they may fail on quite simple queries. Others try and propagate the side conditions to inform the user, though this can rapidly lead to voluminous output. Mathematica and Maple generally attempt to return an answer whenever they can and leave to the user the burden of checking correctness. In [1] we have analyzed this in some detail for symbolic integration, and proposed a solution based on verified look-up tables. We extended our ideas to dynamical systems and mathematical modeling in [26], with a suite of PVS tools to check definedness and continuity, callable from Maple.

However there is a deeper reason for apparent unsoundness than failure to check for side-conditions. Formally CAS compute indefinite integrals and solve differential equations within the algebraic framework of the theory of differential fields [2]: fields with an operator satisfying $d(f.g) =$

$(df).g + f.(dg)$. When using an indefinite integral as part of an analytic calculation, for example solving a differential equation, the answers obtained algebraically may differ significantly from what is expected. For example, viewed as an element of a differential field, the derivative of $f(x) = \tan^{-1}(x) + \tan^{-1}(1/x)$ is zero, and it follows that $f(x)$ is a constant. Viewed analytically it is a step function with the value $-\pi/2$ for $x < 0$ and $\pi/2$ for $x > 0$. Thus an "unexpected" answer to a query involving $f(x)$ may be correct within the theory of differential fields, but incorrect in the usual analytic framework for differential equations we have presented above. Similarly it is easy to get Maple's *dsolve* command to display behavior which is unsound analytically, as it applies (4) without checking continuity of a and b.

This analysis should be kept in perspective however: developers of the symbolic software systems GAP [34], **axiom**[19] and Aldor [19] indicate that the majority of bug reports tend to uncover user misunderstanding, performance, or systems flaws, especially to do with portability, rather than problems with the underlying mathematics or algorithms. For example of approximately 1100 bug reports on Aldor only one reported a problem with an incorrect library implementation, involving a failure to detect a division by zero.

6   Correctness concerns for modeling

We now return to correctness concerns for the modeling process, and consider the pipeline, $\{reality+ \ purposes\} \rightarrow model \ world \rightarrow model \rightarrow implementation$.

One may first ask whether the "implementation" is a correct implementation of the underlying "model". In particular we may ask which aspects of its behavior are artifacts of the "implementation" (for example a poor choice of random number generator) rather than consequences of the model, or what hidden or explicit assumptions about the model have been made and how they affect the uses to which the system has been put. For example, if the system is used in a new application and predicts that $x > 3$, is this a consequence of the model, or of some implementation decision being called upon outside its domain of validity.

Heterogeneous distributed implementations often incorporate large legacy systems where the underlying assumptions may have varied over time, where later implementors may not have fully understood the original assumptions, or have incorpo-

rated variations based on new results, or where the underlying models may be incompatible. Thus for example an implementor may have hard-wired an implicit assumption about, say, the life span of a predator which is totally inaccessible to later users, and may lead to nonsensical results when combined with a different implementation.

The correctness of an implementation concerns how the "implementation" of a model matches the "model": of much greater concern in the modeling community is the assessment of the "model" against its "purposes", or against other models with the same or related purposes. In such discussions the "model" and its "implementation" may often be identified, particularly if we only have numerical information about the model. An excellent account from the point of view of environmental predictions is given in Oreskes [31].

The correctness of a "model" is in any case contingent: it says that under the hypotheses of the "model world" certain consequences occur, and the output of the implementation may be regarded as a prediction, with estimates of error being provided by mathematical analysis in the light of the model and the reliability of the data. The hypotheses of the model world may not necessarily be very clear or explicit, being part of the assumed background knowledge of domain experts. Care needs to be taken with data: for example a famous data-set on Canadian hare and lynx populations was discredited [11] when it was pointed out that the lynx and hares lived far apart and had little opportunity to eat each other. Our ability to test the correspondence of the "model world" with the "real world" depends in part on our understanding of the phenomena, and in part on the availability of sufficiently accurate data. So questions of correctness of a particular model are complicated and often subject to heated debate or compromise.

In some cases predictions may be easy to check: the occurrence of the full moon for example is readily observed and not subject to major disagreement. So if a model with a trustworthy implementation whose purpose is to predict the full moon fails to do so we may reasonably assume the "model", or the "model world" is incorrect. Even then it may not be at all clear which assumption or equation has led to the error. However most models cannot be checked in so straightforward a way: for example the average temperature of the earth needed in models of global warming is hard to measure or estimate, and in other cases it may be infeasible to check the predictions: for example safety thresholds for aircraft

loads or discharge of pollutants.

Models may be known by insiders to be inaccurate, but none-the-less used as a best guess, or treated as accurate even though they are not. Mackenzie [24] reports on the debate surrounding the abandonment of nuclear weapons testing, drawing attention to the importance of tacit knowledge in the practical development of nuclear weapons, and the possibility that they might be "uninvented" if this tacit knowledge is lost. He reports scientists' claims that a computer prediction is "pretty good" if the actual yield is within 25% of prediction, and notes that during the moratorium on nuclear testing in the 1950s dependence on and confidence in computer programs increased: according to an interviewee "people start to believe the codes are actually true, to lose touch with reality.".

Experts may disagree as to the acceptability of the model: Shrader-Frechette [39] reports disagreement among two expert committees in the 1993 assessment of the proposed Yucca Mountain Waste repository site as to whether the large and well-established geological models used could reliably predict volcanic activity. We may have several competing models: for example Gilpin and Alaya [9] used experiments on competing populations of fruit-flies to test different variants of the predator-prey "model" and "model world" against the purpose of accurate prediction of fruit-fly populations. They compared their models against the accuracy of their results, favoring those where the model world made most sense biologically, and those where the model was simple and general[3]. However it may not be the case that we can always chose among competing models so readily.

Matters become more complex when we consider many-layered models, where for example testing against "the real world" may mean in practice testing against another "implementation" of a different "model" that has acquired the standing of "the real world" for practical purposes. In assessing model PP2 for example we would need numbers of hares and lynx: would we do every count by hand or use "implementations" of established "models" of wild-life numbers calibrated with key data from field studies. And how might the assumptions of the latter affect the predictions of PP2?

As we have indicated a particular concern is the combining of different models or implementations.

---

[3] A much argued philosophical point. It has been suggested [31] that the quest for simplicity and generality, identified with Ockham's Razor, owes more to seventeenth century theology and mathematical convenience than any evidence that simple models are better predictors than complex ones.

Different models may address different parts of our purposes differently, or in choosing to model part of a larger scheme we may have to choose between several models none of which are entirely satisfactory. Assumptions may be incompatible or unclear: this is a particular issue for legacy components where assumptions may be concealed, contradictory, or have changed over time.

7 How can formal methods contribute?

Putting together the ingredients described above we may identify the business of modeling with first developing generic mathematical theories, algorithms, and implementations, both numeric and symbolic, and then modeling particular systems by implementing them within the chosen framework as part of the modeling pipeline. Correctness concerns may be raised at all levels of the process: the mathematics, the software systems, the implementations of the model and the correspondence of the model with reality. As far as we can tell this last is of most concern to the modeling community.

Formal methods, broadly construed, offers a variety of approaches.

Mathematical theories

Since the pioneering work of de Bruijn's AUTOMATH [4], developed in 1967, the theories of analysis which underlie differential equations and mathematical modeling have been developed inside various theorem provers: for recent manifestations see Dutertre's implementation of the reals inside PVS [7] or Harrison's development as far as integration in HOL [14]. As far as we are aware a full machine verification of the mathematics outlined in the previous sections has not yet been done, but it is perfectly feasible in a number of systems, using classical or constructive techniques. However while this is possible, it is hard to see how it would serve the needs of the modeling community, who regard the soundness of the underlying math as a "solved problem", established over many years in text-books and courses. They rely on mathematicians, and the usual community mechanisms of mathematics, which are remarkably averse to rigour [25], to establish correctness of the necessary mathematics: I have identified little interest in human or machine formal proof for the classical mathematics underlying the subject, the work of the toolsmith, or its routine application in modeling. This is not to write

off machine checked mathematics as an endeavor, merely to say that this community sees little point to it. While logicians [8] have considered alternative axiomatizations for differential analysis I have identified no interest among the mainstream mathematical or modeling community in these matters.

Once such a development had been done it would, in principle, be possible to investigate our models directly within the prover, recasting the various queries outlined in Section 4 as proof requirements, for example the reachability results of example (2). However it is hard to see how such systems would overcome the difficulties we have already discussed for symbolic computation systems: infeasibility or intractibility mean that often there will not be an automatic proof procedure, and users will need to produce a manual proof of something whose numerical equivalent could be produced automatically. In addition any such system would need a computational component if it was to match the exploratory capacity of existing techniques, and as Section 4 shows many of the computations or proofs would require advanced symbolic computation facilities, for example to calculate eigenvalues.

Against this however we should set the advantages of abstraction, higher level proof and the handling of parameters: for example it takes laborious numerical simulation to investigate changes in the phase plane as a coefficient varies, whereas a symbolic approach merely produces a proof obligation to be discharged.

In addition, as we have argued elsewhere [25] specialized decision procedures may prove useful for some queries, for example quantifier elimination for reachability [20].

Computational techniques

As we have seen general software engineering issues have been identified as a major source of problems in both numerical and symbolic software: since these problems and formal methods approaches to them are not peculiar to modeling we do not discuss them further here. The modeling community relies on the usual mechanisms of software development, which are averse to rigour, to establish trustworthiness of its computer systems: I have identified little interest among commercial vendors in classical formal methods techniques.

It is in principle possible to implement numerical or symbolic computation inside a theorem prover, gaining reliability at a cost in performance, and both approaches have received much attention in recent years. The notorious "Pentium bug" drew attention to the unreliability of floating point implementations, and inspired Harrison's development of floating point arithmetic in HOL [13] which has had considerable commercial impact in the verification of hardware.

Such implementations of computer algebra systems have proved harder, partly because, as we have indicated, they require implementation inside a prover of specialized algorithms such as factorization. In any case, some of the unexpected behaviors of computer algebra systems arise from the algebraic representation of analysis: these would not be solved by re-implementation inside a prover. We report elsewhere [26] on an alternative approach: we built a toolkit in the PVS [32] theorem prover which automatically checks pre and side conditions such as continuity to computer algebra algorithms such as Maple's *dsolve*, thus addressing some of the difficulties caused by unsoundness in using computer algebra systems for analytic work at little extra cost to the user.

Modeling

As we have indicated the main concerns of the modeling community are with the correctness, validation and confirmation of models.

We report elsewhere [5, 6] on our lightweight formal methods approach: we built a verification condition generator in Aldor, an internal language used in developing the computer algebra systems **axiom** and Maple, and are currently developing this work in collaboration with NAG Ltd. The verification conditions are generated at compile time from user annotations, typically recording pre- and postconditions, and can be passed to a theorem prover or used for information or documentation.

Our original motivation was particularly that of assisting the user of libraries where the code itself might be trusted, but the assumptions or preconditions for its correct use were ill-documented. We are currently experimenting with the use of these annotations for documenting requirements and assumptions in legacy models.

However there appear to be some differences between the needs here and those of design or requirements engineering: in particular there are cases where it seems useful to record assumptions or domain knowledge that does not affect the state or output of the module where it is recorded or assumed, but may be significant elsewhere. It is not entirely obvious to us how to map the mod-

eling pipeline to frameworks such as the reference model of Gunter et al [10], which is based on domain knowledge, requirements, specifications, program and program platform.

We note that these matters are beginning to receive commercial attention: Lemma 1 Ltd [23] report on their ClawZ system which translates Simulink diagrams into Z specifications, and the UK company QSS [33] have interfaced their DOORS requirements tool to Simulink.

8   Some new directions

The previous section paints a somewhat depressing picture, suggesting that many areas which have received considerable research attention are unlikely to have much effect on the practice of modeling. We might sum up by observing that the modeling community are users rather than creators of mathematics and software, and by and large take both the mathematical theories underlying their work and the largely commercial computer systems that implement them pretty much for granted as "solved problems". The main concerns lie elsewhere and there is little interest in or motivation for change, and a heavy personal and financial investment in existing technologies.

We can none-the-less outline some ways ahead. The modeling community, like many others, are interested in new methods that fit their present world view, address their main concerns or improve or extend existing techniques or software.

The correctness, validation and confirmation of models is of primary importance to the modeling community, and of particular concern when these impact public policy in matters such as nuclear waste disposal. It is the assumptions, data and choice of model that seem to matter here, not questions about correctness of the underlying mathematics or software once the model has been chosen. We are not even aware of a suitable framework for the analysis of requirements, specifications, assumptions and proof obligations for modeling within our pipeline: an extension of the reference model of Gunter et al [10] may be appropriate. Computational logic has a useful role to play in monitoring and analysis here, and hence in reasoning directly about the assumptions of the "model world", the "model" and the "implementation".

Heterogeneous distributed models are of particular current interest, put together for example across the Internet, with disparate or legacy components where assumptions may be concealed, con-

tradictory, or have changed over time. An engine for managing requirements and assumptions would be a key component technology of robust architectures for linking such heterogeneous models. Particular care would need to be taken over the layering issues indicated above. Projects such as Open Math [30], which attempt to provide reliable interface mechanisms for heterogeneous mathematical systems using type inference seem relevant here also.

New analytic, numerical or visualization techniques which leverage off the established mathematical and computational framework and extend its functionality are of interest. For example, as we have seen, computer algebra systems are useful in the analytic study of dynamical systems, especially those with parameters, but these are error prone: extending them using computational logic engines as we have indicated above adds functionality at little cost to the user.

Symbolic analysis Numerical analysis software does continuous mathematics numerically, computer algebra software does continuous mathematics symbolically by algebraic means, but no software yet does continuous mathematics symbolically by analytic means, and it is not clear how it should be done. As we have indicated this is the underlying reason for the deficiencies of computer algebra systems: solving it would indeed make possible a new generation of useful computational tools. It is not enough to formalize existing computer algebra systems based on differential rings: these will not give us true computational analysis. It is not enough to prove theorems about real analysis inside a theorem prover: we need to be able to do computations like those described in Section 3 as well.

We urge the formal methods and computational logic community to take up these challenges.

Acknowledgements

References

[1] A Adams, H Gottliebsen, S Linton, U Martin. VSDITLU: a verified symbolic definite integral table look-up Proc CADE 16, LNAI 1632, 112-126, Springer 1999

[2] M. Bronstein. *Symbolic integration. I.* Springer, 1997.

[3] Research Index, http://citeseer.nj.nec.com/cs

[4] N de Bruijn, The mathematical Language AUTOMATH, its usage, and some of its extensions, Symposium on Automatic Demonstration, Lecture Notes in Mathematics 125, Springer 1968

[5] Martin Dunstan, Tom Kelsey, Steve Linton and Ursula Martin Lightweight formal methods for computer algebra systems In ISSAC'98, ACM Press, 1998

[6] Martin Dunstan, Tom Kelsey, Steve Linton and Ursula Martin Formal Methods for Extensions to CAS Proc FM'99, LNCS 1709, 1758-1777, Springer 1999

[7] B. Dutertre. Elements of Mathematical Analysis in PVS. Proc TPHOLS 9, LNCS 1125, Springer 1996.

[8] S Feferman, Why a little bit goes a long way: Logical foundations of scientifically applicable mathematics, in PSA 1992, Vol. II, 442-455, 1993.

[9] M E Gilpin and F J Ayala, Global models of growth and competition, Proc Nat Acad Sci USA 70, 3590-3593, 1973

[10] Carl A Gunter et al, A reference model for requirements and specifications, to appear IEEE Software.

[11] C A S Hall, Assessment of theoretical models, Ecological Modeling 43, 5-31, 1988

[12] J Hammersley, Maxims for manipulators, Bull I M A 9 (1973) 276.

[13] J Harrison, Floating point verification in HOL. In Proc TPHOLS 8, LNCS 971, 186-199, Springer 1995

[14] J Harrison, Constructing the Real Numbers in HOL, Formal Methods in System Design 5 (1994) 35-59

[15] Leslie Hatton, The T-experiments: errors in scientific software, IEE Computational Science and Engineering, 4, 27-38, 1997

[16] Leslie Hatton, personal communication.

[17] Thomas Henzinger and Pei-Hsin Ho, HyTech: The Cornell Hybrid Technology Tool, Hybrid Systems II, LNCS 999, 265-294, Springer, 1995.

[18] N Higham, Accuracy and Stability of Numerical Algorithms, SIAM Press 1996

[19] R D Jenks and R S Sutor, **axiom**: The Scientific Computation System, Springer 1992

[20] M Jirstrand, Nonlinear Control System Design by Quantifier Elimination, Journal of Symbolic Computation, 24, 137-152, 1997.

[21] W Kahan, http://www.cs.berkeley.edu/ kahan

[22] B Latour and S Woolgar, Laboratory Life: the Social Construction of Scientific Facts, Sage, London, 1979

[23] ClawZ: Lemma 1 Ltd, http://www.lemma-one.com

[24] Donald Mackenzie, The uninvention of nuclear weapons, Chapter 10 of Knowing Machines, MIT Press 1998.

[25] U Martin Computers, reasoning and mathematical practice In Computational Logic, NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci., Springer 1998

[26] U Martin and H Gottliebsen, Computational logic support for differential equations and mathematical modeling, submitted.

[27] Matlab, http://www.matlab.com

[28] D Mooney and R Swift, A course in mathematical modeling, MAA press, 1999

[29] NAG Libraries, http://www.nag.co.uk

[30] OpenMath, http://www.openmath.org

[31] N Oreskes et al, Verification, validation and confirmation of numerical models in the earth sciences, Science 263, 1994, 641–646

[32] S Owre, S Rajah, J Rushby, N Shankar. PVS: combining specification, proof checking, and model checking. In Proc CAV 8, LNCS 1102, 411-414 Springer, 1996

[33] DOORS, http://www.requirements.com

[34] Martin Schönert et al, GAP: groups, algorithms, and programming, `www-gap.st-and.ac.uk`

[35] J Sethian, Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science, Cambridge University Press, 1999

[36] Simulink, http://www.simulink.com

[37] M Wester, Computer Algebra Systems : A Practical Guide, Wiley 1999

[38] Westpoint Consortium, Interdisciplinary Lively Application Projects, MAA Press, 1997

[39] K Shrader-Frechette, Science Versus Educated Guessing: Risk Assessment, Nuclear Waste, and Public Policy, BioScience 46, 1996

Timing analysis by model checking

Dimitri Naydich and David Guaspari [*]

Odyssey Research Associates
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250
naydich@clarityconnect.com
davidg@oracorp.com

Abstract: The safety of modern avionics relies on high integrity software that can be *verified* to meet hard real-time requirements. The limits of verification technology therefore determine acceptable engineering practice. To simplify verification problems, safety-critical systems are commonly implemented under the severe constraints of a cyclic executive, which make design an expensive trial-and-error process highly intolerant of change. Important advances in analysis techniques, such as rate monotonic analysis (RMA), have provided a theoretical and practical basis for easing these onerous restrictions. But RMA and its kindred have two limitations: they apply only to verifying the requirement of schedulability (that tasks meet their deadlines) and they cannot be applied to many common programming paradigms.

We address both these limitations by applying *model checking*, a technique with successful industrial applications in hardware design. Model checking algorithms analyze finite state machines, either by explicit state enumeration or by symbolic manipulation. Since quantitative timing properties involve a potentially unbounded state variable (a clock), our first problem is to construct a finite approximation that is conservative for the properties being analyzed—if the approximation satisfies the properties of interest, so does the infinite model. To reduce the potential for state space explosion we must further optimize this finite model. Experiments with some simple optimizations have yielded a hundred-fold efficiency improvement over published techniques.

1 The safety of hard real-time software

Modern avionics relies fundamentally on high integrity software that meets hard real-time requirements such as schedulability the guaranty that all tasks meet their deadlines. It is common to implement a high integrity real-time system by means of a cyclic executive, in which programmers explicitly allocate the execution of processes or process fragments to portions of a master control loop. This technique has the strengths of requiring essentially no runtime support and of making schedulability analysis trivial. But the design of a cyclic executive is expensive and time-consuming, relies heavily on trial-and-error rather than systematic design principles, and is highly intolerant of change. Small modifications to individual processes may require complete redesign of the master control loop. In addition, this narrowing of the design space potentially constrains the introduction of automation technologies that could improve both safety and performance.

The alternative to a cyclic executive is some form of preemptive scheduling in which processes are scheduled dynamically. Preemptive scheduling immediately presents two problems: First, static analysis of program behavior becomes much more difficult. Second, the runtime support required to carry out dynamic scheduling must be efficient and must admit an implementation simple enough to satisfy the certification requirements for high integrity systems. Raven [32] is an example of such a runtime.

The best-known analysis technique for preemptive scheduling is Rate Monotonic Analysis (RMA) [19], which applies to a restricted but useful class of systems and reduces schedulability analysis to checking a set of simple algebraic inequalities. However, RMA does not provide information about properties other than schedulability and is not applicable to

---

many common programming paradigms: Figure 1 provides an example of such a program. Nor does RMA cover properties other than schedulability.

This paper describes an ongoing investigation of model *checking* as a supplement to RMA. Model checking comprises automated techniques that apply, in principle, to any system representable as a finite state machine. These techniques are of two general kinds: *explicit search* (clever strategies for visiting all possible states) and *symbolic model checking* (combining symbolic execution and automated reasoning). Both styles can be used to analyze properties other than schedulability and systems that do not meet the design restrictions imposed by RMA. Our work shows that model checking can be applied to some systems beyond the reach of current analytical techniques. The technical barrier to making these applications practical and routine is the possibility of state space explosion. We are investigating optimization techniques that generate efficient representations of the system to be analyzed.

1.1 Ravenscar and Raven

The general principles we employ are not tied to any particular implementation, though the details will necessarily depend on the programming language and runtime system being modeled. The Ravenscar Profile [8] defines a set of Ada tasking features rich enough to support (among other things) rate monotonic scheduling, but requiring a minimal runtime. Ravenscar is supported by the Raven runtime, developed at Aonix to meet the highest FAA certification standards for safety critical systems. The tasking subset we consider can be regarded as a generalization of Ravenscar, together with a technical requirement, which we call *frame synchronization*, that reduces nondeterminism by eliminating arbitrary task phasings. Thus, the analysis we propose can be directly applied to real systems.

1. The main features of the Ravenscar subset are as follows:
2. The number of tasks, and the base priority of each, is fixed and statically determined.
3. Scheduling is preemptive, using the priority ceiling protocol.
4. Tasks interact only through protected objects. No more than one task may ever

be queued on the entry of any protected call. (This limit on the size of the entry queues is a dynamic requirement that cannot in general be enforced by syntactic restrictions.)
5. Task behavior is deterministic.

Figure 1, based on an example from [16], illustrates a simple Ravenscar program to which RMA does not apply. Three sensors periodically sample flight data and send it via a bounded buffer to an analyzer that periodically reads the data from the buffer. The buffer is implemented as a protected object containing a protected entry for writing data and a protected procedure for reading it. A read from an empty buffer returns some conventional value. The buffer's *write* entry blocks the sensors from writing when the buffer is full. The protected *read* procedure blocks the analyzer from reading while the buffer is being written to. (We make the *read* operation a procedure rather than an entry because Ravenscar forbids protected objects with more than one entry. That is why *read* does not block on any empty buffer, but reads some conventional value.) RMA does not apply because each of the periodic sensor tasks contains a protected entry call, at which it can be blocked.

1.2 Model-checking real-time properties

Many existing models for real-time systems are based on timed automata [2] or, more generally, hybrid automata [1]. These models contain state variables that represent the values of real-time clocks. Notice that a direct model of time, by means of a variable containing the current value of the clock, leads to an infinite state space, since the clock may increase without bound. Some form of temporal abstraction is required. The abstraction used to analyze hybrid automata is to represent *regions*—sets of states—symbolically, via logical formulas. Symbolic manipulation of such formulas [20] is the heart of model checking tools such as [4].

In [10], Corbett presents a two-stage construction that models real-time Ada tasking programs (together with the supporting runtime) as hybrid automata. The first stage translates a program to a transition system representing the possible interleavings of the tasks' execution. The second stage captures the timing constraints of the program by transforming the transition
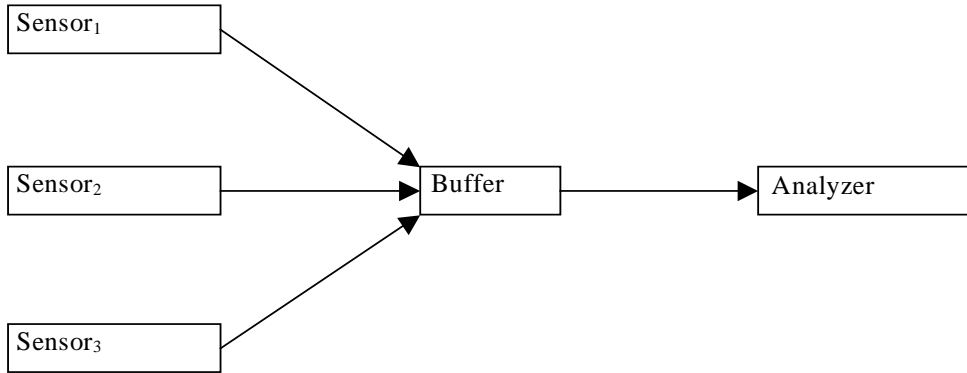
Figure 1: A Ravenscar example to which RMA does not apply

system into a hybrid automaton. This hybrid automaton is then analyzed by the HyTech verifier [11], which can be regarded as symbolic model checker.

In [16], we developed a method for constructing models of real-time tasking programs in Promela [12], a language for specifying communicating sequential processes. The program's tasks and the runtime system are represented as Promela processes. The frame synchronization requirement mentioned above allows us to eliminate the real-time clocks from the system's model altogether and thereby to represent the system as a simple transition system rather than a hybrid automaton. We introduce state variables to keep track of upper and lower bounds on the completion time of each process, and perform a "dynamic abstraction" of these time-related state variables to make the state space finite. In essence, the pair of completion times for each process defines the region of states in which the process is running. This representation is much simpler than representation by a logical formula. We then analyzed the Promela program with the Spin verifier [12].

Many other formal models have been proposed for concurrent real-time systems [3]. These include Petri nets [14], timed automata [2], timed process algebras [17], and real-time logics [13]. For the most part, these models are intended to represent specification, not implementation. In [5], general timed automata are extended to represent such implementation details as the assignment of tasks to processors, priorities, worst-case execution times of operations, and scheduling policies. Our model compares to [5] much as it does to [10].

2 A simple illustration

This section uses a trivial example to show how the "dynamic time abstraction" of [16] can be combined with reduction techniques from [10] and illustrate its effectiveness. Although there are enough differences that a quantitative comparison is not strictly scientific, we obtain a hundred-fold advantage over [10] in both speed and memory usage and a ten-fold advantage over [16].

2.1 A schedulability problem

Consider two periodic, non-interacting tasks, A and B, run on a single processor under preemptive scheduling. Task A has higher priority than B. Although this trivial tasking pattern can be analyzed by RMA, it allows us to illustrate essential features of our proposed strategy and to perform a simple comparison with Corbett's analysis via a hybrid automaton model.

A code skeleton is given in Figure 2. We assume that the variable *StartTime* records the value of the system clock at some moment after the tasks have been initialized but before they start running. In effect, this implements the frame synchronization assumption. *StartTime* can be initialized to satisfy the assumption by using a simple Ada coding idiom given in [16]. The code fragments *<statements1>* and *<statements2>* implement periodic activities whose functionality is irrelevant to the tasks' timing. Let *estimA* and *estimB* be upper bounds on the amount of CPU time necessary to execute the bodies of the loops in task *A* and task *B* respectively. We assume that CPU time is the tasks' only shared resource. The parameters

```
        task A is                              task B is
            pragma Priority(20);                   pragma Priority(10);
        end A;                                 end B;


        task body A is                         task body B is
            nextA: Time = StartTime;               nextB: Time = StartTime;
        begin                                  begin
            loop                                   loop
             <statements1>                          <statements2>
            nextA := nextA + periodA;              nextB := nextB + periodB;
            delay until nextA;                     delay until nextB;
            end loop;                              end loop;
        end A;                                 end B;
```

Figure 2 : A two-task problem

and *periodA* and *periodB* define the periods of task *A* and task *B*. Execution of "**delay until** *t*" blocks a task until the system clock has value *t*. If task *A* reaches its "**delay until** *nextA*" statement when the clock time is greater than *nextA,* then task A has missed a deadline. We can characterize a missed deadline for task *B* similarly.

With this definition of deadline, we analyze the schedulability of tasks *A* and *B* in terms of the task periods *periodA* and *periodB*, and the CPU time estimates *estimA* and *estimB*. As noted, RMA handles the problem easily, but the point of the example is to exhibit simple optimization strategies that can dramatically improve the efficiency of analysis by model checking.

2.2 A discrete model

In the program of Figure 2, the only variables affecting the timing behavior of the program are *nextA* and *nextB*. They are the only data variables represented in our model.

To model the program's control state, we completely abstract from the code fragments within the task loops. We represent the fragments as abstract actions whose executions take time, and whose executions can be preempted by higher priority actions. We model execution of tasks *A* and *B* as periodic invocations of these abstract actions.

In [16] we represented the runtime and each task as a separate process. As observed in [10], this simple-minded representation introduces unnecessary states because the actions of the

runtime are so tightly coupled to the actions of the tasks. That is, we know a strong *invariant* that permits a more efficient abstraction of the state space. Because task *A* has higher priority than task *B*, we can partition the system states as follows: task *A* can be either running or blocked by its "delay until" statement; task *B* can be running, or blocked by its "delay until" statement, or preempted by task *A*; and the system as a whole enters an error state if either task misses a deadline. Thus, we represent the status of the program by introducing a variable *runtime_status* that can have the following symbolic values: *runningA_preemptedB, blockedA_runningB, blockedA_blockedB, runningA_blockedB,* and *missed_deadline*.

We also introduce several variables to model timing information:

1. The integer variables *lb* and *ub* specify lower and upper bounds for the clock time at which the currently executing abstract action will (if not preempted) complete. The values of these time bounds vary dynamically, according to the program's control flow.

2. The integer variable *delta* contains an upper bound for the CPU time needed to complete the currently executing abstract action. When a preempted action resumes its execution the value of *delta* will typically be revised to reflect the progress made before preemption.

3. The integer variable *preemptB*, called the *preemption bound*, stores the value of *delta* when task *B* is preempted by A.

We specify the schedulability requirement by

asserting that the runtime status *missed_deadline* never occurs:

```
Invariant "hard deadline"
    ! runtime_status = missed_deadline
```

The states and transitions of our model are shown graphically in Figure 3. We define the effect of each transition using the notation of the Murphi model-checker [33]. The meaning of

$$guard ==> \text{ Begin <statements> End}$$

is that the transition may take place when the boolean *guard* is true; and, if it does take place, the effect on the state variables is defined by the Pascal-like code in *statements*. If several transitions may take place, then the choice of which transition to fire is non-deterministic. (Even if the Ada code is deterministic our model may be a conservative, non-deterministic, approximation.) The simple model shown here does not represent the overhead attributable to runtime actions such as preempting a task or restoring the state of a preempted task. Those costs are accounted for explicitly in [16].

Figure 4 provides definitions for three representative transitions: 1, 2, and 4. Transition rules 1 and 2 describe the program's behavior when *A* is running and *B* is preempted. Rule 4 describes one of the possible behaviors of the system when task *A* is blocked and task *B* is running—namely, the possibility that task *A* may preempt task *B*.

Rule 1: If the upper estimate of the clock time for completing task *A* is greater than or equal to the next deadline—that is, $ub \geq nextA+periodA$—then it is possible that *A may* miss its deadline; and therefore a deadline violation *will be* reported. Our model is a conservative approximation of the program. The program will satisfy any invariant satisfied by the model, but the converse need not be true.

Rule 2: If $ub < nextA+periodA$, this iteration of task *A* will meet its deadline. Transition 2 represents the successful completion of *A*, after which *A* becomes blocked until the beginning of its next period, and hands off to task *B* (as reflected by changing the value of *runtime_status* to *blockedA_runningB*). To do the necessary bookkeeping, the other state variables are modified as follows:

- *nextA*, the next clock time at which task *A* becomes ready to run, is incremented by the value of its period,
- *delta*, the estimate of the remaining CPU time to complete task *B*, is restored to the preemption bound of B,
- *ub*, which now represents an upper estimate of the clock time at which task *B* will complete, is increased by *delta*,
- since the preemption of B has now been accounted for, we reset *preemptB* to zero.

Rule 4: The guard for transition 4 represents the following possibility: task *B* will, if not preempted, meet its deadline; but task *A* becomes ready before the action of task *B* completes and therefore preempts *B*. Among the actions of rule 4, the interesting new feature is a call to procedure *time_wrap*, which is essential for making our model finite.

The state variables *nextA, nextB, lb,* and *ub* are regularly incremented. If we allowed them to increase without bound our model's state space would be infinite. However, the presence or absence of a deadline violation depends only on the *relative* values of these variables, not on their absolute values. Therefore, the relevant timing behavior of our model does not change if we recalibrate by simultaneously decreasing *nextA, nextB, lb,* and *ub* by the same amount. Procedure *time_wrap* does the recalibration, decrementing all these variables by the current value of *lb*. Our transition rules will invoke *time_wrap* immediately after any increment to *lb*. This is a form of rolling, dynamic time abstraction.

This recalibration strategy will succeed in bounding the values of these variables if the *differences* between the values of *nextA, nextB, lb,* and *ub* are bounded. It is shown in [16] that, for all the executions of the model in which no deadline is missed, the absolute values $|nextA-lb|, |nextB-lb|,$ and $|ub-lb|$ will all be less than $2*max(periodA, periodB)$. Therefore we can statically restrict the range of the time variables to $-MAX .. MAX$, where $MAX=2*max(periodA, periodB)$. To be more precise, if there is a deadline violation in the infinite model (from which all occurrences of *time_wrap* have been deleted), then there is a deadline violation in the recalibrated model, and it will be detected before
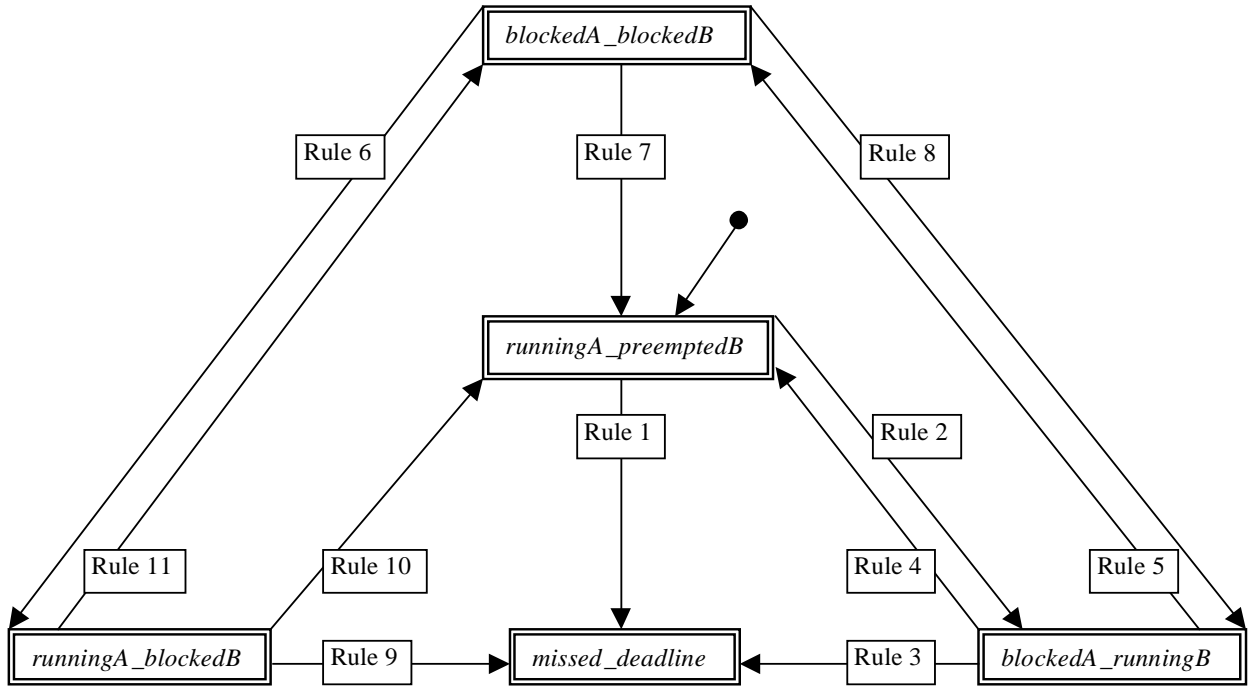
Figure 3: The transition system model

```
    Rule "1"                          Rule "2"                              Rule "4"
    runtime_status=                   runtime_status           =           runtime_status = blockedA_runningB
runningA_preemptedB               runningA_preemptedB                       & ub < nextB + periodB
    & ub >= nextA + periodA           & ub < nextA + periodA                & nextA < ub
       ==>                               ==>                                ==>
    Begin                             Begin                                 Begin
                                      nextA := nextA + periodA;
    runtime_status            :=      runtime_status           :=           runtime_status:=runningA_preempte
missed_deadline;                  blockedA_runningB;                        dB;
    End;                              delta := preemptB;                    preemptB := (ub - nextA < delta) ?
                                      ub := ub + preemptB;                              (ub  -  nextA)  :
                                      preemptB := 0;                    delta;
                                      End;                                  delta := estimA;
                                                                            lb := nextA;
                                                                            ub := nextA + estimA;
                                                                            time_wrap();
                                                                            End;
```

Figure 4: Representative transition rules

execution of the model attempts an update that puts these variables out of range.

2.3   A comparison

Our experiment analyzed the example of section 2.1 in three ways: We applied Murphi to the transition system defined in section 2.2; we applied HyTech to the hybrid automaton constructed by the methods of [10] alone; we applied SPIN to the model constructed by the methods of [16] alone. The comparison with [10], for various values of the parameters, is shown in the charts below.

We suspect that that the advantage of these

estimA=5, periodA = 10, estimB = 10, periodB = 30	Transition system	Hybrid automaton
Number of states/regions	11	8
CPU time (sec)	0.10	0.24
Memory used	1K	0.82M

estimA = 29, periodA = 59, estimB = 61, periodB = 181	Transition system	Hybrid automaton
Number of states/regions	1002	480
CPU time (sec)	0.10	13.73
Memory used	25K	4.53M

estimA = 167, periodA = 353, estimB = 313, periodB = 997	Transition system	Hybrid automaton
Number of states/regions	5013	2700
CPU time (sec)	0.40	106.95
Memory used	163K	20.13M

Figure 5 : A comparison

optimizations will increase as the timing constraints become more complex, because manipulating integers is more efficient than manipulating linear formulas with integer coefficients. We cannot quantify how much of the difference might be attributable to the fact that Murphi is a more mature tool than HyTech.

The advantage over [16] is not quite so dramatic—the improvement is one order of magnitude, not two.

2.4 Other properties

This section briefly considers problems other than schedulability. The model and the size of the state space depend on the property analyzed. For example, in the terminology of Figure 2, it is easier to analyze the assertion that "Both tasks always meet their deadlines" than to analyze the assertion "Task *B* always meets its deadlines," because uncertainty about the behavior of *A* would add nondeterminism to the model. Since the tasks of Figure 2 do not interact (except

implicitly, via preemption) there is not much to ask about this example aside from its schedulability.

When tasks do interact, things become more interesting. The Ravenscar rules require that no more than one task be waiting on the entry of any protected call. The main purpose of this requirement is to avoid the overhead of maintaining queues. In general, it is undecidable whether a program meets the requirement, though compliance could be guaranteed by making severe static semantic restrictions on the code. The Raven runtime raises an error dynamically if execution ever violates the requirement. Thus, it is important to be able to check this rule by static analysis. A schedulability model of the kind suggested in this section already encodes enough information in its state to answer this question. Analysis of the length of entry queues is insensitive to the recalibration trick.

Deadlock freedom is another interesting

question that should be amenable to our techniques. The priority ceiling protocol itself suffices to guarantee that a certain class of tasking programs cannot deadlock, but the general question is undecidable. (This problem is also insensitive to recalibration.)

2.5 Limitations

We might hope for a divide-and-conquer approach whereby knowing that the system is schedulable—for example, in cases where RMA is applicable—might permit us to produce a simpler model with which we might verify other properties. However, if the precise timing behavior of the program is necessary to guarantee those properties, we must represent that behavior in our model and therefore encode the schedulability problem within it. In effect, verifying schedulability is automatically part of verifying any property at all. Unfortunately, the intricacies and timing of task interleavings are the principal source of state space explosion.

Our experience thus far suggests that the effectiveness of our methods will depend more on the underlying set of tasking primitives than on a discipline restricting the patterns in which they are used. Interrupts are especially interesting, and present special problems. In the model of [16] we found that code with interrupts typically resulted in a state space explosion. Symbolic model checking may be applicable to this case. On the other hand, several tasking constructs omitted by the Ravenscar Profile seem amenable to model checking analysis: absolute delay statement; rendezvous; select statements.

3 More realistic examples

This section briefly describes the application of our model-checking techniques to more realistic examples. We summarize experiments using the methods of [16] on a modest work station, which we have not had the opportunity to repeat with the optimizations proposed above. These examples employ the main Ravenscar tasking constructs such as "**delay until**" statements, protected procedures and entries, interrupts, and sporadic tasks triggered by interrupts.

The modeling of interrupts and sporadic tasks is the most complicated part of the model of [16]. Conceptually, a sporadic task is triggered by an interrupt and must complete its response interrupt within a specified *response time*. Each interrupt is characterized by its *minimum interarrival time*—the minimum time between two consecutive occurrences of the interrupt. The minimum interarrival time and the response time for each interrupt are parameters of the model.

To implement sporadic tasks we use an Ada idiom required by the Ravenscar programming discipline: The response to an interrupt I is performed by a sporadic task T whose body is a loop. The head of that loop is a call on a protected entry E, so that task T is blocked at the head of the loop so long as entry barrier of E is false; and the last act of the loop is to reset the entry barrier of E to true. The text of an Ada program binds interrupt I to a protected procedure P, which will be executed by the runtime whenever I occurs; and, in this programming idiom, P must be implemented so that its only effect is to change the entry barrier of E from false to true. Thus, when interrupt I occurs, the runtime executes P, which sets the barrier of E to true; that unblocks task T, which performs the response to the interrupt, resets the barrier of E to false, and becomes suspended.

We permit tasks to contain both "**delay until**" statements and entry calls. For our purposes, a task containing a "**delay until**" statement is *periodic*. A *sporadic* task contains a call on a protected entry whose barrier is set by an interrupt handler. Since we impose no upper limit on the interrupt interarrival time, a sporadic task cannot be guaranteed to satisfy any periodic deadline. For this reason, sporadic tasks may not contain 'delay until' statements. The Promela code checks that all periodic tasks meet their deadlines and that the response to every interrupt completes within the response time.

We have analyzed several systems containing both periodic and sporadic tasks, all on a SparcServer20 with 64 megabytes of memory.

One is a toy pump control system [29] often used as a benchmark example, which our techniques handled in seconds. With some more complicated systems, however, the model of [16] encountered a state space explosion. We describe two such examples:

1. the Olympus attitude and orbital control system (AOCS) [30],
2. a brewery control program [31].

A pump controller

The pump control system has the following components:

1. four periodic tasks getting data from the four sensors and controlling the pump,
2. a sporadic task, triggered by the interrupt from a high/low water level detector, that controls the pump, and
3. two protected objects for the pump and the interrupt interface.

Verification of this program took 20 seconds.

The AOCS

The AOCS design contains 17 protected objects, 4 sporadic tasks driven by interrupts (with short interarrival times), and 10 periodic tasks (with relatively long periods). We were able to verify a reduced version with all 10 periodic tasks and only one sporadic task (roughly 1.5 hours of computation). Adding a second sporadic task resulted in a state space explosion that SPIN could not handle.

A Brewery controller

Our techniques successfully identified a timing error in the brewery control program, but the analysis required some abstractions performed by hand, not merely the "standard" abstractions used to represent the pump controller.

The brewery control program contains no interrupts. It consists of an alarm task suspended on a protected entry, several short-period tasks, and one long-period task that calculates a "pattern temperature." One of the short-period tasks compares the actual temperature to the pattern and, if the difference between the temperatures is too great, opens the entry barrier to trigger the alarm. We model the decision about whether to trigger the alarm as a completely nondeterministic event (a conservative approximation).

We may eliminate the long-period task altogether if we assume that the pattern temperature is constant. Under that assumption (also conservative) our methods took 6 minutes of computation to find a timing violation.

If we do not assume that the pattern temperature is constant, the combination of a long-period task with a short-period task nondeterministically triggering another task results in a state space explosion (as explained below).

The size of our model's state space is proportional to S^P, where:
1. P is the number of possible patterns of the periodic tasks' arrival times. (A task *arrives*

whenever it begins a new period.). P is roughly proportional to *(M/D)*, where M is the least common multiple of the task periods and interrupt interarrival times, and D is their greatest common divisor.
2. S is the average number of non-deterministic choices exercised by the model during the execution of any one pattern of arrival times. A common source of non-determinism is the runtime process controlling task preemption. However, this nondeterminism is usually restricted, since the control-delegating conditions in the runtime process are often mutually exclusive. Thus, the runtime process does not contribute much to the size of S. On the other hand, nondeterministic behavior in a short-period task will increase S, since this behavior is exercised in the many patterns where the task is running.

Our problem with the brewery control program is that the short-period task nondeterministically triggers the alarm, which increases S. We can still analyze the program if P is low, but including the long-period task increases P. This combination increases S^P sufficiently to cause a state explosion.

As for the interrupts, in [16] we model each interrupt by a Promela process representing a "quasi-task" that makes calls on the protected procedure that is its handler. The behavior of such a task is in many respects similar to the behavior of a periodic task that non-deterministically executes the interrupt handler and has a period equal to the interrupt's minimum interarrival time.

4 Future research

Our primary technical problem is how to optimize the model for efficient model-checking. The optimizations described in section 2—the runtime status abstractions, the encoding of regions as pairs of integers—are specific to our problem domain and to the kinds of properties being analyzed. There is an extensive literature on general-purpose algorithms for abstractions and optimizations of untimed transition systems, and on the automated discovery of invariants. (See, for example, [21-24]). Future research will consider the applicability of that literature to our problem.

Symbolic model checking is another possibility for dealing with state space explosion. Problems that do not yield to explicit search

techniques can sometimes be solved by symbolic model checking (and vice versa). The state-machine model accepted by a symbolic model checker is typically quite low-level and constrained. Not all symbolic model checkers permit variables of integer type. But some, such as WSMV [9], are able to treat integers and certain integer operations symbolically by using special encoding techniques that permit efficient representation of addition and integer comparisons, and those are precisely the arithmetical operations our methods require. Thus, WSMV is a promising engine for extending our results with symbolic model checking.

References

[1] R. Alur, C. Coucoubetis, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, pp. 3-34, 1995.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science* 126, pp. 183-235, 1994.

[3] R. Alur and T.A. Henzinger. Logics and Models of Real Time: A Survey. In *Real-Time: Theory in Practice, REX Workshop,* LNCS 600, pp. 74-106, 1991.

[4] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering* 22, pp. 181-201, 1996.

[5] K. Brink, J. Katwijk, R. Spelberg, and H. Toetenel. Analyzing schedulability of Astral specifications using extended timed automata. *Proceedings of the Third International Euro-Par Conference*, LNCS 1300, pp. 1290-1297, Springer-Verlag, 1997.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond, *Information and Computation*, vol. 98, pp. 142-170, 1992.

[7] A. Burns. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems*, S. H. Son, Ed.: Prentice Hall, pp. 225-248, 1994.

[8] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. Proceedings of *Reliable Software Technologies - Ada-Europe'98*, LNCS 1411, pp. 263-275, 1998.

[9] E.M. Clarke, M. Khaira, and X. Zhao. Word-level symbolic model checking: a new approach for verifying arithmetic circuits. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, 1996.

[10] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on software engineering*, 22(7), pp. 461-483, 1996.

[11] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer* 1, pp. 110-122, 1997.

[12] G. J. Holzmann. *Design and validation of computer protocols*: Prentice Hall, 1991. (The current version of Spin can be found at http://netlib.bell-labs.com/netlib/spin/whatispin.html.)

[13] C. Ghezzi, D. Mandriolli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2), pp. 107-123, 1990.

[14] C. Ghezzi, D. Mandriolli, S. Morasca, and M. Pezze. A unified high-level Petri net model for time-critical systems. *IEEE Transactions on software engineering*, 17(2), 1991.

[15] K. G. Larsen, P. Pettersson and Wang Yi. UPPAAL in a Nutshell. In *Springer International Journal of Software Tools for Technology Transfer* 1(1+2), 1997.

[16] D. Naydich and D. Guaspari. Analyzing Ravenscar Profile tasks by model checking. Technical report TM-98-0034, Odyssey Research Associates, 1998.

[17] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249-261, June 1988.

[18] G. Romanski *Safety critical software handbook*. Aonix, 1997.

[19] L. Sha, R. Rajkumar, and S. S. Sathaue. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of IEEE*, vol. 82, pp. 68--82, 1994.

[20] S. Wolfram. *Mathematica: A system for doing mathematics by computer*. Adisson-Wesley, 1988.

[21] The SAL Group. The SAL Intermediate Language.

[22] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. To appear in *Formal Methods of System Design*.

[23] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically.

[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[25] M. Bickford and D. Naydich. Hardware verification technology transfer: Application of formal methods and modeling to the ARM6. Tech. Rep. TM98-0021, ORA, 1998.

[26] Sast User Manual (version 0.2). Odyssey Research Associations, 1997.

[27] Z. Chen and D. Hoover. TableWise, a decision table tool. *Proceedings of the Tenth Annual Conference on Computer Assurance (Compass '95).*

[28] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering,* vol. 16, no. 9, September, 1990. Reprinted in proceedings of the First International Workshop on Larch, Springer-Verlag, 1993.

[29] A. Burns and A. J. Wellings, *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*: Elsevier, 1995.

[30] A. Burns, A. J. Wellings, C. M. Bailey, and E. Fyfe, "The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation," Proceedings of *Ada sans frontiers -- 12th Ada-Europe Conference*, LNCS 688, pp. 19-35, 1993.

[31] G. Romanski, "Ada, Concurrency and a Safety Critical Subset," Personal communications, 1998.

[32] "Raven Fact Sheet", Aonix, 1999.

[33] David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, "Protocol Verification as a Hardware Design Aid," 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.

Modeling and Verification of Real-Time Software
Using Extended Linear Hybrid Automata

Steve Vestal

steve.vestal@honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

Linear hybrid automata are finite state automata augmented with real-valued variables. Transitions between discrete states may be conditional on the values of these variables and may assign new values to variables. These variables can be used to model real time and accumulated task compute time as well as program variables. Although it is possible to encode preemptive fixed priority scheduling using existing linear hybrid automata models, we found it more general and efficient to extend the model slightly to include resource allocation and scheduling semantics. Under reasonable pragmatic restrictions for this problem domain, the reachability problem is decidable. The proof of this establishes an equivalence between dense time and discrete time models given these restrictions. We next developed a new reachability algorithm that significantly increased the size of problem we could analyze, based on benchmarking exercises we carried out using randomly generated real-time uniprocessor workloads. Finally, we assessed the practical applicability of these new methods by generating and analyzing hybrid automata models for the core scheduling modules of an existing real-time executive. This exercise demonstrated the applicability of the technology to real-world problems, detecting several errors in the executive code in the process. We discuss some of the strengths and limitations of these methods and possible future developments that might address some of the limitations.

1 Introduction

The first goal of the work described in this paper was to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory. Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors.

The second goal was to verify software implementations of systems. Task schedulers and communications protocols are reactive components that respond to events like interrupts, service calls, task completions, error detections, etc. We would like to model important implementation details such as control logic and data variables in the code. We would like the mapping between model and code to be clear and simple to better assure that the model really does describe the implementation.

Discrete event concurrent process models are widely used to model control flow within and interactions between concurrent activities. Classical discrete event concurrent process models do not deal with resource allocation and scheduling or data variables, which limits their usefulness for real-time systems and makes it awkward to model some implementation details. Classical preemptive scheduling models do not deal well with complex task sequencing and interaction, which limits their usefulness for describing distributed systems and implementation details. Discrete time models have been developed for real-time scheduling of concurrent processes[23, 13, 11, 31], and some work has been done on dense time real-time process algebras[10, 14]. This paper describes the use of dense time linear hybrid automata models to perform schedulability analysis and to verify implementation code.

The first problem we faced was the modeling of resource allocation and scheduling behaviors using hybrid automata. The applicability in principle of hybrid automata to the scheduling problem was already known[4]. We wanted a model that would admit a variety of complex allocation as well as scheduling algorithms, e.g. load balancing, priority inheritance. We wanted to be able to change the allocation and scheduling algorithms easily without changing the models of the real-time tasks themselves. We wanted to minimize the number of states and variables added to model allocation and scheduling. We found it most general and efficient to extend the definition of hybrid automata to include resource allocation and scheduling semantics rather than try to model the scheduling function as a hybrid automaton.

We use integration variables to record the accumulated compute time of tasks in preemptively sched-

uled systems. Allowing integration variables is known to make the reachability problem undecidable[22, 17]. We were curious about whether analysis of real-time allocation and scheduling in distributed heterogeneous systems is itself a fundamentally difficult problem, or if general linear hybrid automata are more powerful than is really necessary for this problem. We were able to show that the reachability problem becomes decidable when some simple pragmatic restrictions are placed on the model.

The second problem we faced was the computational difficulty of performing a reachability analysis. We began our work using an existing linear hybrid automata analysis tool, HyTech[18], but found ourselves limited to very small models. We developed and implemented a new reachability method that was significantly faster, more numerically robust, and used less memory. However, our prototype tool allows only constant rates (not rate ranges) and does not provide parametric analysis.

Using this new reachability procedure we were able to accomplish one of our goals: the modeling and verification of a piece of real-time software. We developed a hybrid automata model for that portion of the MetaH real-time executive that implements uniprocessor task scheduling, time partitioning and error handling[1]. We successfully analyzed these models, uncovering several implementation defects in the process. There are limits on the degree of assurance that can be provided, but in our judgement the approach may be significantly more thorough and significantly less expensive that traditional testing methods. This result suggests the technology has reached the threshold of practical utility for the verification of small amounts of software of a particular type.

However, we do not believe existing reachability methods are adequate yet for schedulability analysis of real systems. In our judgement, we would need to be able to analyze systems having a few dozen tasks on a few processors in order for the technology to begin finding use in this area. We discuss approaches that might lead to such improvements.

2   Resourceful Hybrid Automata

A hybrid automaton is a finite state machine augmented with a set of real-valued variables and a set of propositions about the values of those variables. Figure 1 shows an example of a hybrid automaton whose discrete states are `preempted, executing` and `waiting`; and whose real-valued variables are c and t. `Waiting` is marked as the initial discrete state, and c and t are assumed to be initially zero.

Each of the discrete states has an associated set of differential equations, e.g. $\dot{c} = 0$ and $\dot{t} = 1$ for the discrete state `preempted`. While the automaton is in a discrete state, the continuous variables change at the rates specified for that state.

Edges may be labeled with guards involving continuous variables, and a discrete transition can only occur when the values of the continuous variables satisfy the guard. When a discrete transition does occur, designated continuous variables can be set to designated values as specified by assignments labeling that edge.

A discrete state may also be annotated with an invariant constraint to assure progress. Some discrete transition must be taken from a state before that state's invariant becomes false. For example, the hybrid automaton in Figure 1 must transition out of state `computing` before the value of c exceeds 100.

The hybrid automata of interest to us are called linear hybrid automata because the invariants, guards and assignments are all expressed as sets of linear constraints. The differential equations governing the continuous dynamics in a particular discrete state are restricted to the form $\dot{x} \in [l, u]$ where $[l, u]$ is a fixed constant interval (our current prototype tool is further restricted to a singleton rate, $\dot{x} = [l, l]$).

We want to verify assertions about the behavior of a hybrid automaton. Although it is possible in general to check temporal logic assertions[4], we make do by annotating discrete states and edges with sets of linear constraints labeled as assertions. These constraints must be true whenever the system is in a discrete state or whenever a transition occurs over an edge.

The cross-product construction used to compose concurrent finite state processes can be extended in a fairly straight-forward way to systems of hybrid automata. The invariant and assertion associated with a discrete system state are the conjunction of the invariants and assertions of the individual discrete states. The guards, assertions and assignments of synchronized transitions are the conjunction and union of the guards, assertions and assignments of the individual discrete co-edges. If there is a conflict between the rate assignments of individual discrete states, or a conflict between the variable assignments of co-edges, then the system is considered ill-formed. Note that concurrent hybrid automata may interact through shared real-valued variables as well as by synchronizing their transitions over co-edges.

The application of interest in this paper is the analysis and verification of real-time systems. Figure 1 shows an example of a simple hybrid automata model for a preemptively scheduled, periodically dispatched task. A task is initially waiting for dispatch but may at various times also be executing or preempted. The variable t is used as a timer to control dispatching and to measure deadlines. The variable t is set to 0 at each dispatch (each transition out of the waiting state), and a subsequent dispatch will occur when t reaches 1000. The assertion $t \leq 750$ each time a task transitions from executing to waiting (each time a task completes) models a task deadline of 750 time units. The variable c records accumulated compute time, it is reset at each dispatch and increases only when the task is in the computing state. The invariant $c \leq 100$ in the computing state means the task must complete before it receives more than 100 time units of processor service, the guard $c \geq 75$ on the completion transition means the task may complete after it has received 75 time units of processor service (i.e. the task compute time is uncertain and/or variable but always falls in the interval $[75, 100]$).

In this example the edge guards `selected` and `unselected` represent scheduling decisions made at

if selected

if c $\geq$ 75

assert t $\leq$ 750

preempted

$\dot{c} = 0$
$\dot{t} = 1$

executing

c $\leq$ 100
$\dot{c} = 1$
$\dot{t} = 1$

waiting

t $\leq$ 1000
$\dot{c} = 0$
$\dot{t} = 1$

if unselected

c := 0
t := 0

if t = 1000
and selected
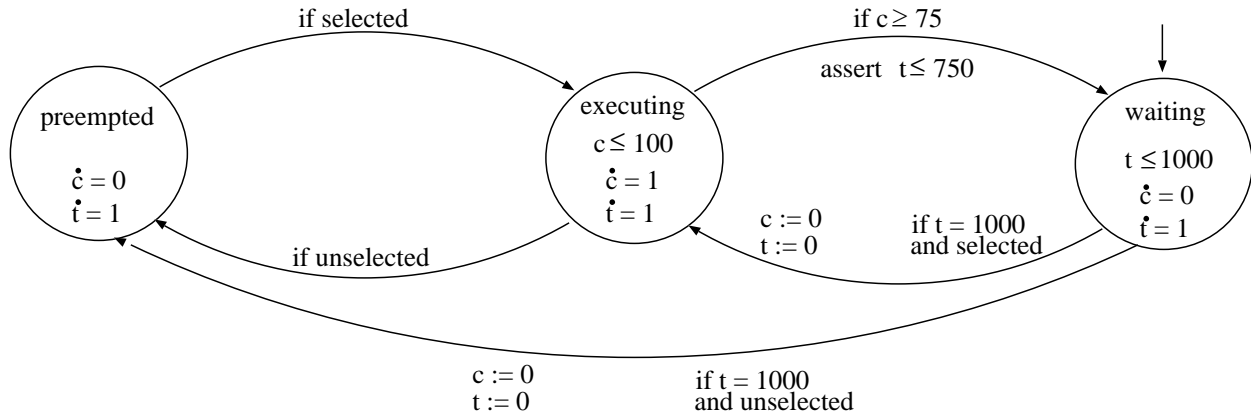
c := 0
t := 0

if t = 1000
and unselected

Figure 1: A Hybrid Automata Model of a Preemptively Scheduled Task

scheduling events (called scheduling points in the real-time literature). These decisions depend on the available resources (processors, busses, etc.) being shared by the tasks. There are several approaches to introduce scheduling semantics into a model having several concurrent tasks.

Scheduling can be introduced using concepts taken from the theory of discrete event control[26]. A concurrent scheduler automaton can be added to the system of tasks. The scheduling points in the task set become synchronization events at which the scheduler automaton can observe the system state and make control decisions. Many high-level concepts from discrete event control theory carry over into this domain, such as the importance of decentralized control and limited observability in distributed systems.

Discrete event control theory provides an approach to synthesize optimal controllers, which in this domain translates to the automatic construction of application-specific scheduling algorithms. However, classical discrete event control theory does not deal with time. The theory has been extended to synthesize nonpreemptive schedulers for timed automata[9, 2], but this excludes preemptively scheduled systems. It is possible to develop scheduling automata by hand using traditional real-time scheduling policies such as preemptive fixed priority. Some examples have been given in the literature, where each distinct ready queue state is modeled as a distinct discrete state of the scheduler automaton[4]. This would allow a very large class of scheduling algorithms to be modeled, but the size of the scheduler automaton may grow combinatorially with the number of tasks.

It is possible to model preemptive fixed priority scheduling by encoding the ready queue in a variable rather than in a set of discrete states. A queue variable is introduced that will take on only integer values. At each transition where a task i is dispatched, 2^i is added to this queue variable; at each transition where task i completes, 2^i is subtracted. The queue variable can be interpreted as a bit vector whose i^{th} bit is set whenever task i is ready to compute. There is no

separate scheduler automaton, the scheduling protocol is modeled using additional guards and states in the task automata. This is the approach we took when we started our work using HyTech. This encodes a specific scheduling protocol into each task model, and adds additional discrete states, variables and guards to the model. It is awkward to model any scheduling policy other than simple preemptive fixed priority.

In the end, we found it simpler and more general to define a slightly extended linear hybrid automata model that includes resource scheduling semantics[28]. The discrete state composition of the task set is performed before any scheduling decisions are made. A scheduling function is then applied to the composed system discrete state to determine the variable rates to be used for that system state. In essence, the composed system discrete state is the ready queue to which the scheduling function is applied, very much analogous to the way run-time scheduling algorithms are applied in an actual real-time system. It is not necessary to have different discrete states for preempted and computing, since this information is now captured in the variable rates. It is not necessary to model a scheduling algorithm as a finite state control automaton added to the system, it is not necessary to encode a specific scheduling semantics into the task automata. One simply codes up a scheduling algorithm in the usual way and links it with the rest of the reachability analysis code. This approach significantly reduces the number of discrete states in the model (from 3^t for our HyTech models to 2^t for our extended models, where t is the number of tasks). This also simplifies the modeling of the desired scheduling discipline. The details of this model and its semantics are recorded elsewhere[28].

3 Decideability

Most traditional real-time schedulability problems are solvable in polynomial time or are NP-complete. However, hybrid automata models that allow multiple rates and integration variables are undecideable[22, 17]. The hybrid automata models we are using are much more powerful than traditional allocation and

scheduling models, and most existing tasking and scheduling models can be viewed as special cases of the more general hybrid automata model. This raises the question of whether the schedulability problem for complex interacting tasks that are dynamically allocated in distributed heterogeneous systems is in fact undecideable, or whether models of such systems are decideable special cases of the more powerful linear hybrid automata models.

The undecideability of hybrid automata reachability analysis was proved by reducing the reachability problem for two-counter machines, which is known to be undecideable, to the reachability problem for hybrid automata[22, 17]. The construction used in the proof is fairly straightforward in our slightly extended model and can be accomplished using a single processor. However, a pragmatic real-time system designer would reject the theoretical construction as a bad design because it relies in places on exact equality comparisons between timers and accumulated compute times. In a real system, these would be regarded as race conditions or ill-defined behaviors. The problem becomes decideable given a few simple practical restrictions, which are captured in the following theorem.

Theorem 1 *The reachability problem is decideable for resourceful linear hybrid automata if the following conditions hold.*

- *The set of possible outputs of the scheduling function for each possible system discrete state is finite and enumerable.*

- *For every task activity integrator variable, the rate interval remains fixed between resets of that integrator (i.e. the scheduler does not dynamically reallocate any task activity in mid-execution to a new resource having a different rate for that activity).*

- *For every task activity integrator variable, every edge guard is a set of rectangular constraints of the form $x \in [l, u]$, and either the edge guard has a non-singular interval ($x \in [l, u]$ with $l < u$) or else the rate interval for $\dot{x}$ is non-singular (i.e. system behavior does not depend on exact equality comparisons with exact drift-free clocks or execution rates).*

- *However, we allow as a special exception task activity integrator variables with singular rate interval and singular rectangular edge guards, providing the integrator variable is only reset or stopped or restarted at a transition having at least one edge guard $y \in [m, m]$ with $[m, m]$ and $\dot{y}$ singular (y may but need not be x), and for every such singular constraint on that edge $\dot{x} = k\dot{y}$ for some positive integer k (i.e. some types of noninteracting or harmonically interacting behaviors may be modeled exactly).*

This result should not be surprising. The ability to test for exact equality is known to add theoretical

power to dense time temporal logics[3], and similar restrictions are known to make certain other hybrid automata models decideable[25]. The proof of this theorem, which we provide elsewhere[28], is by reduction to a discrete time finite state automaton.

4 Reachability Analysis

A state of a linear hybrid automaton consists of a discrete part, the discrete state at some time t; and a continuous part, the real values of the variables at time t. It turns out that, although this state space is uncountably infinite, the reachable state space for a given linear hybrid automaton is a subset of the cross-product of the discrete states with a recursively enumerable set of convex polyhedra in $\Re^n$ (where n is the number of variables)[4]. A region of a linear hybrid automaton is a pair consisting of a discrete state and a convex polyhedron, where convex polyhedra can be represented using a finite set of linear constraints. Model checking consists of enumerating the reachable regions for a given linear hybrid automaton and checking to see if they satisfy the assertions.

Figure 2 depicts the basic sequence of operations that, given a starting region (a discrete state and a polyhedron defining a set of possible values for the variables), computes the set of values the variables might take on in that discrete state as time passes; and computes a set of regions reachable by subsequent discrete transitions.

The first step is the computation of the time successor polyhedron from the starting polyhedron (often called the post operation). For each point in the starting polyhedron, the time successor of that point is a line segment beginning at that point whose slope is defined by the variable rates specified for the discrete state. This is the set of variable values that can be reached from a starting point by allowing some amount of time to pass. The time successor of the starting polyhedron is the union of the time successor lines for all points in the starting polyhedron. A basic result of linear hybrid automata theory is that the time successor of any convex polyhedron is itself a convex polyhedron (which in general will be unbounded in certain directions)[4].

The second step is the intersection of the time successor polyhedron with the invariant constraint associated with the discrete state. Polyhedra are easily intersected by taking the union of the set of linear constraints that define the two polyhedra. This is the time successor region that is feasible given the invariant specified for the discrete state.

The remaining steps are used to compute new regions reachable from this feasible time successor region by some transition over an edge. For each edge out of the current discrete state, the associated guard is first intersected with the feasible time successor region. This polyhedron, if nonempty, defines the set of all variable values that might exist whenever the discrete transition could occur. Any variable assignments associated with the edge must now be applied to this polyhedron. This is done in two phases. First, a variable to be assigned a new value $x := l$ is unconstrained (often called the free operation). This oper-
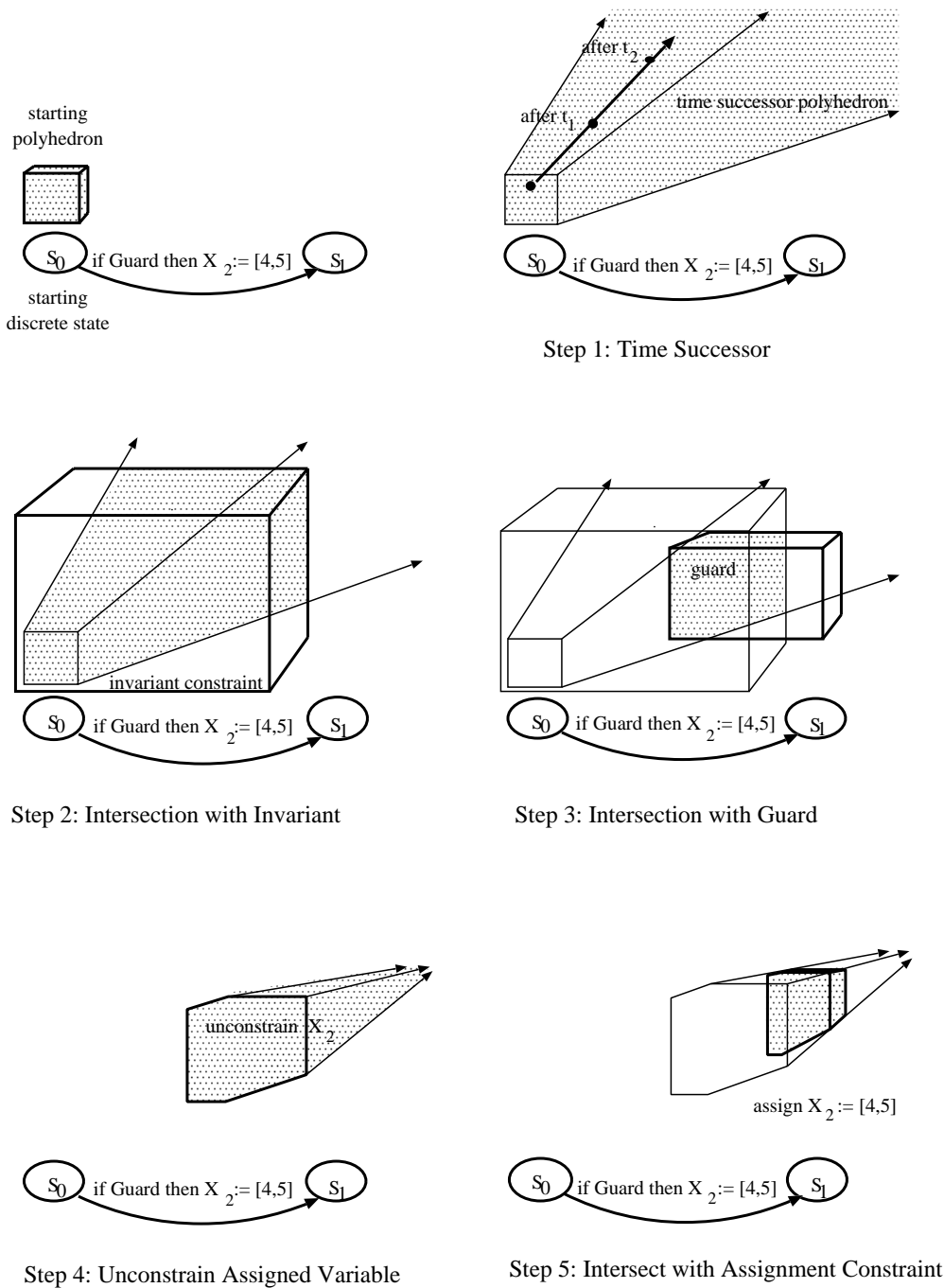
starting
polyhedron

starting
discrete state

S_0 if Guard then $X_2 := [4,5]$ S_1

after t_2

after t_1

time successor polyhedron

S_0 if Guard then $X_2 := [4,5]$ S_1

Step 1: Time Successor

invariant constraint

S_0 if Guard then $X_2 := [4,5]$ S_1

Step 2: Intersection with Invariant

guard

S_0 if Guard then $X_2 := [4,5]$ S_1

Step 3: Intersection with Guard

unconstrain X_2

S_0 if Guard then $X_2 := [4,5]$ S_1

Step 4: Unconstrain Assigned Variable

assign $X_2 := [4,5]$

S_0 if Guard then $X_2 := [4,5]$ S_1

Step 5: Intersect with Assignment Constraint

Figure 2: Hybrid Automata Reach Forward Operations

ation leaves unchanged the relationships between all other variables, i.e. the polyhedron is projected onto the subspace $\Re^{n-1}$ of the remaining variables. This result is then intersected with the constraint $x = l$. This polyhedron, together with the discrete state to which the edge goes, is a new region for which the

above steps may be repeated. In general a set of assignments whose right-hand sides are linear formula are allowed, with some restrictions. The variables to be assigned are unconstrained and the resulting polyhedra are then intersected with the appropriate linear constraints in some order. With care, fairly complex

sequences of assignments to program variables can be modeled on a single edge[30].

The overall method begins at the initial region of a hybrid automaton. The operations described above are applied to enumerate feasible time successor regions and the new regions reachable from these via discrete transitions. As new regions are enumerated, they must be checked to see if they have been visited before (otherwise the method will not terminate even when there are a finite number of regions). This is done by comparing the discrete states of regions for equality, and by checking to see if the new polyhedron is contained in the polyhedron of a previously visited region.

The earliest reachability tool of which we are aware, HyTech, represented polyhedra as finite sets of linear constraints[4]. Operations on polyhedra used quantifier elimination, a method to manipulate and make decisions about systems of linear constraints in which some of the variables are existentially quantified. Subsequent tools, Polka and a later version of HyTech, used a pair of representations: the traditional system of linear constraints together with polyhedra generators consisting of sets of vertices and rays[16, 18]. Different operations required during reachability are more convenient in the different representations, and methods are used to convert between the two as needed.

Both of these methods are subject to the theoretical risk that some polyhedra operations may require a combinatorial amount of time. Another potential performance problem occurs when the reachable discrete state space is completely enumerated first followed by an enumeration of the polyhedra. This might result in enumerating discrete states that are actually not reachable due to edge guards involving the continuous variables. Finally, in our experiments we found that a significant fraction of a set of benchmark schedulability problems we tried to solve using HyTech resulted in numeric overflow errors.

We developed a new set of algorithms for the polyhedra operations used during reachability analysis and implemented a prototype on-the-fly reachability analysis library. Our prototype operates on lists of linear constraints of the form $l \le e \le u$ where l and u are fixed constant integer bounds and $e = c_1 x_1 + c_2 x_2 + ...$ is a linear formula with fixed constant integer coefficients. Our current algorithms restrict variable rates to be fixed scalar constants, $\dot{x} = i$ rather than the more general $\dot{x} \in [l, u]$.

We convert a polyhedron P into $\text{Post}(P, \dot{\mathbf{x}})$, the time successor of P given a vector of variable rates $\dot{\mathbf{x}}$, by applying the two steps

1. Let each constraint $l_i \le e_i \le u_i$ where $\dot{e}_i \ne 0$ be written so that $\dot{e}_i > 0$, which can be achieved by multiplying the constraint by -1 if needed. For each distinct pair of constraints

$$l_i \le e_i \le u_i$$
$$l_j \le e_j \le u_j$$

where $\dot{e}_i > 0$ and $\dot{e}_j > 0$, add to the set the

constraint

$$\dot{e}_j l_i - \dot{e}_i u_j \le \dot{e}_j e_i - \dot{e}_i e_j \le \dot{e}_j u_i - \dot{e}_i l_j$$

2. Replace each constraint $l \le e \le u$ where $\dot{e} > 0$ by $l \le e \le \infty$.

We compute $\text{Free}(P, x)$, the result of unconstraining variable x in polyhedron P, using the two steps

1. Let each constraint $l \le e \le u$ in P where e has an instance of x be written in the form $l \le cx - e' \le u$, where e' involves the remaining variables and their coefficients and $c > 0$. For every distinct pair of such constraints in P

$$l_i \le c_i x - e_i \le u_i$$
$$l_j \le c_j x - e_j \le u_j$$

combine the two in a way that cancels the x terms, adding to $\text{Free}(P, x)$ the constraint

$$c_j l_i - c_i u_j \le c_i e_j - c_j e_i \le c_j u_i - c_i l_j$$

2. Each constraint $l \le e \le u$ where e has no instances of variable x is added to $\text{Free}(P, x)$.

These algorithms might be viewed as generalizations of the difference methods used for timed automata[12, 8] and exhibit some similarity to the pragmatic algorithm used earlier for quantifier elimination[4]. Our prototype invokes a Simplex algorithm as part of the operations to test for feasibility and containment. We use a bounds tightening procedure to reduce the size of the constraint list after certain operations and to rapidly detect most infeasible polyhedra. Simplex-based reduction and feasibility testing is only applied when the bounds tightening procedure is ineffective. Details of our reachability analysis methods and implementation and proofs of correctness are documented elsewhere[29].

We benchmarked our prototype tool against HyTech and Verus[11] (a discrete timed automata reachability analysis tool that uses BDD techniques) using randomly generated uniprocessor workloads containing mixtures of periodic and aperiodic tasks. Figure 3 shows the percentage of problems that were solved by each of the tools, together with the primary reasons that solution was not achieved. Figure 4 compares the time required for solution for problems that were solved by all the tools using a logarithmic scale (a point appears for both HyTech and our prototype only for problems that were solved by both). We further increased the size of model we could analyze by applying some results from traditional scheduling theory to simplify the models, and by using a simple partial order reduction technique, these results are reported elsewhere[29].

5 Verifying the MetaH Executive

MetaH is an emerging SAE standard language for specifying real-time fault-tolerant high assurance software and hardware architectures[1, 24, 27]. Users
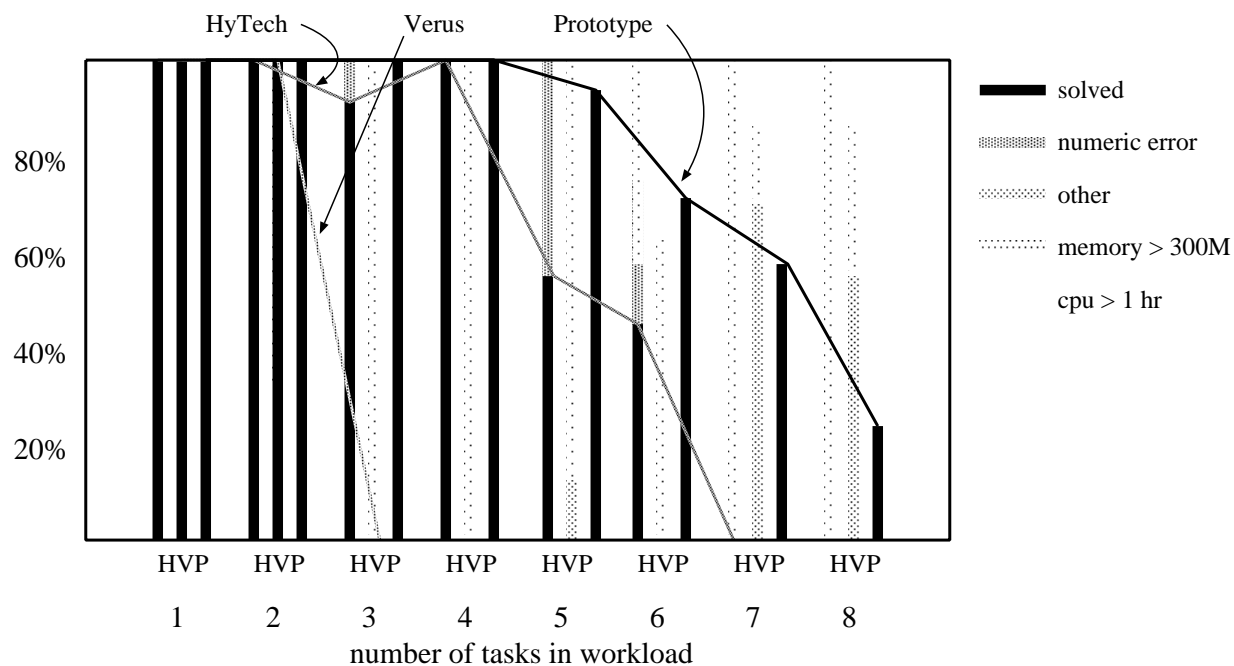
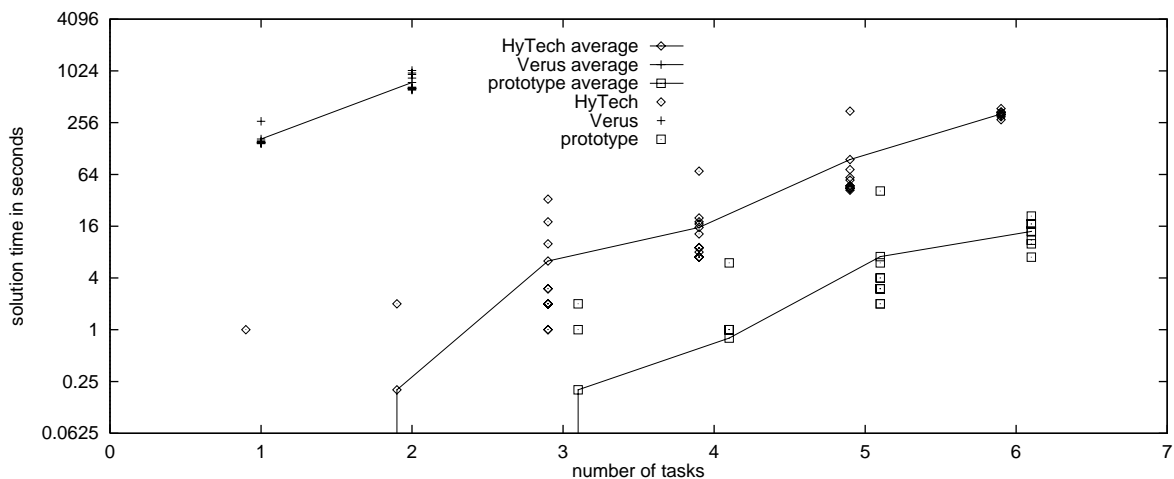Figure 3: Percentage of Generated Problems That Were Solved

Figure 4: Solution Times for Problems That Were Solved

specify how software and hardware components are combined to form an overall system architecture. This specification includes information about one or more configurations of tasks and message and event connections; and information about how these objects are mapped onto a specified hardware architecture. The specification includes information about timing behaviors and requirements, fault and error behaviors and requirements, and partitioning and safety behaviors and requirements.

Our current MetaH toolset, illustrated in Figure 5, can generate and analyze formal models for schedula-

bility, reliability, and partition isolation. The toolset can also configure an application-specific executive to perform the specified task dispatching and scheduling, message and event passing, changes between alternative configurations, etc. Unlike many conventional systems that rely on a large number of run-time service calls to configure a system by dynamically creating and linking to tasks, mailboxes, event channels, timers, etc., our toolset builds most of this information into an application-specific executive. There are relatively few run-time service calls, and the effects of these are tailored based on the specified application
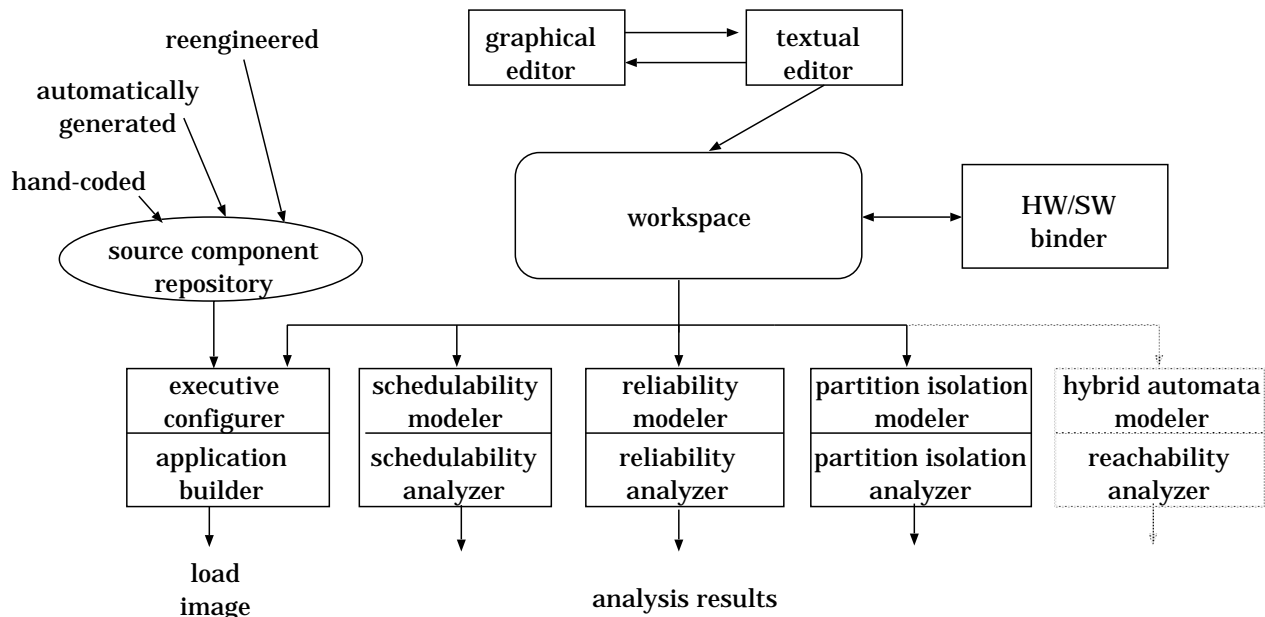
Figure 5: MetaH Toolset

architecture and requirements.

Our MetaH executive supports a reasonably complex tasking model using preemptive fixed priority scheduling theory[5, 6, 7]. Among the features relevant to this study are period-enforced aperiodic tasks, real-time semaphores, mechanisms for tasks to initialize themselves and to recover from internal faults, and the ability to enforce execution time limits on all these features (time partitioning). Slack stealing in support of aperiodic and incremental tasks is also supported, but as we will mention later these were not modeled or verified.

Figure 6 shows the high-level structure of the MetaH executive. The core task scheduling operations are implemented by module `Threads`, e.g. start, dispatch, complete. These operations implement transitions between the discrete task scheduling states, e.g. dispatch may transition a task from the awaiting dispatch state to the computing state. These operations must take into account details such as the task type, optional execution time enforcement, event queueing, etc. Module `Threads` invokes operations in module `Time_Slice`, which encapsulates arithmetic operations and tests on two execution time accumulators maintained by the underlying RTOS and hardware for each task: an accumulator that increases while a task executes, and a time slice that decreases while a task executes. `Time_Slice` may set these variables to desired values using services provided through the `MetaH RTOS interface`. If time slicing is enabled for a task, then a trap will be raised by the underlying hardware and RTOS when the time slice reaches zero. This trap is handled by one of the operations in `Threads`. Module `Clock_Handler` is periodically invoked by the underlying system (it is the han-

dler for a periodic clock interrupt) and makes calls to `Threads` to dispatch periodic tasks and start and stop threads at mode changes. Modules `Events`, `Modes` and `Semaphores` contain data tables and operations to manage user-declared events, dynamic reconfiguration, and semaphores.

We produced hybrid automata models for the `Threads` and `Time_Slice` modules, about 1800 lines of code. We did not write a separate model using a special modeling language, instead we inserted calls to build the model into the executive code itself. For example, in the code that implements the dispatch operation there is logic to decide if a task can be dispatched, assignments to change program variables, and calls to set the time slice and execution time counters. Into this code we inserted a call to a modeling procedure to create an edge between the corresponding states of the linear hybrid automata model. The guards for this edge are the conditional expressions appearing in the code, and the assignments on this edge are the assignments appearing in the code. This provides a high degree of traceability between the implementation and the model.

The generation of the hybrid automata models resembled all-paths unit testing. We developed several simple application specifications that included most (but not all) of the tasking features. We wrote a test driver that exercised all relevant paths in the core scheduling modules. For each application specification, the test driver thus triggered the generation of a linear hybrid automata model of the possible behaviors of the core scheduling operations for a particular combination of tasks and features.

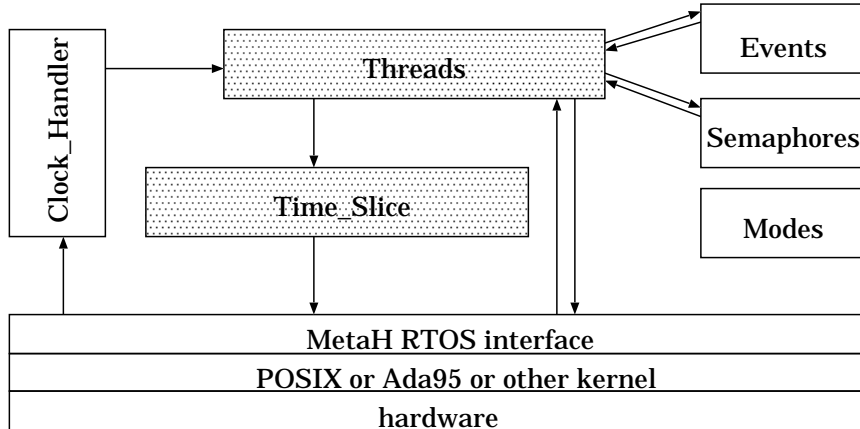The conditions we checked during reachability analysis were that all deadlines were met whenever

Figure 6: MetaH Executive Structure

the schedulability analyzer said an application was schedulable; no accessed variables were unconstrained (undefined) and no invariants were violated on entry to a region; and no two tasks were ever in a semaphore locking state simultaneously. Assertion checks appearing in the code were modeled by edges annotated with `assert False`.

We also collected information about which edges were used by some transition during reachability analysis and compared this with all the possible edges that might be created (all instances of calls inserted into the code to create edges). This allowed us to insure that all modeled portions of the code were covered by at least one reachability analysis.

A total of 14 real-valued variables and 15 discrete states were defined to model each task. No single task model used all 14 variables and 15 states, different task types with different specified options used different combinations. Figure 7 shows the simplest linear hybrid automata model we generated, a periodic task with period and deadline of 100000us, compute time between 0 and 90000us, recovery time between 0 and 10000us. States are also annotated with processor scheduling priorities, which are not shown here. The variable rates were derived from the scheduling priorities by the analysis tool, which used preemptive fixed priority scheduling semantics for this study. Table 1 summarizes the complete set of applications we analyzed. A more detailed discussion of the modeling methods and results is provided elsewhere[30].

We discovered nine defects in the course of our verification exercise. Four of these were tool defects, two that could cause bad configuration data to be generated and two that could cause erroneously optimistic schedulability models to be generated. Six of these defects could cause errors only during the handling of application faults and recoveries, three of these six only in the presence of multiple near-coincident faults and recoveries. In our judgement, of the nine defects we found, one would almost certainly have been detected by moderately thorough requirements testing, while three would have been almost impossible to de-

tect by testing due to the multiple carefully timed events required to produce erroneous behavior. The other five may have been detected by thorough requirements testing of fault and recovery features, providing the tester thought about possible execution timelines and arranged for tasks to consume carefully selected amounts of time between events.

There are a number of significant limitations on the degree of assurance provided. In our initial exercise, we chose not to model many behaviors that could have been modeled in a fairly straight-forward way, e.g. mode changes, inter-processor communication protocol, non-preemptable executive critical sections. In some cases different behaviors and subsystems can be modeled and analyzed almost independently, but it is not clear at what point the reachability analysis will become intractable as the extent of the model grows. Some behaviors might be more difficult to model, e.g. slack scheduling. The MetaH processor interface, underlying RTOS and hardware are unlikely to be fully model-able for a variety of practical and technical reasons. The MetaH tools were not verified, only a few specific generated modules and reports for a few example applications. Although our approach provides good traceability between code and model, there is still a very real possibility of modeling errors. The reachability analysis tool may contain defects; we discovered two in our tool in the course of this work. The modeled code does not change from application to application, and the analyzed applications fully exercised the code model, but to rigorously assert this code is correct for all possible applications would require some sort of induction argument. Even if the source code is correct, defects in the compiler, linker or loader software could introduce defects into the executable image.

Nevertheless, we estimate that the effort required for this exercise was roughly comparable to that required for traditional unit testing, but the results were more thorough than would have been achieved using traditional requirements testing. The method must be used in conjunction with traditional verification tech-
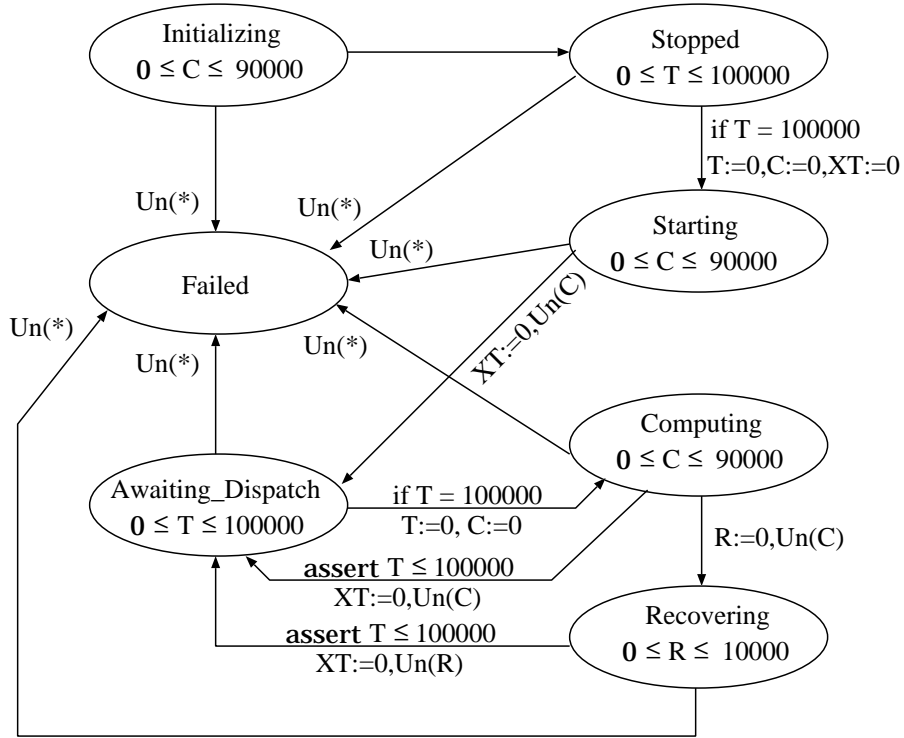
Figure 7: Generated Hybrid Automata Model for a Simple Periodic Task

niques such as testing, but it is at least intuitively reasonably easy to distinguish requirements that will be verified using hybrid automata from requirements that must be verified using other techniques.

6   Future Work

Our experience leads us to believe that linear hybrid automata are very powerful and well-suited for this domain. We were able to achieve one of our goals, the modeling and verification of a piece of real-world real-time software, with a number of limitations. We do not believe we have achieved the other goal yet, modeling and schedulability analysis for complex distributed systems of real-world size. However, there are a number of potential future developments that might reduce the verification limitations and provide useful schedulability analysis capabilities.

It should be possible to use the set of reachable regions produced by the analysis tool to automatically generate tests. This could significantly reduce the cost and increase the quality of requirements testing (which might still be required by the powers-that-be). Such tests could also detect defects that could not be found by model analysis, such as defects in the compiler, linker, loader, RTOS or hardware. One of the issues that must be confronted is the ease of constructing, running and observing the results of tests; for example, in theory one might encounter transitions in the model that occur only when two values are extremely close, which could be practically impossible to do in a test. Another issue is that such tests would not take into

account the internal logic of unmodeled modules such as the RTOS; a systematic method for testing multiple points within each reachable polyhedron might help address this.

There are a number of potentially useful improvements in analysis methods and tools. Approximation and partial order methods might significantly increase the size of the model that could be analyzed[16, 19, 15, 29]. Preprocessing models to modify numeric parameters in certain ways can result in much more easily solved models[29]. It is possible to apply theorem proving methods to linear hybrid automata[21], and some work has been done on dense-time process algebras[10, 14]. Decomposition and induction methods currently being explored for discrete state models might be extensible to linear hybrid automata. There are a number of possible ways to visualize and navigate the reachable region space that would be of practical assistance during model development and debugging and during reviews. Concise APIs and support for inline modeling could reduce both the modeling effort and the number of modeling defects.

Changes will inevitably be required to the design, implementation and verification processes to make good use of these methods. Much of the benefit of other formal methods has been due to subsequent changes in development methods that resulted in more verifiable and defect-free specifications, designs and code in the first place. An important and not completely technical question is how verification processes might be changed to beneficially use these methods.

Description	Discrete States	Distinct Polyhedra	Sparc Ultra-2 CPU Seconds
one periodic task	7	7	0
one periodic task, enforced execution time limits	7	10	0
one periodic task, enforced execution time limits, one semaphore	8	29	15
one period-enforced aperiodic task	9	18	0
one period-enforced aperiodic task, enforced execution time limits	9	27	2
one period-enforced aperiodic task, enforced execution time limits, one semaphore	11	124	125
two periodic tasks	36	60	3
two periodic tasks, enforced execution time limits	36	108	24
two periodic tasks, one with period transformed into two pieces,	41	97	10
two periodic tasks, one shared semaphore	48	118	36
two periodic tasks, one with period transformed into two pieces, enforced execution time limits	41	174	87
two periodic tasks, one with period transformed into four pieces, enforced execution time limits, recovery limit greater than compute limit	40	334	103
two tasks, one periodic and one period-enforced aperiodic	44	623	115
two periodic tasks, one with period transformed into four pieces, enforced execution time limits	41	351	170
two tasks, one periodic and one period-enforced aperiodic, enforced execution time limits	44	425	184
two tasks, one periodic and one period-enforced aperiodic, one shared semaphore	70	638	840
two periodic tasks, one with period transformed into two pieces, enforced execution time limits, one shared semaphore	55	963	5658

Table 1: Modeled Applications

What evidence would be required, for example, to convince a development organization or regulatory authority to replace selected existing verification activities with modeling and analysis activities, or to add modeling and analysis to current verification activities?

References

[1] *MetaH User's Guide*, Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN, www.htc.honeywell.com/metah.

[2] K. Altisen, G. GöBler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, "A Framework for Scheduler Synthesis," *Real-Time Systems Symposium*, December 1999.

[3] Rajeev Alur, Tomás Feder and Thomas A. Henzinger, "The Benefits of Relaxing Punctuality," *Proceedings of the Tenth Annual ACm Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 19-21, 1991.

[4] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[5] Pam Binns, "Scheduling Slack in MetaH," *Real-Time Systems Symposium*, work-in-progress session, December 1996.

[6] Pam Binns, "Incremental Rate Monotonic Scheduling for Improved Control System Performance," *Real-Time Applications Symposium*, 1997.

[7] Pam Binns and Steve Vestal, "Message Passing in MetaH using Precedence-Constrained Multi-Criticality Preemptive Fixed Priority Scheduling," *submitted Real-Time Applications Symposium*.

[8] Johan Bengtsson and Fredrik Larsson, *UPPAAL, A Tool for Automatic Verification of Real-Time Systems*, DoCS 96/97, Department of Computer Science, Uppsala University, January 15, 1996.

[9] B. A. Brandin and W. M. Wonham, "Supervisory Control of Timed Discrete-Event Systems," *IEEE Transitions on Automatic Control*, v39, n2, February 1994.

[10] Patrice Brémond-Grégoire and Insup Lee, "A Process Algebra of Communicating Shared Resources with Dense Time and Priorities," University of Pennsylvania Department of Computer Science Technical Report MS-CIS-95-08, June 1996.

[11] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems," *Real-Time Systems Symposium*, December 1994.

[12] David L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 12-14, 1989, also in *Lecture Notes in Computer Science 407*, J. Sifakis (Ed.), Springer-Verlag, pp 197-212.

[13] Andre N. Fredette and Rance Cleaveland, "RSTL: A Language for Real-Time Schedulability Analysis," *Proceedings of the Real-Time Systems Symposium*, December 1993.

[14] Andre N. Fredette, *A Generalized Approach to the Analysis of Real-Time Computer Systems*, Ph.D. Dissertation, North Carolina State University, March 1993.

[15] Nicolas Halbwachs, Pascal Raymond and Yann-Eric Proy, "Verification of Linear Hybrid Systems by Means of Convex Approximations," *Workshop on Verification and Control of Hybrid Systems*, Piscataway, NJ, October 1995.

[16] Nicolas Halbwachs, Yann-Erik Proy and Patrick Roumanoff, "Verification of Real-Time Systems using Linear Relation Analysis," *Formal Methods in System Design*, 11(2):157-185, August 1997.

[17] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[18] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "HyTech: The Next Generation," *Real-Time Systems Symposium*, December 1995.

[19] Thomas A. Henzinger and Pei-Hsin Ho, "A Note On Abstract Interpretation Strategies for Hybrid Automata," *Hybrid Systems II*, also *Lecture Notes in Computer Science 999*, Springer-Verlag, 1995.

[20] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "A User Guide to HyTech," University of California at Berkeley,
`www.eecs.berkeley.edu/~tah/HyTech`

[21] Thomas A. Henzinger and Vlad Rusu, "Reachability Verification for Hybrid Automata," *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, also *Lecture Notes in Computer 1386*, Springer-Verlag, 1998.

[22] Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, "Integration Graphs: A Class of Decideable Hybrid Systems," in R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer-Verlag, 1993.

[23] Insup Lee, Patrice Brémond-Grégoire and Richard Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," Department of Computer Science, University of Pennsylvania.

[24] Bruce Lewis, "Software Portability Gains Realized with MetaH, an Avionics Architecture Description Language," 18th *Digital Avionics Systems Conference*, St. Louis, MO, October 24-29, 1999.

[25] Anum Puri and Pravin Varaiya, "Decidability of Hybrid Systems with Rectangular Differential Inclusions," Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

[26] Peter J. G. Ramadge and W. Murray Wonham, "The Control of Discrete Event Systems," *Proceedings of the IEEE*, v77, n1, January 1989.

[27] Steve Vestal, "An Architectural Approach for Integrating Real-Time Systems," *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997.

[28] Steve Vestal, "Linear Hybrid Automata Models of Real-Time Scheduling and Allocation in Distributed Heterogeneous Systems," Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, 1999.

[29] Steve Vestal, "A New Linear Hybrid Automata Reachability Procedure," Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, 1999.

[30] Steve Vestal, "Formal Verification of the MetaH Executive Using Linear Hybrid Automata," Honeywell Technology Center, Minneapolis, MN 55418, December 1999.

[31] Jin Yang, Aloysius K. Mok and Farn Wang, "Symbolic Model Checking for Event-Driven Real-Time Systems," *ACM Transactions on Programming Languages and Systems*, v19, n2, March 1997.

Orpheus: A Self-Checking Translation Tool Arrangement for Flight Critical Hardware Development

David Greve[*]   Matthew Wilding[*]   Mark Bickford[†]   David Guaspari[†]

Abstract

We describe *Orpheus*, our vision for a development and verification environment for flight critical hardware devices. Orpheus provides an arrangement of translation tools that are self-checking and that integrate synthesis, high-speed simulation, and formal analysis. Implementation of the Orpheus architecture would allow tight integration of these formerly distinct activities and facilitate the use of formal analysis in flight-critical system certification. Further, flexibility in the choice of design representation provided by Orpheus would support both current design practice and hardware/software codesign. This paper describes the notion of self-checking tools, the Orpheus tool architecture, and how commercially-available tools could be used to implement such a system.

1   Current Practice

1.1   Background

Certification of flight critical systems is today a labor-intensive, manual process. Verification and certification of flight critical software and application-specific integrated circuits (ASICs) require an almost heroic effort of intense inspections and process documentation. The complexity of systems and devices will increase, because increases in cockpit automation and application integration offer important safety benefits, and because astonishing improvements in digital computing technology can potentially improve performance and decrease cost. The current approach to verification and certification may not be adequate in the face of this increased complexity. In order to reap fully the safety benefits of these technological advances we must develop new methods for verification and certification of flight critical devices.

Several recent developments permit a superior approach to verification and certification. First, flight critical ASICs can now be developed using standard hardware description languages (HDLs) because recent advances in equivalency-checking tools provide an independent check that synthesis preserves functional correctness. Second, theorem proving tools have emerged that enable mechanical formal analysis of device properties. Third, translation tools are emerging that allow the integration of mathematical analysis into the conventional fabrication/simulation-based development environment.

1.2   Flight Critical HDL Use

Modern hardware devices are typically developed using one of several hardware description languages (HDLs), such as Verilog or VHDL.

---

[*]Rockwell Collins, Inc. Advanced Technology Center, Cedar Rapids IA
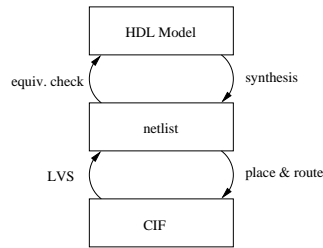
[†]Odyssey Research Associates, Ithaca NY

Figure 1: Fabrication toolsets for flight critical HDL provide self-checking

Figure 2: Designers typically build two device models

In the area of flight critical hardware, however, this has been the case only within the last few years. The delay in adopting these design techniques has been a result of concerns about the reliability of the process by which an implementation expressed in an HDL is used to fabricate the actual device. The complexity of HDLs means that tools that manipulate HDL designs are complex. As a result, the move toward using standard HDLs was hindered because requirements could not be traced to the device without trusting the synthesis tools and supporting libraries.

Fortunately, tools now exist that allow highly-dependable HDL fabrication. Figure 1 shows how 4 fabrication-oriented tools can be used to make the fabrication process immune from corruption by a fault in any single tool. A synthesis tool converts an HDL design into a netlist, and a place-and-route tool converts the netlist into CIF data that can be fabricated. The CIF data is checked against the netlist using an LVS (layout-versus-schematic) tool. The netlist is checked against the VHDL model using equivalence-checking tools.

The dependability of the connection between the design and physical device afforded by an independent tool chain as presented in Figure 1 has changed how flight critical hardware is developed. Incorporation of this in-
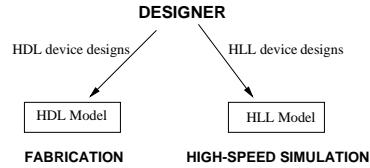
novation into the development process has allowed developers of airborne hardware to benefit from modern design practices such as synthesis and optimization.

1.3 Device simulators

It is commonly the case that a high-speed simulator is developed in parallel with an HDL model of a device. There are several reasons for this.

- Execution of the VHDL model is often too slow to support testing activities. This is especially true for large test suites such as are typical for regression testing.

- Software or other parts of the system that rely on the device must be developed before the HDL model is complete.

High performance is critical for device simulators, so simulators of this type are typically constructed using a high-level language (HLL) such as C or C++ for which there are compilers that generate efficient code[1].

---

[1]Multiple simulators are routinely built during device development. For example, a microcoded microprocessor's simulators would typically include both an instruction-level simulator and a microarchitecture simulator. The device simulator we are describing here is a low-level, cycle-accurate simulator.

The required functionality of complex computational devices is typically implemented using a combination of hardware and software, and an early design decision in the development of these systems is where to draw the line between these two kinds of implementations. The distinction between hardware and software in implementions adds complexity to these systems, since it requires that an interface be defined. Furthermore, this interface between hardware and software can change during a design cycle as implementation issues make clearer the tradeoffs between implementing various functions in hardware or in software. It would therefore be desirable to develop hardware and software using the same languages and tools, and delay decisions about the exact form in which they will be implemented. Designing could be done, for example, using C. Functions whose design will ultimately appear in hardware can be fabricated using the HDL representation. This has the potential to simplify development efforts since no hardware/software interface need be considered during development.

Figure 2 shows the artifacts resulting from current practice: two models that are expected to be identical in substance but that are written in different languages. This is typical of the current state-of-the-art design practice for airborne hardware devices.

2 Formal Analysis

Current certification processes provide some hard-to-quantify assurance that critical airborne hardware devices meet their requirements. Teams of inspectors "walk through" a design, assessing whether the implementation indeed meets the stated requirements. This process generates a paper trail that documents the level of effort of the inspectors and ensures that all relevant parts of the design have in fact been examined against the requirements. For complex designs this type of examination is very labor-intensive, but there is currently no viable alternative. Even so, the quality of the device is, to a large extent, measured indirectly via the inspection process.

Several aspects of the current process for developing and certifying safety-critical devices are not ideal. It would be better if certification practice measured the quality of the device directly, rather than measuring the effort applied to the verification. Further, as the trend is toward using more complex devices for critical airborne activities, current verification and certification threaten to become increasingly inadequate. It has long been hoped that mathematical reasoning — rather than careful documentation of the efforts of inspectors — could ferret out design flaws more effectively than manual inspections. The potential for establishing by direct, formal reasoning that a device meets its requirements has obvious appeal, and is increasingly recognized as a viable verification methodology by certification authorities.

Mathematical proofs about computing devices tend to be very complex and detail-laden, which makes them impractical to develop or check by hand. There has been considerable research applied to the development of automated theorem provers that are capable of checking and/or generating mathematical proofs. Leading tools include ACL2, HOL, and PVS, and each is increasingly finding application in industrial settings where safety or wide product distribution makes establishing design correctness imperative. Various verification projects have used theorem provers to analyze computer system models [1, 2, 4, 6, 8, 13, 14, 17]. A dramatic recent example of the possibilities of applying formal analysis to computing systems is the ACL2-checked verification of AMD's Athlon (formally "K7") floating-point operations [16].
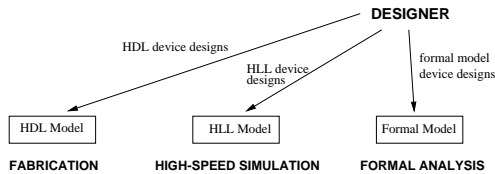
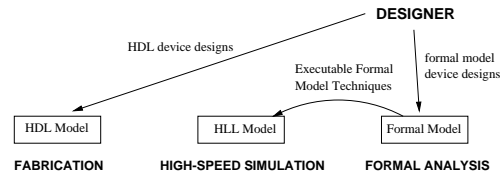Figure 3: Formal analysis requires designers build yet another model.

Figure 4: Executable formal models reduce the number of models

The increased industrial use of automated theorem provers results from improvements in the tools themselves and increased availability of reusable libraries of results [7, 9, 16]. Although we expect these tools will be increasingly common, poor integration with other aspects of the design environment remains an impediment to their adoption [12]. We believe that formal analysis will become pervasive *only when the tools are properly integrated with other aspects of the design environment.*

Figure 3 identifies the artifacts resulting from a process augmented to support formal verification: three models of the same device written in three different languages each supporting its own development or verification activity. In a recent effort Rockwell Collins developed three separate device models — one each for fabrication, simulation, and formal analysis — in order to benefit from each of these activities [11]. However, the high cost of building and maintaining models alone makes this approach unsustainable. Even more troublesome is that the multiple models might be inconsistent with each other, so a property proved about the formal model or the observable behavior of the simulator used to develop other parts of the system might not be reflected in the actual fabricated device.

3 Orpheus

We propose a comprehensive development and verification environment for safety-critical hardware devices called *Orpheus*. In Greek mythology, Orpheus subdues the fearsome, three-headed, dog-like Cerebus. As we have seen, verification and certification of increasingly-complex safety-critical devices requires us to overcome another three-headed challenge: to support device fabrication, high-speed simulation, and formal analysis in an integrated way. Orpheus does so without requiring the development of multiple models that are expensive and possibly inconsistent. The Orpheus approach can be integrated into current approaches for flight critical device development. The Orpheus tools are self-checking, so as to guarantee that no single translation tool can introduce an error into the verification process. The approach allows flexibility of design paradigm: it supports HDL development, hardware/software codesign, and designs derived from formal specification.

3.1 Reducing Three Models to Two

In part to address the issue of multiple distinct models, Rockwell Collins recently developed techniques that allow formal models written in a particular style in the ACL2 logic to be

compiled into C for use as a high-speed simulator [10, 18]. This work effectively combines the formal and simulator models, thereby reducing the number of models from three to two. Figure 4 shows the impact of this innovation. The integration increases confidence in the validity of the unified model, since the *same model* is used both as a simulator and as a target of formal analysis. This important capability—high speed execution of formal logic definitions—has since been added to two theorem proving systems:

- A recent PVS extension provides a translator from PVS functions into Common Lisp. Rockwell Collins' preliminary tests using a version of the benchmark from [18] in PVS 2.3 [15] indicate that execution speeds are within an order of magnitude of the speed of a model written conventionally in C. We expect that PVS will ultimately develop the capability to integrate models expressed in the PVS logic into other tools.

- *Single-threaded objects* have been added to ACL2 2.4 and provide for high-speed execution of certain definitions [5]. Single-threaded objects are an extension of the notion of ACL2 "state" that permits the introduction of user-defined state elements. ACL2 enforces syntactic restrictions on the use of single-threaded objects to guarantee that the optimizations are legitimate. Rockwell Collins' experiments suggest that complex device models can be expressed despite the syntactic restrictions enforced on single-threaded objects, indicating that these restrictions do not make the ACL2 language impractical. Rockwell Collins has recently shown that ACL2 code can be integrated with other tools [18].
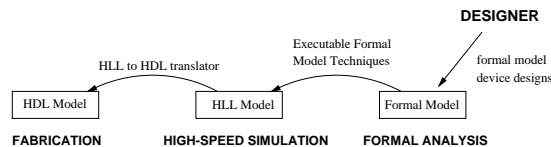

Figure 5: Formal models could provide a single, unified model

3.2 Reducing Two Models to One

An approach has recently emerged that potentially allows the integration of high-speed simulation models and device designs written in HDL. Several commercial tools are now available to translate high-level language (HLL) models into HDL models suitable for fabrication. Among the leading tools of this type are CynApps' C++-to-Verilog converter and C level's C-to-HDL converter, which generates either Verilog or VHDL. These tools promote an HLL-based design methodology that integrates simulation and fabrication. The existence of such tools and the emerging push for system level design and hardware/software codesign practices suggest that the commercial world will continue to develop and improve these tools.

The ability to compile a formal model into a simulation model, as described above, reduces the three models to two. Figure 5 suggests an obvious way to reduce the two models to one, by compiling the simulation model into a fabrication model expressed in an HDL. We discuss in Section 3.4 our initial testing of one of these tools, C level's C-to-HDL tool, and this experience suggests that Orpheus may be a realistic path for some applications.
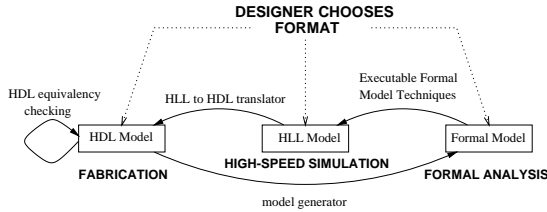
Figure 6: The Orpheus translation circle uses a single model to combine fabrication, high-speed simulation, and formal analysis

3.3 Closing the Loop with Orpheus

Although the tools outlined above allow the translation of high-level artifacts to HDL, and while such a process supports methodology changes that could reduce design errors, there are problems with using these tools for flight critical applications. First, the process outlined above requires that device development be accomplished by constructing a model or specification in formal logic. This is impractical, as hardware development is most appropriately done in an HDL or, in the case of hardware/software codesign, in an HLL.

The second issue is tracibility. Specifically, there must be a way to trace the requirements to the device through the tools. This issue is analogous to the one discussed in Section 1.2 that has until recently bedeviled those who wished to use an HDL for flight critical hardware design. Note that, unlike the fabrication tools of Figure 1, the compilation tools described in Figure 5 are not arranged to be self-checking. As a result the two compilers employed in this process would have to be thoroughly vetted before they could be used in a process for developing flight critical devices, which is problematic.

The Orpheus system addresses these two important issues by adding to the chain an-

other tool, a *model-generator*, that converts an HDL design into a formal model. Figure 6 shows how the Orpheus tools are arranged. The translators form a circle in which a representation is converted in turn into each of the other representations and ultimately back into its original representation language. For example, a device model could be developed in an HDL that supports fabrication. The model-generator then creates a formal model that can be analyzed using a theorem prover. Using executable formal models techniques, the formal model is translated into an HLL model that supports high-speed simulation. Finally, the HLL model is translated back into HDL, and shown to be equivalent to the initial HDL model using an equivalency checker of the kind used in the HDL fabrication process.

There is only a single model, yet three distinct device representations are involved to support the three different uses: fabrication, high-speed simulation, and formal analysis. These three activities support each other, both for model validation and in the fabrication/verification process, because they involve the single model in different ways.

As previously discussed, although the necessary formats can be generated without completing the circle of translations, the question of translation correctness remains open. The certification of flight critical devices must address this issue. If the circle is completed, and the initial design and final design are shown equivalent, then each representation of the design is guaranteed correct so long as at most one of the tools has erred. Much as the fabrication tools diagrammed in Figure 1 are arranged to be self-checking, so too are the Orpheus translation tools. Even if more than one tool errs, the probability of catching the error is still very high since otherwise the multiple mistaken tools would have to fail in ways that mask each other's errors.

This kind of self-checking tool arrangement

provides a very strong argument for the absence of translator-induced errors, and makes this kind of development practical just as modern self-checking fabrication tools permit HDL use in safety-critical devices. Orpheus therefore provides a framework for tight and highly reliable integration of formal analysis, simulation, and fabrication.

3.4 Orpheus Translation Circle Example

To assess the technical feasibility of the Orpheus approach, we have done a small experiment with using current versions of Orpheus components in a manner consistent with the tool arrangement of Figure 6.

As discussed previously, one of the advantages of Orpheus is that it allows a developer to use any of the representations for his device. We might expect VHDL to be the language of choice for hardware designers, while C might be preferred for hardware/software co-design. This experiment begins from a formal ACL2 model of an interrupt controller that forms part of a proprietary device developed by Rockwell Collins. We will navigate around the Orpheus circle to generate a simulation model, a VHDL model, and a second formal model. We have already discussed the benefits accruing from these different representations. The point of the experiment is to observe that the two formal models have sufficient similarities in structure, complexity, and level of detail to indicate that a proof of their equivalence — and therefore a self-check of all the translations — is feasible.

The Common Lisp model of this device uses a macro package developed by Rockwell Collins to ease modeling in Common Lisp. The line

```
(ST. SYNC1 = (& (ST. SYNC0) (HxFFDF)))
```

expresses the following behavioral detail: SYNC1 is an element of the machine state, a register. It is updated each clock tick with the result of applying a constant bit-mask to another state variable, SYNC0.

We also wish to simulate this device. We might choose merely to execute the Common Lisp code. However, there would be two disadvantages to that approach. First, it would be slow. Our experiments with running applicative Common Lisp models indicates that these models execute roughly 100 times slower than equivalent C language models [18]. Second, it is difficult to integrate raw, applicative Common Lisp into other tools.

Rockwell Collins has been working on this challenge for two years and, as described above, has sped applicative Common Lisp execution and integrated this code into other applications. This approach, broadly called "executable formal models," is outlined in two recent publications [10, 18]. Using these optimizations and a Lisp compiler, we generate a C program that executes at roughly the same speed as hand-coded C, and can be integrated with other software. Rockwell Collins in the past has integrated code of this type into various simulation and development environments [18].

We apply this technique to the example above. The line of the resulting C code that corresponds to the given line of Common Lisp reads as follows:

```
V12= (D.SYNC1 = ((((((V11)), Q.SYNC0)) &
  ((-(33))))),((V11)));
```

We also wish to fabricate this device. To do so we have applied a C-to-HDL tool (developed by C level) to convert the auto-generated C program produced into VHDL. Many transformations are done, such as converting variables in the C code that maintain state into registers in the VHDL. The line of C code

shown above translates into the following line of VHDL:

```
D_var (SYNC1_2'range) := (Q_var (SYNC0_2'range)
   and "1111111111011111");
```

Ultimately, we wish to fabricate devices from VHDL using the approach outlined in Figure 1. We applied a Synopsys VHDL synthesizer to this VHDL code, and the result appears correct. As described in Section 1.2, it is this synthesis step from VHDL that current-available tools such as the Chrysalis equivalence checker can verify.

We really want to check much more than this final step. We want to verify that the synthesized design implements the formal model with which we began, so we complete the circle with a model-generator developed by ORA [2, 3]. This tool currently generates a description in first-order logic, rather than ACL2 code, and there are other modest problems related to differences between the VHDL libraries assumed by the C level tool and the libraries assumed by the ORA tool. However, with minor manual changes to the VHDL needed to overcome the library issue, we were able to use the model-generator to construct a specification in first-order logic. In this notation, the value assigned to SYNC1 is:

```
(slice(s.q, 79, 64) and
   flip(shift(vector("1111111111011111"), 78))))
```

The "slice" expression denotes the 16-bit slice of vector q that, by definition, represents SYNC0. The "flip" expression is of course the mask. (It is "flipped" because q has been defined to run down from 79 to 64 rather than up from 64 to 79.) Although it is expressed in a different syntax (i.e. Larch/VHDL rather than ACL2) the generated formal model corresponds term-by-term to the original Common Lisp (ACL2) model.

Sophisticated digital design, simulation and test-generation, and machine-checked formal
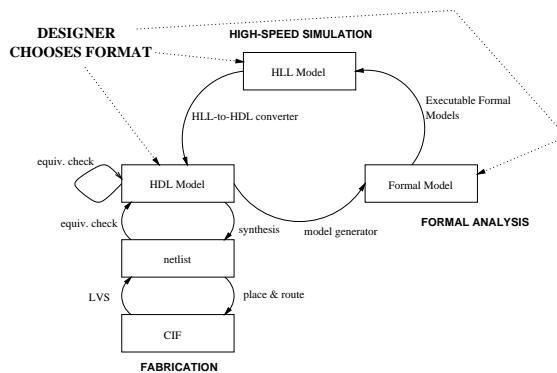

Figure 7: Orpheus Supports and Integrates Each Design Activity

analysis, each individually pose technical challenges that are not solved by using the Orpheus approach. However, Orpheus provides a framework for integrating these separate domains, and we believe that the simple experiment reported here indicates that this novel technical approach can succeed.

4 Summary

Current verification and certification of devices appears increasingly inadequate in the face of increasing complexity of flight critical systems. Figure 7 summarizes the Orpheus approach. Orpheus supports hardware development and hardware/software codevelopment in a way that allows for formal analysis, fabrication, and high-speed simulation. The Orpheus tools are self-checking, just as modern HDL fabrication tools are, to insure their reliability. Orpheus supports a verification approach that forms the basis of a superior certification approach that provides a way to meet this looming challenge.

References

[1] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[2] Mark Bickford. Technical Information Report - Final Report for Formal Verification of VHDL Design. Technical Report TM-96-0025, ORA, July 1996. Delivered to Rome Lab under contract F30602-94-C-0136.

[3] Mark Bickford and Damir Jamsek. Formal specification and verification of VHDL. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[5] Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2, 1999. `http://www.cs.utexas.edu/users/moore`.

[6] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[7] Bishop Brock. ACL2 integer hardware specification (IHS) books, 1998. Standard ACL2 distribution at `http://www.cs.utexas.edu/users/moore`.

[8] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[9] Ricky Butler, Paul Miner, et al. PVS libraries for arithmetic, sets, and graphs. `http://shemesh.larc.nasa.gov/fm`.

[10] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, to appear. `http://www.pobox.com/users/hokie/docs/hsas.ps`.

[11] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[12] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. `http://pobox.com/users/hokie/docs/concept.ps`.

[13] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

[14] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[15] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[16] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor, January 28 1998. `http://www.onr.com/user/russ/david`.

[17] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. `ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/wilding-cav93.ps`.

[18] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/hokie/docs/efm.ps`.

FormalCORE™ PCI/32
A Formally Verified VHDL Synthesizable PCI Core

Bhaskar Bose, M. Esen Tuna and Ingo Cyliax
Derivation Systems, Inc.
Carlsbad, California, USA.
www.derivation.com

Abstract

This paper describes an integrated design methodology for the use of formal methods with existing tools in the context of developing FormalCORE PCI/32. The primary goal is to develop technology for the design and verification of formally verified IP cores that includes all the features, documentation, and support necessary to insure integration into designs with the high degree of reliability provided by the application of formal methods. Validation techniques used in developing these cores include formal specification, formal synthesis, simulation, hardware emulation, theorem proving, and model checking.

1 Introduction

The PCI[6,7] Local Bus is a high performance, 32-bit or 64-bit bus with multiplexed address and data lines. The bus is designed for use as a high-speed interconnect mechanism between peripheral components and processor/memory subsystems.

FormalCORE™ PCI/32 is a synthesizable VHDL[4] 32-bit, 33MHz PCI interface core targeted to programmable hardware. The VHDL description is formally synthesized using our DRS[1,2] formal synthesis system and formally verified using the Verysys PropertyProver model checker to be compliant with the v2.1 PCI specification.

The overall goal of the project is increased assurance by using a variety of formal methods technologies in concert to attack a practical problem. We have developed the methodology for the design and validation of VHDL cores with a variety of tools that can serve as documentation, and increase assurance. In meeting the primary goal of the project we achieve a reduction in the development time as well. Once the design flow was in place, correcting specification bugs and rechecking the properties was a routine task rather than a challenge.

A key benefit to this approach is that it allows for the deployment of formal methods into current engineering practice via pre-designed, pre-verified components that meet the stringent reliability and safety requirements that are necessary in avionics and space applications. These components can then be integrated into larger designs providing the building blocks for complex designs and the foundation for design reuse.

In developing the FormalCORE technology we rely heavily on both formal and traditional design and verification tools. We recognize at the early stages of planning that a comprehensive approach to the integration of formal verification techniques to an existing design flow is critical to the success of the technology. A well implemented design and verification strategy, incorporating formal techniques at key points in the design flow minimizes the likelihood of design errors.

2 The PCI Bus Protocol Standard Revision 2.1

The PCI bus specification was first developed by Intel Corporation and was released in June 1992. It was intended to define an industry standard for local bus architectures. Revision 2.1 became available in early 1995 and is managed by a consortium of industry partners known as the PCI Special Interest Group. The specification is a 282-page English language document describing the protocol, electrical, mechanical, and configuration specification for PCI components and expansion boards.

The PCI specification defines two possible PCI agents, a *master* and a *target*. The master is the device that initiates a transfer. The target is the device currently addressed by the master for the purpose of performing a data transfer. The master and target state machines are independent. However, a master device must incorporate a target device for the purpose of responding to system configuration requests.

The minimum PCI compliant device satisfies the requirements of a target-only device. This device requires 47 pins and can only respond to a master initiated transaction. A master device requires two additional signals, (REQ# and GNT#), for it to handle data and addressing, interface control, arbitration, and system functions. Figure 1 illustrates the required and optional signals for a PCI compliant device. The signals on the left are required pins for target and master devices. The signals on the right are optional pins and are used to support the 64-bit extension to the specification, exclusive access (LOCK#), interrupts, cache support, and the JTAG (IEEE 1149.1) boundary scan interface.
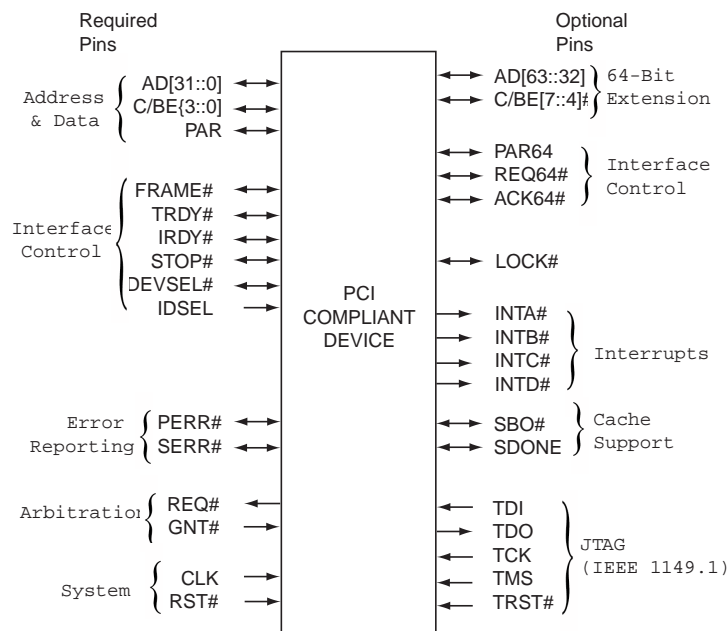
Figure 1: PCI Compliant Device Signals

The heart of the PCI Bus Protocol is the burst transfer mechanism. A burst transfer consists of a single address phase followed by two or more data phases. The start address and transaction type are issued during the address phase. The target device latches the start address into an address counter and is responsible for incrementing the address from data phase to data phase. Figure 2 illustrates a sample read transaction.

A basic bus cycle involves the FRAME#, IRDY#, TRDY#, C/BE# control signals as well as the multiplexed address/data AD[31:0] lines and the parity signal PAR and DEVSEL#. The bus cycle starts with an *address phase*. This is the first clock after FRAME# is asserted by the master. During this cycle, the address lines carry the desired address and the C/BE# signals the bus command. Bus commands encode the address space and direction of transfer. There are also some special bus cycles, like interrupt acknowledge and various memory transfer modes. After the address phase, the master goes into the *data phase*.

The addressed target, will then decode the address to determine if it needs to take the bus cycle. It can decode either as a fast/medium/slow decoder, which are 1,2,3 cycles after the address phase. Once it has decoded and accepted the bus cycle, it asserts the DEVSEL# signal to signal that it will take the bus cycle. When the master has sent data via the AD[31:0] or when it is ready to receive data, it will assert the IRDY# signal. The target indicates its readiness with the TRDY# signal. Only when the TRDY# and IRDY# signals are both asserted, will a data transfer take place. Otherwise wait states are inserted. The master controls how much data is
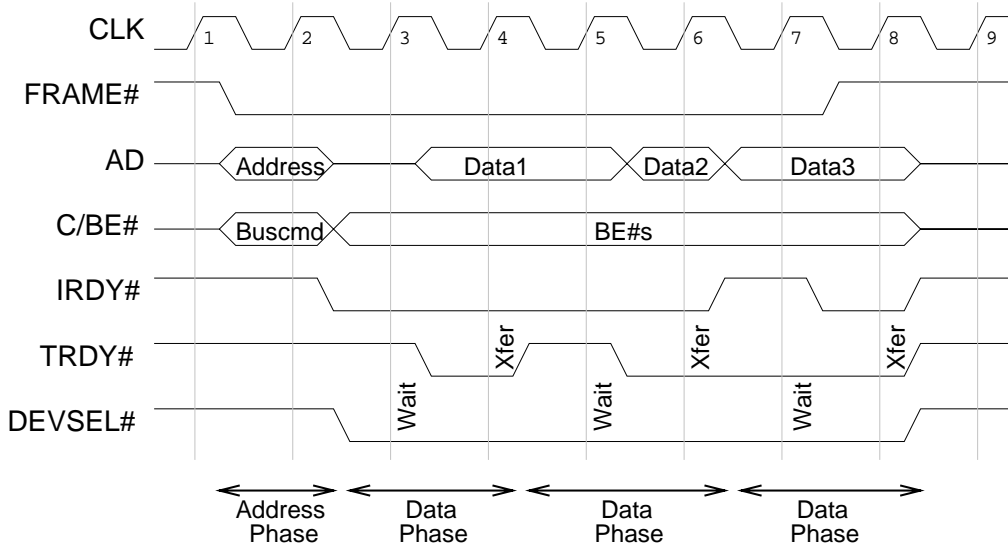
Figure 2: PCI Timing Diagram

transferred. When it is done transferring data, it will de-assert FRAME# on the last data phase. When the target sees neither FRAME# or IRDY#, the master has finished.

The target uses the STOP# signal to signal the master that it has to terminate the current transaction. The PCI Target asserts combinations of TRDY#, DEVSEL#, and STOP# to signal different termination conditions. The PCI protocol is specified in plain English. The specification contains rules such as:

"Data is transferred when IRDY# and TRDY# are asserted."

"When either TRDY# or IRDY# is deasserted, a wait cycle is inserted and no data is transferred."

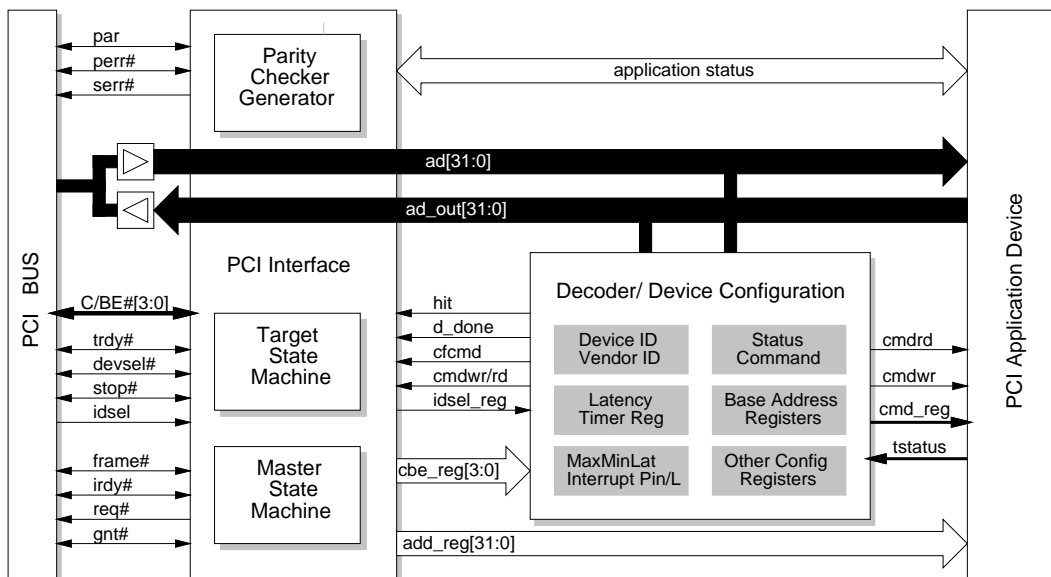## 3 FormalCORE PCI/32 -- A Formally Verified PCI Interface

Figure 3: FormalCORE PCI/32 System Architecture

FormalCORE PCI/32 is a synthesizable VHDL 32-bit, 33 MHz PCI interface targeted to programmable hardware, formally verified to be compliant with the v2.1 PCI specification. Figure 3 is a block diagram of the FormalCORE PCI/32 system architecture.

The design is composed of three primary modules. A PCI Interface Module, Decoder/Device Configuration Module, and PCI Application Module. The PCI Interface Module is the primary interface to the PCI bus and user application. It contains the Target and Master state machines, parity circuit, and implements the bus protocol. The Decoder/Device Configuration Module contains the PCI configuration registers and address decode circuitry. The PCI Application Module is a stub module defining a backend interface. This module is used to integrate the user's application into the PCI core. It is not specified in v2.1 since it is dependent on the specific device. For example, the Application Interface would vary widely between a video device and a modem. This partitioning allows us to swap different application backends to the existing core with minor modifications.

4 Design and Verification Tools

The software tools comprising our design and verification suite included:

- DRS (Derivational Reasoning System), formal synthesis system from Derivation Systems, Inc. to develop high-level formal behavioral specification, high-level simulation, hardware emulation, and formal synthesis to VHDL and gate-level netlist. We use DRS to derive a structural specification from the top-level behavioral description, synthesize VHDL code and PVS theories. The system was also used for functional simulation of the top-level specification, and as the interface to hardware emulation of the synthesized design.

- PVS[5] (Prototype Verification System) from SRI for validating safety and liveness properties of the top-level behavior specification.

- Verysys PropertyProver[8] and StructureProver[8]. PropertyProver is a state-of-the-art model checker that can verify model properties at the Behavior, RTL and Gate levels. StructureProver is a high-performance, high capacity equivalence checking tool that can be used at the RTL and Gate levels. The Verysys tool suite was chosen for its support of the IEEE 1076 VHDL standard and hierarchical verification. In addition, PropertyProver generates an input sequence and a VHDL testbench for counter-examples. The built in VHDL simulator can be used to simulate the counter example.

- Verysys Circuit Interface Language[3,8] to formally describe circuit properties. These properties are described using temporal relationships between the various input and output ports of the circuit. CIL is used to describe the PCI Compliance Model to validate the VHDL core. Properties are written in an assumption-commitment style. Predicates in the logic are written using VHDL syntax.

- ModelSim from Model Technologies for VHDL simulation. ModelSim is chosen because it is a full featured VHDL simulator providing accurate modeling of the language. It provides a rich set of features.

- Xilinx Foundation Express[9] for VHDL synthesis, gate-level timing analysis, gate-level simulation, and FPGA programming. Foundation Express incorporates the Synopsys Express VHDL compiler and Aldec gate-level timing analyzer and simulator. Foundation Express provides a low-cost, comprehensive solution for FPGA programming. The entry to the tool can be VHDL, Verilog, Schematic entry, or gate-level netlist. Xilinx offers a variety of chips that are PCI compatible and is an industry leader in programmable hardware.

5 Design and Verification

The primary design criteria for FormalCORE PCI/32 was to synthesize a VHDL model from DRS that would run at 33Mhz, optimized for size, and compatible with the various VHDL level tools. The generated VHDL had to be compatible with the Verysys model checker, Synopsys FPGA Express compiler, and Model Technologies VHDL simulator.

From the PCI Specification document, we developed a formal PCI compliance model in CIL, Verysys circuit interface language. These properties are described using temporal relationships between the various input and output ports of the circuit. They are extracted from the PCI rules in the specification document.

Formal design and verification is a theme that runs throughout the lifecycle of the FormalCORE PCI/32 development. Verification tools were used continuously once the design reached a state where the tools were applicable. DRS synthesis served as a backplane for the design flow. Changes in the design were reflected in the DRS top-level specification and the VHDL was re-synthesized.

The need for verification in this project was two fold. First the specification had to be proven to meet the PCI specification properties. The correctness of the specification in derivation is assumed, not proven. Secondly, even though DRS guarantees correctness of its transformations in the original specification, the state representation and the VHDL translation are not reasoned about. Therefore, the generated VHDL had to be shown to satisfy the same properties as the initial DRS specification.

Once a stable DRS specification was established, PVS was employed to validate the DRS top-level description. DRS was then used to derive a structural description from the top-level specification and generate VHDL. Verysys model checker, Model Technologies VHDL simulator, and Synopsis VHDL compiler were used for VHDL property verification, simulation and synthesis. The synthesized gate-level design was simulated with the Xilinx simulator.

Several modes of validation were always running in parallel. We performed functional simulation of the top-level and structural DRS descriptions. We simulated the design both at the VHDL and gate-level. Formal verification at the high-level, and formal verification at the VHDL level were used to validate properties of the design. The design flow (Figure 4), from high-level formal specification to running hardware can be characterized as five stages of design.
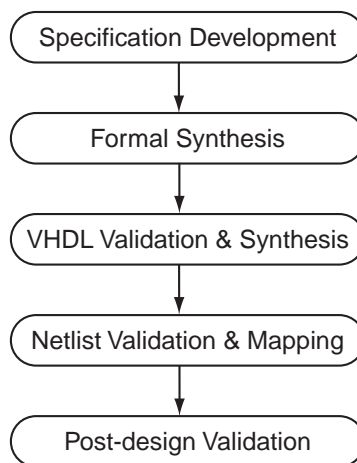

Figure 4: Design Flow

The design flow reflects a top-down design methodology. It provides for the formal specification and verification at an abstract behavioral level, and a systematic process to refine the design to a concrete VHDL implementation. The design flow incorporates formal and traditional validation techniques. The use of DRS

and formal methods contributes to the soundness of the specification and implementation, and VHDL provides an industry standard language to interface to other tools. Figure 5 details the design and verification flow and the tools used. Shaded boxes denote formal tools. Shaded ovals denote formal specifications. Clear boxes denote traditional design tools.
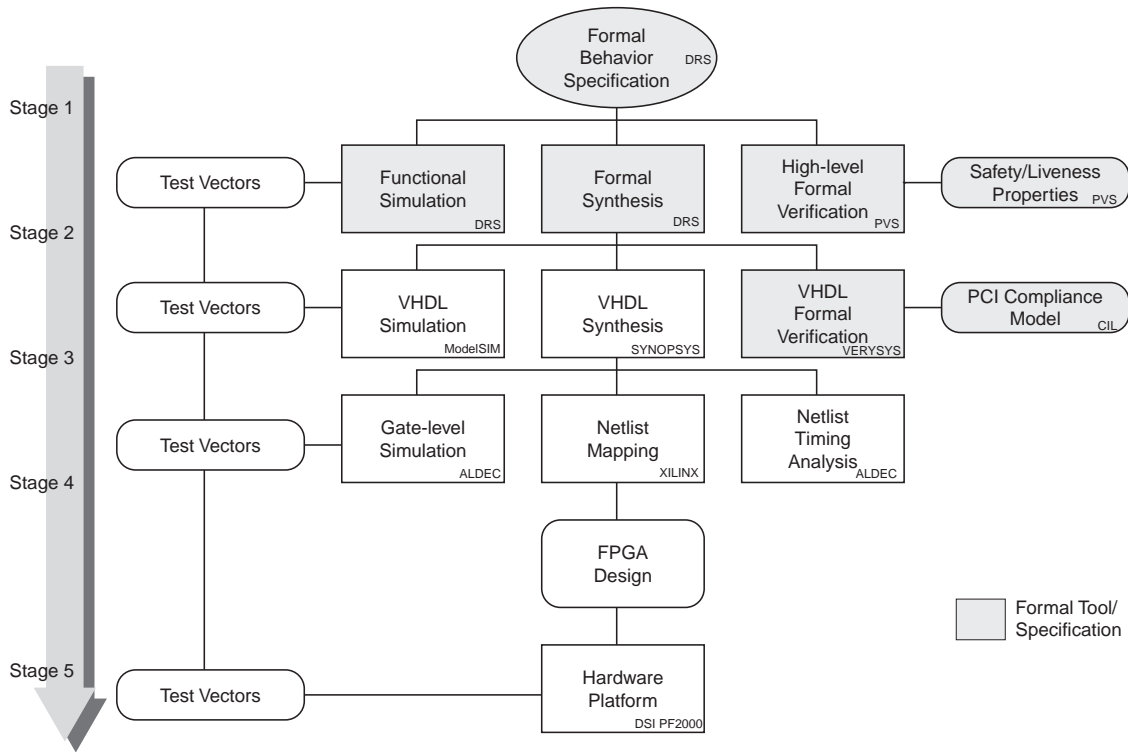

Figure 5: Design and Verification Flow

5.1 Specification Development

In the first stage the top-level behavior specification is developed and validated using simulation and formal verification. Verification begins early using the DRS functional simulator. A high-level behavioral model is written in DRS and run against test vectors. This behavior model becomes the reference model for all subsequent verification and synthesis.

```
[b_busy
 (lambda (add_reg cbe_reg idsel_reg ...)
   (let ([devsel_lo_o HI]    [serr_lo_o HI]    [trdy_lo_o HI]
         [stop_lo_o (not (and (or t_abort term)
                              (or wrcmd (and rdcmd tar_dly))))]
          ...)
     (if (and (or frame (not d_done)) (not hit))
         (b_busy ...)
         (if (and (or frame irdy)
                  (and hit (and (or (not term) (and term ready))
                        (or free (and locked l_lock_lo)))))
             (s_data ...)
             ...)))))]
```

Figure 6: Code fragment for Target Interface b_busy state

The top-level DRS specification is a collection of communicating state machines. Each state machine is defined in terms of a set of mutually recursive function definitions. A fragment of the b_busy state of the Target Interface is depicted in Figure 6. Because of the reactive nature of the protocol specifications, the specification is written at a fine level of granularity. The specification captures the complete synchronous behavior of the PCI core circuit.

DRS descriptions were written for the master and target state machines along with their lock machines, the configuration/decode circuit, the parity circuit, and a basic application backend. The chip-level glue-logic was also written integrating all the modules into a single core. Figure 7 illustrates the modules and their interconnectivity.
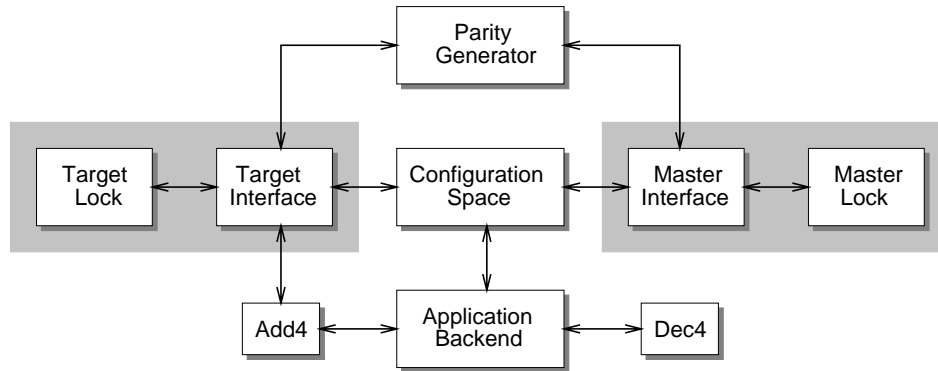

Figure 7: DRS Specification Hierarchy

An abbreviated form of the top-level DRS description is shown below. The module instantiations are show in bold.

```
(define mchip
  (lambda (cbe_lo ad par idsel frame_lo irdy_lo trdy_lo stop_lo lock_lo
           perr_lo serr_lo devsel_lo gnt_lo)
    (stream-letrec
        ([tsbuf  (lambda (o oe) (if oe o #\z))]
         [parity (lambda ((d0 ... d31) (c0 c1 c2 c3)) (b-xor ...))]
         ...)
      (letrec (...) ;; -- Component descriptions
  (system-letrec
    ([(add_reg cbe_reg idsel_reg ...) (target_xface ad cbe_lo par ...)]
     [(conf_data hit d_done cfcmd ...) (target_conf add_reg ad ...)]
     [(ad_o cbe_lo_o tstatus t_abort ...) (backend mxfer add_reg ...)]
     [(lock_lo_oe own_lock ...) (master_xface par idsel frame_lo ...)]
     [(tfree tlocked) (target_lock frame_lo lock_lo l_lock_lo hit ...)]
     [(lock_free) (master_lock frame_lo lock_lo own_lock)]
     [par_c (parity (if par_dir ad_out ad)
                    (if par_dir cbe_lo_out cbe_lo))]
     [ad_out (tsbuf32 (if (b-or ior cmdwr) ad_o conf_data)
                      (b-or ad_oe m_ad_oe))]
     [frame_lo_out (tsbuf frame_lo_o frame_lo_oe)]
     [irdy_lo_out (tsbuf irdy_lo_o irdy_lo_oe)]
     ...)
  (list ad_out cbe_lo_out par_out frame_lo_out trdy_lo_out irdy_lo_out
        stop_lo_out perr_lo_out serr_lo_out devsel_lo_out req_lo
        lock_lo_out ...))))))
```

The DRS behavior model is automatically translated into a PVS theory to perform formal verification. The primary goal is to verify that the specification satisfies a set of high-level safety and liveness properties. Inconsistencies in the top-level specification found by PVS are then manually corrected in the DRS specification.

The DRS->PVS translator generates a PVS function corresponding to the state to state transition of the DRS model. PVS was used to analyze the functional properties of the specification. For example, we show that the `trdy_lo_o` signal is asserted only when `t_abort` is false and `ready` is true with the PVS theorem:

```
 trdy_on_write: THEOREM
         (FORALL (t_abort: bit, tar_dly: bit, ready: bit):
             compute_trdy_lo(write, t_abort, tar_dly, ready) = true_lo
                 IFF  NOT(t_abort) AND ready).
```

The `From_idle_goto_busy` theorem states that from IDLE, only when `frame_lo_i` is asserted, the Target sequencer goes to the BUS BUSY state.

```
From_idle_goto_busy: THEOREM
  (FORALL ((frame_lo_i: bit), (irdy_lo_i: bit), (trdy_lo_i: bit),
           (stop_lo_i: bit), (perr_lo_i: bit), (serr_lo_i: bit),
           (devsel_lo_i: bit), (ready: bit), (t_abort: bit),
           (term: bit), (state: state_type),
           (cbe_reg: [bit, bit, bit, bit]), (tar_dly: bit),
           (par_dat: bit), (par_en: bit), (par_i: bit),
           (perr_dat: bit), (r_perr: bit), (rperr_reg: bit)):
               idle(frame_lo_i, irdy_lo_i, trdy_lo_i, stop_lo_i,
                    perr_lo_i, serr_lo_i, devsel_lo_i, ready,
                    t_abort, term, state, cbe_reg, tar_dly, par_dat,
                    par_en, par_i, perr_dat, r_perr, rperr_reg)
                = bus_busy
                IFF (frame_lo_i = true_lo))
```

Many of the functional properties verified in PVS were also verified in the Verysys model checker. Both PVS and Verysys were useful in finding errors in the design. Early in the design process, we used sample equations from the PCI specification as a guide to developing the DRS specification. PVS uncovered overlaps in some of the equations. A set of conditions would satisfy two different equations.

5.2 Formal Synthesis using DRS

In the second stage, formal synthesis is used to manipulate the design hierarchy and derive a VHDL description from the top-level behavior specification. This process requires manual guidance from the designer. DRS provides automated support for transforming the specification to a concrete implementation, however, design decisions are made by the designer. DRS maintains correctness and does not allow the introduction of errors. The key benefit is that it provides the designer with direct control over the synthesis process.

DRS can manipulate a large class of designs including datapath and/or control oriented circuits. The PCI specification is a control-dominated circuit geared for bus protocol. DRS allowed us to manipulate the PCI design hierarchy providing a means of managing the complexity of the verification and defining the synthesized VHDL modules. We found that manipulating the design hierarchy of the VHDL would impact how the VHDL compiler would synthesize the design. Hierarchy played an important role in the speed of the synthesized circuit. The synthesizer did better when the design was in logically organized major blocks than a totally flat description or when there were many small modules instantiated in the larger ones.

The derivation was limited to obtaining a structural specification and generating the support modules from DRS libraries. We added four valued logic libraries to DRS. This enabled DRS to generate tristated input/output signals which are essential in a bus implementation.

The following table summaries the number of derivation steps, the specification and implementation size for each of the modules, along with the top-level mchip module.

	Add4	Dec4	Backend	Txface	Mxface	Tconf	Tlock	Lock	mchip
DervSteps	14	14	15	128	77	30	19	9	55
Spec Size	899	899	4440	12800	13906	5507	732	347	6209
Imp. Size	3194	3434	14286	12554	7471	10632	563	416	55790
VHDL Size	2669	2849	11909	11832	8425	8792	1002	858	48973
VHDL Comp	2669	2849	5493	9163	8425	8792	1002	858	9722

DRS and VHDL sizes include all the modules that make up the component. The component VHDL size lists only the size of that component. All sizes in bytes.

5.3 VHDL Generation, Validation and Synthesis

5.3.1 VHDL Generation

Once the design is refined to a concrete architecture in DRS, VHDL files are automatically generated and the VHDL Validation and Synthesis process begins. Model Technologies ModelSIM is used to simulate the VHDL. To streamline our simulation environment, we created interfaces from the DRS simulator to the VHDL and netlist simulator. This provided us the ability to localize our test vector generation within the DRS framework, and then automatically generate test vectors to validate the netlist generated by the VHDL compiler, and VHDL simulator.

The tools we used understood only a restricted subset of the VHDL language. We had to tune the VHDL generation toward the common syntax used among these tools. For example, the Verysys VHDL type checker could not resolve predicates of the form: `ad(1:0) = "00"`. The DRS VHDL generation had to produce expressions of the form: `ad(1) = '0' and ad(0) = '0'`.

The VHDL compiler infers registers in a design depending on the way the code is written. Rather than an implicit mechanism to infer registers, we controlled the introduction of registers in the design by an explicit register entity, that served as a state holding abstraction and directly corresponded to DRS registers. The combinational logic is expressed as simple equations of assignments and entity instantiation. The resulting VHDL follows the intended implementation architecture closely.

To improve performance we experimented with several hierarchical design layouts. When flattening hierarchies the circuits were logically equivalent. However the circuit speed varied widely.

In generating VHDL, DRS constructs had to be mapped carefully over to VHDL constructs to ensure the semantics of the DRS expression is maintained. One problem we ran into was generating VHDL code for nested DRS if-then-else expressions. These expressions cannot be converted to selected signal assignments (WITH statement) unless the else branch guard is ANDed with the negated test expression. However, conditional signal assignment behaves just like a nested DRS if-then-else expression and is used instead of the WITH statement. In fact, the Verysys model checker uncovered this bug in the DRS VHDL generation.

5.3.2 VHDL Validation

The Verysys model checker is used to validate the VHDL against the PCI compliance model written in CIL. The underlying model checking technology used by the Verysys tools is the Siemens Circuit Verification Environment (CVE) [3]. The system is a BDD based symbolic model checker. It supports EDIF and VHDL, and generates VHDL test benches for counter examples.

Circuit properties are written in CIL (Circuit Interval Language). CIL formulae are built up from timed predicates that consist of a state predicate and a temporal specification. The temporal specification describes when the machine should be in a state that satisfies the state predicate. The state predicate is given in the subset of Boolean expressions in VHDL. The temporal specifications refer either to a particular point of time, or to a whole period. A point of time is specified after the keyword `at`. A period is specified by an interval, which is a uniform representation of three different types: `[t1, t2]`, refers to the time between `t1` and `t2` inclusively, `[t1, infinite]`, refers to `t` and every point after `t`, `[t, p]`, refers to the time between `t` and the last point of time before the state predicate `p` is satisfied for the next time.

An interval is preceded by `during` or `within` to specify whether the state predicate holds during the whole period or at least once in the interval. Times are either integer constants or defined relative to a variable `t` which is universally or existentially quantified by `always` and `finally`.

As an example, we express the property that the "Target Sequencer will never deadlock" as:

```
theorem target_deadlock;
  assume: (set = '0' during [0, infinite]);
  prove: always(possibly state = idle within [t, infinite]);
end theorem;
```

The assumption eliminates the reset state, and the proof guarantees that no matter what state the Target Sequencer is in, there exists a path to the idle state.

We prove that the Target Sequencer that implements the sustained tristate signals correctly with the following theorem:

```
theorem target_sustained_tristate_trdy;
  assume: (set = '0' during [0, infinite]);
  prove: always((trdy_lo_oe = '1' at t-1) and
                (trdy_lo_oe = '0' at t)
                     implies (trdy_lo_o = '1' at t-1));
end theorem;
```

In order for a signal to adhere to the sustained tristate property, it must drive the signal high one clock cycle before tristating the signal.

Most of the effort at this stage was spent developing the PCI compliance model. It was critical to be able to ask the "right" question. This was difficult since we had no prior understanding of the PCI protocol. Once the protocol was understood, writing the CIL properties from the PCI specification was fairly straight forward and the actual running of the model checker was automatic. Counter examples generated by the model checker were validated with the ModelSIM simulator at the VHDL level as well as in the DRS simulator. This capability allowed us to pinpoint if the problem was in the top-level DRS specification, VHDL generation, or VHDL code.

The design environment of this project consisted of two dynamic aspects: on the one hand the engineering process and on the other the formal process. From initial specification to working hardware the model checker did not find any errors that our hardware engineer did not find using traditional techniques. The model

checking was lagging behind in this process. Errors uncovered by the engineering process led to revisions in the DRS specification.

After working hardware was achieved the model checker started finding errors in the design that the simulator did not uncover. This was due to three facts. First, the simulation tests were not exhaustive. Second, hardware and specification reached a level of maturity where the core appeared to work for most cases. Thirdly, we developed a better understanding of the PCI protocol.

The compliance model provides a comprehensive formal validation of PCI compliance and becomes extremely valuable in providing exhaustive analysis of the VHDL model. Inconsistencies found in the PCI specification were documented, and design decisions were made to resolve them.

5.3.3 VHDL Synthesis

The VHDL files are input to Synopsys FPGA Express compiler for netlist synthesis. The issue in this process is that minor changes to the VHDL would result in significant performance changes in the synthesized netlist.

5.4 Netlist Validation and Mapping

The next stage involves simulating the netlist, and using the model checker to validate that the VHDL synthesis has not introduced any errors. Timing analysis is also done at this time. The netlist is then mapped to the appropriate target technology for hardware programming. At this stage, the logic netlist is validated using the Aldec netlist simulator. Test vectors written for the DRS architectural simulation are used at the VHDL and netlist level.

The logic netlist is formally verified using Verysys StructureProver. This ensures that the synthesized netlist behaves identical to the VHDL model in order to eliminate the possibility that logic bugs that would be introduced during VHDL synthesis. The equivalence checker compares the finite state machine models of the VHDL source and EDIF files of the synthesized netlist. There were no errors in the VHDL synthesis.

The Xilinx mapper then synthesized the appropriate configuration files for the target device.

5.5 Post-design Validation

Traditional hardware techniques were used for post-design validation.

The DRS Functional Test Environment (FTE) was used for hardware emulation of the synthesized PCI core. The FTE consists of the DRS simulation environment communicating with a Ampro EBX form factor Pentium based single board computer (SBC) and the PF2000 PC/104 FPGA module. The synthesized core is downloaded on to the PF2000 FPGA module. Then the DRS simulator drives the inputs of the circuit, single steps the clock, and samples the outputs, displaying them in the DRS simulator. In contrast to the functional simulation of the model in DRS, the FTE was used to compare the functional behavior of the model to that of a design that has been processed by implementation specific back end tools.

The core has been targeted to Xilinx XC4000 and Virtex family of FPGA devices. A working prototype is running in two different environments. The first system is a standard PCI/ISAbus AT motherboard with a AMD-K5 processor clocked at 133MHz. It includes an NE2000 compatible ISAbus based Ethernet card and a PCI VGA card.. The second system is an AMPRO PC/104+ system consisting of a Ampro EBX form factor Pentium based single board computer (SBC). Both systems are configured with 32Mb of memory and runs Linux RedHat 6.0, which is based on a 2.2.5 Linux kernel.

6 Conclusions

The methodology developed to build the FormalCORE PCI/32 is an example of how formal tools and traditional simulation and synthesis tools are integrated for the design and validation of VHDL IP cores. These cores can then be integrated into larger designs providing the building blocks for complex designs.

The FormalCORE PCI/32 and associated PCI compliance model consists of pre-designed, pre-verified VHDL components that can be integrated into larger designs and a validation suite providing exhaustive analysis of the VHDL models using a commercial model checker. The core has been designed to be flexible and can be adapted to a variety of designs with little or no modification to the VHDL or compliance model.

One observation is in the early stages of this project, traditional techniques led the design process. The ModelSIM VHDL simulator, Aldec netlist simulator, and hardware Logic Analyzer were used to debug the design. The model checker did not find any errors that either simulation or hardware debugging did not catch. The traditional techniques were satisfactory in achieving a working prototype. In the later stages of the project, the formal techniques led the design process. The model checker was able to find errors in the design that were not tested for in simulation. Using the DRS system, we were able to routinely make changes to the top-level specification, manipulate the design hierarchy, and re-synthesize the VHDL core. We could then re-validate the core against the compliance model automatically.

Both PVS and the Verysys model checker were useful in developing the PCI core. PVS was used to verify functional properties of the DRS top-level specification. Verysys was used to verify functional and temporal properties of the DRS generated VHDL. The Verysys verification effort was more extensive since the end goal was to develop a verified VHDL PCI core and compliance model.

This work has significantly enhanced our capability to design and validate VHDL cores. The enhancements added to the DRS system are general and can be used to synthesize a wide array of designs.

The future work on this topic is to extend the PCI core and Compliance model to the 64-bit PCI standard, retarget the core to operate at 66Mhz, and update the design to Revision 2.2 of
the PCI specification. In addition, we would like to perform an independent validation of the compliance properties.

References

1. Bose, B. DRS – Derivational Reasoning System: A Digital Design Derivation System for Hardware Synthesis. In *Safety and Reliability in Emerging Control Technologies* (1996), S. Zaleswki, Ed., Elsevier.
2. Bose, B., Tuna, E., and Choppell, V., A Tutorial on Digital Design Derivation Using DRS., In *Formal Methods in Computer-Aided Design*, M. Srivas and A. Camilleri (eds.), Springer, 1996, pp. 270-274.
3. Bormann, J., Lohse, J., Payer, M., and Venzl, G., Model Checking in Industrial Hardware Design, In *Proceedings 32nd Design Automation Conference*, pp. 298-303, June 1995.
4. *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., New York, IEEE Std 1076-1993 edition, 1994.
5. Owre, S., Rushby, J., and Shankar, N. PVS: A Prototype Verification System. In *The 11th International Conference on Automated Deduction (CADE)*, Volume 607 of Lecture Notes in Artificial Intelligence (Saratoga, NY, June 1992), D. Kapur, Ed., Springer-Verlag, pp. 748-752.
6. PCI SIG. PCI Local Bus Specification, Revision 2.1., June 1995.
7. Shanley, T., and Anderson, D. *PCI System Architecture*. Addison Wesley, ISBM 0-201-40993-3.
8. Verysys Design Automation: Verysys Prover Environment Manual, VT-0050, Version 2.1, Rev. A, Verysys Design Automation, 42707 Lawrence Place, Fremont, CA 94538.
9. Xilinx, Foundation Series Software User Manual, Version 1.5i., Xilinx, 2100 Logic Drive, San Jose, CA 95124.

Structuring Formal Control Systems Specifications for Reuse: Surviving Hardware Changes*

Jeffrey M. Thompson, Mats P.E. Heimdahl and Debra M. Erickson
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS; 200 Union Street S.E.
Minneapolis, MN 55455 USA
+1 (612) 625-1381
{thompson,heimdahl,erickson}@cs.umn.edu

Abstract

Formal capture and analysis of the required behavior of control systems have many advantages. For instance, it encourages rigorous requirements analysis, the required behavior is unambiguously defined, and we can assure that various safety properties are satisfied. Formal modeling is, however, a costly and time consuming process and if one could reuse the formal models over a family of products, significant cost savings would be realized.

In an ongoing project we are investigating how to structure state-based models to achieve a high level of reusability within product families. In this paper we discuss a high-level structure of requirements models that achieves reusability of the desired control behavior across varying hardware platforms in a product family. The structuring approach is demonstrated through a case study in the mobile robotics domain where the desired robot behavior is reused on two diverse platforms—one commercial mobile platform and one build in-house. We use our language $RSML^{-e}$ to capture the control behavior for reuse and our tool Nimbus to demonstrate how the formal specification can be validated and used as a prototype on the two platforms.

Keywords: Requirements, Formal Models, Requirements Reuse, Control Systems, $RSML^{-e}$

1 Introduction

Reuse of software engineering artifacts across projects has the potential to provide large cost savings. Traditionally, the research in the reuse community has focused on how to construct reusable software components, and how to classify and organize these components into libraries where they can be retrieved for use in a particular application. We know, however, that coding errors are not the main source of problems and delays in a software project; incomplete, inconsistent, incorrect, and poorly validated requirements are the primary culprit [4]. Thus, we hypothesize that reuse of requirements in conjunction with reuse of design and code will provide greater benefits in terms of both cost and quality. In this paper we present an approach to structuring formal requirements models for control systems that make the control requirements reusable across platforms where the hardware (sensors and actuators) may vary. We also illustrate the structuring approach with an example from the mobile robotics domain.

The beginnings of our approach is a high-level requirements structuring technique based on the relationship between *system requirements* and the *software specification*. We developed this structuring technique to enable a software development approach we call *specification-based prototyping* [23] where the formal requirements model is used as a prototype (possibly controlling the actual hardware—hardware-in-the-loop-simulation) during the early stages of a project. Here we present how this structuring approach also enables reuse of the high-level requirements across members of a product family with variabilities in the hardware components. The approach is

demonstrated via a case study in the mobile robotics domain where the desired robot behavior is reused on two diverse platforms—one commercial mobile robot and one built in-house. We use our language RSML^{-e} to capture the desired control behavior for reuse and our tool Nimbus to demonstrate how the formal specification can be validated and used as a prototype on both platforms.

The rest of the paper is organized as follows. Section 2 describes related work on requirements reuse and product families. Then, Section 3 describes our approach to structuring the high-level system requirement and the software specification. Section 4 describes the mobile robotics platforms that we are using as the case study in the paper and presents a simple analysis of their commonalities and variabilities. The requirements of the mobile platforms in the family are presented in Section 5. The refinement of these *system requirements* to a *software specification* is presented in Section 6. In this section we also show how the system requirements are reused across the members of the product family. Finally, Section 7 presents a summary and conclusion.

2 Related Work

The foundations for reuse of can be traced back to the early work on program structure and modularity pioneered by David Parnas and others [3, 20, 21, 22]. This work establishes the basis for reuse: the concept of a self contained module with well-defined interfaces. Nevertheless, the guidelines for how to encapsulate and structure a model (in this case implementations) for reuse is not sufficiently addressed in this early work. Thus, subsequent research in the field of software reuse seeks to further define and provide additional tools and techniques for reuse.

In the area of requirements reuse, Lam *et al.* provides some guidance on specific techniques which can be used by organizations to introduce requirements reuse into their software process [15]. In addition, Lam addressed requirements reuse in the context of component-based software engineering [14]. Our area of interest is more in structuring of specifications to achieve reuse; nevertheless, this work presents some ideas about how to package and specify generic requirements and how to factor requirements into pluggable requirements parts [15]. Of particular interest is the relationship of their work to the product families work being done at Lucent Technologies [2, 24].

Product family engineering is related to the work presented in this paper; in particular, the FAST (Family Oriented Abstraction, Specification and Transla-tion) approach is of interest. FAST provides a process for how to identify commonalities and variabilities across a product family. This commonality analysis can then be used to provide domain specific development tools that will greatly reduce the development costs for later generations of the product family. FAST does not explicitly address the structuring of product requirements. The FAST concepts of the domain analysis and the commonality analysis can, however, be directly applied to our work with formal specifications; FAST provided some of the inspiration for the work presented here.

Little work has been done on how to structure and develop a formal specification in a language such as RSML^{-e}. One notable exception is the CoRE methodology [5, 6, 7] developed by the Software Productivity Consortium. CoRE includes much useful information on how to perform requirements modeling in a semi-formal specification language (similar to the formal SCR defined at the Naval Research Laboratory [12]). Even so, the structuring mechanism proposed in the CoRE guidebook is based on the physical structure of the system as well as which pieces of the system that are likely to change together—these two (often conflicting) structuring mechanisms may or may not be beneficial to reuse. Furthermore, the way in which the structuring techniques achieve reuse is not specified in the guidebook—reuse is not specifically addressed. Our work is based on many ideas similar to those found in CoRE, but we have extended and refined these ideas to address structuring of state-based requirements models to achieve (1) conceptual clarity, (2) robustness in the face of the inevitable requirements changes to which every project is subjected, (3) robustness of the requirements as hardware evolves, and (4) reuse of models as well as V&V results.

3 Structuring

In our work we are primary interested in safety critical applications; that is, applications where malfunction of the software may lead to death, injury, or environmental damage. Most, if not all, such systems are some form of a process control system where the software is participating in the control of a physical system.

3.1 Control Systems

A general view of a software controlled system can be seen in the center of Figure 1. This model consists of a process, sensors, actuators, and a software controller. The process is the physical process we are
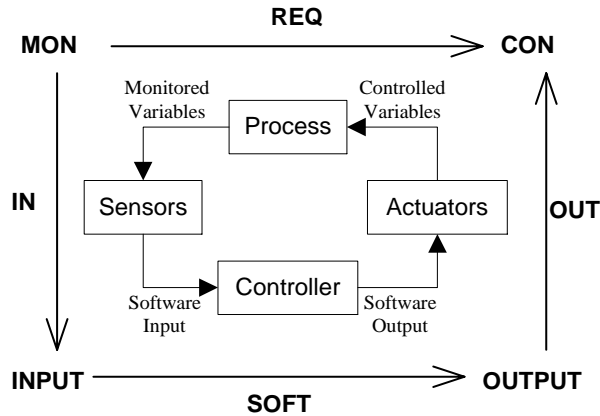
Figure 1. A traditional process control model (center) and how it is captured with the four variable model

attempting to control. The sensors measure physical quantities in the process. These measurements are provided as input to the software controller. The controller makes decisions on what actions are needed and commands the actuators to manipulate the process. The goal of the software control is to maintain some properties in the physical process. Thus, understanding how the sensors, actuators, and process behave is essential for the development and evaluation of correct software. The importance of this systems view has been repeatedly pointed out in the literature [19, 17, 12].

To reason about this type of software controlled systems, David Parnas and Jan Madey defined what they call the four-variable model (outside square of Figure 1) [19]. In this model, the monitored variables (MON) are physical quantities we measure in the system and controlled variables (CON) are physical quantities we will control. The requirements on the control system are expressed as a mapping (REQ) from monitored to controlled variables. For instance, a requirement may be that *"in case of a collision, the robot must back up and turn 90 degrees left."* Naturally, to implement the control software we must have sensors providing the software with measured values of the monitored variables (INPUT), for example, an indication if the robot has collided with an obstacle. The sensors transform MON to INPUT through the IN relation; thus, the IN relation defines the sensor functions. To adjust the controlled variables, the software generates output that activates various actuators that can manipulate the physical process, for instance, a means to vary the speed of the robot. The actuator

function OUT maps OUTPUT to CON. The behavior of the software controller is defined by the SOFT relation that maps INPUT to OUTPUT.

The requirements on the control system are expressed with the REQ relation; the system requirements shall always be expressed in terms of quantities in the physical world. To develop the control software, however, we are interested in the SOFT relation. Thus, we must somehow refine the *system requirements* (the REQ relation) into the *software specification* (the SOFT relation).

3.2 Structuring SOFT

The IN and OUT relations are determined by the sensors and actuators used in the system. For example, to determine if the robot has collided with an obstacle we may use a bumper with micro-switches connected to a digital input card. Similarly, to control the speed of a robot we may use a digital to analog converter and DC motors. Armed with the REQ relation, the IN relation, and the OUT relation we can derive the SOFT relation. The question is, how shall we do this and how shall we structure the description of the SOFT relation in a language such as RSML^{-e}?

As mentioned above, the system requirements should always be expressed in terms of the physical process. These requirements will most likely change over the lifetime of the controller (or family of similar controllers). The sensors and actuators are likely to change independently of the requirements as the controller is reused in different members of a family or new hardware becomes available; thus, all three relations, REQ, IN, and OUT, are likely to change over time. If either one of the REQ, IN, or OUT relations change, the SOFT relation must be modified. To provide a smooth transition from system requirements (REQ) to software specification (SOFT) and to isolate the impact of requirements, sensor, and actuator changes to a minimum, the structure of the software specification SOFT should be based heavily on the structure of the REQ relation [18, 23].

We achieve this by splitting the SOFT relation into three pieces, IN^{-1}, OUT^{-1}, and SOFT_{REQ} (Figure 2). IN^{-1} takes the measured input and reconstructs an estimate of the physical quantities in MON. The OUT^{-1} relation maps the internal representation of the controlled variables to the output needed for the actuators to manipulate the actual controlled variables. Given the IN^{-1} and OUT^{-1} relations, the SOFT_{REQ} relation will now be essentially isomorphic to the system requirements (the REQ relation) and, thus, be robust if it is reused on a new platform (manifested as changes
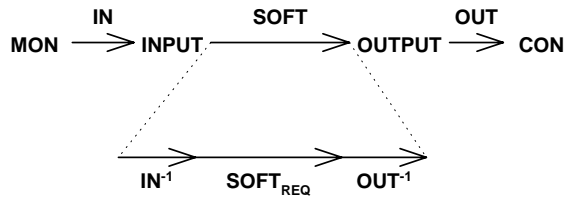
Figure 2. The SOFT relation can be split into three composed relations.

Figure 3. A picture of the robotic platforms used in this paper

in the IN and OUT relations). Such changes would only effect the IN^{-1} and OUT^{-1} portions of the software specification. Thus, the structuring approach outlined in this section will makes the SOFT_{REQ} portion of the software specification reusable over members of a product family exhibiting the same high-level behavior.

4   Mobile Robotics Platforms

When evaluating our work, we wanted to find a domain were a variety of similar platforms could be constructed on a university budget in a timely and cost effective manner. Furthermore, we wanted this domain to be realistic—with the inclusion of noisy sensors and actuators and the possibility of complex sensor fusion and error detection. The mobile robotics domain seemed ideally suited for these needs.

The mobile robotics platforms that we are using in our research range in size from about the size of the Mars Pathfinder to a small lego-bot. The robots have a limited speed, and can operate either autonomously (via a radio modem or radio Ethernet) or via a tether cable going to a personal computer. The robotics platforms come from various vendors and have a wide variety of sensors and actuators available.

The platforms that are discussed in this paper are shown in Figure 3[1]. One platform, the Pioneer [1], is built and sold by ActivMedia, Inc. The Pioneer includes an array of sonar sensors in the front and sides that allow it to detect obstacles. To detect collisions, the Pioneer monitors its wheels and signals a collision when the wheels stall. The Pioneer includes an extensive control library called Saphira. The Pioneer is controlled by a radio modem that plugs in to the personal computer's serial port. Saphira manages the communication over the radio modem. Saphira is capable of implementing complex rule-based control functions; however, in our work we are using only the simplest
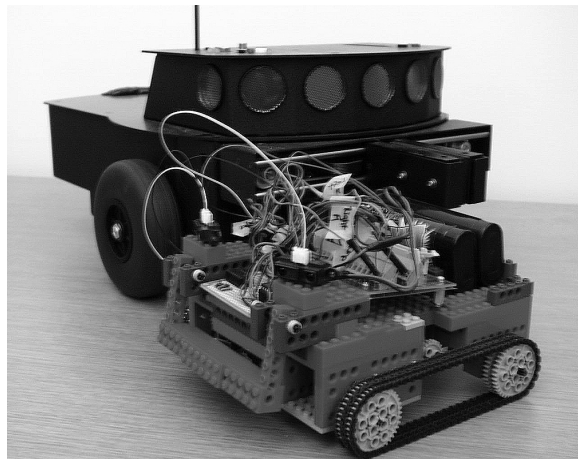
---

[1] Photograph by Timothy F. Yoon

of Saphira functions that allow us nearly direct access to the sensors and actuators. Nevertheless, the level of abstraction presented by the Saphira library is significantly higher than on the other platform in this case study: the lego-bot.

The lego-bot is a smaller platform built from Lego building blocks and small motors and sensors. The lego-bot uses a tank-like track locomotion system and has infrared sensors for range detection. The lego-bot is controlled via a tether to the robot from the personal computer. This tether is connected to a data-acquisition card and the software specification for the lego-bot behavior must directly manage the low-level voltages and signal necessary to control the robot; there is very little support for the actuators and sensors.

Despite the significant difference between the platforms, we wanted them to exhibit nearly identical visible behaviors; the only difference would be in the hardware determined speed of the robot's movements. Therefore, the visible behavior (the REQ relation) for each robot is the same. Note that we are not addressing non-behavioral requirements such as power consumption and wear and tear of hardware components in our discussions of reuse. We have focused solely on the *behavior* captured in the requirements.

5   The REQ relation

The first step in a requirements modeling project is to define the system boundaries and identify the monitored and controlled variables in the environment. In this paper we will not go into the details of how to

scope the system requirements and identify the monitored and controlled variables—guidelines to help identify monitored and controlled variables have been discussed in numerous other places [6, 13, 18]. Here it suffices to say that the monitored and controlled variables exist in the physical system and act as the interface between the proposed controller (software and hardware) and the system to be controlled.

For the mobile robots, the goal was to construct a simple reactive control behavior that would cause the robot to explore its environment. To accomplish this objective, the robot must be able to perform several tasks:

- If the robot detects an obstacle, it shall attempt to avoid it.
- If the robot collides with an obstacle, it shall attempt to recover from the collision and continue exploration.
- In the absence of a collision or obstacle, the robot shall proceed to move forward at full speed.

In this case study, we wanted all robots of the product family to exhibit the same exploratory behavior. To capture this behavior we must discover monitored and controlled variables in the environment that will allow us to build the formal model. In addition, while evaluating candidates for monitored and controlled variables we must keep in mind that the REQ model shall apply to all members of the product family.

We identified a robot's *speed* and *heading* as controlled variables. *Speed* ranges from 0 to 100 and can be mapped into a speed for each family member using the maximum speed of the particular robot. *Heading* ranges from -180 to 180 and indicates the number of degrees that the robot may have to turn to avoid an obstacle.

We identified *CollisionDetected*, *Range*, and *ObstacleOrientation* as monitored variables. The *CollisionDetected* variable is simply a Boolean value which is true when there is a collision and false otherwise. The *Range* variable is the distance from the robot to the nearest obstacle and the *ObstacleOrientation* denotes whether the obstacle is straight ahead, or on the right or left of the robot. These variables clearly reside in the system domain and are sufficient to model the desired behavior. If the monitored and controlled variables are chosen appropriately, the specification of the REQ relation will be focused on the issues which are *central* to the requirements on the system.

Since our work is based around a modeling language called RSML^{-e} (Requirements State Machine Language without events), a state-based language suitable for modeling of reactive control systems, we pro-

vide a short introduction to the notation before we continue with a discussion of the REQ relation for the mobile robots.

5.1   Introduction to RSML^{-e}

RSML^{-e} is based on the language Requirements State Machine Language (RSML) developed by the Irvine Safety Research group under the leadership of Nancy Leveson [17]. RSML^{-e} is a refinement of RSML and is based on hierarchical finite state machines and dataflow languages. Visually, it is somewhat similar to David Harel's Statecharts [10, 8, 9]. For example, RSML^{-e} supports parallelism, hierarchies, and guarded transitions. The main differences between RSML^{-e} and RSML are the addition in RSML^{-e} of rigorous specifications of the interfaces between the environment and the control software, and the removal of internal broadcast events. The removal of events was prompted by Nancy Leveson's experiences with RSML and a new language called SpecTRM-RL that she has evolved from RSML. These experiences have been chronicled in [16].

An RSML^{-e} specification consists of a collection of *state variables*, *I/O variables*, *interfaces*, *functions*, *macros*, and *constants*, which will be briefly discussed below.

In RSML^{-e}, the state of the model is the values of a set of *state variables*, similar to mode classes in SCR [12]. These state variables can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

For example, consider the behavioral requirements for our mobile robots outlined in the introduction to this section. The state variable hierarchy used to model the requirements on this system can be represented as in Figure 4. This representation includes both parallel and hierarchical relationships of state variables: *Failure* and *Normal* are two parallel state variables and *Robot_Recover_Action* is a child of *Normal*.

Next state functions in RSML^{-e} determine the value of state variables. These functions can be organized as *transitions* or *conditional assignments*. Conditional assignments describe under which conditions a state variable *assumes* each of its possible values. Transitions describe the condition under which a state
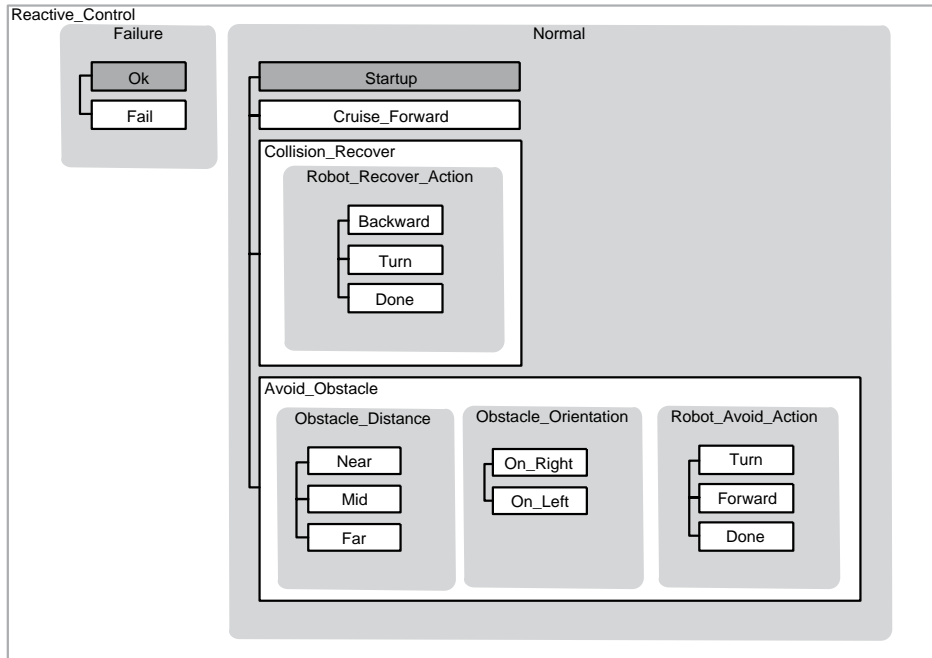
Figure 4. The REQ relation state hierarchy

variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. The two state function types are logically equivalent; mechanized procedures exist to ensure that both types of functions are complete and consistent [11].

The next state functions are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other state variables, macros, or I/O variables that are named in their transitions or condition tables. If a variable is a child variable of another state variable, then it is also dependent on its parent variable. The value of the state variable can be computed after the items on which it is data-dependent have been computed. For example, the value of the *Robot_Avoid_Action* state variable would be computed after the *Obstacle_Distance* state variable because the action to take is dependent upon the range of the obstacle.

Conditions are simply predicate logic statement over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [17] The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all

of its elements match the truth values of the associated columns. An asterisk denotes "don't care." Examples of AND/OR tables can be found later in this section and in the next section.

I/O Variables in the specification allow the analyst to record the monitored variables (MON) or values reported by various external sensors (INPUT) (in the case of input variables) and provide a place to capture the controlled variables (CON) or the values of the outputs (OUTPUT) of the system prior to sending them out in a message (in the case of output variables).

To further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, RSML^{-e} allows expressions used in the predicates to be defined as functions and familiar and frequently used conditions to be defined as macros. Finally, RSML^{-e} requires rigorous specification of *interfaces* between the environment and the model.

5.2 REQ Relation Overview

Due to space constraints, the entire model of the REQ relation cannot be given in this paper and we will focus on an illustrative subset. Figure 4 shows that the REQ relation definition at the top level is split between two state variables: *Failure* and *Normal*. The *Failure* state variable encapsulates the failure conditions of the REQ relation, whereas the *Normal* state variable de-

scribes the how the robot transitions between the various high-level behaviors discussed at the introduction to this section (obstacle avoidance, collision recovery, etc.). For the reminder of our discussion of REQ, we will focus on the *Normal* state variable where this aspect of the requirements is captured (Figure 5).

The *Normal* variable defaults to the *startup* value. This allows the specification to perform various initialization tasks and checks before the main behavior takes over. The first transition in Figure 5 states that after two seconds, the specification will enter the *Cruise_Forward* state.

The next two transitions govern the way that the *Normal* state variable can change from the *Cruise_Forward* value. If a collision is detected, then the state variable changes to the *Collision_Recover* state. If an obstacle is detected, then the specification will enter the *Avoid_Obstacle* state. Otherwise, the value of the *Normal* state variable will remain unchanged.

If a collision or obstacle is detected, the machine needs to begin the *Cruise_Forward* behavior when the avoidance/recovery action has been completed. We accomplished this in the mobile robotics specification by providing a "done" state in each of the sub-behaviors. This is illustrated by the fifth and sixth transitions in Figure 5.

Finally, it is also possible to transition from *Avoid_Obstacle* directly to *Collision_Recover* if, for example, the robot hits an undetected obstacle; this case is covered by the final transition in Figure 5.

Given this definition of the REQ relation high-level behaviors, the definitions of the sub-behaviors can be constructed in a similar and straightforward manner. For example, if the robot hits an obstacle, it will attempt to back up, turn, and then proceed forward again. This behavior is specified in the *Robot_Recover_Action* state variable by having the variable cycle though the values *Backward*, *Turn*, and finally *Done*.

6   The SOFT relation

When refining the specification from REQ to SOFT, we select the sensors and actuators that will supply the software with information about the environment, that is, we must select the hardware and define the IN and OUT relations for each platform. Consequently, we will also need to define the IN^{-1} and OUT^{-1} for each platform. We do not have the space to discuss the IN, OUT, IN^{-1}, and OUT^{-1} for every monitored and controlled variable. Instead, we will focus our discussion on two areas where the Pioneer
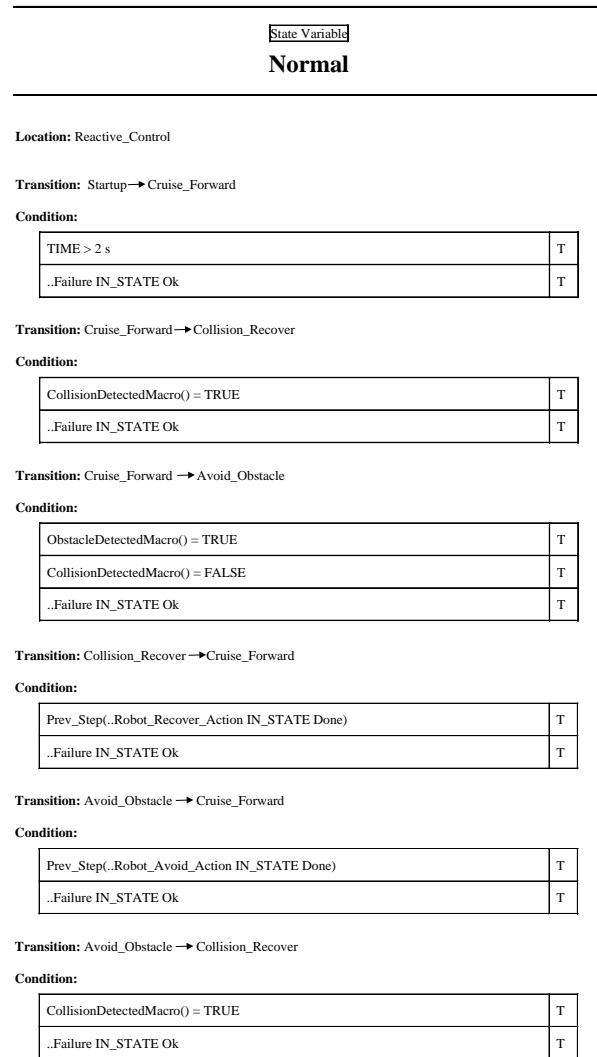
State Variable

Normal

Location: Reactive_Control

Transition: Startup → Cruise_Forward

Condition:

TIME > 2 s	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward → Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward → Avoid_Obstacle

Condition:

ObstacleDetectedMacro() = TRUE	T
CollisionDetectedMacro() = FALSE	T
..Failure IN_STATE Ok	T

Transition: Collision_Recover → Cruise_Forward

Condition:

Prev_Step(..Robot_Recover_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle → Cruise_Forward

Condition:

Prev_Step(..Robot_Avoid_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle → Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Figure 5. The definition of the *Normal* **state variable**

and the lego-bot presented illustrative and challenging differences.

6.1 Obstacle Detection— Sonar versus Infrared

As members of the mobile robot product family that we specified in Section 5, both the Pioneer and the lego-bot have the ability to sense the distance to objects in their surroundings. Distance sensors typically function by emitting some sort of signal (for example, a sound in the case of sonar) and then measuring the amount of time between the emission of the signal and its being received back at the sensor. Given how fast the signal can travel, the distance to the closest object can be determined. Although the distance sensors may be somewhat similar in their operation, different sensors provide very different accuracies and ranges. For example, a laser range finder is far more accurate and has much less noise than the sonar sensors.

The Pioneer uses sonar sensors and the Saphira software package to accomplish obstacle detection whereas the lego-bot uses a set of simple infrared range finders. This significant difference in the type of sensors as well as differences in the number and placement of the sensors leads to two quite different IN relations. The differences of the IN relations necessitate different IN^{-1} in the computation of the estimated value of the *Range* monitored quantity.
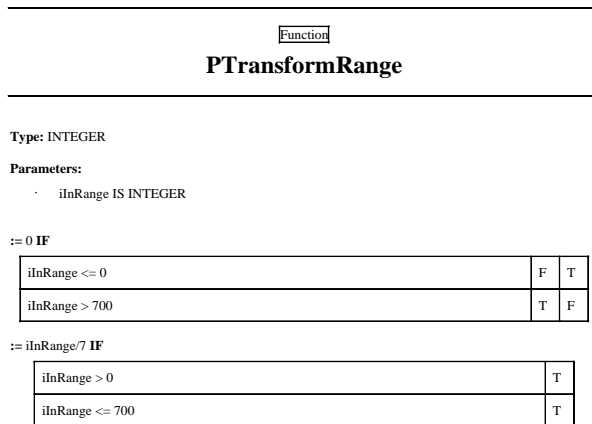
Function
PTransformRange

Type: INTEGER

Parameters:
· iInRange IS INTEGER

:= 0 **IF**

iInRange <= 0	F	T
iInRange > 700	T	F

:= iInRange/7 **IF**

iInRange > 0	T
iInRange <= 700	T

Figure 7. IN^{-1} Range for the Pioneer

The difference between the SOFT relations for the two platforms (with respect to the range to obstacles) can be encapsulated in a function which transforms the input variables from the range sensors to estimates of the monitored quantity *Range*. The computation of
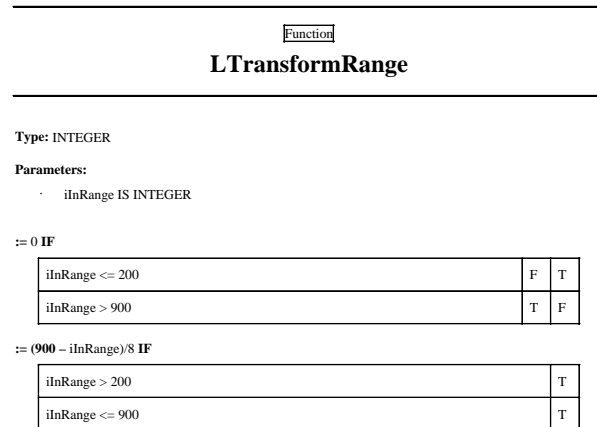
Function
LTransformRange

Type: INTEGER

Parameters:
· iInRange IS INTEGER

:= 0 **IF**

iInRange <= 200	F	T
iInRange > 900	T	F

:= (900 – iInRange)/8 **IF**

iInRange > 200	T
iInRange <= 900	T

Figure 8. IN^{-1} Range for the lego-bot

IN^{-1} for the Pioneer is pictured in Figure 7 and for the lego-bot is in Figure 8. For the Pioneer, the sonar inputs range from 0 to 700 and must be scaled appropriately to a number between zero and 100.

For the lego-bot, the transformation is more complex. Both the sonar and the infrared distance sensors have a certain range close to the sensor where the signals cannot be used for range detection (in the case of the sonars, the signals that are emitted bounce back to the sensor too fast for the sensor to detect). Thus, the sensor will report that no obstacle is present when, in fact, an obstacle is very close. In the case of the Pioneer, this problem is handled by the Saphira library. For the lego-bot, however, the $RSML^{-e}$ specification must include a minimum threshold as well as a scaling factor for the maximum values. In our case, readings below 200 from the infrared sensor cannot be trusted and we simply treat any reading below 200 as if the distance is 0, indicating that no obstacle has been (or can be) detected (Figure 8).

Thus, we have shown that even though the sensors and the way in which we have access to the sensors differs widely between the Pioneer and the lego-bot, we can still use the same $SOFT_{REQ}$ model for both robot platforms. In this way, we make the high-level behavior robust and reusable in the face of changes in the range finder.

6.2 Speed— Saphira versus Pulse Modulation

The previous section focused on platform dependent variabilities in the IN and IN^{-1} relations. The Pioneer and the lego-bot have more significant differ-
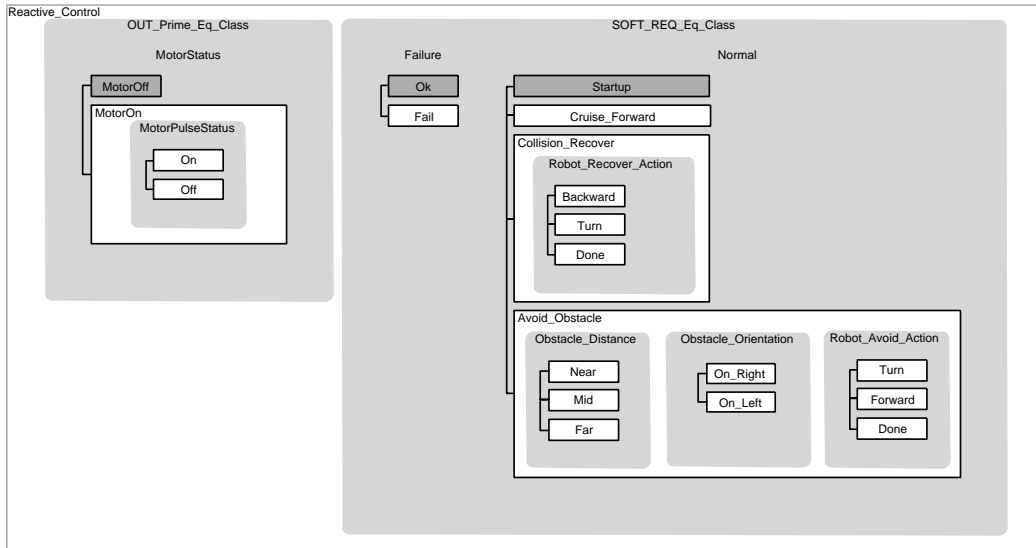
Figure 6. The state machine for the lego-bot

ences in the way that they control their propulsion and in their steering systems (the OUT and OUT^{-1} relations).

The Pioneer's Saphira library provides a high-level control of the Pioneer's motors so that the specification for SOFT on the Pioneer platform is very similar to REQ. The transformation of the desired speed performed in OUT^{-1} for the Pioneer (Figure 9) only requires some minor scaling with respect to the Pioneer's maximum speed. The result of this transformation can then be directly sent to the Pioneer platform and Saphira will control the hardware to achieve the desired speed.
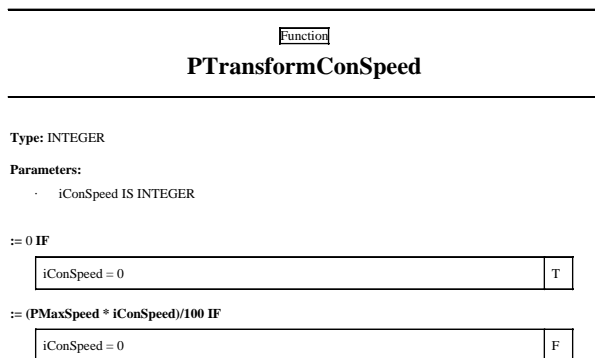
<u>Function</u>

PTransformConSpeed

Type: INTEGER

Parameters:

· iConSpeed IS INTEGER

:= 0 IF

| iConSpeed = 0 | T |

:= (PMaxSpeed * iConSpeed)/100 IF

| iConSpeed = 0 | F |

Figure 9. OUT^{-1} Speed for the Pioneer

On the other hand, the OUT^{-1} specification for the speed of the lego-bot is significantly more complex.

<u>State Variable</u>

MotorPulseStatus

Location: ..MotorOn

Transition: Off ® On

Condition:

| TIME >= PREV_STEP (..MotorPulseStatus TIME_ENTERED Off) + LMotorPWMTimeOut(Heading, ConSpeed) | T |
| PREV_STEP(..MotorPulseStatus IN_STATE Off) | T |

Transition: On ® Off

Condition:

TIME >= PREV_STEP(..MotorPulseStatus TIME_ENTERED On) + LPWMOnTimeOut	T
PREV_STEP(..MotorPulseStatus IN_STATE On)	T
ConSpeed = 100	F

:= On

Condition:
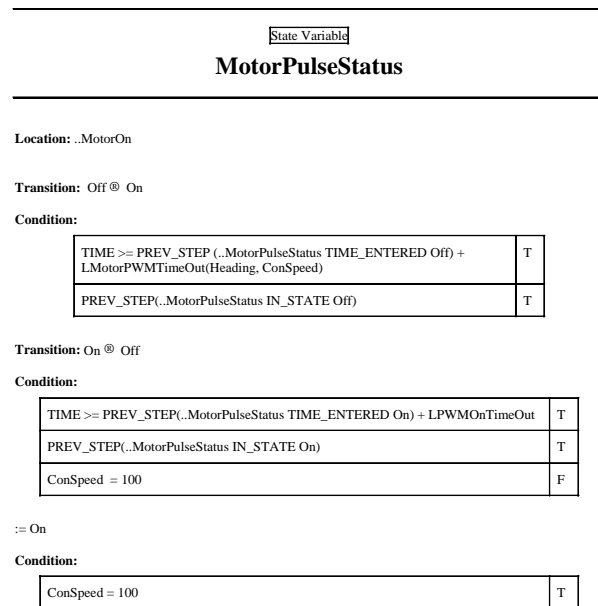
| ConSpeed = 100 | T |

Figure 10. The part of OUT^{-1} for the Lego-bot that performs the pulsing on the motors

This is because the SOFT relation for the lego-bot must control the motors directly with low-level hardware signals. The speed of the lego-bot is controlled by a technique called *pulse-width modulation* of the DC motors: the speed of the motors is determined by the length of time which passes between pulses of current applied to the motor. Therefore, the SOFT specification cannot simply output the speed value with some transformation applied; instead, we must use the computed value for the controlled variable *Speed* to determine the pulse width for the motors and then output the pulses accordingly; the motors will then provide enough propulsion to move the lego-bot at the desired speed.

This control strategy necessitates a more complex OUT^{-1} relation for the desired speed; the OUT^{-1} relation can no longer be a simple function—in this case we need to add an additional state machine. To model the pulse modulation we add a state variables to the SOFT specification so that the machine can output the required pulses. These additions are shown in Figure 6. The *MotorPulseStatus* state variable is the part of the OUT^{-1} specification that determines the pulse width. Figure 10 shows the definition of this state variable.

A key component of the pulse-width modulation is the LMotorPWMTimeOut function which determines the length of time to pulse the motors (Figure 11). Notice that because of the lego-bot's tank-track propulsion system, the motors must be pulsed both in the case of a turn and in the case that the robot is moving forward. Thus, the LMotorPWMTimeOut function takes as parameters the controlled variables for speed and heading and produces the correct timeout values.

The values for the pulse intervals were were chosen by running experiments to determine which pulse interval would achieve which speed. We have, therefore, encapsulated these constants so that if we were to change motors on the lego-bot in the future we could simply change the constants rather than having to revisit the pulse-width modulation process.

Thus, despite the fact that the Pioneer and the lego-bot differ significantly in the way that the motors are controlled, the $SOFT_{REQ}$ relation can again be reused across the platforms. Furthermore, changes in the REQ relation (and analogous changes to $SOFT_{REQ}$) will be independent of changes in the OUT and OUT^{-1} relations.

7 Conclusions

This paper describes how structuring the requirements based on the relationship between the system

Function
LMotorPWMTimeOut

Type: TIME

Parameters:
- iHeading IS INTEGER
- iConSpeed IS INTEGER

:= LSlowPWMOffTimeOut IF

iHeading = 90	T	F	F
iHeading = -90	F	T	F
iConSpeed = 25	F	F	T

:= LMidPWMOffTimeOut IF

iHeading = 45	T	F	F	F
iHeading = -45	F	T	F	F
iConSpeed = 50	F	F	T	F
iConSpeed = -50	F	F	F	T

:= LFastPWMOffTimeOut IF

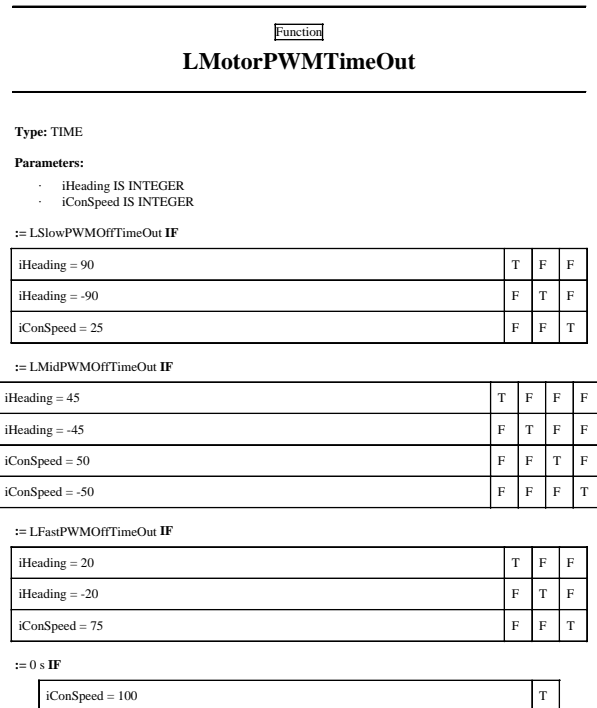|			
iHeading = 20	T	F	F
iHeading = -20	F	T	F
iConSpeed = 75	F	F	T

:= 0 s IF

iConSpeed = 100	T

Figure 11. The timeout function for pulse-width modulation on the Lego-bot.

requirements and the software specification can lead to benefits in terms of maintainability and reusability. Specifically, we describe a technique for structuring high-level requirements for reuse in the face of hardware changes.

From the four variable model for process control systems, we have described how the REQ relation can be refined to the SOFT relation while maintaining a separation between the part of SOFT which is related to REQ (SOFT_{REQ}) and the parts of SOFT which handle the particular sensors and actuators in the system design (IN^{-1} and OUT^{-1}). This allows us to separate changes in the requirements from sensor and actuator changes and achieve better maintainability and reusability.

This techniques was demonstrated on a case study in the mobile robotics domain using two quite different robots. One robot is commercially produced and is equipped with a rich control library that provides many complex control functions, for example, traveling at a requested speed. The other robot was build in-house from Lego building blocks and off-the-shelf motors and sensors. This robot is controlled completely by the software specification in RSML^{-e} through our NIMBUS toolset.

We demonstrated the usefulness of the structuring approach by reusing the high-level requirements (REQ) across a (currently quite small) family of mobile robots. Nevertheless, there are numerous issues left to address. In the future, we plan to define more complex control behaviors and investigate how individual behaviors (or operational modes) can be successfully reused.

References

[1] Activmedia robotics website. Makers of the Pionner robot. http://www.activrobots.com/.

[2] Mark A. Ardis and David M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 649–650, 1997.

[3] K.H. Britton, R.A. Parker, and D.L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Fifth International Conference on Software Engineering*, 1981.

[4] F. P. Brooks. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[5] Consortium requirements engineering handbook. Technical Report SPC-92060-CMC, Software Productivity Consortium, December 1993.

[6] S. Faulk, J. Brackett, P. Ward, and J Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[7] Stuart Faulk, Lisa Finneran, James Jr. Kirby, Sudhir Shah, and James Sutton. Experience applying the CoRE method to the lockheed C-103J software requirements. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS)*, pages 3–8, 1994.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[9] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[10] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[11] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, pages 363–377, June 1996.

[12] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[13] Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[14] W. Lam. Developing component-based tools for requirements reuse: A process guide. In *Eighth International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 473–483, 1997.

[15] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematic requirements reuse. In *Third IEEE International Symposium on Requirements Engineering (RE'97)*, pages 6–15, 1997.

[16] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing specification languages for process control systems: Lessons learned and steps to the future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, September 1999.

[17] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, pages 684–706, September 1994.

[18] Steven P. Miller. Modeling software requirements for embedded systems. Technical report, Advanced Technology Center, Rockwell Collins, Inc., 1999. In Progress.

[19] David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.

[20] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[21] D.L. Parnas. Designing software for ease of extension and contraction. In *Third International Conference on Software Engineering*, 1978.

[22] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):256–266, 1985.

[23] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, September 1999.

[24] David M. Weiss. Defining families: The commonality analysis. Technical report, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Naperville, IL 60566, 1997.

Automated V&V for high integrity systems, a targeted formal methods approach

Simon Burton, John Clark, Andy Galloway, John McDermid
Department of Computer Science.
University of York,
Heslington, York.
Y010 5DD, England.
+44 1904 432749
{burton, jac, andyg, jam}@cs.york.ac.uk

Abstract

This paper describes the intermediate results of a project to develop automated, high integrity, software verification and validation techniques for aerospace applications. Automated specification validation and test case generation are made possible by the targeted use of formal methods. Specifically, the restricted domain of use is exploited to reduce the set of mathematical problems to those that can be solved using constraint solvers, model checkers and automated proof tactics. The practicality of the techniques is enhanced by the tight integration of the formal methods to intuitive specification notations, existing specification modelling tools and a traditional software development process.

This paper presents evidence to support an emerging appreciation amongst the software engineering community that, for the benefits of formal methods to be widely exploited in industry, an approach must be taken that integrates formal analysis with intuitive engineering notations, traditional engineering approaches and powerful tool support.

1. Introduction

It is widely accepted that verification and validation (V&V) activities for high integrity systems are expensive (typically over 50% of total software development costs [2]). The requirements for such systems are often subject to change throughout the project so the high V&V costs are normally incurred not only once, but many times. Also, the cost of fixing errors later in the development life-cycle can be many times more than if they were identified during the phase in which they were introduced. Additionally, commercial pressures to reduce time to market, technological conservatism and the need to meet standard test metrics make the software V&V process a highly fragile and risky component of system development.

The use of formal methods has long been advocated as a means of improving the development of high integrity systems. Despite evidence to support this claim, e.g. [14, 17], formal methods have still to gain widespread use in the software industry. Industrial acceptance of formal methods requires the development of powerful tools to support formal analysis, pragmatic approaches to using these tools within a software process and more industrially applicable examples of the successful use of formal methods [6, 15]. Also, for the engineers with the system domain knowledge to be able to perform V&V there is a need, as Ould [22] put it, to "disguise" the formality so that an impractical amount of formal methods skill is not a pre-requisite to effective V&V.

This paper describes the results of a project that has achieved practical integration of automated formal methods for V&V into an industrially applicable software development process. The paper is structured as follows: Section 2 introduces the background and objectives of the work reported here. Section 3 describes the translation of domain specific, intuitive engineering notations into formal specifications. Section 4 describes how these intermediate formal specifications can be used to automatically analyse certain properties of the requirements specification. Section 5 describes a method of automated test case design and test data generation, based again on the intermediate formal specification. Section 6 presents some results of applying the techniques in practice and gives an evaluation of the work so far. Section 7 presents some conclusions and suggests directions for future work.

2. Background and objectives

The work reported here is being undertaken as part of a process improvement programme to demonstrate a "better, faster, cheaper" software process for developing Electronic Engine Controllers (EECs) for aircraft engines. These are real-time, safety critical, fault-tolerant computer systems embedded in complex engineering products. The contribution of the V&V strand of the process improvement work (the subject of this paper) is to develop efficient and effective V&V techniques that can be smoothly integrated into a practical engineering process.

The use of formal methods is intended to increase the integrity of engineering activities already performed such as specification and testing. These improvements must be implemented within a process that engineers can use with the minimum amount of re-training. Therefore intuitive engineering notations have been retained as a means of software specification and techniques have been developed to increase the integrity of these specifications through the use of automated formal analysis.

Although this research is targeted towards specification validation and software testing, it is acknowledged that significant benefits in these areas can not be attained without improving the rigour and consistency of the requirements specifications. Specification notations are therefore used that are both "engineer friendly" and amenable to formal analysis. The savings demonstrated in the validation and testing phases serve as drivers to encourage investment in these improved specification activities. It is expected that the most significant cost-benefits can be achieved by capturing more requirements and software specification errors at the specification validation phase (therefore reducing the number of iterations of the software design, coding and testing phases) and by automating test case generation (one of the most time consuming parts of the present process).

3. Translating engineering notations into formal specifications

Domain specific graphical engineering notations are popular with engineers, but their semantics are often unclear from inspection of the diagrams alone. In reality, it is also unlikely that only one notation will be used to specify a system, or indeed that notations will be used consistently between projects. The resulting loose specification and inconsistency complicates the task of automating specification validation and test case generation based on these notations. Indeed as the notations change so frequently (as a result of commercial trends, new or outdated tool-support etc.) it may not even be

cost effective to invest in automated V&V tool support which may in practice only have a limited life-span and audience.

Translation of the graphical requirements into a core formal notation removes the vagueness of the original notations and makes the behaviour implied by the specification explicit for the purposes of V&V. Validation may thus be supported by rigorous (or formal) reasoning using the formal representation. Also, by explicitly rendering all specified behaviour, the intermediate representation is a strong basis for automated test case design. The use of a common formal notation to model several engineering notations facilitates re-use of the analysis and test techniques. The introduction of a new notation requires only a translation to the formal notation, after which the previously developed tools and heuristics can be re-used. A strict translation process allows a fixed structure to be enforced on the resulting formal specification that can be exploited in the development of automated heuristics, e.g. test data generation procedures and proof tactics.

The work reported here focuses on the specification, validation and verification of the discrete aspects of engine controllers (well-established mathematically based processes were already in place for modelling and validating the continuous aspects of the control software – e.g. the control laws). The Practical Formal Specification (PFS) notation [7, 19] is used to specify the functional software requirements. The PFS notation consists of hierarchical state machines (in particular, a dialect of statecharts[1] [9]) and tabular forms, such as those employed in SCR (Software Cost Reduction) e.g.[12]. The notation has so far proven popular with the engineers introduced to PFS[2]. PFS also provides a theory for combining components specified in the notation – based on weakest precondition reasoning – and a set of suggested "healthiness properties" that specifications should display to be considered valid.

One of the cornerstones of the PFS approach is that engineers are not only required to specify the intended behaviour of components of the system, they are also obliged to state explicitly the assumptions on which each component relies, i.e. its domain of applicability. Healthiness conditions can then be stated and discharged to demonstrate that, for instance, within the assumptions of each component the behaviour is completely and unambiguously defined. Additionally, healthiness conditions are stated to demonstrate that where behaviour is scoped by assumptions, it is only ever used when the required assumptions hold.

---

[1]The state-based notation employed in this paper, a sub-set of Statecharts, differs slightly from that usually employed in PFS.

[2]Who have stated that they find the notation valuable *even without* the added rigour provided by a formal underpinning.
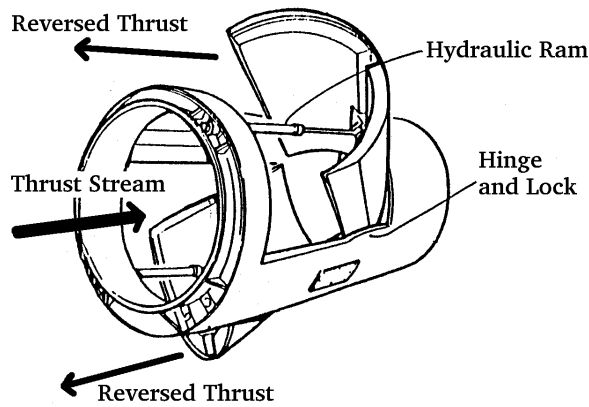
Figure 1. Thrust Reverser System (Deployed)

Due to the restricted domain, the specifications shared some common attributes which reduced the set of (mathematical) problems that needed to be solved when applying the formal analysis:

- The software requirements did not involve the storage and maintenance of complex information structures, typically only fixed-point numbers, conditions and enumerated types were used.

- In the control software domain, non-determinism (looseness) as a means of abstraction is difficult to apply[3]. In the example given here, the requirements were tightly specified, only one outcome was to be specified for each situation[4].

PFS components are either reactive (the outputs depend only on the current set of inputs) and specified using tables, or else state-based (the outputs depend on both the current inputs and the current state of the system) and specified using annotated state-machines.

The example used to illustrate the techniques reported in this paper is the specification of a thrust reverser deployment mechanism. The thrust reverser provides part of the retarding force for an aircraft on landing (see figure 1). It slows the aircraft by using pivoting doors to redirect the engine thrust. For the purpose of clarity and brevity, we will present a much simplified version of the specifications, although their essence is retained. A real thrust reverser system was used as the primary case study for evaluating the techniques described here. The software specification used for the case study consisted of 70 pages of PFS tables and statecharts, component combination diagrams and supporting text.

Examples of software requirements written in PFS are given in figures 2 and 3. Figure 2 describes a function that returns a boolean value (*DoorDeployed*) corresponding to whether a door is locked into its deployed position or not. The assumption defines the context in which the component may be safely used (in this case, the conditions under which sensor values may be deemed to be valid). The guard/definition pairings define the conditions (guards) under which the function returns particular values (definitions). A state-machine that specifies which commands should be sent to the door actuators based on the pilot actions and current door position is given in figure 3. The part of the transition labels before the '/' defines the condition under which the transition is taken. The rest of the label defines the action to be performed on taking the transition. The values for *DoorDeployed*, *DoorStowed* and *Pilot-Command* are calculated based on functions defined in the reactive notation. Likewise the *DoorActuators* command would be transformed into actuator signals based on the command and a number of environmental and positional inputs. This function would also be specified in the reactive notation.

Both the reactive and state-based components are translated into formal specifications (we use the model-based notation Z [24] due to the large amount of local experience and existing in-house tool support). The semantics of PFS notations has been formally specified also using Z and this is used to define the translation from the reactive components into Z. The state-based components are modelled using Statemate [10] (a commercially available tool that allows Statecharts to be entered and animated via a mouse-driven interface). The semantics used by the tool are well-documented [11] and have also been formally specified [20, 29]. These semantics are used to define the translation from the Statecharts into Z. The formal specifications for both notation types are structured as follows:

- *Auxiliary definitions:* These may include definitions of types, constants and relations used to constrain the system according to the static semantics of the engineering notation.

- *Global State (for Statecharts only):* Contains all information relating to the persistent state of the system. This may include a set of currently active behavioural states, active events and the values of all data variables local to the statechart. The global state is constrained by semantic relations specified

---

[3]The controlled environment is understood in terms of the great many relationships between physical quantities, as a result the expression of requirements are highly explicit and deterministic.

[4]Although PFS notation does allow some non-deterministic abstractions to be used in certain situations [7].

Function:	DoorDeployed
Assumption:	FullyRetracted $\leq$ RamPosition $\leq$ FullyExtended **AND** $0 \leq$ Hinge ≤ 90
Guard 1:	RamPosition $>$ DeployedPosition **AND** Hinge > 80 **AND** DeployLock $=$ Activated
Definition 1:	True
Guard 2:	RamPosition $<$ DeployedPosition **OR** Hinge < 80 **OR** DeployLock $=$ Deactivated
Definition 2:	False

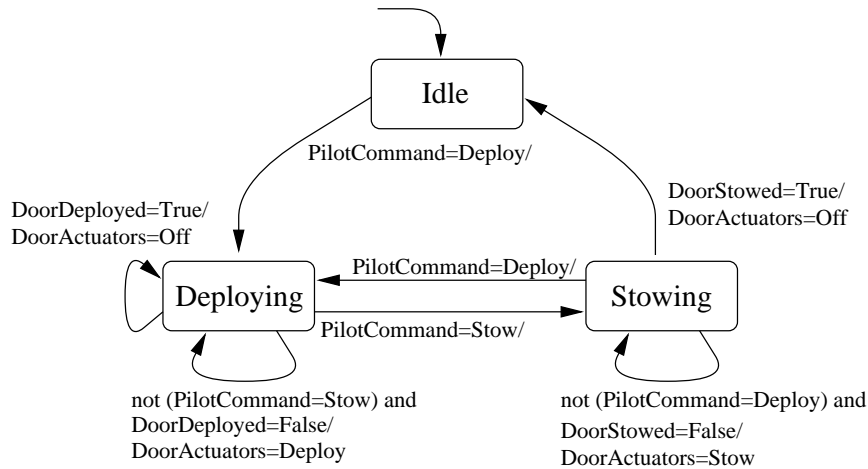**Figure 2. Reactive component for sensing thrust reverser door deployment**

Figure 3. State-based component for controlling door deployment

in the auxiliary definitions (defined in terms of a state invariant).

- *Operations:* The dynamic behaviour of the system is specified as a set of operations. Each operation defines a transformation of the global state (for statechart operations only) and inputs to the component in terms of a pre-condition. A post-condition constrains the next value of the global state (for Statecharts operations only) and a definition of the outputs of the component. One operation is specified for each reactive definition and for each statechart transition. These operations form the basis of the specification validation and test case generation activities described in the following sections.

The Statemate [10] tool provides an "Application Programming Interface" (API), that allows direct access to the internal form of the specification. An interfacing tool, called StateZ, was written by the authors, that takes this internal form and, based on an understanding of the formal semantics of the Statecharts, directly generates a formal specification of the statechart in Z. Included in this specification are the accompanying proof conjec-

tures required to discharge particular healthiness checks of the specifications (see section 4) and automatically generated English language annotations. This informal text provides the traceability between the formal operations and their corresponding part in the original requirements or a description of the property which the conjectures are used to prove. These annotations not only allow the generated Z document to be reviewed for correctness with respect to the original requirements (verifying the automated translation) but also form the basis of the test descriptions which are used to associate each test with the property in the requirements being checked. The addition of the informal text generation to the translation tool was found to greatly increase the readability and usability of the formal specification and associated tests.

The Statemate and StateZ tools can be run in parallel, allowing the Z to be re-generated whenever a change is made to the Statecharts. Coupled with the automated theorem proving described below, this allows the formal analysis to be used as a development aid rather than a separate post development activity.

At present, no tool support exists for translating the PFS reactive notation into Z (this step is done by hand)

and therefore the checking of the reactive components was not as tightly integrated into the specification process as for the Statecharts, however we predict that this should not present any technical difficulties, given a suitable method of electronically recording and managing the reactive tables.

4. Specification validation

In current industrial practice, many requirements errors are only found once the system has been implemented. Detecting them at an earlier stage in the development would greatly reduce the cost of (both implementation and V&V) re-work. This can best be achieved by applying a variety of diverse methods to validate the requirements specifications. These can include peer review, model animation (as supported by tools such as Statemate) and automated formal analysis. The use of intuitive engineering notations would normally exclude the possibility of applying formal analysis. However, based on the same mapping used to generate the formal specification, specification healthiness conditions can be couched as formal constraints. Formal analysis can then be used to show the truth (or otherwise) of these constraints.

Completeness[5] and determinism[6] are two of the healthiness conditions suggested by the PFS approach. If the behaviour of a component is defined as a set of operations $\{Op_1, Op_2, ...Op_n\}$ over the inputs and state, then a conjecture on the completeness of the specification of that component can be formulated as follows:

$$\vdash \forall \, GlobalState, Inputs \bullet Assumptions \Rightarrow$$
$$\text{pre} \, Op_1 \lor \text{pre} \, Op_2 \lor ... \text{pre} \, Op_n$$

Informally, for each possible value of the global state (if there is one) and each combination of inputs that satisfy the validity assumptions of the component, the precondition of *at least one* operation is satisfied.

A similar conjecture can be defined to show the determinism of the operations. Each combination of global state and inputs that satisfies the component validity assumption must satisfy *at most one* operation.

$$\vdash \forall \, GlobalState, Inputs \bullet Assumptions \Rightarrow$$
$$\forall \, i : 1..n - 1 \bullet \forall j : i + 1..n \bullet$$
$$\neg (\text{pre} \, Op_i \land \text{pre} \, Op_j)$$

---

[5]The behaviour of a system is defined for each combination of inputs and current state.

[6]The behaviour of a system is unambiguously defined for each combination of inputs and current state.

Completeness and determinism conjectures for the example reactive definition and state-based component are shown in figures 4 and 5 respectively.

$\vdash \forall \, RamPosition : \mathbb{N}; \; Hinge : \mathbb{N};$
$DeployLock : Activated \mid Deactivated \bullet$
$(FullyRetracted \leq RamPosition \leq FullyExtended$
$\land \, 0 \leq Hinge \leq 90) \Rightarrow$
$\quad (RamPosition > DeployedPosition \land$
$\quad Hinge > 80 \land DeployLock = Activated) \lor$
$\quad (RamPosition < DeployedPosition \lor$
$\quad Hinge < 80 \lor DeployLock = Deactivated)$

Figure 4. Completeness conjecture for DoorDeployed

$\vdash \forall \, State : Idle \mid Deploying \mid Stowing;$
$PilotCommand : Off \mid Deploy \mid Stow;$
$DoorDeployed : Boolean;$
$DoorStowed : Boolean \bullet$
$State = Stowing \Rightarrow$
$\quad \neg (DoorStowed = True \land$
$\quad \neg PilotCommand = Deploy \land$
$\quad DoorStowed = False) \land$
$\quad \neg (DoorStowed = True \land$
$\quad PilotCommand = Deploy) \land$
$\quad \neg (\neg PilotCommand = Deploy \land$
$\quad DoorStowed = False \land$
$\quad PilotCommand = Deploy)$

Figure 5. Determinism conjecture for Stowing

Closer inspection of these two conjectures show that they are invalid. For the reactive component, no outcome is specified if the hydraulic ram is exactly at the deployed position or the hinge is at exactly 80 degrees. For the state-based component, it is not clear to which state the machine should move while in the *Stowing* state if the pilot requests deployment at the same moment as the doors become stowed. Depending on the behaviour specified within the *Idle* and *Deploying* states (these could be super-states encapsulating more detailed behaviour) taking one transition over another may have a serious impact on the behaviour of the system.

The conjectures that arose from the case studies were proven using CADiℤ [26, 28]. CADiℤ is a general purpose Z type checker and theorem prover that allows a user to interactively browse, type check and perform
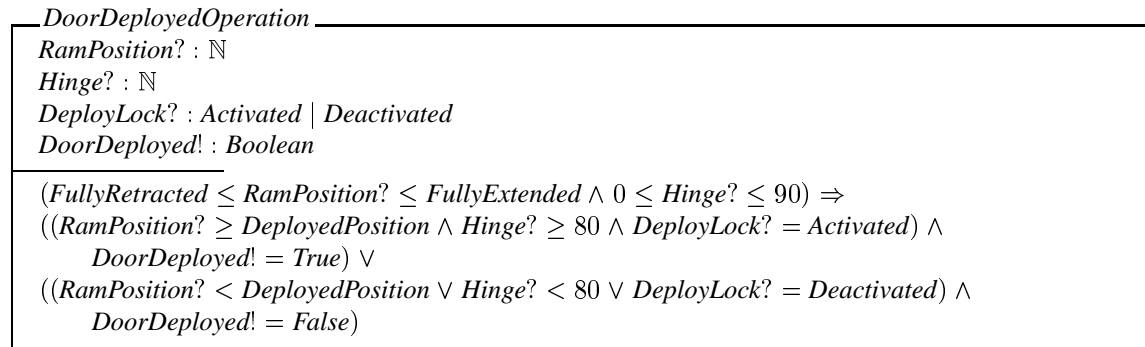
```
┌─ DoorDeployedOperation ─────────────────────────────────────────────┐
│ RamPosition? : ℕ                                                     │
│ Hinge? : ℕ                                                           │
│ DeployLock? : Activated | Deactivated                               │
│ DoorDeployed! : Boolean                                             │
├─────────────────────────────────────────────────────────────────── │
│ (FullyRetracted ≤ RamPosition? ≤ FullyExtended ∧ 0 ≤ Hinge? ≤ 90) ⇒ │
│ ((RamPosition? ≥ DeployedPosition ∧ Hinge? ≥ 80 ∧ DeployLock? = Activated) ∧ │
│     DoorDeployed! = True) ∨                                          │
│ ((RamPosition? < DeployedPosition ∨ Hinge? < 80 ∨ DeployLock? = Deactivated) ∧ │
│     DoorDeployed! = False)                                           │
└─────────────────────────────────────────────────────────────────────┘
```

Figure 6. Z operation schema for DoorDeployed

proofs upon a Z specification. CADiℤ allows general purpose proof tactics to be written in a lazy functional notation [27], these can be invoked from within a CADiℤ window and applied to any proof obligation on the screen. This level of proof tactic re-use is possible because of the consistent structure of the completeness and determinism conjectures. A proof tactic has been written to prove the determinism and completeness conjectures. The tactic first simplifies the constraint and then calls either the SMV [3] model checker (most suitable for predicates involving finite types) or a simulated annealing based constraint solver [4] (used for counter-example generation for predicates involving mixed numeric types including integers and reals). If the check fails, a counter-example is given. This information has been found to be extremely valuable when tracking the error in the specification. Conjectures that can not be automatically discharged in this way involve a mixture of enumerated and infinite numeric types. This combination is not currently supported by the constraint solvers. Restricting the numeric types to sensible finite ranges allows these constraints to be checked automatically.

The healthiness checks that failed have been found to be due to areas of omission or ambiguity in the original system requirements that were not detected through review or animation. This illustrates that there is much benefit to be obtained by verifying relatively simple properties of the specifications and the high level of automation ensures that the only additional work required is that of locating the errors in the specification based on the counter-examples. This work would otherwise be done at a later stage with perhaps less illuminating data to work from.

The high level of automation allows the analysis to be re-run each time the specification is changed, further reducing the cost of rework. Although the interactive version of CADiℤ allows the proof effort to be automated

it still requires some repetitive work from the user to load the generated Z file and select each proof obligation in turn to apply the proof tactics. Work is underway to encapsulate the functionality of CADiℤ within an API. This will provide the opportunity to fully integrate the formal analysis into specification modelling tools. Instant feedback on the properties being analysed can then be presented to the user using the same format as the original specification. The details of the analysis would be recorded (as the intermediate specification and proofs in Z) for review by engineers with the relevant formal methods skills.

5. Automatic test case generation

The formal specification describes each atomic action defined by the requirements specification. These operations can be used as basic test specifications. If data can be found to satisfy these constraints, the results of applying the data to the implementation can be used to gain confidence in its correctness with respect to the specification. The success of testing depends on the ability to select data that demonstrates the presence of a fault in the program. Category-partitioning [21] is a method of deriving tests based on a formal specification and testing heuristics based on common error types. Test data generated for the partitioned specification is then assumed to have a greater chance of detecting errors in the implementation (at least errors of the type used to formulate the testing heuristic). This approach was first applied to formal specifications by Ostrand and Balcer [21] and has been developed and applied to the formal specification notation Z by Stocks and Carrington [25].

The category-partition method is based on the theory of equivalence classes [8]. The input domain of the test specification is partitioned into sets of data that exhibit the same behaviour in the specification. If the equiva-

$\underline{\quad DoorDeployedOperationPartition1 \quad}$
$RamPosition? : \mathbb{N}$
$Hinge? : \mathbb{N}$
$DeployLock? : Activated \mid Deactivated$
$DoorDeployed! : Boolean$

$(FullyRetracted \leq RamPosition? \leq FullyExtended \wedge 0 \leq Hinge? \leq 90) \Rightarrow$
$(\boxed{\textbf{\textit{RamPosition? = DeployedPosition}}} \wedge$
$(RamPosition? \geq DeployedPosition \wedge Hinge? \geq 80 \wedge DeployLock? = Activated) \wedge$
$\quad DoorDeployed! = True) \vee$
$((RamPosition? < DeployedPosition \vee Hinge? < 80 \vee DeployLock? = Deactivated) \wedge$
$\quad DoorDeployed! = True)$

$\underline{\quad DoorDeployedOperationPartition2 \quad}$
$RamPosition? : \mathbb{N}$
$Hinge? : \mathbb{N}$
$DeployLock? : Activated \mid Deactivated$
$DoorDeployed! : Boolean$

$(FullyRetracted \leq RamPosition? \leq FullyExtended \wedge 0 \leq Hinge? \leq 90) \Rightarrow$
$(\boxed{\textbf{\textit{RamPosition? = DeployedPosition + 1}}} \wedge$
$(RamPosition? \geq DeployedPosition \wedge Hinge? \geq 80 \wedge DeployLock? = Activated) \wedge$
$\quad DoorDeployed! = True) \vee$
$((RamPosition? < DeployedPosition \vee Hinge? < 80 \vee DeployLock? = Deactivated) \wedge$
$\quad DoorDeployed! = True)$

$\underline{\quad DoorDeployedOperationPartition3 \quad}$
$RamPosition? : \mathbb{N}$
$Hinge? : \mathbb{N}$
$DeployLock? : Activated \mid Deactivated$
$DoorDeployed! : Boolean$

$(FullyRetracted \leq RamPosition? \leq FullyExtended \wedge 0 \leq Hinge? \leq 90) \Rightarrow$
$(\boxed{\textbf{\textit{RamPosition? > DeployedPosition + 1}}} \wedge$
$(RamPosition? \geq DeployedPosition \wedge Hinge? \geq 80 \wedge DeployLock? = Activated) \wedge$
$\quad DoorDeployed! = True) \vee$
$((RamPosition? < DeployedPosition \vee Hinge? < 80 \vee DeployLock? = Deactivated) \wedge$
$\quad DoorDeployed! = True)$

Figure 7. Test partitions for DoorDeployed

lence class hypothesis is assumed to hold in the implementation, only a selection of data from each equivalence class is needed to show that the implementation satisfies the specification for all data in that class.

As an example, the operation defining the *DoorDe-ployed* function (from figure 2, but corrected based on the completeness analysis described above) will now be partitioned to verify that the boundary used to define when the hydraulic ram is in the deployed position is correctly implemented in the code. The Z specification

$$\forall X, Y : \mathbb{N} \bullet X \geq Y \Leftrightarrow$$
$$X = Y \vee$$
$$X = Y + 1 \vee$$
$$X > Y + 1$$

Figure 8. Generic test heuristic for $\geq$

of the operation is given in figure 6. The ? and ! decorations are used to distinguish between the input and output parameters to the schema. Based on the assumption that errors often occur on or around boundaries, applying a boundary value analysis partitioning strategy would result in the partitions shown in figure 7. The additional constraints added by the partitioning are shown in bold. These partitions together with those generated for the condition where the hydraulic ram is not in the deployed position, from the second guard in figure 2, fully test the boundary.

The category-partition method has been automated as extensions to the CADi$\mathbb{Z}$ theorem prover. Partitioning heuristics are specified as lemmas and general-purpose proof tactics are used to apply the heuristics via the graphical user interface. The predicate to be partitioned is highlighted and a proof tactic invoked via a menu option which automatically introduces the partitioning heuristic into the operation conjecture, instantiates the generic heuristic with the operands of the predicate and simplifies the whole conjecture to reveal a disjunction of partitions. Each partition is then converted into a separate schema operation. The lemma used to create the partitions shown in figure 7 is given in figure 8. The user is also given the opportunity to amend the supporting English language description of the operation being partitioned, to include for example the rationale behind using the particular partitioning strategy.

The method of specifying the heuristics as lemmas, stored in a separate Z library file, which are then 'cut' into the operation has several important advantages. Properties of the heuristics themselves can be proven (e.g. that the partitions together maintain the state-space of the operation). If more heuristics are required (e.g. based on common errors specific to the system under development), they can be added without making a change to the software itself. The test specifications can be instantiated with test data via a similar mechanism to the test partitioning. The test specification is highlighted and an option called from within a CADi$\mathbb{Z}$ menu. A proof tactic is then automatically applied that simplifies the constraint and applies either the SMV model checker or simulated annealing constraint solver to generate a set of data satisfying the test specification.

Once the test data has been generated, CADi$\mathbb{Z}$ pro-

duces a corresponding AdaTEST [16] test script. AdaTEST provides a harness for automating the execution, checking and documentation of tests for software written in the Ada language. AdaTEST can also record the structural code coverage achieved by running the tests. Manually producing these test scripts, consumes a large proportion of the test engineers time. By automating this step, effort that was previously required for test implementation can now be redirected towards more rigorous test design. The generated test scripts also include the informal text derived from the original requirements and annotated with the test rationale during partitioning. This text is automatically included in the AdaTEST test results file and provides the traceability between any suspected fault in the program, the requirement under test and the heuristic used in designing the test.

The test specifications for the case study were first partitioned to give Modified Condition/Decision Coverage[7] (MC/DC) of each operation. Additional tests were then generated based on other heuristics, such as boundary value analysis. If full MC/DC (as mandated by certification standards such as D0-178B [23]) was not achieved by running these tests, it was assumed that the untested code represented refinements in the design or potential errors. Additional manual test effort then concentrated on writing tests for and reviewing these potentially problematic areas of code. The targeting of testing resources in this way was made possible by the high amount of automation achieved in generating the requirements covering tests.

6. Results and Evaluation

A summary of the specification validation and testing work performed for the thrust reverser case study is shown in figure 9. The numbers include only automatically generated proof obligations and tests and the requirements errors found by discharging the proofs. An activity is said to be automated if it requires at most a single interaction from the user to perform (e.g. a proof is discharged by selecting a completeness conjecture and choosing "Completeness Check" from the on-screen menu). Consequently these activities take very little time to perform. Many of the proof obligations stretched over 4 pages of formal text. Each of these would have taken an engineer a significant amount of time to prove or disprove. On a Pentium II 400 MHz computer running the linux operating system, the largest of these proofs took no more than a second to discharge.

The activities in the process described in this paper that have so far been automated are: automatic generation of a formal specification and associated healthiness

---

[7]MC/DC is achieved by showing that each condition within a decision can independently affect that decision's outcome[23].

State-based components:	
State-machines	9
States	48
Transitions	112
Z operations	112
Specification validation proofs	74
Automatically discharged proofs	74
Requirements errors found	18
Automatically generated tests	262
Reactive components:	
Tables	34
Definition/Guard pairings	84
Z operations	34
Specification validation proofs	62
Automatically discharged proofs	52
Requirements errors found	18
Automatically generated tests	237

Figure 9. Summary of results

property proof obligations from a Statechart modelled using STATEMATE, automatic proof or generation of a counter-example for PFS (Statechart and tabular) completeness and determinism healthiness properties, automatic partitioning of formally specified operations (derived from Statecharts of tabular requirements) into test cases based on pre-defined heuristics and the automatic generation of test data for the partitions and associated AdaTEST test script. Activities that we believe can be automated or are already in the process of being automated are; automatic generation of a formal specification and associated healthiness proof obligations from PFS tabular requirements (given a consistent form of recording and managing these requirements), automatic identification of the healthiness property proof obligations within the Z specification and application of the appropriate proof tactics and the automatic selection of partitioning strategies to generate test sets to satisfy particular structural specification coverage criteria.

The results show that a significant number of requirements errors were detected for little additional effort. All the requirements errors detected using this method manifested themselves as either non-determinism or incompleteness of the specification (as would be expected based on the nature of the checks). On analysis of these errors we discovered two distinct causes. The first type of error was the result of a mis-interpretation of some higher level requirements that resulted in an incorrect specification with respect to these requirements. These errors accounted for the greater proportion of total requirements errors found. The second type of error was non-determinism or incompleteness as the result of some omission or ambiguity in the higher level require-

ments. Although these errors were less frequent (potentially because the analysis was not specifically targeted at validating the higher level requirements) they were deemed to be very valuable.

A far greater number of tests were generated than would have been written for a specification of this size using the traditional process. The number of tests that can be written are typically restricted by the time it takes to design, implement and evaluate each test, in the process described here much of this effort has been automated, greatly reducing the amount of effort per test case. When the analysis or test revealed an error, the time taken to review and rework the error varied according to the nature of the problem. However, the impression amongst those involved was that the counter-example information and supporting informal text greatly contributed to the process of tracking down the errors in the requirements. It was also noted that as the case study progressed the number of requirements errors being detected decreased significantly. The feedback from the formal analysis was thought to have influenced the style of requirements specification, i.e. the author of the requirements was consciously writing specifications to meet the healthiness conditions specified by the PFS methodology.[8]

In [18], Knight presented the following criteria for industrial acceptance of formal methods. Based on the evidence from the case study and experience of working with our industrial partners we can now assess the industrial suitability of our work along similar lines.

1. Formal methods must not detract from the accomplishments achieved by current methods

2. Formal methods must augment current methods so as to permit industry to build "better" software

3. Formal methods must be consistent with those current methods with which they must be integrated

4. They must be compatible with the tools and techniques that are currently in use

These criteria emphasise the need to develop formal methods for the types of practical tools and notations used in industry and also for formal methods to complement and not preclude existing practices. It is the authors' opinion that the work described in this paper has gone some way to satisfying these criteria, although admittedly for a particular domain and set of V&V activities. This was accomplished by basing the formal

---

[8]The final validation of the system will tell whether the requirements indeed improved or whether errors were instead being introduced into areas not covered by the healthiness conditions.

analysis and test case generation activities on an automatically generated formal representation of the intuitive requirements specifications, written using existing modelling tools. In addition, the activities performed here complement methods already in use such as review and animation. As such they can be seen as natural extensions to the existing modelling process.

Formal specifications are typically very sensitive to change. However, due to automation, the formal specifications can be re-generated and verified whenever a change in the requirements occurred, at little extra cost. The intuitive engineering requirements remained the first class citizens of the process and the standard interface to the engineers. The ongoing extensions to CADiℤ to provide a 'silent' interface via an API will allow modelling environments such as Statemate to exploit an intermediate formal representation of the requirements to perform checks and generate tests while hiding the details of the analysis from the engineer. This would further encourage an iterative development of the requirements (i.e. do not pass onto the coding phase until the requirements have been properly validated) and increase the efficiency of the test generation process.

7. Conclusions and Future Work

In this paper we described our experiences in integrating formal methods into an industrial software development process. Intuitive engineering notations were translated into intermediate formal specifications which formed the basis of automated proof and test case generation activities. The high level of automation was made possible by restricting the work to a particular domain (discrete engine control requirements) and a tight subset of an otherwise highly expressive formal notation (Z). The automated analysis and tests allowed a significant amount of errors to be detected earlier than would have been possible had a manual, ad-hoc approach been taken.

The findings support other work [5, 13, 1] that has similarly used formal semantics of engineering notations to facilitate an effective approach to verification. The work presented here contributes to this field by showing that general purpose formal analysis tools such as CADiℤ and SMV can be used to support automated analysis and test generation based on different engineering notations given a suitable translation from the notations to a formal specification.

In developing these techniques we have identified the following generic process for applying formal methods to engineering notations for automated V&V.

1. **Use intuitive Engineering notations with fixed semantics:** Record and maintain the requirements specifications in notations most suitable for the domain and whose semantics can be formally specified.

2. **Couch healthiness conditions as mathematical constraints:** Identify "healthiness properties" that should be common to all specifications e.g. completeness and determinism. Based on the formal semantics of the notations, couch these properties as mathematical constraints. Automate the translation if possible.

3. **Formally specify the behaviour under test:** Define a translation between the original notation and a formal specification of the properties under test based on the formal semantics. Automate the translation if possible.

4. **Exploit existing tool-support:** Apply a combination of automated proof, model checking and constraint solving to analyse the healthiness properties, to generate test specifications and to generate test data.

5. **Complement the formal specification and tests with informal text:** To aid the review and documentation of the analysis and test results, informal text descriptions should be generated and maintained to describe the formal specification of the healthiness properties and tests.

Ongoing work aims to increase the level of automation and integration of the formal techniques into existing specification modelling environments with the development of a CADiℤ API. In addition to this, we also hope to expand the generic process and the toolset to cover more areas of the verification process. In particular, the refinement of the intermediate formal specification into program annotations that can be used to discharge correctness proofs on the code and the automatic efficient sequencing of test cases to reduce the amount of effort required to physically run the generated tests. Other work will concentrate on developing further the constraint solving abilities of the CADiℤ theorem prover. This will allow a wider range of specifications and properties to be automatically verified than the current system.

8. Acknowledgements

References

[1] Jim Armstrong. Industrial integration of graphical and formal specifications. *Systems Software*, 40:211–225, 1998.

[2] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[3] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 1999. The latest version of SMV and its documentation may be downloaded from this site.

[4] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment Work-Bench Feasibility Assessment.

[5] Nancy Day, Jeffrey Joyce, and Gerry Pelletier. Formalization and analysis of the seperation minima for aircraft in the north atlantic region. *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, pages 35–49, September 1997.

[6] David Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, April 1996.

[7] Andy Galloway, Trevor Cockram, and John McDermid. Experiences with the application of discrete formal methods to the development of engine control software. *Proceedings of DCCS '98. IFAC*, 1998.

[8] John B Goodenough and Susan L Gerhart. Towards a theory of test data selection. *IEEE Transactions On Software Engineering*, 1(2):156–173, June 1975.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, , Michael Politi, Rivi Sherman, Aharon Shtull-Truaring, and Mark Trakhenbrot. Statemate, a working environment for the development of complex reactive systems. *IEEE Transactions On Software Engineering*, 16:403–414, 1988.

[11] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *IEEE Transactions On Software Engineering And Methodology*, 5(4):293–33, Oct 1996.

[12] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1), 1980.

[13] Constance Hietmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. *In Proceedings of International Symposium on Requirements Engineering*, pages 56–63, 1995.

[14] M.G. Hinchey and J. P. Bowen (editors). *Applications of formal methods*. Prentice-Hall, 1995.

[15] C. Michael Holloway and Ricky W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, April 1996.

[16] Information Processing Ltd., Bath, UK. *AdaTEST 95 User Manual*, June 1997.

[17] Steve King, Jonathon Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1527–1545. Springer-Verlag, September 1999.

[18] John Knight, Colleen DeJong, Matthew Gibble, and Luis Nakano. Why are formal methods not used more widely? *Proceedings of the Fourth NASA Langley Formal Methods Workshop*, pages 1–12, September 1997.

[19] John McDermid, Andy Galloway, Simon Burton, John Clark, Ian Toyn, Nigel Tracey, and Samuel Valentine. Towards industrially applicable formal methods: Three small steps, one giant leap. *Proceedings of the International Conference on Formal Engineering Methods*, October 1998.

[20] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On the formal semantics of statecharts as supported by Statemate. *Proceedings Of The Second Northern Formal Methods Workshop*, July 1997.

[21] Thomas J Ostrand and Marc J Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[22] Martyn Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.

[23] RTCA. *RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[24] J. M. Spivey. *The Z Notation: A Reference Manual, second edition*. Prentice Hall, 1992.

[25] Phil Stocks and David Carrington. Test template framework: A specification-based case study. *Proceedings Of The International Symposium On Software Testing And Analysis (ISSTA'93)*, pages 11–18, 1993.

[26] I. Toyn. Formal reasoning in the Z notation using CADiℤ. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[27] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.

[28] Ian Toyn. The CADiℤ web site. http://www.cs.york.ac.uk/~ian/cadiz/, 1999. The latest version of CADiℤ and its documentation may be downloaded from this site.

[29] Sam Valentine. *Modeling Statemate Statecharts In Z (DCSC/TR/98/15)*. Dependable Computer Systems Centre (DCSC), University of York, October 1998.

Integrating Z and Cleanroom

Allan M. Stavely
Computer Science Department
New Mexico Tech

Abstract

We describe an approach to integrating the Z specification notation into Cleanroom-style specification and verification. In a previous attempt, a group at IBM used formal refinement of the Z in their development. They concluded that this was not cost-effective in a commercial environment, and the attempt was not judged successful. The current approach avoids formal refinement, and instead begins by converting the Z to a fully constructive form, expressing all state changes using an assignment notation. The development then proceeds in Cleanroom style, with sections of the Z specification simply distributed among the program components to serve as their specifications. In a pilot project, this approach was found to work quite well, with development proceeding smoothly and predictably as normally expected with Cleanroom methods.

1   History of the problem

In the early 1990s, a group of technical staff at the IBM laboratory at Hursley Park (near Winchester, England) attempted to integrate two software engineering technologies which IBM had previously used separately with considerable success: the Z specification notation and the Cleanroom method.

The Z notation [15] [6] [13] [17] [18] is based on set theory and other basic elements of discrete mathematics, and incorporates novel structuring constructs (schemas and the schema calculus). Z technology also includes methods for the formal refinement of specifications into designs and code.

The core of the Cleanroom method [10] [8] [16] is formal or semiformal specification, and corresponding verification done by a development group in review meetings. Other elements of the method include notations and techniques for stepwise refinement, testing based on expected usage patterns, statistical analysis of test results to predict product quality, and incremental development.

IBM had had considerable experience with both technologies. The Cleanroom method was developed largely at IBM, by Harlan Mills and his colleagues in the Federal Systems Division. By the time of the Hursley experiment, it had been used successfully on a number of industrial-sized projects at IBM and elsewhere. The results were striking: very low levels of defects in the products, with no net loss and often a net gain in productivity [8] [3].

IBM had just finished a substantial development project at Hursley using Z, in collaboration with its developers at Oxford University [5]. The project was a major new release of the CICS transaction processing system: 268,000 lines of new and modified code, of which 37,000 lines were specified and designed using Z and another 11,000 lines were partially specified in Z. For the parts produced using Z, IBM reported a higher percentage of defects eliminated early in the development, a lower level of defects in the final product, and an estimated 9% reduction in development costs. IBM and Oxford were jointly given the Queen's Award for Technological Achievement for 1992 on the basis of this work.

The CICS group at Hursley hoped that Z and Cleanroom methods could be used together, and would complement each other to produce products of even higher quality than with either separately. The approach that they took was to write specifications in Z initially; to proceed with formal refinement steps as normally done in Z; to write the correctness criteria for these refinements as mathematical theorems; and to prove these theorems in review meetings, as normally done in Cleanroom.

The experiment was not judged a success. In particular, the group found it too hard to do the formal refinement from Z into code. The postmortem [12] concluded that "it is not cost-effective in a commercial software environment to do even semi-formal refinement without machine assistance" (which was not available).

Despite this discouraging result, we felt that there

was much to be gained if Z and Cleanroom methods could be integrated successfully. In the following, we describe a quite different approach. We avoid formal refinement in Z altogether, and instead begin by translating Z specifications into a form that more closely resembles Cleanroom-style specifications. From there, the development proceeds in Cleanroom style, but retaining fragments of Z notation where appropriate. We found that, using this approach, the Z notation can complement Cleanroom methods quite effectively.

2   Z and Cleanroom specification styles

The Z notation is well suited to expressing the specification of a system as a whole, or of major parts of a system. It provides a great deal of useful mathematical vocabulary, and the vocabulary of discrete mathematics in particular, which can be used very effectively to specify aspects of an information-processing system at a high level. Furthermore, it provides the schema notation and the schema calculus, by means of which many different aspects of a specification, each perhaps derived from a different requirement of the system, can be expressed separately and then combined into a single specification.

The Cleanroom method, on the other hand, provides relatively little built-in notation. Indeed, one of its strengths is that many kinds of notation, from a wide variety of domains and at many levels of formality, can be imported into it and used in its specifications. What it does provide is, in particular, a straightforward method of placing specifications on the lower-level components of a program, down to the level of the control construct or statement, and verifying that these components satisfy their specifications.

It would seem to be a natural idea, then, to begin by writing the top-level specification of a system using Z, and then to proceed with the development in Cleanroom fashion, distributing the Z specification among the program components and verifying those components against the specification fragments using Cleanroom protocols in review meetings.

However, there is a gap that must be bridged before the Z notation can be incorporated into Cleanroom-style specifications. This is because there are fundamental differences in the styles of the specifications of Z and Cleanroom.

The Z notation is based on predicates, which express preconditions and postconditions on operations, invariants on data, and other assertions and constraints on the data objects and inputs and outputs of a system. In particular, the specification of an operation defines a relation among inputs, outputs, previous values of state vari-

ables, and new values of those variables.

A fundamental property of Z is that such specifications may be *nonconstructive:* they may express properties that outputs and new values of variables must satisfy, without giving any clue as to how these values can be calculated from inputs and previous values of variables. In fact, specifications may even be *nondeterministic:* they may not constrain each output and updated variable to a unique value.

Here is an example which is both nonconstructive and nondeterministic, from the specification of a text-processing system: [6, p. 172]:

$$[CHAR]$$
$$TEXT == \text{seq } CHAR$$

$$
\begin{array}{|l}
\hline
\_Format_____ \\
t, t' : TEXT \\
\hline
words\ t' = words\ t \\
\forall l : \text{ran } (lines\ t') \bullet \#l \le width \\
\hline
\end{array}
$$

In the specification of an operation in Z, the name of a state variable is "decorated" with a $'$ symbol to refer to its new value; the undecorated variable name refers to its previous value. (Input variables are decorated with ? and output variables with !.) Thus, this schema says that a *Format* operation leaves the sequence of words in t unchanged and that each line of t after the operation must be no longer than *width* (the functions *words* and *lines* and the constant *width* are defined elsewhere). The specification says nothing about how to achieve this result and, in fact, there will usually be many ways of dividing t into lines that will satisfy this specification.

Z practitioners see the ability to write nonconstructive and nondeterministic specifications as an advantage:

> Non-deterministic operations are important because they sometimes allow specifications to be made simpler and more abstract [15, p. 131].

> Nonconstructive specifications achieve expressivity and brevity at the expense of executability ... they leave the programmer free to choose among different implementation strategies [6, p. 38].

In the Cleanroom method, on the other hand, the general rule is that specifications are both deterministic and constructive. Specifications are written in the "functional" style [9], in which each operation, control construct and statement in a program is viewed as computing a function on the program's state:

$$X := f(X)$$

Here X is a state vector that encompasses all of the program's state variables, including its input and output streams. Specifications are written in the form of *intended functions* which explicitly give values for every state variable which changes value. The usual notation is the *concurrent assignment*, such as the following:

$$[\; sum, \; i, \; trend :=$$
$$sum + a[i], \; i + 1, \; (sum + a[i])/(i+1) \;]$$

A variant is the *conditional concurrent assignment*, which specifies a state change by cases, such as the following:

$$[\; i > 0 \; \rightarrow \; trend := sum/i$$
$$| \;\; i = 0 \; \rightarrow \; sum, trend := 0, trend_0 \;]$$

Each case has a precondition and a concurrent assignment which is the state change to be performed when the precondition is satisfied; the computation is undefined whenever no precondition is satisfied.

The usual situation is that the preconditions of a conditional concurrent assignments are mutually exclusive (there is no state in which any two are both true) and that the right-hand side of each concurrent assignment contains only single-valued expressions which are obviously computable. In this case, the specification is deterministic and constructive. Exceptions are occasionally made, and occasionally a specifier will depart from this notation entirely. However, the rest of the Cleanroom method, and the verification in particular, will usually proceed more smoothly if the above conventions are followed. One reason for this is that a common manipulation in verification is to substitute the result of a computation into the specification of a following computation and then simplify.

3 The transition from Z to Cleanroom

The first step in our adaption of a Z specification to Cleanroom-style development and verification, then, is to transform the Z into a deterministic and constructive specification, so that it can be expressed using the intended functions required by the Cleanroom method. It might seem that this would be a nontrivial task, requiring a great deal of effort and introducing many opportunities to make mistakes that will jeopardize the success of the project.

However, in our experience thus far, we find that the job is usually not as hard as one might think. This is largely because many parts of typical Z specifications are already deterministic and constructive. In particular, we find that many Z predicates are of the form

$$v_1 = e_1 \wedge v_2 = e_2 \wedge \ldots$$

or

$$P \wedge v_1 = e_1 \wedge v_2 = e_2 \wedge \ldots$$

or

$$(P_1 \wedge v_{11} = e_{11} \wedge v_{12} = e_{12} \wedge \ldots) \vee$$
$$(P_2 \wedge v_{21} = e_{21} \wedge v_{22} = e_{22} \wedge \ldots) \vee \ldots$$

where each v is a changed state variable or an output variable (i.e., an variable decorated with $'$ or $!$) and such variables do not occur in any P or e, and where (in the third form) the P_i are mutually exclusive. Such predicates define computations that are clearly both deterministic and constructive, assuming that each P and e is single-valued and there is an obvious way to compute it. Furthermore, it is trivially easy to rewrite any such predicate in conditional-concurrent-assignment form. In fact, they are essentially in that form already, except for the symbols used.

Fortunately, such forms are natural to use in many situations in Z specifications, and Z users seem to use them rather commonly. In 28 case studies presented in six prominent Z books [15, ch. 1] [4, parts B–D] [6, ch. 16–25] [13, ch. 8] [17, ch. 15 and 20–23] [18, appendix A], over 67% of the 353 schemas which imply state changes or output are already in one of the above forms, once the schemas that are defined by including or combining other schemas are expanded out into their full forms. Another 6% contain instances of (for example) the new value of one variable being defined in terms of the new value of another, in contexts like

$$a' = f(a) \wedge$$
$$b' = g(a')$$

in which the departure from the above forms can easily be eliminated by an obvious substitution. Again, the translation to conditional-concurrent-assignment form is straightforward.

We could proceed, then, by translating all of the specifications of operations directly from Z predicates to conditional concurrent assignments, routinely in the easy cases and using more complex transformations in the other cases. However, to make the transition smoother, we devised an intermediate form which combines characteristics of both notations. It is very much like standard Z — in fact, it can be considered a nonstandard dialect of Z — except that all state changes are specified explicitly and constructively.

Here are the principal differences between this notation and standard Z.

- State changes are written in the form

$$x := E$$

This is equivalent to the standard Z

$$x' = E$$

but the change in notation emphasizes the explicit, constructive definition of the state change. The same assignment notation is used to specify the computation of outputs.

- Every change to a state component is specified in this way; it is implied that no other state component changes its value. With this convention, all assertions of the form

$$x' = x$$

are omitted as redundant from schemas that specify state changes.

There are no implicit changes to one state component induced by changes to another state component and constraints between them. All state changes are written out explicitly.

- In the same spirit, where it is asserted that part of a structured state component is changed, it is implicit that the rest of the component remains unchanged. In particular, where the state component is a mapping (in Z represented as a function), a change to its value on one element of its domain can be written in the form

$$f\, a := E$$

If this is the only change to f that is specified in the schema in which this appears, it is implied that f remains unchanged otherwise, and the above is equivalent to the standard Z

$$f' = f \oplus \{\, a \mapsto E \,\}$$

where $\oplus$ is the "override" operator.

More than one change to the same function can be specified:

$$f\, a_1 := E_1 \wedge f\, a_2 := E_2$$

means

$$f' = f \oplus \{\, a_1 \mapsto E_1, a_2 \mapsto E_2 \,\}$$

which, of course, is well-defined (i.e., f' is still a function) only if $a_1 \neq a_2$ or $E_1 = E_2$.

A change to a (curried) function of two arguments can be written as

$$f\, a\, b := E$$

which (if no other changes to f are specified in the schema in which this appears) is equivalent to

$$f' = f \oplus \{\, a \mapsto ((f\, a) \oplus \{\, b \mapsto E \,\}) \,\}$$

and so on for functions of more arguments.

- Since the syntax $x := E$ is really a predicate, it can appear anywhere a predicate can appear, such as within the scope of a quantifier. An example is

$$\forall x : T \mid x \in S \bullet$$
$$f\, x := a$$

which means

$$f' = f \oplus \{\, x : T \mid x \in S \bullet x \mapsto a \,\}$$

- The symbols Δ and Ξ are now superfluous in most places and may be omitted.

- All computations of new states and outputs appear only in contexts which are unconditional, or in conditional structures (using $\vee$ and $\wedge$) with mutually exclusive conditions.

Many of these notation conventions are similar to constructs in the notation of the B method [19], although that notation is more restrictive than AZ in a number of ways.

Since state changes are specified in the form of assignments, we tentatively call this variant of the Z notation "Assignment Z", or AZ. (We considered the name "Constructive Z", but this name is already in use with a somewhat different meaning [11].) We present AZ not as another formal specification notation, but merely as an informally-defined intermediate form between Z and conditional concurrent assignments.

Where the Z specifications are not already constructive, we transform them into a constructive form as we rewrite them in AZ notation. For example, we would rewrite

---
Pop
$stack, stack' : \text{seq } Item$
$x! : Item$

$stack = \langle x! \rangle \frown stack'$

---

as

```
┌─ Pop ─────────────────────────
│ stack : seq Item
│ x! : Item
├───────────────────────────────
│ x! := head stack
│ stack := tail stack
└───────────────────────────────
```

Often, as here, making a state change constructive is rather easy, but it can require considerable manipulation.

There is sometimes more than one way to express the constructive version, and whatever choice is made will usually suggest a design or implementation possibility more strongly than the nonconstructive version did. Similarly, where the specification is nondeterministic, making it deterministic typically involves either making arbitrary choices as to the result that is specified, or making choices influenced by design or implementation considerations. An example is an allocation of a resource from a set of numbered resources:

```
┌─ Allocate ────────────────────
│ free : 𝔽 ℕ
│ allocated' : ℕ
├───────────────────────────────
│ allocated' ∈ free
│ free' = free \ allocated'
└───────────────────────────────
```

(Here 𝔽 means "finite set of" and ℕ denotes the natural numbers.) As we convert this to AZ form, we might make it deterministic by arbitrarily choosing the free resource with the minimum number:

```
┌─ Allocate ────────────────────
│ free : 𝔽 ℕ
│ allocated : ℕ
├───────────────────────────────
│ allocated := min free
│ free := free \ {min free}
└───────────────────────────────
```

Another way of resolving the nondeterminism would be to define *free* to be a sequence rather than a set, and choosing the first element of the sequence every time:

```
┌─ Allocate ────────────────────
│ free : seq ℕ
│ allocated : ℕ
├───────────────────────────────
│ allocated := head free
│ free := tail free
└───────────────────────────────
```

Clearly, this version encourages a different implementation. It is important to realize that the transformations that we perform to make the specification constructive and deterministic are not just changes in notation, but are true development steps, and may involve nontrivial and significant design decisions.

It is probably not necessary to be too dogmatic about removing all nonconstructive and nondeterministic aspects of the specification at this stage. Consider this example:

```
┌─ DisplayPeople ───────────────
│ knownPeople : 𝔽 PERSON
│ people! : 𝔽 PERSON
├───────────────────────────────
│ people! = knownPeople
└───────────────────────────────
```

This is reasonable Z, but of course if a set is displayed as an output, it must be displayed in some order. A possible conversion to AZ might be:

```
┌─ DisplayPeople ───────────────
│ knownPeople : 𝔽 PERSON
│ people! : seq PERSON
├───────────────────────────────
│ people! = alphabeticalSort knownPeople
└───────────────────────────────
```

where *alphabeticalSort* denotes sorting by name in phone-book order. One might object that this is still both nonconstructive and nondeterministic, since it does not suggest how the sorting is to be done, and it may allow more than one outcome if more than one person can have the same name. But any other way of writing this specification is likely to be more complicated and less satisfactory. Furthermore, it is obvious that any competent programmer can create an implementation that will satisfy it. In such situations the pragmatic thing to do may be to allow specifications such as this one, although we should do so only after careful consideration.

4 A pilot project

To try out the techniques presented above, we attempted use them on a development project of modest size. As it happened, we had a project that we were already planning to undertake.

The project was to develop a "rehearsal scheduler's assistant": a program to help with the planning and scheduling of the rehearsals and other preparation for a theatrical production. The central job to be done is to manage the interacting schedules of many activities and many people. We had a real client, Doug Dunston, the faculty member in charge of the music program in our college. (The project is described in somewhat fictionalized form in section 11.3 of [16].)

The first step of the development, after discussing requirements with the client, was to prepare a specification

in standard Z. As is common practice with Z, the specification took the form of a document, with sections of Z interspersed with explanatory text in English. The specification contained 43 schemas, and 16 other Z sections containing definitions of various kinds.

The specification document contained one other important specification notation: color pictures of the screens and other components of the graphical user interface (GUI). Here is an example:
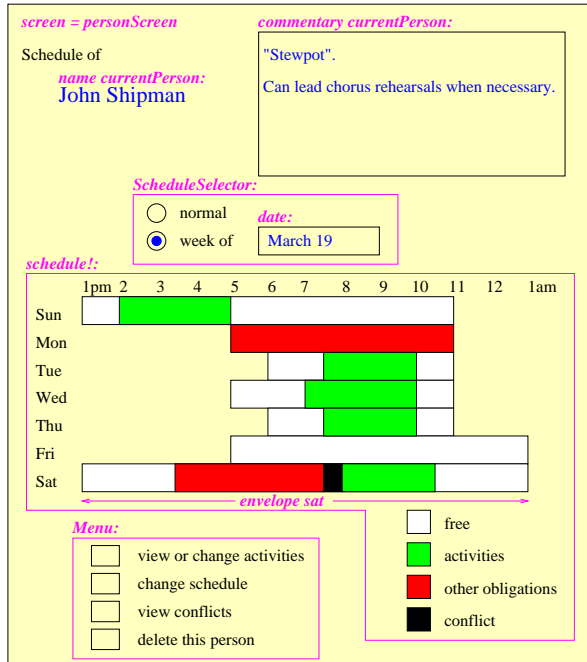
We used pictures like these to include in the specification a general idea of what the GUI would look like. However, we adopted the convention that the pictures would represent only an approximation to the appearance of the interface, which might vary somewhat according to the eventual implementation. Colors and dimensions (for example) might be slightly different from the way they appear in the pictures, and there might be implementation-dependent features not shown in the pictures, such as additional ways to move from one screen to another.

On the other hand, some aspects of the pictures are quite specific. In particular, some of the elements of each picture are tied to the Z specifications through labeling conventions. In the picture, an annotation of the form *someName:* (which appears in a distinguished color, magenta, when the specification is printed or displayed in color) is not to appear in the actual GUI as displayed, but indicates that the corresponding part of the GUI corresponds to the construct of the same name

in the Z specification. If there might be any doubt as to what part of the display is being referred to, a box of the same color is drawn around the relevant part; again, this will not actually appear in the GUI.

Here is the Z schema that corresponds to the above picture:

$$
\begin{array}{l}
\underline{\quad PersonScreen \quad}\\
PersonData \\
ScheduleSelector \\
PersonScheduleDisplay \\
Menu[SCREEN] \\
\hline
screen = personScreen \\
preselected! := normal \\
date = today \\
menuChoices! := \\
\quad \{ editPersonScreen \mapsto \\
\quad\quad\quad \text{``view or change activities''}, \\
\quad\quad personScheduleScreen \mapsto \\
\quad\quad\quad \text{``change schedule''}, \\
\quad\quad showConflictsScreen \mapsto \\
\quad\quad\quad \text{``view conflicts''}, \\
\quad\quad deletePersonScreen \mapsto \\
\quad\quad\quad \text{``delete this person''} \} \\
screen := chosenItem?
\end{array}
$$

The annotation **screen = personScreen** in the picture indicates that *screen* has the value *personScreen* when what the user sees is the screen shown in the picture. The variables *date* and *preselected!* are defined in the schema *ScheduleSelector*; *preselected!* defines which of the two selector buttons is initially shown as selected. Whenever a schema defines a GUI component, we define informally in the accompanying text how the Z components relate to what the user sees and can manipulate.

We adopted several other conventions to reduce the amount of repetitive detail in the specification. For example, in many places in the GUI there is a box in which the user can fill in or edit a value. Wherever a picture contains such a box labeled with the name (for example) *x*, and *x* is a variable of type *T*, we treat that annotation as implicitly introducing a Z schema of the form

$$
\begin{array}{l}
\underline{\quad Edit\_x \quad}\\
displayed\_x! : TEXT \\
entered\_x? : TEXT \\
x : T \\
\hline
displayed\_x! := TtoTEXT\ x \\
x := TEXTtoT\ entered\_x?
\end{array}
$$

where *TtoTEXT* and *TEXTtoT* are appropriate conversion functions. The box labeled *date:* is an example

of this; the schema *personScreen* also specifies that the value initially displayed for *date* is today's date.

The specification document, then, contains English text, Z notation, and pictures, all interrelated. It should be apparent that some parts of the specification are formal and other parts are quite informal. In all, the document is 42 pages long.

The next step, after meeting again with the client to discuss that specification and obtain his approval, was to prepare another version of the document in which the parts of the Z sections were rewritten in AZ form. This turned out to be quite easy in most places, especially since most of the state changes were specified in such a way that the translation was trivial, as discussed in the previous section. In many cases, the resulting specifications turned out to be considerably simpler than the original, largely because of the AZ convention for expressing changes to components of structured objects. For example, the specification used curried functions like the following in many key places:

$$SCHEDULESTATUS ::=$$
$$\quad free \mid booked \mid otherObligations \mid conflict$$
$$DAYSCHEDULE ==$$
$$\quad TIME \nrightarrow SCHEDULESTATUS$$
$$WEEKSCHEDULE ==$$
$$\quad DAYOFWEEK \rightarrow DAYSCHEDULE$$

NormalSchedules
. . .
People
normalSchedule :
$\quad PERSON \nrightarrow WEEKSCHEDULE$

. . .

This was a natural way of constructing *normalSchedule*, especially since we sometimes wanted to refer to the whole weekly schedule of a person, sometimes for that schedule on a particular day, and sometimes to *that* schedule at a particular time. But then specifications of state changes like the following became complex and tedious:

$$normalSchedule' = normalSchedule \oplus$$
$$\quad \{ currentPerson \mapsto$$
$$\quad (normalSchedule\ currentPerson) \oplus$$
$$\quad\quad \{ day \mapsto$$
$$\quad\quad (normalSchedule\ currentPerson\ day) \oplus$$
$$\quad\quad\quad \{ t \in possibleTimes\ day \mid$$
$$\quad\quad\quad\quad from \leq t < to \bullet$$
$$\quad\quad\quad\quad t \mapsto selected?\ \} \} \}$$

The AZ form of this is much more straightforward:

$$\forall t \in possibleTimes\ day \mid from \leq t < to \bullet$$
$$\quad normalSchedule\ currentPerson\ day\ t$$
$$\quad\quad := selected?$$

In determining what needed to be rewritten to make it constructive, we were guided by pragmatic considerations. For example, the original specification contained a number of state changes specified using set comprehensions, in forms such as

$$result := \{ a \in S \mid P(a) \}$$

But in each such case, *S* was a finite set and so, in principle at least, the set of its elements satisfying *P* could be constructed by a simple-minded enumeration of the set, testing each element. Indeed, for this reason a mathematician would probably consider such an expression quite constructive, and we judged all such specifications to be "constructive enough" for our purposes. In fact, in the implementation, each such set turned out to be reasonably small, and so this is exactly how almost every such state change was actually implemented.

For a number of reasons, including portability (the program was to be developed on our Linux machines but would eventually run on Dr. Dunston's Macintosh), we chose the Python programming language and the Tkinter GUI library for the implementation.

We found it easy to implement many parts of the AZ specification using Python constructs, in ways that so obviously matched the specification that verification was hardly necessary. This was especially true of state changes that called for modifying values of functions. We implemented the function *normalSchedule*, for example, as a dictionary indexed by *Person* and *Activity* objects, containing lists indexed by numbers representing days of the week, where those lists contained dictionaries indexed by *Time* objects and containing *ScheduleStatus* objects. Thus, for example, the implementation of the state change specified by

$$\forall t \in possibleTimes\ day \mid from \leq t < to \bullet$$
$$\quad normalSchedule\ currentPerson\ day\ t$$
$$\quad\quad := selected?$$

turned out to be simply

```
for t in possibleTimes(day):
   if fromTime <= t < toTime:
      normalSchedule[currentPerson] \
          [day][t] \
      = selected
```

which is essentially identical to the specification.

We used one other significant piece of software engineering technology in the project: a form of "literate

programming" [7]. This means that the program is prepared and presented in the form of a document, with explanatory text accompanying each section of program code. Thus the program and its documentation are integrated, and stored in a single file. There are software tools that process that file either to strip out and order the code sections for compilation and execution, or to format the document for viewing or printing.

In the usual kind of literate programming, the code fragments may appear in the document in any order, but the author must use markup commands to define their ordering and nested structure in the final program. We adopted a much more "lightweight" approach, in which the code fragments appear in the program in the same order in which they are presented in the document. This is reasonable with Python, in which the order of elements in a program is relatively unconstrained. And it means that the program that strips out the code fragments for execution does not need to do any reordering, which made it a very simple program. Even more important, it means that the programmer does not need to include markup to structure the program fragments. In fact, the only extra markup necessary is a pair of commands defined in the markup language (LaTeX in our case) to mark the beginning and end of a code fragment.

With very little extra effort, then, we were able to maintain the code and a description of it as a readable document. In many places we also included sections of Z and pictures of the GUI, cut-and-pasted from the specification documents. The result is a document very similar to those documents in style and appearance, but with code sections added. Each code section is accompanied by commentary and, in many cases, the Z and pictorial specifications from which the code was derived.

In many places we also extracted fragments of Z from the specification and incorporated them into intended functions in the code fragments. Here is an example:

```
                                                    V
def emptyDaySchedule(d):
[ returned value := a dictionary
     { t ∈ possibleTimes(d) • t ↦ free) } ]
   result = { }
   for t in possibleTimes(d):
      result[t] = free
   return result
```

By the way, the annotation "**V**" in the corner of the box indicates that this section of code has been verified. An annotation "**VT**" would indicate that it had also been tested. We used such annotations to keep track of the status of each fragment directly in the program document during the development, and we found this very helpful.

We did not follow Z protocols for formal refinement at all. The implementation was constructed using ordinary programming skills, as well as Cleanroom-based methods in which intended functions are implemented in a stepwise manner in terms of code and lower-level intended functions [16, ch. 5]. We constantly used the structure of the Z specification as a guide, and this made many parts of the implementation almost trivial to construct.

We verified both the translation of the Z to AZ, and the program code developed from the AZ. Unfortunately, we were unable to adhere strictly to Cleanroom methods in doing this. Cleanroom is inherently a group process; in particular, verification is done in review meetings, with the author and colleagues discussing each correctness criterion and examining the program for other aspects of quality. The goal is to discover and eliminate as many defects as possible while attempting the verification. Normally this requires a group of at least three people, since each person often notices defects that the others miss.

But only one other person trained in Cleanroom methods was available at the time, and the amount of time that he had available was quite limited. Therefore, parts of the program were verified in a two-person group, and some parts were done strictly as a solo effort. We found that verification under these circumstances was far less reliable than normally expected with Cleanroom methods, which typically achieve a level of defects of three per thousand lines of code or better before first execution [8] [3].

To add to our difficulties, we were somewhat unfamiliar with the Python language and the Tkinter library, and made a number of minor mistakes in usage, especially early in the project. Since Python is an interpreted language, the mistakes that escaped our notice during verification were not caught in compilation, but in first execution.

To attempt to compensate, we eventually developed an alternate protocol. The project plan called for the program to be developed in rather substantial increments, as is normal with the Cleanroom method. We divided each of these into a number of very small increments, each adding perhaps only one simple new feature to the program; these increments ranged in size from about thirty to two hundred new and changed lines.

After each of these increments was coded, we inspected it several times, using a checklist and checking different aspects each time. We checked such things as points of syntax and usage which had caused us prob-

lems before, matching of each function and method call against its definition (comparing both intended functions and number and types of parameters), and correspondence of intended functions with the Z in the specification document. Finally we inspected for correctness of each section of code with its intended functions. In some cases, as in the *normalSchedule* example above, we judged the code obviously correct "by inspection"; in other cases we carried through more detailed correctness arguments, mentally or on paper [16]. We caught and eliminated many defects by means of these inspections, about four defects per hundred lines on average.

Each increment was then integrated into the program; thus we were, in effect, "growing" the program gradually, as advocated by Brooks [1, p. 18]. At each integration step we ran a few cursory tests to execute each new piece of code for the first time, and many more defects showed up immediately. The defect density on first execution was about five per hundred lines on average, not nearly as good as normally expected with Cleanroom methods. Thus, our one-person inspection protocol does not come close to competing with a full Cleanroom-style verification review by this measure.

Fortunately, this had little effect on the effectiveness of the development! Almost all of the defects that survived the inspections were caught on first execution and were simple oversights: typographical errors, mistakes in punctuation, mistakes in names of variables, omitted initialization, and the like. Each took only a few minutes to track down and fix. There were no deep algorithm flaws, no subtle bugs which would cause malfunctions only rarely, and no places in which we had implemented algorithms that would do something quite different from what was specified. This is typical of what normally happens in a Cleanroom-style development: the really nasty bugs are the ones that specification and verification seem to be most effective at preventing or eliminating.

Most important for the subject of this paper, the entire development went very smoothly. At no time did we feel that the mixture of notation was a hindrance or added excessive complexity to the process. On the contrary, we felt that it was definitely helpful to have a Z specification to use a a basis for the development, and that specifying the program using the vocabulary of discrete mathematics right from the beginning probably made the design cleaner than it would otherwise have been. We also felt that using Cleanroom-style intended functions and stepwise refinement definitely contributed to the quality of the product, as did inspections, imperfect as the latter were. These are subjective judgments for the most part, of course, but we think they are justified.

At the time of writing, the third of the five major increments called for in the project plan has been com-

pleted, resulting in 1409 nonblank, non-comment lines of code in the Python language. (We estimate that several times as many lines would have been needed in a lower-level language such as C or Java.) We found only five defects in further testing; this means that the defect density that we obtained after inspection and first test during integration is comparable to the defect density normally obtained after verification in Cleanroom.

Dr. Dunston has begun to use the program experimentally, and intends to put it into full production use for his next musical comedy. By that time the remaining increments will be constructed and installed. Meanwhile, we have begun to use the program in our own work, to help schedule the activities of the staff of an introductory computer science course (lecturers, teaching assistants, tutors and graders) around all of their other obligations. The program has been quite helpful with this. As of the time of writing, no further defects have been found in the program.

5   Conclusions

We consider that the integration of Z and Cleanroom, as described above, was successful. We believe that the use of specification via pictures and of "lightweight" literate programming contributed to the success of the project as well. Results obtained from one project of this size are not conclusive, of course, but all indications are positive thus far.

We definitely intend to use similar combinations of technologies in future projects, and are eager to try them on substantially larger projects. Since Z and Cleanroom have been used separately on projects of substantial size with considerable success, we see no reason why the same should not be true when they are used together, but only actual experience will tell us with certainty.

Beyond this, we believe that our results confirm and support several ideas already noted by other writers and researchers regarding the way to use formal methods most effectively. First, formal methods are not monolithic: it is quite possible to use some parts or aspects of a method without using all of the method. For example, it makes perfect sense to write specifications in Z even if one has no intention of using the accompanying methods for formal refinement, and doing this seems to be rather common among Z users.

Similarly, it is perfectly reasonable to use more than one formal method or notation in a project, according to which is most suitable for each part of the project. A notable example of this was the development project for the CDIS air traffic control display system [2], which successfully used a variety of formal notations: VDM, VVSL, CSP and CCS, as well as data-flow diagrams and

finite-state machines.

Finally, full formality is not only not necessary to obtain the benefits of formal methods, but is frequently not even productive or cost-effective. In the postmortem to the Hursley experiment [12, p. 293], Mark Pleszkoch of the IBM Cleanroom Software Technology Center is quoted as saying:

> I believe that the key to applying Cleanroom in a cost-effective, highly productive manner is to not force developers to go to a level of formality beyond their needs (and abilities), while at the same time not losing the benefits of precise documentation that makes clear what each piece of code is designed to do.

A number of other writers have been expressing similar opinions in recent years (see, e.g., [14] and [2, pp. 74–75]). The general principle is that there is an appropriate level of formality for every situation, and more rigor is not always better. If this is not yet the consensus of the formal methods community, perhaps it eventually will be.

References

[1] Frederick P. Brooks, Jr. "No silver bullet: Essence and accidents of software engineering." *Computer 20,* 4, pp. 10–19, April 1987.

[2] Anthony Hall. "Using formal methods to develop an ATC information system." *IEEE Software 13,* 2 (March 1996), pp. 66–76.

[3] P. A. Hausler, R. C. Linger and C. J. Trammell. "Adopting Cleanroom software engineering with a phased approach." *IBM Systems Journal 33,* 1 (1994), pp. 89–109.

[4] Ian Hayes, ed. *Specification Case Studies.* London: Prentice Hall International (UK) Ltd, 1987.

[5] Iain Houston and Steve King. "CICS project report: Experiences and results from the use of Z in IBM." In *VDM '91: Formal Software Development Methods,* pp. 588–596. Berlin: Springer-Verlag, 1991.

[6] Jonathan Jacky. *The Way of Z.* Cambridge, England: Cambridge University Press, 1997.

[7] Donald E. Knuth. "Literate programming." *The Computer Journal 27,* 2 (May 1984), pp. 97–111.

[8] Richard C. Linger. "Cleanroom process model." *IEEE Software 11,* 2 (March 1994), pp. 50–58.

[9] Harlan D. Mills. "The new math of computer programming." *Commun. ACM 18,* 1 (January 1975), pp. 43–48.

[10] Harlan D. Mills, Michael Dyer and Richard C. Linger. "Cleanroom software engineering." *IEEE Software 4*, 5 (September 1987), pp. 19–24.

[11] Seyed-Hassan Mirian-Hosseinabadi and Raymond Turner. "Constructive Z." *J. Logic Computat. 7,* 96-48 (1997), pp. 49–70.

[12] Glyn Normington. "Cleanroom and Z." In *Z User Workshop, London 1992.* London: Springer-Verlag, 1992.

[13] Ben Potter, Jane Sinclair and David Till. *An Introduction to Formal Specification and Z* (second edition). Hemel Hempstead, England: Prentice Hall Europe, 1996.

[14] Hossein Saiedian, ed. "An invitation to formal methods." *Computer 29,* 4 (April 1996), pp. 16–30. See particularly the contributions of Jones, Jackson and Wing, and Lutz.

[15] J. M. Spivey. *The Z Notation: A Reference Manual* (second edition). Hemel Hempstead, England: Prentice Hall International (UK) Limited, 1992.

[16] Allan M. Stavely. *Toward Zero-Defect Programming.* Reading, Mass.: Addison Wesley Longman, 1999.

[17] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof.* Hemel Hempstead, England: Prentice Hall Europe, 1996.

[18] J. B. Wordworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Wokingham, England: Addison Wesley, 1992.

[19] J. B. Wordworth. *Software Development with B: An Introduction.* Harlow, England: Addison Wesley Longman Limited, 1996.

Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks

Rossana M. C. Andrade[1]

TSERG, School of Information Technology and Engineering, University of Ottawa
150 Louis Pasteur, MCD 409, Ottawa, Ontario, K1N 6N5, Canada
E-mail: randrade@site.uottawa.ca

Abstract. Over the last few years, several alternatives for adding mobility to Asynchronous Transfer Mode (ATM) signaling protocols have been presented in the literature. However, most of the current approaches for wireless mobile ATM (WmATM) network development include basically text and information flows. As a result of the complexity involved in handling mobility, communication and handoff procedures for WmATM networks, current approaches can lead to ambiguities, gaps, inconsistencies and undesirable interactions at the later stages of the development process where changes can be costly and provoke backward incompatibility. With these problems in mind, this work proposes a development approach that includes a technique called Use Case Maps (UCMs), and the following formal methods: Language of Temporal Ordering Specifications (LoTOS) and Message Sequence Charts (MSCs). UCMs are applied at the *requirements capture* and *analysis* stages, followed by LoTOS and MSCs at the *design* stage. Besides providing a better and precise description of the system at the early stages, our main goal is to combine these techniques and help to solve design problems like the ones mentioned above. As a case study, WmATM network procedures are specified using the proposed approach.

Keywords. Causal Scenarios, Use Case Maps, Formal Techniques, Language of Temporal Ordering Specifications, Message Sequence Charts, Wireless Mobile ATM Networks.

1. INTRODUCTION

Although **F**ormal **D**escription **T**echniques (FDTs) have been successfully used to specify and validate protocols in different application domains achieving clear and concise specifications, most current development approaches for **W**ireless **m**obile **ATM** (WmATM) networks include basically text and information flows at the early stages. As a result of the complexity involved in handling mobility, communication and handoff procedures, these approaches can lead to ambiguities, gaps, inconsistencies and undesirable interactions at the later stages.

FDTs, such as LoTOS [21], the **L**anguage **o**f **T**emporal **O**rdering **S**pecifications, and **M**essage **S**equence **C**harts (MSCs) [18], have not only shown resiliency in the usability, but also tool support and training have improved in the last 15 years [4][12][13]. Even though LoTOS and MSCs can be used at different levels of abstraction, it requires precision on the description of action sequences and exchanged messages. Thus, these formal techniques are more suitable to be applied at intermediate stages of the development process. In contrast, visual techniques such as **U**se **C**ase **M**aps (UCMs) [8][9] give to the designer capability to work with whatever amount of detail is available being appropriate for the early stages.

In this context, we propose the application of UCMs, LoTOS, and MSCs at different stages of the system development process. UCMs are applied at the *requirements capture* and *analysis* stages, followed by LoTOS and MSCs at the *design* stage. The proposed approach is applied to the development of a prototype for WmATM networks. Mobility, communication and handoff procedures are firstly described with UCMs and, in conformance to that, formally specified and validated with LoTOS. MSC scenarios are automatically generated from the LoTOS specification in order to represent the results of the validation and facilitate the implementation of protocols.

This paper is divided into 7 sections. An overview of the WmATM networks is given in Section 2. Section 3 illustrates the proposed development approach. A big picture of the relationship among mobility, communication and handoff procedures for WmATM networks are described with UCMs in Section 4. After that, the corresponding LoTOS specification is presented in Section 5. Section 6 shows

---

the generated MCSs scenarios and, last, Section 7 discusses our main contributions. Related works are also mentioned in Sections 4, 5 and 6.

2.   CASE STUDY: WIRELESS MOBILE ATM NETWORKS

Asynchronous Transfer Mode (ATM) was developed in the 90s to support high-bandwidth multimedia applications and provide bandwidth on demand, traffic integration, cost effectiveness, as well as flexible data networking [23]. Nowadays, ATM is viewed as a strong candidate to extend these services to portable systems using wireless technologies [1][28][30]. Accordingly, several alternatives for adding mobility to ATM signaling protocols have been presented in the literature [5][6][7][10][24][31]. For example, [7] and [28] present WmATM networks as a wireless extension of ATM networks with mobility and any modification in the existing ATM signaling protocols. On the contrary, [24], [30] and [31] believe that minimum challenges should be done in the ATM networks to support mobility and achieve a global WmATM network environment. In [5] and [6], the authors present two different signaling protocols to support both alternatives.

2.1.   A Typical WmATM Network Environment

Figure 1 illustrates a possible environment that can support the concepts involved in designing a global WmATM network. The wireless service area is divided into cells and each cell is equipped with a *base station transceiver* (BST) that is responsible for the use of the allocated spectrum. A *base station* (BS) is responsible for a set of BSTs that are connected to the BS through *wireless access ports*. Several *mobile stations* (MSs) share the capacity of each BST. A *wireless ATM network backbone* is composed of *WmATM switches* attached through high-speed *transmission links*. *Databases* are responsible for keeping information about mobile users. The wireless backbone can communicate with the *ATM network backbone* using *wired access ports*.

   We choose a simplified wireless mobile ATM network as a case study, since it is representative of large and complex systems and touches upon common problems in the development process of these systems. The WmATM reference architecture considered in our work includes *mobile stations*, *WmATM switches* and *databases*. An *ATM network* composed of *ATM switches* and *fixed stations* is also described to allow the communication between fixed and mobile stations. Since we focus on signaling protocols for upper layers, *base stations* are not considered. Mobile stations communicate directly with WmATM switches and mobility occurs every time the mobile station changes a location area (represented by changing the WmATM switch).
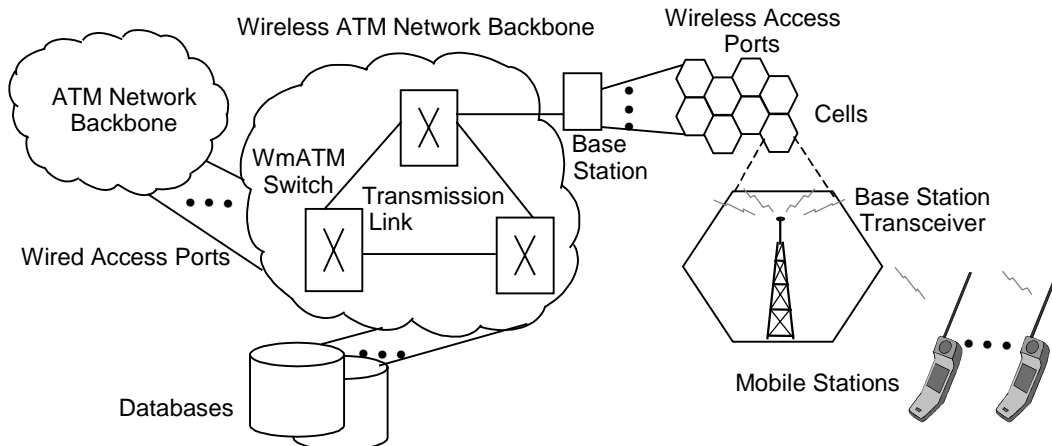

Figure 1 A Possible Wireless Mobile ATM Network Environment

(adapted from Figures 2, 3 and 4 of [24][7] and [1], respectively)

In the ATM fixed networks [17], there is no need for databases since each fixed station has a user's identification that determines where the user is and how to route a call to the user. Our work focuses only

on the specification and validation of connections between mobile users and between mobile and fixed users.

During the development of the WmATM network environment, each component of the reference architecture is specified with its corresponding protocols related to mobility, communication and handoff. Informally, *mobility management functions* provides a secure environment for mobile users, updates location information and perform the user de-registration in an old location area when a mobile user roams and registers in a new location area. *Communication management functions* are used to establish, release and maintain calls between two mobile users, from mobile to fixed users, and from fixed to mobile users at their request. Meanwhile, *handoff functions* give to the mobile user freedom of motion beyond a wireless coverage area by maintaining the quality of a link whenever a user moves from one location to another.

2.2. Current Development Approaches

Several signaling protocols alternatives for wireless mobile ATM networks have been presented in the literature and their development approaches involve *informal descriptions* as text at the early stages followed by *flow charts* [7][31], *state models* [10], or *information flows* [5][6][24] as shown in Figure 2.
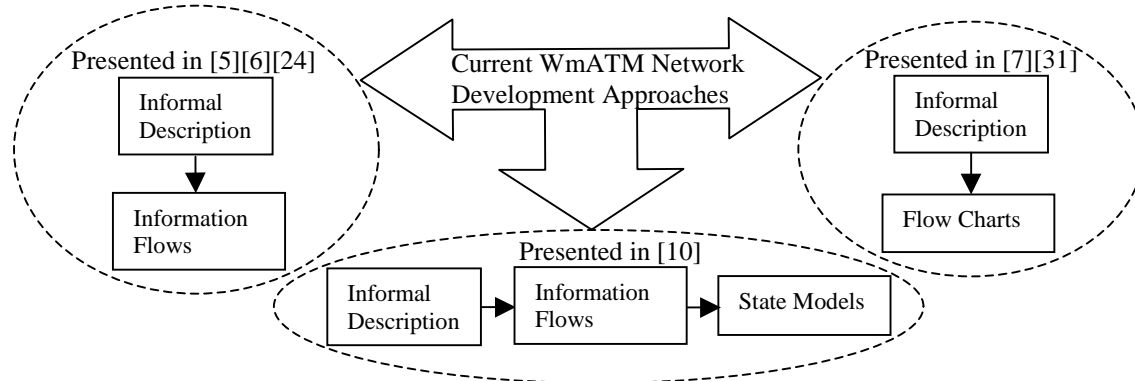
Figure 2 Different Development App roaches for WmATM Networks

When signaling protocol requirements are described only with text, they can lead to redundancies and become cumbersome to read, understand and manage at the later stages. An attempt to solve these problems is usually done following *informal description* by *information flows* (also known as sequence diagrams or message sequence charts). However, they are only necessary for detailed design, when design decisions about messages, parameters, data, and system components need to be taken. *State models* are also suitable for later stages, since they demand full precision during the definition of each state and underlying architecture. As a result, from the informality of the text to these formal models, a description gap can be identified that leads to protocol inconsistencies and undesirable interactions at the later stages. Even though *flow charts* are more adequate after informal descriptions to reduce this gap, they quickly become difficult to manage due to the increasingly complexity involved in the description of architecture and protocols of large systems such as WmATM networks. Besides this, *information flows* and *flow charts* produce disjoint scenarios that can not be validated. Thus, completeness and consistency can only be checked at the implementation stage.

3. THE PROPOSED DEVELOPMENT APPROACH

In order to overcome the problems mentioned earlier, this work proposes the combination of techniques such as UCMs, LoTOS and MSCs in the system development process. The proposed approach splits this process into a number of steps, called *stages*, each of which produces a more detailed view of the system. By decomposing a system into manageable units, we are applying a strategy used by object- and function-oriented software community when dealing with the complexity of large systems [11]. Besides this, we are adding rigor to the approach by using formal techniques. Figure 3 depicts the proposed development approach with *requirements capture*, *analysis* and *design* stages. Arrows show how these stages interact and represent the relation *dependency on*. Several development cycles represent the gradual and iterative

characteristics of the approach. Since implementation and testing are not considered for our work, we omit these stages in the figure.
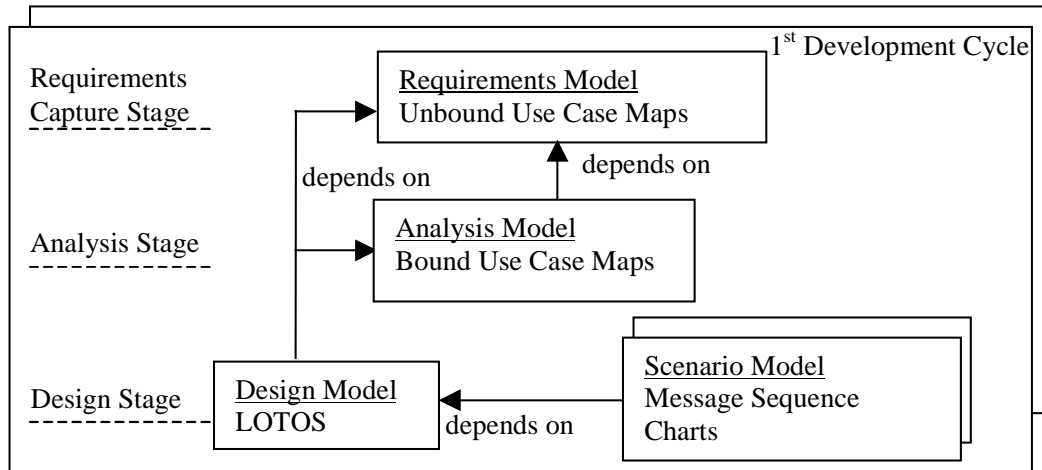

Figure 3 Proposed Development Approach

The *requirements capture* stage is the first input for the development of a system. The elicitation of meaningful requirements identifies and documents what the system is supposed to do and which are the main functions to be described. Use cases are used for most software community to describe the sequence of events of an actor (an external agent) using a system to complete a process [19]. To avoid ambiguities caused by narrative documents such as simple text or even textual use cases, these informal descriptions are replaced by a *requirements model* developed with UCMs. Since the notation is informal and intuitive, it is suitable for the early stages when the user needs are described in a high-level of abstraction and designers are discussing about, visualizing and explaining the overall behavior of a system. For example, at the beginning, when organizational structure details are not available, this visual technique describes high-level scenarios in terms of *causal relationships* between *responsibilities* (called *unbound UCMs*). The *stub* notation is used to hide functions that are detailed at the later stages.

Design decisions regarding which system component is responsible for a specific action, event or transaction are taken during the *analysis* stage. The functional behavior is further investigated and mapped to system components (part of the reference architecture). The *analysis model* is generated with *bound UCMs*. Detailed descriptions about what the system does are represented in terms of UCM notation: *plug-ins*, detailed *responsibilities*, detailed *pre-* and *post-conditions*.

Even though UCMs are supported by a drawing tool (the UCM Navigator) [20] and by a user group [29], due to its informality, validation and verification techniques are not possible. Two formal methods are included at the *design* stage, LoTOS [21] and MSCs [18], to describe how system components communicate or interact in order to fulfill the analysis model. Details regarding data types, parameters and exchange messages are introduced in a *design* model (behavior and reference architecture are described with LoTOS) and successful and unsuccessful outcomes are shown in several MSCs (*scenario* models).

LoTOS specifications represent a system prototype by describing temporal relations with externally observable behavior. Abstract data types are also included in this formal technique. Details about LoTOS, standardized by ISO, can be found in [21]. This FDT is supported by tools that offer ways of checking completeness and consistency. For instance, **LO**tos **LA**boratory (LOLA) is a set of tools developed by the Department of Telecommunication Engineering (ETSIT) of the University of Madrid [22] that includes: a step-by-step executor, a tool for obtaining the labeled transition system, and a tool for testing.

MSCs [18], standardized by ITU-T, describe interactions between system components. Each MSC represent exactly one scenario by focusing on the communication behavior of system components and their environment through message exchanges. We use MSCs to represent the results of the LoTOS validation and these scenarios are used as input to the implementation and testing stages. Recently, a

Lotos2MSC Converter is being developed by Bernard Stepien [25] in order to generate MSCs directly from LoTOS traces. The first version is already available and applied to our work.

As a case study, we iterative and gradually specify and validate a simplified WmATM network environment. The system behavior increases with designer and user needs. Each development cycle brings more details regarding new functional requirements as well as new system components. At the beginning, mobility management are described (development cycle 1), followed by communication and handoff functions (development cycle 2 and 3). Next sections present the description of the system functional behavior and reference architecture using our approach.

4. WmATM DESCRIPTION WITH USE CASE MAPS

This section presents the *requirements capture* and *analysis* stages of the proposed development approach depicted in Figure 3. These models are based on the description of WmATM signaling protocols presented in [5][6] as well as on our experience with wireless network standards [3][4]. By focusing on the functional requirements with the UCM notation, firstly, it is possible to describe the whole scenario of how the simplified WmATM network environment works. The system is decomposed in the following functions: mobility (authentication, registration and de-registration), communication (connection establishment and disconnection) and handoff functions. These functions are gradually described in terms of sequential actions with *unbound* UCMs (the requirements model) followed by more details about the system behavior and the addition of the reference architecture with *bound* UCMs (the analysis model). We present the first development cycle related to mobility management functions. Scenarios related to the other functions as well as exceptions (such as network failure, lack of network resources, database failure and so on) are left to the next development cycles.

4.1. Requirements Model: Unbound UCMs

The whole behavior of the system and, consequently, the relationship among the functions mentioned earlier are better understood by following the UCM flows shown in Figure 4. Based on the root map, users and designers can discuss about early decisions regarding the sequence in which these functions are performed. This map describes the system behavior that starts when a pre-condition is triggered, for example, the user powers on a mobile station (filled circle labeled S). A *stub* (such as MM, HP and CM in the figure) identifies places where details are delayed to a sub-UCM, called *plug-in*. The *stub* notation is applied to our work not only to hide details, but also to decompose the system into small manageable units. In this paper, we focus on the MM Stub to show the development approach step-by-step. Communication management and handoff functions are not described for space limitation.
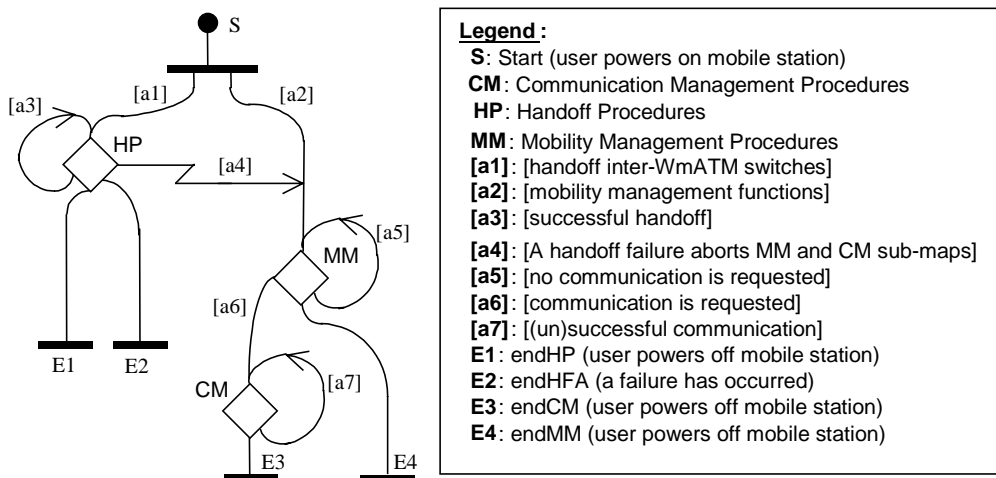

Figure 4 WmATM Network Root Map: *Unbound* UCMs

This scenario ends with one or two of the following post-conditions: user powers off mobile station (bars labeled E1, E3 and E4) or a handoff failure occurs (bar labeled E2). A *route* is a path that links an initial cause to a final effect. For example, <S, [a2], MM, E4> represents a *route* for registration followed by an user powers off event. The zig-zag notation ([a5] path) exists to describe exception paths, however, we propose its use also to describe synchronous interactions between stubs such as HP and CM (HP replaces CM in case of handoff failure). Direction arrows help designers to visualize the UCM flow as in the HP *stub* where an outgoing path returns to the same stub in order to trigger the *plug-in*. *And-forks* represent composite UCMs that split a path into parts (sub-paths) that proceed concurrently ([a1] and [a2] in the figure). *Or-joins* represent composite UCMs that can be concatenated in only one path (represented by [a3] joining [a1], [a5] joining [a2], and [a7] joining [a6]). There is no level of concurrency associated with *OR-joins*.

Figure 5 depicts the second level of the requirements model when mobility management procedures are decomposed into small units represented by Auth and Update *stubs* in the Location Registration *plug-in* bound to the MM *stub* in the root map. Alternative paths (called *OR-forks*) represent composite UCMs that can be split into two different paths (no level of concurrency is associated with them). For instance, a responsibility point (cross labeled cR in the figure) is activated along the [b2] path to decide whether the mobile station is registered or not at the current location area. The alternatives sub-paths (labeled [b3] and [b4]) are generated after this decision. Auth stub has two outgoing paths labeled [b1] and [b2] that correspond to end points of the authentication *plug-in* (respectively, unsuccessful and successful outcomes). The Update stub groups all the functions related to updating user information.
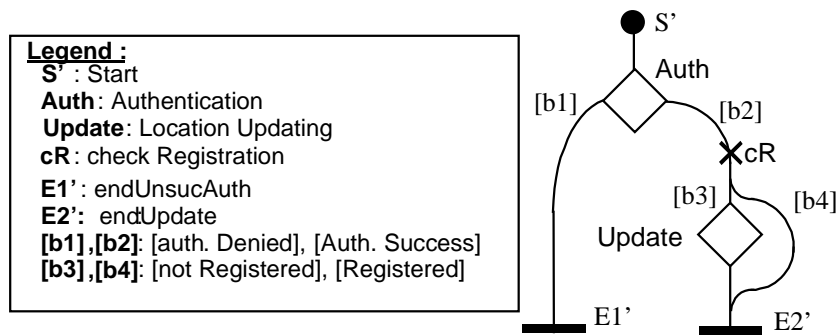

Figure 5 (a) Location Registration *Plug-in* for MM *Stub*

The main advantage of applying *unbound* UCMs when comparing with information descriptions is the visual representation of the overall system behavior since the early development stages. Under such circumstances, the system description becomes more readable and design decisions regarding to the mapping of the reference architecture, exchanged messages and data types are easier to handle at the later stages.

4.2. Analysis Model: Bound UCMs

At the *analysis* stage, the previous *stubs* are detailed with r*esponsibility* points along the paths that identify actions, events, or operations on data items. Figure 6(a) details the Auth *stub*. First, the mobile station processes the user authentication and sends the authentication result to the network (sI responsibility). Then, the aAA responsibility performs the same authentication operation at the network side. The cAR responsibility generates the successful or unsuccessful outcomes (respectively, E2'' or E1'' end points). In case of denied authentication, the mobile user is notified (nAD responsibility). Otherwise, the network is notified (nN responsibility).

Figure 6(b) shows what happens inside the Update stub. For example, cL generates different outcomes according to whether the mobile user is roaming or not. uP and uTP responsibilities are operations on database items. Sub-paths labeled [c1] and [c2] are concatenated after the network is notified (nN responsibility) about the successful operation.
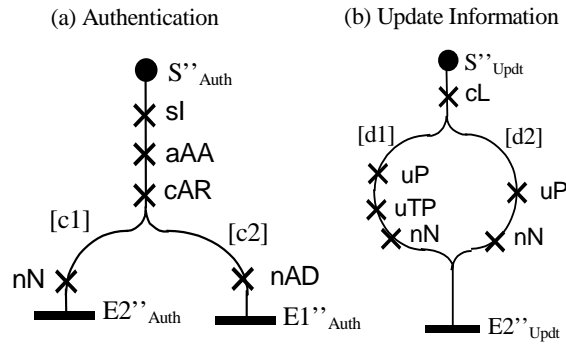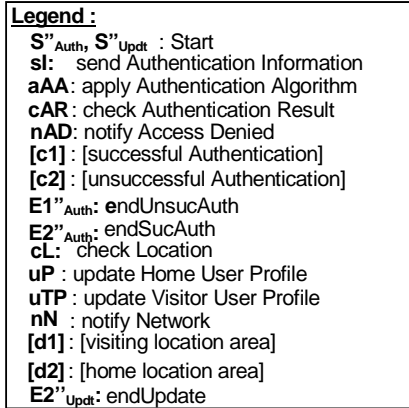
Figure 6 (a) Authentication and (b) Update Information *Plug-ins*

Besides describing detailed scenarios as causal paths with new plug-ins bound to the stubs at this stage, organizational structures of system *components* (represented by rectangular boxes as shown in Figure 7) are added. For instance, WmATM *components* involved in the authentication and update information functions, *Mobile Stations*, *WmATM switch* and *Home and Visitor Databases* are mapped to the *unbound UCMs* described at the requirements model.
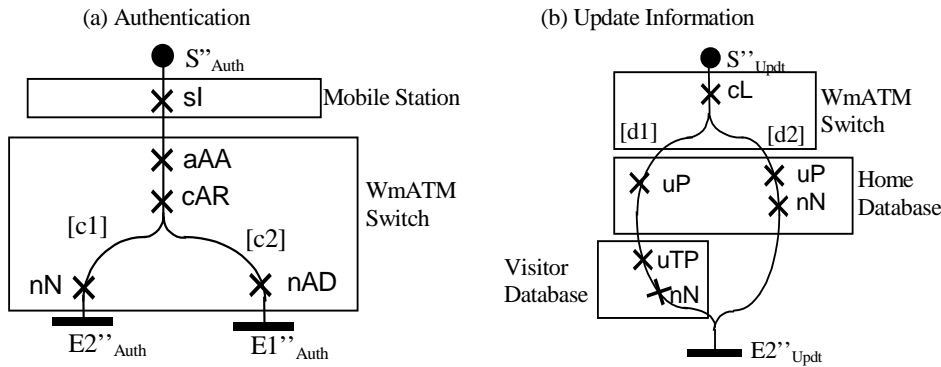
Figure 7 *Bound* UCMs: Authentication and Update Information *Plug-ins*

Related and Future Work. With the increasing popularity of the Unified Modeling Language (UML) for modeling systems using object-oriented concepts, UCMs are currently being investigated as another UML artifact to help the system development process. According to [2], UCMs can help to bridge the gap between the use case model and the analysis and design models represented by behavioral diagrams (sequence, state charts, and activity diagrams) in the UML. We intend to apply UCMs as an alternative for the early stages of the function-oriented development process combined with a strong formal method such as LoTOS and MSCs. In short, our approach brings more powerful tools to tackle the verification problem (how can a designer solve a given problem systematically so that requirements are realized) in large systems. Object-oriented analysis and design that are the subject of UML is one step further and it is considered as future work.

Besides the comparison with UML, as a graphical notation, UCMs resembles petri nets at a first sight, but many differences can be quickly perceived. For instance, use case maps are based on causality events while petri nets are based on states or events. Also, semantics are not defined for UCMs, they are often applied to the early stages of the development process to give a global picture of the system, they are light-weight, easy to learn, and the underlying architecture can be also expressed using this notation. On the other hand, petri nets have strict semantics, are rich in analysis methods, have automated tools that are rigorous and soundness. Besides this, the latter is more appropriate to the design stage and more

feasible to be compared to formal languages like LoTOS and Specification and Description Language (SDL). A mapping of UCMs to petri nets can be investigated as future work.

5. WMATM SPECIFICATION AND VALIDATION WITH LOTOS

At the design stage, a formal model is generated based on the *bound* UCMs described earlier. LoTOS has many advantages to specify and validate complex and large systems. For example, different levels of abstraction can be used to describe functional behavior at different development cycles, not to mention the LoTOS ability of process instantiation and parallel composition to specify the system reference architecture with the sequence of responsibilities defined previously at the requirements and analysis models. LoTOS tools like the LOLA environment are available to automatically support validation and verification methods. These methods allow the detection of design errors, inconsistencies and incompleteness at the time the LoTOS specification is being developed.

Since the behavior and structure models are iterative and incrementally generated at the *requirements* and *analysis* stages, the LoTOS specification becomes easier to develop. In addition, the gap between stages is also reduced by moving from *bound UCMs* (such as Figure 7) to LoTOS *processes* and *gates* shown in Figure 8. Even though the processes are derived from the UCM system components, design decisions related to how they communicate through gates are not always straightforward and depend on real interfaces and synchronization needs.
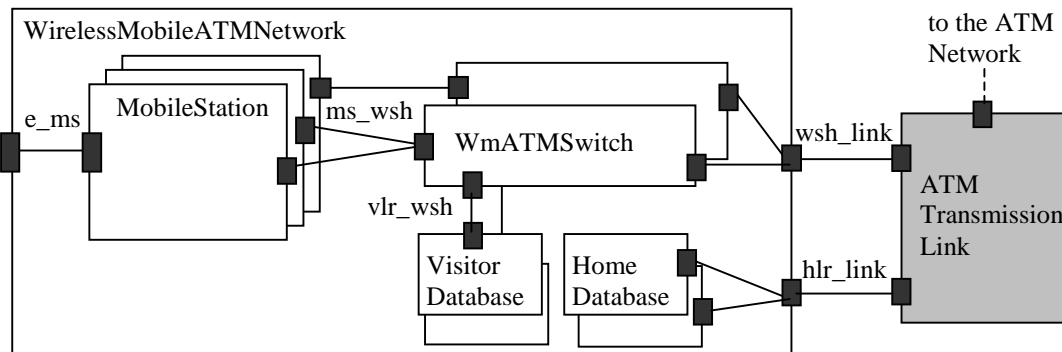

Figure 8 Graphical Representation of the LoTOS Specification Architecture

At the highest level of abstraction, the specification is composed of *Wireless mobile ATM Network*, *ATM Transmission Link* (depicted in gray in the figure to differentiate from the processes described also as UCM system components at the previous stages) and *ATM Network processes*. *Wireless mobile ATM Network* includes *mobile stations* (originating and terminating sides), *WmATM switch*es (same process for previous and current *WmATM switches*), *home databases* (referred to Home Location Register - *HLR* in the specification), and *visitor databases* (referred to Visitor Location Register - *VLR*) *sub-processes*. These processes are synchronized through the following gates: **ms_wsh**, **vlr_wsh**, **wsh_link**, and **hlr_link**. *ATM network process* contains *fixed stations* and *ATM switches* connected through gate **fs_sh** (not shown in the figure for lack of space). *ATM transmission link* process are added to this stage in order to overcome a LoTOS limitation and provide the communication among *WmATM switch processes* (through gate **wsh_link**) and among *ATM switches processes* (through gate **sh_link**). Gates **e_ms** and **e_fs** provide the interaction of mobile and fixed stations with the environment (for simulation purpose).

Figure 9 depicts how theses processes synchronize through the gates (||| represents the interleaving operator and |[gate list]| the selective parallel operator). The use of these LoTOS operators allows process synchronization and the ability to simulate and test the whole system behavior. Data types are designed to guarantee information exchange among processes. In particular, each *MobileStation* is identified by its identification number (*user_A, user_B,* and *User_C* in the figure), electronic serial number, random variable*, secret key* (these identifiers are represented by *info_A, info_B* and *info_C*), home database (hlr_1 and hlr_2) and current zone (zone_1 and zone_2). Each *WmATM switch* has its identification (*zone_1* and *zone_2* in the figure). *HLR* and *VLR* processes keep an identity and information about mobile stations in a set of database record (called *HLRRecSet* and *VLRRecSet*, respectively).

```
behavior
hide ms_to_wsh, vlr_to_wsh, hlr_to_link, wsh_to_link, fs_to_sh, sh_to_link  in
(( WirelessMobileATMNetwork [e_to_ms, ms_to_wsh, wsh_to_link, vlr_to_wsh,
hlr_to_link] ||| ATMNetwork [e_to_fs, fs_to_sh, sh_to_link]    )
      |[wsh_to_link, sh_to_link, hlr_to_link]|
     ATMTransmissionLink [wsh_to_link, sh_to_link, hlr_to_link] )
where
process WirelessMobileATMNetwork [e_to_ms, ms_to_wsh, wsh_to_link, vlr_to_wsh,
hlr_to_link]: exit :=
   ( (* users power on the mobile station *)
   ( MobileStation [e_to_ms, ms_to_wsh] (user_A, info_A, zone_1, hlr_1, 0)
 ||| MobileStation [e_to_ms, ms_to_wsh] (user_B, info_B, zone_1, hlr_1, 0)
 ||| MobileStation [e_to_ms, ms_to_wsh] (user_C, info_C, zone_2, hlr_2, 0)  )
 |[ms_to_wsh]|
   ((WmATMSwitch [ms_to_wsh, wsh_to_link, vlr_to_wsh] (zone_1)
 |[vlr_to_wsh]| VLR [vlr_to_wsh] (vlr_1, InitialVLRSet1))
 |||(WmATMSwitch [ms_to_wsh, wsh_to_link, vlr_to_wsh] (zone_2)
 |[vlr_to_wsh]| VLR [vlr_to_wsh] (vlr_2, InitialVLRSet2))    )
 |[hlr_to_link]|(HLR [hlr_to_link] (hlr_1, InitialHLRSet1)
 |||HLR [hlr_to_link] (hlr_2, InitialHLRSet2) )   ) …
```

Figure 9 Highest Level of Abstraction of the LoTOS Specification

The behavior of each process is first generated based on the sequence of UCM responsibilities (for instance, from Figure 4, Figure 5, and Figure 7 to Figure 10). After that, both informal descriptions and information flows presented in [5] and [6] are considered to add more details to the specification such as data types and specific messages. During this stage, duplicate behavior and incomplete scenarios related to the signaling protocols are detected and corrected using the simulation and testing Lola tools. For example, in our specification the same procedure is used for connection establishment between two mobile users as well as between mobile and fixed users minimizing duplicated behaviors. Also, more unsuccessful scenarios are described, such as power off, handoff failure and disconnection (represented by the [> disable operator). These scenarios happen at any time after the user powers on or the connection establishment. Figure 10 depicts part of the behavior of the *MobileStation* process when a mobile user powers on and authentication and update information plug-ins are triggered as shown in Figure 7.

```
process  MobileStation [e_to_ms, ms_to_wsh] (usrid: UserIDN, userInfo: InfoIDN,
myzoneid: ZoneIDN, hlrid:DatabaseIDN, n: Nat) :exit :=
( … e_to_ms !usrid ?czid:ZoneIDN;  (* change location area *)
    ( [h(myzoneid) ne h(czid)] ->  (* registration begins *)
     (  ms_to_wsh !usrid !czid !InitiateRegREQ;
        ms_to_wsh !usrid !czid ?M:Message [h(M) eq h(InitiateRegCONF)];
          (* authentication process takes place *) …
        ms_to_wsh !usrid !czid !hlrid !r !AuthUserResult;
        ms_to_wsh !usrid !czid ?M:Message;
        ( [h(M) eq h(AuthSuccess)] ->
          MobileStation [e_to_ms, ms_to_wsh] (usrid, userInfo, czid, hlrid, 0)
         [] [h(M) eq h(AuthDenied)] ->
          … MobileStation [e_to_ms, ms_to_wsh] (usrid,…,czid, hlrid, n)))) … )
      [> e_to_ms !usrid !myzoneid ?M:Message[h(M) eq h(PowerOff)]; stop
     >> MobileStation [e_to_ms, ms_to_wsh] (usrid, myzoneid, hlrid, 0)
endproc (* MobileStation *)
```

Figure 10 Partial Behavior of the *MobileStation* Process

LOLA is a transformational and state exploration tool that supports execution and testing of LoTOS specification. To do this, LOLA provides a set of tools that help designers to analyze the behavior of a system before the implementation stage. The following tools are applied to our specification: *simulation* or *debugging* that simulates the behavior step by step and evaluates data value expressions; and *testing* that calculates the response of a system specification to a test according to testing equivalence. The one

expansion transformation tool is also used to generate a file with a trace (one possible scenario). Next sub-section presents MSC scenarios that are automatically generated from LoTOS validation traces contained in these files.

6. SCENARIOS WITH MESSAGE SEQUENCE CHARTS

MSC is the favorite notation to describe scenarios of current systems and many basic sequence diagrams are used at the early phases of the development of large systems, standards and to represent early behavior model in object-oriented approaches. Nevertheless, these diagrams are static and disjoint, only one sequence of events can be observed at once. Due to these characteristics, validation and verification techniques are not possible and in [4], we propose their use as a complement of formal methods such as LoTOS and SDL [16]. Recently, High-Level MSCs include control structures that can combine several MSCs representing more than one scenario, however, they are not considered in our work. By adding these scenarios to the proposed approach, we aim to represent the results of the LOLA validation activities. Successful and unsuccessful MSC scenarios can be more readable and attractive than LoTOS traces and they can be used for implementers to generate the protocols.

The generation of MSCs is done automatically with the Lotos2MSC converter tool. This tool uses a configuration file that interprets the LoTOS traces and generates proper MSC scenarios. To make this possible, the converter uses conventions and additional configuration information to decode a LoTOS action and its elements (the sequence of values) to derive MSCs components, messages and parameters. The converter restricts LoTOS capability of full-duplex communication through gates (no direction is associated to the execution of LoTOS actions among processes) by demanding that gates represent directions and components. Since LoTOS generic concept of action defines messages and parameters implicitly in terms of abstract data types, as mentioned above, the tool can only recognize messages and parameters when they are described in the LoTOS action. A direct mapping of LoTOS to MSC concepts is not done due to LoTOS synchronization of many simultaneous actions in contrast to MSCs exchange of messages between components. This converter also allows filtering specific LoTOS actions that the designer wants to be displayed on the MSC graph using gate names as filtering criteria. More details about this converter can be found in [25].

Figure 11 illustrates two disjoint scenarios that represent specific behaviors of the system in conformance not only with the LoTOS specification (Figure 10) but also the *bound* UCMs (Figure 7(a)). For sake of clarity, we represent *WmATM switch process* as current WmATM switch.
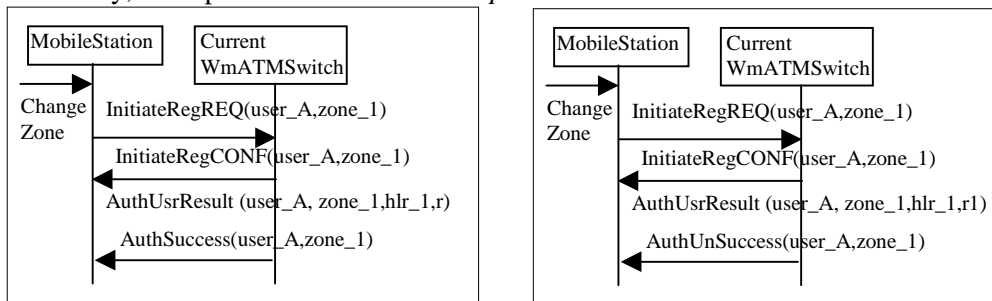

Figure 11 (a) Successful Authentication (b) Unsuccessful Authentication

By comparing these MSCs with some of the protocols presented in the literature, decisions such as the authentication result initially being done at the MS side and also network side is clearer and guarantee security through the air interface (more complete design). Also, details about parameters make the implementation easier.

Due the popularity of sequence diagrams, most tools for formal methods bring options in how to generate MSCs from the validation results. For instance, the **SD**L Development **T**ool set (SDT) and the SPIN tool support the process of going from the formal design to MSCs. Work on providing the requirements first in terms of sequence diagrams and then applying more formal verification techniques such as the ones supported by the SPIN model checker [15] to these diagrams is presented in [14]. This process is also a recent research interest for the SDL and LoTOS communities. We believe that these

solutions as well as our approach are valuable and lead to a more effective and attractive way to design a system and present the validation and verification results to users and developers.

7. CONCLUSION

Current development approaches for wireless mobile ATM (WmATM) networks describe all specific information related to the signaling protocols at once. However, a good approach should iterative and gradually add details during different development stages and life cycles, while checking for ambiguities, inconsistencies, and undesirable interactions. In this context, the main contribution of this work is to introduce the combination of different techniques at appropriate stages of the system development process. As a case study, mobility, communication and handoff procedures for WmATM networks are developed using the proposed approach.

In short, at the *requirements capture* stage, *unbound* Use Case Maps (UCMs) are used as first scenarios by focusing on the causality relationship between responsibilities, without any concern about components. At the *analysis* stages, system components and more behavior details are added to these maps, generating *bound* UCMs. This notation provides a better human understanding of the system and it helps network designers to produce descriptions of the requirements more legible as well as facilitates the system development and maintenance. At the *design* stage, a formal specification is developed with LoTOS adding rigor to the approach and many possible behaviors are described concurrently with details such as data types, parameters and specific events. A set of LoTOS tools assures the completeness of the system and verifies correctness and consistency properties. MSCs scenarios are automatically generated from the results of the LoTOS validation in order to facilitate future protocol implementation.

The proposed approach improves the existing current development process for wireless mobile ATM networks in different ways as follows: by achieving a better model, by helping human understanding and by reaching technical quality with the formal specification for future maintenance. Using our approach, inconsistencies of parameters and incompleteness of the informal description are detected and corrected. In addition, the UCM technique can reduce the gap between early and later stages. Our results also intend to show how the combination of informal and formal techniques at the appropriate development stages can really aid designers on generating good systems, ready to be reused and easy to maintain and add new features.

The motivation for choosing WmATM networks resides in their under development status and also the amount of information available about the signaling protocol alternatives. This makes feasible to produce the design prototype with the proposed approach. Our approach can also be applied to other wireless mobile communication systems. The Ottawa University LoTOS Group has successfully applied LoTOS to the specification and validation of mobile network standards, such as Global System for Mobile Communication (GSM) [27], and UCMs to the description of Wireless Intelligent Network standards [26] as presented in [3]. Currently, the combination of these techniques is being one of the main research topics of our group.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] Acampora, A., "Wireless ATM: A Perspective on Issues and Prospects", IEEE Personal Communications, Vol. 3, No. 4, pp. 8-17, August 1996.

[2] Amyot, Daniel, Use Case Maps and UML for Complex Software-Driven Systems, Technical Report, August 1999. www.usecasemaps.org

[3] Amyot, D., Andrade, R., "Description of Wireless Intelligent Networks with Use Case Maps", *Proc. Brazilian Symposium on Wireless Networks (SBRC 99)*, pp.418-433, Salvador (BA), Brazil, 25-28 May 1999.

[4] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., and Yi, Z. (1999) "Formal Methods for Mobility Standards". *IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems*, Dallas, USA, April 1999.

[5] Akyol, B. A., "Signaling Alternatives in a Wireless ATM Network", *IEEE Journal on Selected Areas in Communications*, Vol. 15, No. 1, pp. 35-49, January 1997.

[6] Akyol, B. A. and Cox, D. C., "Rerouting for Handoff in a Wireless ATM Network", *IEEE Personal Communications*, Vol. 3, No. 5, pp. 26-33, October 1996.

[7] Ayanoglu E. et al., "Wireless ATM: Limits, Challenges, and Proposals", *IEEE Personal Communications*, Vol. 3, No. 4, pp. 18-36, August 1996.

[8] Buhr, R.J.A. and Casselman, R.S., *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM_book95.pdf

[9] Buhr, R.J.A. (1998) "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998. http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

[10] Cheng, Fang-Chen, Holtzman, Jack M., "Wireless Intelligent ATM Network and Protocol Design for Future Personal communication Systems", *IEEE Journal on Selected Areas in Communications*, Vol. 15, No. 7, Sept. 1997.

[11] Corriveau J.,-P., Nel, D. N., "Introduction to Object-Oriented Software Engineering", Version 1.0, Course Notes, School of Computer Science, Carleton University, 1997.

[12] Courtiat, J.,-P., Dembinski, P., Holzmann, G., J., Logrippo, L., Rudin, H., Zave, P., "Formal Methods after 15 years: Status and trends", *Computer Networks and ISDN Systems 28*, pp. 1845-1855, 1996.

[13] Faci, M., Logrippo, L., Stepien, B., "Structural Models for Specifying Telephone Systems", *Computer Networks and ISDN Systems 29*, pp. 501-528, 1997.

[14] Holzmann, Gerard, J., Formal Methods for Early Fault Detection, Proc. Of Formal Techniques for Real-Time and Fault Tolerant Systems, LNCS Vol. 1135, pp. 40-54, 1996.

[15] Holzmann, Gerard, J., Design and Validation of Computer Protocols, Prentice Hall Software Series, 1991.

[16] ITU-T, *Recommendation Z. 100: Specification and Description Language (SDL).* Geneva, 1994.

[17] ITU-T, *Recommendation Q.2931*: ATM Network Signaling Specification, 1995.

[18] ITU-T, *Recommendation Z. 120: Message Sequence Chart (MSC).* Geneva, 1996.

[19] Jacobson, Ivar et. al, Object-Oriented Software Engineering (A Use Case Driven Approach), ACM Press, Addison-Wesley, 1992.

[20] Miga, A., *Application of Use Case Maps to System Design with Tool Support*, M.Eng. Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998.

[21] OSI - IS 8807, "Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", International Standard IS 8807 (E. Brinksma, ed.), 1989.

[22] Pávon, S., Larrabeiti, D., and Rabay, G., "LOLA – User Manual", version 3.6. DIT, Universidad Politécnica de Madrid, Spain, LOLA/N5/V10, February 1995.

[23] Prycker, Martin de, Asynchronous Transfer Mode: solution for broadband ISDN, Prentice Hall International (UK) Limited, 1995.

[24] Raychaudhuri, D., "Wireless ATM Networks: Architecture, System Design and Prototyping", *IEEE Personal Communications*, Vol. 3, No. 4, pp. 42-49, August 1996.

[25] Stepien, Bernard, Lotos2MSC Converter – User's Manual, University of Ottawa LoTOS Group, January 2000.

[26] TIA/EIA (1998) *Wireless Intelligent Networks (WIN).* Additions and modifications to ANSI-41 (Phase 1). TR-45.2.2.4, PN-3661 Ballot Version, May 1998.

[27] Tuok, R., *Modelling and Derivation of Scenarios for a Mobile Telephony System in LOTOS*, M.Sc. Thesis, Computer Science Program, SITE, University of Ottawa, Canada, 1996.

[28] Umehira, M., et al., "ATM Wireless Access for Mobile Multimedia: Concept and Architecture", *IEEE Personal Communications*, Vol. 3, No. 5, pp. 39-48, October 1996.

[29] *Use Case Maps Web Page*: http://www.UseCaseMaps.org , since 1999.

[30] Varshney, U., "Supporting Mobility with Wireless ATM", *Computer*, Vol. 30, No. 1, pp. 131-137, Jan. 1997.

[31] Veeraraghavan, M. et al., "Mobility and Connection Management in a Wireless ATM LAN", *IEEE Journal on Selected Areas in Communications*, Vol. 15, No. 1, pp.50-68, January 1997.

Formal Analysis of the Remote Agent
Before and After Flight

Klaus Havelund[1], Mike Lowry, SeungJoon Park[2],
Charles Pecheur[2], John Penix, Willem Visser[2], Jon L. White[3]

The Automated Software Engineering Group
NASA Ames Research Center,
Moffett Field, California, USA.

[1] Recom Technologies, [2] RIACS, [3] Caelum

Abstract

This paper describes two separate efforts that used the
SPIN model checker to verify deep space autonomy flight
software. The first effort occurred at the beginning of a
spiral development process and found five concurrency
errors early in the design cycle that the developers ac-
knowledge would not have been found through testing.
This effort required a substantial manual modeling effort
involving both abstraction and translation from the pro-
totype LISP code to the PROMELA language used by
SPIN. This experience and others led to research to ad-
dress the gap between formal method tools and the de-
velopment cycle used by software developers. The Java
PathFinder tool which directly translates from Java to
PROMELA was developed as part of this research, as well
as automatic abstraction tools. In 1999 the flight software
flew on a space mission, and a deadlock occurred in a
sibling subsystem to the one which was the focus of the
first verification effort. A second quick-response "clean-
room" verification effort found the concurrency error in a
short amount of time. The error was isomorphic to one
of the concurrency errors found during the first verifica-
tion effort. The paper demonstrates that formal methods
tools can find concurrency errors that indeed lead to loss
of spacecraft functions, even for the complex software
required for autonomy. Second, it describes progress in
automatic translation and abstraction that eventually will
enable formal methods tools to be inserted directly into
the aerospace software development cycle.

1 Introduction

Complex concurrent software is difficult to debug and
even more difficult to test with adequate coverage. With
the increasing power of flight-qualified microprocessors,
NASA space enterprises are experimenting with a new
generation of non-deterministic flight software that pro-
vides enhanced mission capabilities. A prime example is
the Remote Agent (RA) autonomous spacecraft controller
developed at NASA. In May 1999, the RA was success-
fully demonstrated in flight on Deep Space 1 (DS-1), the
first flight of NASA's experimental New Millennium pro-
gram. The RA is a complex, concurrent software system
employing several automated reasoning engines using ar-
tificial intelligence technology. The verification of such
complex software is critical to its acceptance by science
mission managers.

This paper describes formal methods verification ef-
forts for one of the three subsystems of the RA – specifi-
cally, the RA Executive, which provides operating-system
level capabilities for goal-directed software. Two differ-
ent verification activities were conducted, before and af-
ter flight, using different technologies and in very differ-
ent contexts. As such, this paper provides two succes-
sive snapshots of progress towards making formal meth-
ods verification cost-effective.

In 1997, while the RA was still in the development
stage, we modeled and verified a subset of the core ser-
vices of the RA Executive using the SPIN [10] model
checker. That verification unveiled several concurrency

bugs that were acknowledged by RA Executive developers [7].

As a result of this effort, it was decided to develop model checking technology for a main stream programming language in order to reduce the amount of time spent on modeling the behavior of programs in SPIN. The result was a translator, called Java PathFinder, from Java to the modeling language PROMELA of SPIN. In addition, a tool was developed for abstracting Java programs to reduce their state space, making model checking tractable.

Then, during the actual RA experiment in 1999, a deadlock occurred within less than 24 hours of operation. Although the problem was promptly identified and circumvented by the DS-1 team, we took the challenge of trying to diagnose the error in a fast-response "clean room" experiment [1]. After isolating a suspicious part of the program by visual inspection, we modeled it in Java, and then used Java PathFinder to exhibit a concurrency error that indeed turned out to be the one that had occurred in flight.

One key observation of these two successive experiments is that the error that caused the deadlock is exactly isomorphic to one of those found using SPIN two years before in another part of the code. It is a concurrency error, whose activation depends on a priori unlikely scheduling conditions between concurrent tasks. In fact, this error did not appear in over 300 hours of system-level testing on JPL's flight system testbed. The conditions under which it occurred in flight were not anticipated during testing. A principal benefit of model checking technologies is to be able to exhaustively cover scheduling alternatives. This paper gives a compelling illustration of how model checking found an error that was a priori unlikely but did actually occur. It also discusses gaps between previous formal method tools and requirements for making them easily accessible to system developers for 'in the loop' verification. Technological advances towards narrowing this gap are described in the context of the RA verification.

Section 2 describes the RA experiment. Section 3 describes the verification effort before flight, while Section 4 describes the verification effort after flight. The section also presents Java PathFinder. Section 5 describes the Java abstraction tool, and finally, Section 6 contains a conclusion.

---

[1] By "clean room" we are referring to the fact that, while the verification was post-facto, the team had no interaction with the actual debugging team.

2 The Remote Agent Experiment

To prepare for space exploration programs of the next decades within a reduced budget, NASA has set up the New Millennium program: a series of technology validation flights whose objective is to accelerate the qualification for flight of new spacecraft technology. One of the objectives of the New Millennium program is to increase spacecraft autonomy, moving from the low-level control sequences currently in use towards mission-level planning and autonomous health monitoring and recovery.

Deep Space 1 (DS-1), the first New Millennium Mission, was launched from Cape Canaveral on October 24, 1998 and ended its primary mission in September 1999 (it is still operating and is on its way for a comet encounter in 2001). During that mission, it successfully tested 12 cutting-edge technologies such as ion propulsion, on-board optical navigation, and the AI-based Remote Agent, marking the first operational use of artificial intelligence during space flight.

In its initial design, the RA Experiment (RAX) on DS-1 consisted of a short, limited 12-hour scenario designed to gain confidence in the RA, followed by a complete 6-day scenario that was the full RA test. Later, the experiment had to be compressed into a single 2-day scenario, to accommodate external mission constraints. The original scenarios were designed to cover a formal list of validation objectives. To protect the main DS-1 mission from possible misbehaviors of RA, the design included a "safety net" that allowed the RA experiment to be completely disabled with a single command, issued either from the ground or by on-board fault protection.

The RA went through a thorough qualification process before being allowed to run on DS-1. Though some formal verification tasks, such as the one reported here, were performed as feasibility studies, the formal qualification process relied on more conventional testing approaches. However, since the RA was a flight experiment, and not flight software, it was not subjected to the testing standards of the latter.

This section is a short summary of the flight qualification and experience of the RA [2, 13].

2.1 Remote Agent

The RA is an autonomous spacecraft controller developed by NASA Ames conjointly with the Jet Propulsion Laboratory (JPL) [12]. It comprises three components:

- The Planner and Scheduler (PS) [11] generates flexible plans, specifying the basic activities that must take place. Given a mission goal, it produces sequences of tasks for achieving this goal using available system resources.

- The Smart Executive (EXEC) [14] receives the plan from the Planner/Scheduler and then commands spacecraft systems to take the necessary actions to achieve and maintain the specified spacecraft states.

- The Mode Identification and Recovery component (MIR), called Livingstone [16], monitors the state of the spacecraft, detects and diagnoses failures and suggests recovery actions to the Executive.

The Executive subsystem is the focal point of the verification work discussed in this article. It combines features of multi-threaded operating systems with aspects of AI languages based on sub-goaling, such as Prolog. It is conceptually composed of three layers: a set of *core services* that implement a robust operating system for executing concurrent tasks, a set of *engine modules* including a plan runner, and a set of mission-specific *task programs*. The Executive schedules the execution of concurrent tasks. It also monitors a set of *properties* associated with system resources, and takes recovery actions on property violations. The Executive is written in a multi-threaded LISP, using a set of LISP macros called the *Executive Sequencing Language* (ESL) developed at JPL.

2.2   Testing the Remote Agent

Because autonomous systems such as the RA need to respond robustly in a wide range of situations, verifying that they respond correctly in all situations would require a huge number of test cases. Furthermore, these tests ideally have to be run on high-fidelity testbeds that are highly oversubscribed, hard to configure, and, running at real time speeds, take hours or days for a single run.

To address these problems, the RAX team followed a "baseline testing" approach, starting from nominal scenarios and testing a number of nominal and off-nominal variations around these scenarios. A wide range of variations were run on more available and faster low-fidelity testbeds, leading to the identification and resolution of 100-200 bugs during 18 months. An automated testing tool was designed for this purpose. Some of the

most likely off-nominal variants were run on medium-fidelity testbeds, while only nominal scenarios and certain performance and timing related tests were performed on high-fidelity testbeds. The final stage was a pair of "dress rehearsal" *operational readiness tests* (ORTs), involving actual communication with the mission control center. The bulk of the problems identified during testing were found with the low-fidelity testbeds. The ORTs only identified minor shortcomings that were resolved prior to flight.

2.3   Remote Agent in Flight

On Monday, May 17th, 1999, 11:04 am PDT, a telemetry packet confirmed that the RA had taken control of DS-1. The scenario went on smoothly, achieving 70% of the objectives, until Tuesday 7:00 am, when it became apparent that a command had not been executed as expected by the RA. The RA Executive was blocked, although the rest of the RA and the spacecraft were otherwise healthy. The Executive's low-level commands were used to gather a maximum of information, and then the experiment was interrupted.

By late Tuesday afternoon, the RAX team had found the source of the problem in the Executive code. They designed a 6-hour scenario that was run on Friday morning and went successfully through the remaining 30% of the objectives. A patch was also generated, but the DS-1 mission decided not to uplink it, considering the insufficient testing of the patch and the very low probability of the problem recurring.

The blocking was due to a missing critical section that had lead to a race condition between two concurrent threads. Under some very precise and unlikely timing circumstances, both threads could end up in a deadlock condition in which each one was waiting for an event that only the other one could provide, which is exactly what happened in flight.

3   Formal Analysis Before Flight

In April-May 1997 we analyzed part of the RA Executive using the SPIN model checker [7]. This effort lead to the discovery of five errors in the LISP code which are described below. As discussed in Section 4.3, one of these errors is isomorphic to the error that actually occurred during flight, causing a deadlock. First we give a short description of SPIN and its modeling language PROMELA.

Then we explain how a PROMELA model was extracted from the LISP code, and how properties were stated and verified in the model, leading to the discovery of the five errors. We conclude with a discussion of the methodology that has been followed.

3.1 The SPIN Model Checker

SPIN [10] is a tool for analyzing the correctness of finite state concurrent systems with respect to formally stated properties. A concurrent system is modeled in the PROMELA modeling language, and properties to be verified are formalized as assertions in the program or as formulae in the temporal logic LTL (*Linear Temporal Logic*). SPIN provides a model checker, which automatically examines all program behaviors in order to decide whether the PROMELA program satisfies the stated properties. In case a property is not satisfied, an error trace is generated, which illustrates the sequence of executed statements from the initial state to the state that violates the property. These error traces can then be executed in a simulator. The set of states reachable from the initial state must be finite in case a property needs to be proven correct for the whole state space.

A PROMELA program consists of a set of sequential processes that communicate via message passing through bounded buffered channels and via shared variables. Processes can be created dynamically. The behavior of an individual process is described using the statement language which provides many standard constructs such as variable assignments, channel communications, loops, conditionals, and sequential composition. Variables are typed, where a type can either be primitive, such as integer, or composite in the form of arrays and records. PROMELA provides inline procedures, which is a limited notion of procedural abstraction that is implemented via macro expansion.

Each process represents a finite automaton, and the global behavior of the system is then obtained by computing on-the-fly an *asynchronous* interleaving product of all these automata, creating the global state space. To perform model checking, SPIN translates (the negation of) any LTL formula into a Büchi automaton, and computes the *synchronous* product of this and the global state space. The result is again a Büchi automaton. If the language of this automaton is empty it means that the formula is satisfied. SPIN searches the state space depth-first, creating the states on-the-fly. A partial-order reduction technique

is used to prune the set of transitions to be explored.

3.2 Creating a PROMELA Model

The modeling activity focused on the core services of the plan execution module. The RA Executive core is designed to support execution of software-controlled *tasks* on board the spacecraft. A task often requires specific *properties* to hold during its execution. When a task is started, it first tries to *achieve* the properties on which it depends, after which it starts performing its main function. Several tasks may try to achieve *conflicting* properties; for example, one task may try to turn on a camera while another task tries to turn it off. To prevent such conflicts, a task has to *lock* in a lock table any property it wants to achieve. Once, a property is locked, it can be achieved by the task locking the property.

Properties may, however, be unexpectedly *broken* while tasks depending on them are executing. A property is defined as broken when it is locked in the lock table by some task, has been achieved (an extra boolean field in the lock table), but for some reason fails to hold on board the spacecraft. For the purpose of detecting which properties hold on board, a database is maintained of all properties being true at any time. Hence, an inconsistency can be detected by relating the lock table with the database. Tasks depending on a broken property must be interrupted and informed about the anomaly. For this purpose, a daemon monitors the changes on board the spacecraft, and in particular the consistency between the lock table and the database. The daemon is normally asleep, but is awakened whenever there is a change in the lock table or the database, where upon it checks their consistency.

The PROMELA model focuses on operations on the lock table. Hence, it is an abstraction of the LISP program, omitting details irrelevant for the lock table operations. The LISP program is approximately 3000 lines of code while the PROMELA model is 500 lines of code. Furthermore, the model only deals with a limited number of tasks and properties in order to limit the search space the SPIN model checker has to explore. Most abstractions were made in an informal manner without any formal proofs showing that bugs are maintained. Hence, in the abstraction phase we may have left out errors in the LISP code. However, all the errors we found in the model were also errors in the LISP code.

To give an idea of the modeling, we show how the daemon was translated, since it was the daemon that con-

```
(defun daemon ()
  (loop
    (if (check-locks)
        (do-automatic-recovery))
    (unless
        (changed?
          (+ (event-count *database-event*)
             (event-count *lock-event*)))
        (wait-for-events
          (list *database-event*
                *lock-event*)))))
```

Figure 1: Daemon in LISP

tained the error pattern which also occurred during flight, and which was found using the model checker. The actual LISP code describing the behavior of the daemon is given in Figure 1.

The daemon goes through a loop, where in each iteration it checks the lock table, comparing it to the database, and recovers any inconsistencies that may be detected (if the `check-locks` function returns `true`). After that, it goes to sleep by calling the `wait-for-events` function, which as parameters takes a list of events to wait for. Whenever one of these events is signaled, i.e. the database or the lock table is modified, the daemon will wake up and continue.

In order to catch events that occur while the daemon is executing, each event has an associated event counter that is increased whenever the event is signaled. The daemon only calls `wait-for-events` in case these counters have not changed, hence, there have been no new events since it was last restarted from a call of `wait-for-events`.

The PROMELA model of this LISP code is presented in Figure 2. The if-construct decides whether the daemon should stop and wait for a new database event or lock event to occur (call of `wait_for_events`), or whether it should continue for another iteration. Another iteration is needed if a database event or a lock event has occurred since the daemon was restarted last time; that is, in case the event counter `event_count` differs from the sum of the event counters for the database and lock events. If there is a difference, it means that there has been an event since the last time `event_count` was updated, and the daemon must perform another iteration before calling `wait_for_events`, first updating `event_count` to hold the new event counter sum.

```
proctype daemon(TaskId this) {
  byte event_count = 0;
  do
  :: check_locks_and_recover;
     if
     :: (Ev[DATABASE_EVENT].count +
         Ev[LOCK_EVENT].count
         == event_count )
          ->
         wait_for_events(this,
           DATABASE_EVENT,LOCK_EVENT)
     :: else ->
         event_count =
           Ev[DATABASE_EVENT].count +
           Ev[LOCK_EVENT].count
     fi
  od
};
```

Figure 2: Daemon in PROMELA

3.3 Stating and Verifying Properties

The model was analyzed with respect to the following two properties, here expressed informally. The *release* property reads: *"A task releases all of its locks before it terminates"*. The *abort* property reads: *"If an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon in terms of an abort"*. The release property was formulated by inserting an assertion in the code at the end of each task. This assertion stated that all locks should be released at this point. The second property was stated as a linear temporal logic property of the form:

```
[](property_broken -> <>tasks_informed)
```

This property says: whenever a property is broken, then eventually all tasks depending on this property will be informed about it (in fact terminated). The names `property_broken` and `tasks_informed` are macro names standing for predicates on the state space.

The attempted verification of the two properties led to the direct discovery of five programming errors – one breaking the release property, three breaking the abort property, and one being a non-serious efficiency problem where code was executed twice instead of once. The first four of these errors are classical concurrency errors in the sense that they arise due to processes interleaving in unexpected ways.

The error we want to focus on in this presentation is the one isomorphic to the RAX anomaly. The error caused the abort property to be violated. The error trace generated by SPIN demonstrated the following situation. The daemon is prompted to perform a check of the lock table. It finds everything consistent and checks the event counters to see whether there have been any new events while it has been running. This is not the case, and the daemon therefore decides to call `wait-for-events`. However, at this point an inconsistency is introduced, and a signal is sent by the environment, causing the event counter for the database event to be increased. This is not detected by the daemon since it has already made the decision to wait, which it then does, and the inconsistency now is not discovered by the daemon. Our suggested solution at the time was to enclose the test and the wait within a critical section, which does not allow scheduling interrupts to occur between the test and the wait. Furthermore, two other flawed code fragments violated the abort property.

The release property was violated in the sense that locks did not always get released by a task. The error trace generated by SPIN demonstrated that *during* a task's release of a lock, but before its actual release, the task may get interrupted by the daemon if the property gets broken. This means that the task terminates without releasing the lock. The error is particularly nasty in the sense that all code, *except* the lock releasing itself, had been protected against this situation: in case of an interrupt the lock releasing would be executed.

The model was verified exhaustively using SPIN's partial order reduction algorithm and state compression. Typically between $3,000$ - $200,000$ states were explored in the different models, using between 2 - 7 Mb of memory, and using between 0.5 - 20 seconds.

3.4 Discussion of Methodology

The verification effort has been regarded by all involved parties as a very successful application of model checking, and of SPIN in particular. According to the RA programming team, the effort has had a major impact, locating errors that would probably not have been located otherwise, and identifying a major design flaw.

The modeling effort, i.e. obtaining a PROMELA model from the LISP program, took about 12 man weeks during 6 calendar weeks, while the verification effort took about one week. The modeling effort consisted conceptually of an *abstraction* activity combined with a *trans-*

lation activity. Abstraction was needed to cut down the program to one with a reasonably small finite state space, making model checking tractable. Translation, from LISP to PROMELA, was needed to obtain a PROMELA model that the SPIN model checker could analyze.

The abstraction was done without any knowledge about the properties to be verified, since these were stated later. The abstraction maintained important operations on the lock table and ignored most other details of the original LISP program, hence, a kind of program slicing. No formal attempt was made to show that the abstractions preserved errors. It is interesting that such an ad hoc approach still was extremely effective. The translation phase was non-trivial and time consuming due to the relative expressive power of LISP when compared with PROMELA.

Based on these observations, two research efforts were initiated that should make application of model checking within the software development cycle less resource demanding. In one effort a translator from the Java programming language to PROMELA has been developed; see Section 4.2. In another effort, an abstraction tool has been developed, which can perform so-called predicate abstractions on Java programs; see Section 5. Both tools have been applied in the verification of the RA as described in the following.

4 Formal Analysis After Flight

Shortly after the anomaly occurred during the Remote Agent Experiment, on Tuesday May 18, the ASE team at NASA Ames heard that something had broken down in the RA while it was in control of the spacecraft and offered their help to the RAX team. On Friday morning, after a few email exchanges, the RAX team provided access to the source code of the Executive, without identifying where the error was, and offered the ASE group the challenge of seeing "how long it would take for formal methods to come up with it".

On Friday afternoon, we decided to run a "clean room" experiment to determine whether or not the technology currently used and under development in the group *could* have discovered the bug before it actually happened. At that time, we knew that debugging information collected from the spacecraft had enabled the DS-1 team to identify the bug and continue the experiment, and that the failure had something to do with a "handshaking" communication between a Planner process and an Executive process.

Other than these messages we had no further information, and no one in the ASE group had any contact with RAX personnel during that week.

This section first describes how the experiment was conducted. Then the Java PathFinder translator that was used to model check the flawed code is described. This is followed by a description of the error and how it was found using Java PathFinder. We conclude with a discussion of the methodology that has been followed.

4.1 The Clean Room Experiment

To make this clean room experiment credible, we decided that we would need to complete this exercise over the weekend, prior to the return of the RAX team from the DS-1 mission control at JPL the following Monday. This was both to avoid undue influence by people familiar with the details of the bug, and also to meet the "short-turnaround" challenge, mimicking what would be required if we were actually called on to provide "on-line" assistance.

The experiment was set up as follows. A *front-end* group would try to spot the error by human inspection, or at least identify problematic parts of the code. On the basis of that, it would extract a more or less self contained portion of the code containing the problematic code portions, of a tractable size for a model checker. This extracted code would then be handed over to the *back-end* group without any hints as to what could be the error. The back-end group would then try to locate the error using model checking. The situation was comparable to someone doing visual inspection of code, and finding suspect sections which he wanted to explore further.

The front-end team began perusing the code on Friday afternoon, and extracted roughly 700 lines containing questionable code[2]. The full group met again on Saturday afternoon, and the front-end team gave the back-end team the extracted code. In accordance with the design of the experiment, they did not tell where the suspected bug was, but they briefed the back-end team on the control and data structures of the extracted code. The back-end group spent most of the time understanding that code in order to model it, and on Sunday morning came out with a fairly abstract model of the suspicious code. That model was written in Java and verified with the Java model checker Java PathFinder, as described below. It reported a dead-

---

[2]Though they were not sure that they had indeed captured the concurrency error.

lock, which turned out to be the one that had happened in flight five days before.

4.2 The JPF Translator

Java PathFinder (JPF) [8, 6] is a translator from a nontrivial subset of Java to PROMELA. Given a Java program, JPF translates this into a PROMELA program, which then can be model checked using SPIN. Java is an object-oriented programming language with a built-in notion of threads. Objects are instantiated dynamically from classes, which can be defined using single class inheritance. Threads, which are special objects with an activity, can communicate by making calls to methods defined in shared objects. Such methods can be defined as synchronized, thereby turning these shared objects into monitors, allowing only one thread to operate in the object at a time.

In the default mode, the SPIN model checker will find any deadlocks present in the Java program. Such deadlocks can occur when several threads compete for access to the monitors. Properties can also be formulated explicitly by the user, either as assertions in the program, or as linear temporal logic formulae. That is, a Java program can be annotated with assertions written as calls to a special `assert` method which takes a boolean argument expression over the variables in the Java program. Any such call is translated into a corresponding PROMELA assertion, which will then be checked during the state space exploration whenever reached. Finally, SPIN's own linear temporal logic can be used to formulate properties over the Java program's static variables (a static variable in Java is defined within a class, but is only allocated once, and hence is shared between all objects of the class).

A significant subset of Java is supported by JPF: dynamic creation of objects with data and methods, static variables and static methods, class inheritance, threads and synchronization primitives for modeling monitors (synchronized statements, and the `wait` and `notify` methods), exceptions, thread interrupts, and most of the standard programming language constructs such as assignment statements, conditional statements and loops.

The translator is written in 6000 lines of LISP, and was developed over a period of 8 months. JPF has been applied to a number of case studies, amongst them a 1500 line game server [9], a NASA file transfer protocol for satellites, and a NASA data transmission protocol for the space shuttle ground control.

A related attempt to provide model checking technol-

ogy for Java is described by Demartini et. al. [5], which also translates Java programs into PROMELA. However, their approach does not handle exceptions or polymorphism as does Java PathFinder. In another related approach, Corbett [4] describes a theory of translating Java to a transition model, making use of static pointer analysis to aid *virtual coarsening*, which reduces the size of the model.

4.3 The RAX Error

The suspected and eventually confirmed error was a missing critical section around a conditional wait on an event. The relevant piece of code (anonymized for confidentiality purposes) is shown in Figure 3.

```
   (loop
     (when
(*1*)  (or (/= count (esl::event-count event1))
(*2*)      (warp-safe (wait-for-event event1)))
       (setf count (esl::event-count event1))
       ; ...
(*3*)  (signal-event event2)))
```

Figure 3: The RAX Error in LISP

This is the body of one of the concurrent tasks and consists of a loop. The loop starts with a `when` statement whose condition is a sequential-or statement[3] that states: *if the event counter has not been changed (`*1*`), then wait (`*2*`), else proceed*. This behavior is supposed to avoid waiting on the event queue if events were received while the process was active. However, if the event occurs between (`*1*`) and (`*2*`), it is missed and the process goes asleep. Because the other process that produces those events is itself activated by events created by this one in (`*3*`), both end up waiting for each other, a deadlock situation.

This follows a similar pattern to the code shown in Figure 1 that had been identified as a source of error during the verification of the Executive in 1997, as described in Section 3.3. This similarity was spotted by members of both the front-end and back-end teams, and contributed greatly to narrowing down the verification effort to this particular potential problem.

---

[3] `(or X Y)` is evaluated like `if X then true else Y`.

4.4 Demonstrating the Error with JPF

The modeling focused on the code under suspicion for containing the error. The major two components to be modeled were events and tasks, as illustrated in Figure 4. The figure shows a Java class `Event` from which event objects can be instantiated. The class has a local counter variable and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called `wait_for_event`. Note how the counter is incremented by `signal_event` in order to allow the tasks to check whether new events have arrived. The increment is modulo 3 in order to reduce the state space to be searched by the model checker. This is an informal abstraction in the sense that it has not been proven to preserve errors. Section 5 explains how an alternative counter abstraction for this program can be made and automatically proved correct.

```
class Event{
  int count = 0;

  public synchronized void wait_for_event(){
    try{wait();}catch(InterruptedException e){};
  }

  public synchronized void signal_event(){
    count = (count + 1) % 3;
    notifyAll();
  }
}


class FirstTask extends Thread{
  Event event1,event2;
  int count = 0;

  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      event2.signal_event();
    }
  }
}
```

Figure 4: The RAX Error in Java

Figure 4 also shows the definition of one of the tasks. This is an abstraction (in Java) of the LISP code presented in Figure 3. The task's activity is defined in the `run` method of the class `FirstTask`, which itself ex-

tends the `Thread` class, a built-in Java class that supports thread primitives. The body of the `run` method contains an infinite loop, where in each iteration a conditional call of `wait_for_event` is executed. The condition is that no new events have arrived, hence the event counter is unchanged. After having applied JPF, the SPIN model checker revealed the deadlock situation described in Section 4.3. In the Java context a new event arrived after the test `(count == event1.count)`, but before the call `event1.wait_for_event()`.

4.5 Discussion of Methodology

The formal analysis of the Executive after the occurrence of the anomaly was preceded by a code inspection, which identified the *possible* source of the error. Some of us spotted the potential error situation because it resembled the similar error we had found using SPIN in 1997, as described in Section 3.3. Due to the focus on the particular code fragment, it was relatively easy to perform the abstraction needed to extract a Java program with a small finite state space. This took about two hours. However, the suspicion was only a suspicion, and a demonstration that the code was flawed was provided using JPF. This showed the usefulness of using a model checker to answer focused queries.

Since the original source code was in LISP, we still had to translate it by hand in Java, which goes against JPF's intended purpose. To avoid that, one would need an abstraction tool and a translator for LISP. Since LISP's future within NASA is questionable we have focused on providing these technologies for Java. Java is a very convenient modeling language, providing most of the high level features of the powerful Common LISP Object System (CLOS), such as dynamically created objects with methods and data. The major experience with all experiments done with JPF are obviously that a non-trivial amount of abstraction is needed in order to reduce the size of a program's state space. This problem is addressed in Section 5.

5 An Abstraction Tool for Java

As a part of the JPF project, we have been developing an automated abstraction tool which converts a Java program to an abstract program with respect to user-specified abstraction criteria. The user can specify abstractions by removing variables in the concrete program and/or adding

new variables (currently the tool supports adding boolean types only) to the abstract program. Given a Java program and such abstraction criteria, the tool generates an abstract Java program in terms of the new abstract variables and unremoved concrete variables. To compute the conversion automatically, we use a decision procedure, SVC (Stanford Validity Checker), which checks the validity of logical expressions [1].

The abstraction tool is designed to deal with object-oriented programs. The user can specify abstraction criteria for each class by removing field variables in the class and/or adding new abstract variables to the class. Therefore, it can be used to abstract subcomponents in a program when the whole program is too complicated to apply abstraction globally. In addition, the user can specify new abstract variables which depend on variables from two different classes (inter-class abstraction).

There has been similar work by others [3, 15], all of which require use of only global variables to describe a system in simple languages similar to guarded commands. However, our tool targets a real programming language Java and is able to deal with many problems caused by its object-orientation.

5.1 Application of the Tool to the RA

As we do not have enough space in this paper for a detailed explanation of the abstraction algorithm, let us illustrate the abstraction performed by the abstraction tool on a part of the RA Java code shown in Figure 4. As stated before, state explosion occurs because of the unbounded increase of the count variable in the `Event` class (in the original LISP code) and the assignment of the `count` variable in the `FirstTask` class (as well as in the `SecondTask` class which is not shown). Therefore, we use abstraction to remove those `count` variables by specifying `Abstract.remove(count)` in the classes of `Event` and `FirstTask`. In place of these variables, we add new abstraction predicates which appear in the program with the `count` variables. For instance, we put `Abstract.addBoolean("FcntEqEcnt", count==event1.count)` in the definition of the `FirstTask` class to specify an abstraction predicate: `FirstTask.count` is equal to `Event.count` (For implementation convenience, object names are used to refer to class types.). We also used more inter-class abstractions such as FcntGeEcnt (`FirstTask.count` is greater than or equal to `Event.count`), ScntEqEcnt

(`SecondTask.count` is equal to `Event.count`), etc.

This is an example of an inter-class abstraction. Dealing with such inter-class abstractions is more involved than dealing with the abstractions inside one class. For each inter-class abstraction, the tool generates an additional class definition in the abstract program, which contains new boolean variables corresponding to the specified predicate. The boolean variables in the new class are defined as a two-dimensional array where each index refers to an object in either of the two classes. In Figure 5, the new abstract variable `FcntEqEcnt.pred[Fobj][Eobj]` corresponds to the user-defined predicate `FcntEqEcnt` for an object `Fobj` of `FirstTask` class and an object `Eobj` of `Event` class, i.e., `Fobj.count = Eobj.count`.

Given the abstraction criteria, we now need to compute the value of the abstract variables in the abstract program so that they are consistent with the values of concrete variables in the program. Figure 5 shows how the abstraction tool converts the assignment statement, `count = count + 1` (without the modulo operation) in Figure 4. First, the concrete assignment statement is omitted in the abstract program because the variable to be assigned has been removed. Instead, the tool checks which of the new abstract variables are possibly affected by this assignment and generates corresponding assignments to those abstract variables. For the example statement, a set of boolean variables that refers to 'this' Event object will be affected: `FcntEqEcnt.pred[i][this]` in Figure 5 (Actually, we use functions that return the corresponding index of a given object). To update those abstract variables, a for-statement is used. For each of the abstract variables, the pre-images that leads the abstract variable to be true (or false) by the assignment are computed. Then the pre-images are mapped into the abstract domain by checking validity of the corresponding logical expressions. Finally, the results are used as a guard condition to set the abstract variables to true (or false). In the example, the variable `FcntEqEcnt.pred[i][this]` will be set to false if it was true (or if some condition with another abstract variable holds). Otherwise, the variable is set to a non-deterministic boolean value. Because the concrete assignment statement is regarded as atomic, a set of these abstract assignments are declared as atomic for the JPF model checker. The additional statements for updating other abstract variables such as `FcntGeEcnt` are not shown in the figure.

```
Verify.beginAtomic();
// count = count + 1;
for(int i = 0; i < FcntEqEcnt.numFirstTask; ++i){
  if(FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)]
     || FcntGeEcnt.pred[i][FcntGeEcnt.getEvent(this)])
    FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)] =
      false;
  else FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)]
       = Verify.randomBool();
}

// similar code for updating other inter-class
// abstract variables such as FcntGeEcnt, etc.
Verify.endAtomic();
```

Figure 5: Output of the abstraction tool for the assignment statement

5.2 Discussion of Methodology

Using the tool, we have been able to obtain an abstract Java program of the RA code automatically. In the example, the unbounded integer variables are replaced by a set of boolean variables, hence the abstract program is free from the state explosion. Moreover, use of the tool helps to avoid error-prone abstractions based on human reasoning. The tool generates a sound approximation of the concrete program using an automated validity checker, although it is not necessarily the most accurate one.

However, the user must give reasonable abstraction criteria for the tool to generate a meaningful abstract program in order to check some desired properties. In case the abstraction criteria are not good enough, the result will be a too rough abstract program which can not preserve the properties to be checked.

6   Conclusion

This paper describes two major verification efforts carried out within the Automated Software Engineering Group at NASA Ames Research Center. The first effort consisted of analyzing part of the RA autonomous space craft software using the SPIN model checker. One of the errors found with SPIN, a missing critical section around a conditional wait statement, was in fact reintroduced in a different subsystem that was not verified in this first pre-flight effort. This error caused a real deadlock in the RA during flight in space.

Such concurrency-related errors only happen as the result of particular scheduling circumstances. Scheduling is totally uncontrolled when tests are run, and is highly sen-

sitive to variations in the operating environment (e.g. operating system, other running tasks). This explains why the anomaly happened in flight, though it had not occurred even once in thousands of previous runs on the various ground testbeds.

Developing the formal model of the program was, however, a time consuming task, requiring a manual translation from the RA LISP code to the PROMELA language of the SPIN model checker. In addition, code details had to be abstracted away in order to obtain a small enough finite state system that could be effectively model checked. The translation difficulty spawned the initiative to automate the translation from high level programming languages to modeling languages for formal verification, such as PROMELA. Java was chosen as the source language because of its modern programming language constructs, such as support for object-oriented programming, and the standardization across implementations of its concurrency constructs. An automatic translator from Java to PROMELA was designed and implemented, called Java PathFinder (JPF). With JPF one can model check smaller Java programs for assertion violations, deadlocks, and general linear temporal logic properties. The translator covers a substantial subset of Java, illustrating the feasibility of the approach.

In the second effort, JPF was used for modeling the RAX deadlock after it occurred. That is, after the front-end team isolated a reduced subset of the code that likely included the error, the back-end team developed a Java program which exposed the error. The translator translated this into a PROMELA model, and the model checking of this model then immediately revealed the error. Java turned out to be an excellent choice as a modeling language, with a high level of abstraction, due to its object oriented features. In later work, a system that automates certain aspects of predicate abstraction was developed and successfully demonstrated on the same example.

This experience gave a clear demonstration that model checking can locate errors that are very hard to find with normal testing and can nevertheless compromise a system's safety. It stands as one of the more successful applications of formal methods to date. In its report of the RAX incident, the RAX team indeed acknowledges that it "provides a strong impetus for research on formal verification of flight critical systems" [13].

A posteriori, given the successful partial results, one can wonder why the first verification effort was not extended to the rest of the Executive, which might have spotted the error before it occurred in flight. There are two reasons for that. First, the purpose of the effort was to evaluate the verification technology, not to validate the RA. The ASE team did not have the mission nor the resources needed for a full-scale modeling and verification effort. Second, the part of the code in which the error was found has been written after the end of the first verification experience.

Regarding software verification, the work presented here demonstrates two main points. First of all, we believe that it is worthwhile to do source code verification since code may contain serious errors that probably will not reveal themselves in a design. Hence, although design verification may have the economical benefit of catching errors early, code verification will always be needed to catch errors that have survived any good practice. Code will always by definition contain more details than the design – any such detail being a potential contributor to failure.

Second, we believe that model checking source code is practical. The translation issue can be fully automated, as we have demonstrated. The remaining technical challenge is scaling the technology to work with larger programs - programs that could have very large state spaces unless suitably abstracted. Abstraction is of course a major obstacle, but our experience has been that this effort can be minimized given a set of supporting tools.

Acknowledgments

References

[1] C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.

[2] D. Bernard et al. Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment. In *Proceedings of the AIAA 1999, Albuquerque, NM*, 1999.

[3] M. Colón and T. Uribe. Generating Finite-State Abstractions of Reactive Systems using Decision Procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, July 1998.

[4] J. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis*, March 1998. Clearwater Beach, Florida.

[5] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, November 1998. Paris, France.

[6] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, July and September 1999. Trento, Italy – Toulouse, France (presented at the 6th Workshop).

[7] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998. To appear in IEEE Transactions of Software Engineering.

[8] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. To appear in a special issue of *International Journal on Software Tools for Technology Transfer* (STTT) containing selected submissions to the 4th SPIN workshop, Paris, France, 1998, February 1999.

[9] K. Havelund and J. Skakkebæk. Applying Model Checking in Java Verification. In *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, July and September 1999. Trento, Italy – Toulouse, France (presented at the 6th Workshop).

[10] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[11] N. Muscettola. *HSTS: Integrating Planning and Scheduling*. Morgan Kaufman, 1994.

[12] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[13] P. Nayak et al. Validating the DS1 Remote Agent Experiment. In *Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS-99)*. ESTEC, Noordwijk, 1999.

[14] B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, and B. Williams. An Autonomous Spacecraft Agent Prototype. *Autonomous Robots*, 5(1), March 1998.

[15] H. Saïdi and N. Shankar. Abstract and Model Check While You Prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, July 1999.

[16] B. Williams and P. Nayak. A Model-Based Approach to Reactive Self-Configuring Systems. In *Proceedings of AAAI-96*, 1996.

Taking the hol out of HOL

Nancy A. Day
Oregon Graduate Institute
Portland, OR, USA
nday@cse.ogi.edu

Michael R. Donat and Jeffrey J. Joyce
Intrepid Critical Software Inc.
Vancouver, BC, Canada
{Michael.Donat, Jeffrey.Joyce}@intrepid-cs.com

Abstract

We describe a systematic approach to building tools for the automated analysis of specifications expressed in higher-order logic (hol) independent of a conventional, interactive theorem proving environment. In contrast to tools such as HOL and PVS, we have taken "the hol out of HOL" by building automated analysis procedures from a toolkit for manipulating hol specifications. Our approach eliminates the burden of skilled interaction required by a conventional theorem prover. Our lightweight approach allows a hol specification to be used for diverse purposes, such as model checking, and the algorithmic generation of test cases. After five years of experience with this approach, we conclude that by decoupling hol from its conventional environment, we retain the benefits of an expressive specification notation, and can generate many useful analysis results automatically.

1 Introduction

Formal methods have come a long way. Industrial standards such as IEC 61508, and DO-178B include explicit references to the use of formal methods as a means of increasing confidence in safety-related systems. Formal methods add precision and checkability to various aspects of the system development process.

A decade ago, there was a wide chasm between specialized automated methods such as model checking [6], specification-intensive methods such as the use of Z [33], and general proof-based reasoning found in tools such as HOL [16]. The input notations of the analysis tools matched the analysis capabilities of the tool. For example, the SMV [26] notation describes finite state machines, whereas the use of higher-order logic (hol)[1] as the specification language of PVS corresponds to the intended use of PVS [28] as an interactive theorem prover.

Progress is being made rapidly on bridging this chasm and uniting the capabilities of the various tools

under one roof. For example, the SCR toolset includes a consistency checker, a simulator, and links to a model checker, and a theorem prover [3, 20]. PVS has integrated a number of automated decision procedures [27]. Most of these examples are, however, either application-specific such as the SCR toolset, or start from a heavyweight theorem prover.

We have been exploring a different point in the design space of these combined systems. For the past five years, in an industry/university collaborative research project, we have used hol as a specification notation and applied automated analysis techniques such as refutation-based approaches (i.e., those that generate counterexamples), and test generation to these specifications. We have taken "the hol out of HOL" by building these automated procedures on top of just a parser and typechecker to eliminate the burden of skilled interaction required by a conventional theorem prover.

The combination of hol with automated analysis may seem crippled from the beginning: we do not have all the tools we might need to work with our specification. However, our experience shows that less power is often better. The expressiveness of higher-order logic allows us to embed more familiar notations within hol. The difficulties for new users come when the only tool support available has a high learning curve, and they struggle to understand the feedback the tool provides them about their specification. We offer a solution that lessens the learning curve, delaying the need to use a theorem prover until the problem requires it and the user is ready for it.

In Sections 2, and 3 we present our reasons for choosing to work with higher-order logic outside of a theorem proving environment. In Section 4, we describe our toolkit, a collection of cooperating utilities that manipulate hol expressions in "truth-preserving" ways, i.e., the result of every transformation could also have been produced by a formal derivation using inference rules in HOL. In Section 5, we describe how the blocks are used in combination to construct analysis procedures such as symbolic model checking, and test generation.

---

[1]We will use "hol" or "Hol" for higher-order logic by itself, and "HOL" to refer to the HOL theorem proving system.

Unlike our related presentations of this project [8, 9, 10, 14, 23], in this paper we focus on the capabilities of the tool and how it is engineered. This paper is intended to be a high-level view of the architecture of our analysis tool, illustrating how our toolkit facilitates significant reuse of components for diverse applications such as test generation and model checking. We have also created new analysis methods such as constraint-based simulation. Our focus on automated analysis compels us to provide the user with control of performance factors such as BDD [4] variable order. We have also created methods that allow us to maintain the information necessary to produce readable, traceable results given in terms of the original specification. References are provided to more technical descriptions of the components of our toolkit.

By providing a lightweight interface between a general-purpose notation and automated analysis, we offer a middle ground between special-purpose analysis tools and general-purpose theorem provers. Our goal is to bring the power of a range of automated analysis techniques to specifiers without sacrificing suitability and expressiveness of notation.

2   Why higher-order logic?

Initially, we chose higher-order logic as a specification notation independently of consideration for tool support. Our notation S [23] is a syntactic variant of the object language of the HOL theorem proving system. S was also influenced by Z, in that it includes constructs for the declaration and definition of types and constants. It was developed to support the practical application of formal methods in industrial scale projects. In this section, we explain our reasons for choosing to work with S.

First, S is a *general-purpose notation*; it does not impose any particular style of specification. We have used it to capture a stimulus-response style of specification, as well as embedding other notations such as statecharts [17], and tables in S [2, 9]. By placing specialized notations within a general-purpose environment, we can take advantage of many general-purpose features such as parameterization, and re-usable auxiliary definitions and infrastructure. In the specification of an aeronautical telecommunications network (ATN) written in our embedded statecharts style, we witnessed these benefits, which reduced the specification effort, and resulted in a more concise and readable specification [2]. Also, we do not have to repeat the effort of building analysis tools for particular notations. Once a notation is embedded in S, many of our analysis tools can be applied.

Second, S is a logic. We have found that *uninterpreted constants* in a logic play a key role in allowing

us to match the level of abstraction found in requirements specifications. Joyce has called uninterpreted constants, "a modern-day Occam's razor"[2] and points out their value in filtering non-essential details and in improving the readability of the specification [25]. Uninterpreted constants can be used to represent elements that have meaning to domain experts but whose definition is irrelevant for analysis of a requirements specification. For example, many air traffic control specifications depend on the "flight level" of an aircraft. The details of how the flight level is determined may be irrelevant for analysis of some aspects of the system. The calculation of the "flight level" can be captured by an uninterpreted constant. Analysis results produced for a specification hold for any interpretation of the uninterpreted constants. While a final specification should be complete including definitions for all the constants, the use of uninterpreted constants during the process of writing a specification allows some results to be produced without having to specify all of the details.

Furthermore, a logic contains *quantifiers*, which often allow the expression of formal requirements to more closely correspond to their expression in natural language. Quantified statements can be used to capture domain knowledge that describes the environment of the specification. The ability to use a quantifier eliminates the need to spell out all instances where the environmental assumption is relevant.

Finally, S is expressive; while we will never be able to prove automatically every property of our specifications, our notation is unlikely to limit adding more automated analysis capabilities as they are developed.

3   Why not use a theorem prover?

In our approach, we have focused on automated analysis of our specifications. There have been a variety of successful efforts to combine theorem provers with automated decision procedures, such as PVS and Forte [1]. Our experience with HOL-Voss [24] suggest that having the theorem prover control the link to the decision procedures is not the optimal approach for automated analysis.

First, the infrastructure of the theorem prover is unnecessary for automated analysis and makes the approach clumsy and intimidating to the novice specifier. These difficulties are a factor in industry's resistance to formal methods. For example, we particularly wanted to avoid the need to learn a meta-language to

---

[2]The Aristotelian principle, often attributed to William of Occam (1300-1349), that the simplest theory that fits the facts of a problem is the one that should be used.

accomplish the specification task. Therefore, we made S the input language to our tool, and have very simple commands to invoke our analysis procedures. A second example is that rewriting by means of tactic application was used for expansion of definitions in HOL-Voss. This step was different for each specification analyzed. We have shown that an automatic technique, called symbolic functional evaluation, is sufficient for this task and requires no user intervention.

Second, theorem provers are verification-based analysis tools. The output of a theorem prover is the confirmation of a conjecture. Often, more useful results of analysis are either evidence that refutes an interpretation of the requirements, or truth-preserving rearrangements of the specification in order to distill atomic behaviour. Refutation-based techniques produce a variety of results other than just theorems. For example, when analyzing a table for inconsistency, refutation-based techniques can clearly isolate the source of the inconsistency. Consequently, it is easier to interpret the result of a successful refutation attempt than a failed verification attempt. In using formal methods for an independent validation and verification effort, Easterbrook and Callahan abandoned the use of PVS to carry out completeness and consistency checks because of the difficulty of determining the source of an inconsistency in a failed proof [15].

Third, the results should be expressed in terms of the original specification. In contrast to our approach, translating the specification for input to a specialized decision procedure often results in output in terms of the translated version.

Fourth, most theorem provers do not currently provide hooks to control analysis parameters such as BDD variable order. To work with large examples, control over these parameters is absolutely necessary.

Theorem provers definitely have a role to play in the analysis of complex systems. We advocate an approach that complements the use of theorem provers because we work with the same notation. Novice users and experts can work side-by-side. We have a tool that translates our S specifications to input for the HOL theorem prover [23].

4   The Toolkit

Our toolkit consists of techniques that manipulate S expressions in truth-preserving ways. In this section, we describe the collection of techniques that are combined to build analysis procedures such as symbolic model checking. Figure 1 captures the architecture of our tool. In addition to the specification and commands, the input of semantic definitions allows the specifier to work with notations, such as statecharts, embedded in S.
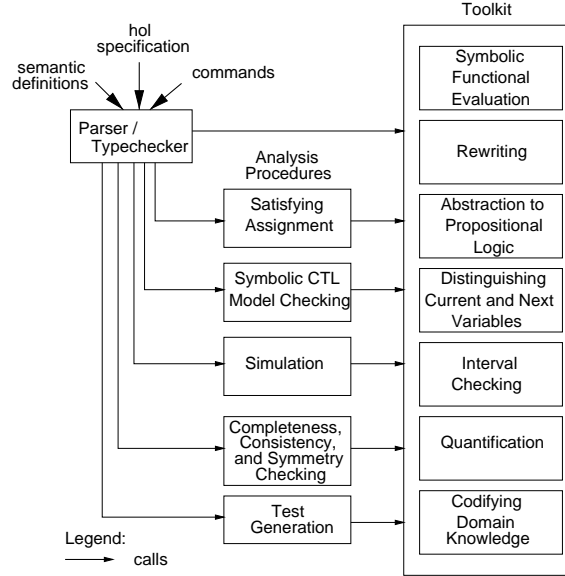

Figure 1: Architecture

The representation of S expressions is encapsulated in an abstract datatype. The representation is created through the process of parsing and typechecking, common to all analysis procedures. Analysis procedures consist of a sequence of calls to the toolkit elements, which manipulate S expressions to accomplish the analysis task. Each of the toolkit elements are independent allowing them to be used systematically in combination to implement analysis procedures. Also the separation of concerns allows each toolkit element to evolve, and additional elements be added, without affecting other components of our tool.

Some of the techniques, such as abstraction to propositional logic, can also be found in tools such as PVS. Others, such as symbolic functional evaluation (SFE) for expanding S expressions, we developed because we wanted to be independent of a theorem proving environment. In some cases, we rely on syntactic conventions for particular styles of specification. For example, we distinguish between the stimuli and responses for test generation based on vocabulary conventions.

We also provide user access to performance tuning for some of these automated techniques. For example, while SFE is automatic, the user can control the depth of evaluation. For BDD-based analysis, we provide a way to input a variable order.

4.1   Symbolic Functional Evaluation

A specification consists of a collection of constant definitions, and declarations of types and constants. If we are using an embedded notation, then a set of semantic definitions is added to this collection. Often,

the first step in analysis is to expand all of these definitions to determine the meaning of the specification.

Symbolic functional evaluation [8] (SFE) is a technique that we developed to "evaluate" or unfold S expressions, i.e., carry out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of quantifiers and uninterpreted constants. It extends mechanisms from functional language evaluation to carry out lazy evaluation of S expressions. Unlike using quote symbols in a language such as Lisp, SFE gives the user control over the depth of evaluation. We illustrate this control with the following declarations and definitions:

$$z_1 : \text{num};$$
$$f_1, f_2, f_3 : \text{num} \rightarrow \text{num};$$
$$z_2 = f_1(z_1);$$
$$z_3 = f_2(z_2);$$
$$f_4(a) = f_3(a);$$

The constants z_1, f_1, f_2, and f_3 are uninterpreted. When we evaluate the expression $f_4(z_3)$, we can instruct SFE to evaluate to one of three levels of evaluation. At the level of "complete" evaluation, it is expands all the definitions and returns the expression $f_3(f_2(f_1(z_1)))$. At the "point of distinction" level, SFE stops after it determines the tip of the expression is an uninterpreted function, and returns $f_3(z_3)$. One further level called "evaluated for rewriting" proved useful and evaluates the arguments of an uninterpreted function at the tip to the point of distinction. In this case, it would return $f_3(f_2(z_2))$.

The choice of level of evaluation is linked with the choice of abstraction to be used for the automated analysis. For example, when abstracting an expression to propositional logic (see Section 4.3), the point of distinction level is most appropriate because any details revealed by evaluation are lost in abstraction.

Our implementation benefits from the use of structure sharing in the representation of expressions, and caching of results.

SFE can be used to carry out symbolic simulation of specifications of hardware circuits as has been done previously in theorem provers, e.g., [34, 35].

SFE provides functionality similar to that of PVS's experimental ground evaluation, which translates a subset of PVS into Lisp for evaluation [32]. However, SFE works for any expression in higher-order logic, including uninterpreted functions, and quantifiers. Our levels of evaluation provide a systematic means of controlling evaluation of these symbolic expressions. A second difference is that we use SFE as the first step in the analysis process. In PVS, evaluation currently is stand-alone and does not affect the proof process. For our purposes, SFE is sufficiently fast for large specifications, however the PVS ground

evaluation is no doubt faster using existing Lisp evaluation and destructive updates where possible.

4.2 Rewriting

Once a specification has been sufficiently unfolded, several analyses require logical manipulation of the resulting S formula. A rewrite toolkit component is useful for performing this task. For example, the following set of rewrite rules could be used to rewrite a specification so that negation ($\neg$) is only applied to predicates:

$$\forall X, Y. X \Rightarrow Y = \neg X \vee Y$$
$$\forall X, Y. \neg(X \wedge Y) = \neg X \vee \neg Y$$
$$\forall X, Y. \neg(X \vee Y) = \neg X \wedge \neg Y$$
$$\forall X. \neg\neg X = X$$
$$\forall P. \neg\forall x. P(x) = \exists x. \neg P(x)$$
$$\forall P. \neg\exists x. P(x) = \forall x. \neg P(x)$$

Some analysis algorithms can be implemented as a series of rewriting operations. An example is the derivation of tests from an S specification using a series of sets of rewrite rules [10, 13]. Implementing the test generator using rewriting is a better way to preserve logical soundness than an implementation as a series of ad-hoc manipulations.

Our lightweight rewrite system differs from some well-known rewrite systems, such as the one found in HOL. For performance reasons, our rewrite system cooperates with other means of simplification such as evaluating expressions with concrete values. The user of the rewrite system must ensure that each set of rewrite rules is confluent – otherwise, rewriting may not terminate. The user must also ensure that the rewrite rules are themselves sound. The checking of the rules need only be performed once as part of the development of an analysis procedure, and can be accomplished using a theorem prover such as HOL or PVS.

Rewrite rules are stated as universally quantified equalities, e.g., $\forall x. E_1(x) = E_2(x)$, where x is a vector of variables. For rules specifying rewrites involving quantifiers and lambda abstraction: 1) variable capture is avoided using alpha conversion; and 2) if variable release occurs, the rewrite fails.

The concept of *variable release* is the opposite of variable capture. During rewriting, if a variable is quantified in an expression matching the left-hand side of the rewrite rule and is not quantified in the corresponding instance of the right-hand side, variable release has occurred. For example, applying the rewrite rule $\forall P, Q. (\forall x. P \vee Q) = ((\forall x. P) \vee Q)$ to $\forall x. f(x) \vee y$ succeeds. However, applying the same rule to $\forall x. f(x) \vee g(x)$ fails because the x of $g(x)$ is released, i.e., x is no longer quantified because it was free

in Q. The rewrite system also recognizes alpha equivalence, e.g., $(\lambda x.E(x)) = \lambda a.E(a)$. By failing rewrites in which variable release occurs and recognizing alpha equivalence, we are able to describe as rewrite rules quantifier manipulation that requires conversions in a theorem prover.

The rewrite system provides routines for applying a single rewrite to an expression, or to an expression and all its subexpressions. Sets of rules can also be applied. The depth of a rewrite operation can be limited by providing a call-back function that examines the current subexpression and signals the rewrite system to continue with this subexpression or go no deeper.

4.3   Abstraction to Propositional Logic

By abstracting our specifications to propositional logic, we can produce conservative analysis results automatically. As in Rajan [29], we decompose our S expression based on the logical operators of conjunction, disjunction, and negation. The fragments are assigned unique Boolean variables with alpha-equivalent subexpressions matched to the same variable. We maintain a table matching the fragments to their Boolean variables to apply and reverse this process.

We also deal with enumerated types so that they are represented by multiple, related Boolean variables as in Ever [22]. Sections 4.5 and 4.7 discuss elements of the toolkit that complement this abstraction process.

We represent the expressions built from the Boolean variables using BDDs. A key to making this process efficient is to cache the match between S expressions and BDD expressions. Once a BDD expression is created, an analysis procedure can manipulate it with the usual BDD package operations such as negation, conjunction, and quantification.

BDD variable order affects the size of the BDD representation of our S expression. For small examples, it is sufficient to create the BDD variable as needed in the abstraction process, but for larger examples, a better method was required. In PVS, it is possible to request that dynamic variable order be carried out within the BDD package doing propositional simplification [31]. But, we found it critical to have direct support for providing the abstraction process with a BDD variable order to allow us to reuse a good order, as well as store and manipulate abstractions of expressions. Furthermore, we wanted the variable order stated in terms of expressions of the specification, not in terms of the Boolean variables that are substituted for those expressions during abstraction.

Therefore, we developed a way of supplying a variable ordering for BDDs as a list of S expressions. There are three types of substitutions: a single Boolean variable matched with a Boolean S expres-

sion, partitions discussed in Section 4.5, and enumerated types. Each type of substitution is accompanied by a list of numbers giving the position in the order of the Boolean variables used to represent the S expressions. We provide some utilities to help the user determine a good variable order by subcontracting the problem to existing verification tools such as the Voss Verification System [30]. Further details on our approach can be found in Day [7].

Creating a Boolean abstraction of an S expression and then reversing the process, can be a useful method of simplifying expressions that include quantification over Boolean variables. The resulting expression is logically equivalent to the original. Our tool provides a command that evaluates an expression to the desired level of evaluation using SFE, creates a BDD representation of the expression, and then creates an S expression from the BDD. We used this process in constructing a large next state relation by constructing conjuncts representing concurrent states individually first.

4.4   Distinguishing Current and Next Values

Specifications written in notations such as finite state machines describe a next state relation. Since S has no built-in notion of dynamic behaviour, a means is required to distinguish the value of a variable in the current state from its value in the next. Our toolkit implements three approaches to this problem based on syntactic conventions.

The first approach is to make each variable a function mapping system states to values for that variable, similar to the concept of variables as functions of time. The approach is well-suited for embedded state transition notations, where the system state is implicit in the use of the variable. In this approach, we avoid the need to group the variables in a record structure explicitly as has been done in PVS [29].

To support this approach to handling dynamic behaviour, an element of the toolkit separates the Boolean variables representing the current state values from those for next state values after abstraction to propositional logic. In the semantics for embedded notations, we adopt the syntactic convention that the variable cf represents the current state, and cf' the next state, thus a Boolean expression such as $x(cf')$ refers to the value of the variable x in the next state. Expressions such as $y(cf') = (y(cf) + 1)$ that contain both cf and cf' are considered as one Boolean variable belonging to the next state.

A second approach is to adopt the convention of Z, where a prime ($'$) is used to distinguish current state values from next state values. Thus, in the specifi-

cation $(z = g(x,5)) \Rightarrow (z' = g(x,10))$, $z = g(x,5)$ refers to the current state because it does not contain a primed variable. The presence of z' indicates that $z' = g(x,10)$ is a condition on the next state.

A third approach uses the syntactic convention that a literal beginning with a lower case letter indicates a next state predicate. A command can specifically label a literal as referring to either state, overriding this convention. This mechanism is appropriate in situations where the vocabulary used to specify next state values is different from that of specifying current state values, e.g., some applications of system-level requirements-based testing [14].

In some cases, the convention used to distinguish values in time is intrinsically linked to the type of analysis, and cannot be supported by an independent part of the toolkit. For example, the test generation process guides the rewrite system to distinguish stimuli from responses, placing expressions in certain forms.

4.5   Interval Checking

The process of abstracting to propositional logic is very conservative. It abstracts expressions such as $x < 5$, $(5 \le x \land x \le 10)$, and $10 < x$ to unrelated Boolean expressions, potentially causing the analysis results to return impossible cases. In this section, we consider options for avoiding this difficulty. One approach is to rewrite predicates involving inequalities into a canonical form to find relationships between expressions such as $x < 5$ and $5 > x$. However, this fails to capture the relationship between $x < 5$ and $10 < x$. A second alternative is to use an external tool to add constraints based on the numeric relationships [5].

Instead of any of these choices, we chose a simple approach that was complementary to the process of abstracting to propositional logic, and that depended on the structure of the notation. Our approach treats related expressions that partition a numeric value as an enumerated type. Based on known structure of a particular notation, we can identify some related expressions without a global search of the complete specification. We encountered linear inequalities in tabular specifications where the cells of a row of a table partitioned the values of a numeric expression.

We can identify the row structure within the specification by searching for the `Row` keyword used in the embedding of the tabular notation. To exploit the structure we extended our tool with a registry mechanism such that when certain keywords are encountered by SFE, particular procedures are carried out. The `Row` keyword is associated with a simple "interval checking" algorithm that takes the list of expressions in a row and determines if they represent a non-overlapping partition. Our registry mechanism makes

it possible to extend easily SFE with other structure-specific rules.

In our current implementation, interval checking works for S expressions that contain numeric comparison operators and have a concrete value on at least one side of the operator. Interval checking also returns any ranges not used in the row entries. By treating the partition as an enumerated type, the related numeric expressions are encoded as related Boolean variables in the abstraction process.

4.6   Readable Results

A significant challenge in requirements analysis is returning results that are understandable and in the same terms as the specification despite the abstractions used in analysis. One step towards this goal is maintaining the information to reverse the Boolean abstraction as described in Section 4.3.

We are able to produce even better results by tracking information through the expansion and logical manipulation processes of SFE and rewriting.

4.6.1   Unexpansion

Through an enhancement of the representation of S expressions, we are able to return an expression in its unevaluated, and usually more compact, form. Technically, lazy evaluation replaces a subexpression with its evaluated form, so the work of evaluation is done only once for all common subexpressions. We have modified our representation of expressions to include a pointer to the original, unevaluated version of the expression.

At the expense of memory, we are able to keep both the evaluated and unevaluated forms of the expressions during SFE. Some analysis procedures choose to output the unevaluated form of the expression to present a more abstract representation of the output.

4.6.2   Traceability

Unexpansion is not sufficient when manipulations other than expansion are performed. For analyses that perform rewriting, it is often critical that the results be traceable to their source in the specification.

For example, tests generated from a specification are logical consequences of it. If a test is produced that represents clearly unintended behaviour, then its source in the specification needs to be located before it can be corrected. In the case of a non-trivial input specification, identifying the source of a test can be surprisingly difficult especially when there is significant "collaboration" between individual requirements.

An extension to our parser allows subexpressions within the S specification to be tagged with user de-

fined identifiers [11]. This use of identifiers is consistent with many requirements specification techniques now used in industry. During rewriting, the tags are propagated. By displaying these tags with the analysis results, the source of the results can be determined.

4.7 Quantification

Our specifications can include quantifiers. In abstraction, a quantified subexpressions can be treated as a single Boolean variable for the purpose of automated analysis. However, we can do better than this conservative approach in certain circumstances. The substitutions described in this section can be done either during SFE or rewriting, or as a separate function.

For quantified variables of types with a finite number of members we can substitute the possible values for the variable, e.g., universal quantification over a finite set of values can be expanded into a conjunction of conditions. For example, given the following type definition and predicate declaration:

: chocolate := Cadburys | Hersheys | Rogers;
$tastesGood$: chocolate → bool;

the expression

$$\forall(x : \text{chocolate}).tastesGood(x)$$

can be rewritten as:

$$tastesGood(\text{Cadburys}) \; \wedge$$
$$tastesGood(\text{Hersheys}) \; \wedge$$
$$tastesGood(\text{Rogers})$$

For quantified variables of infinite or uninterpreted types, we have experimented with methods for instantiating universally quantified variables. When the antecedent of a logical implication is a universally quantified term, the universally quantified variable can be instantiated by any uninterpreted constant of the appropriate type. This substitution is a form of precondition strengthening. Because $(\forall x.P(x)) \Rightarrow P(a)$, we can prove $(\forall x.P(x)) \Rightarrow Q$ by proving $P(a) \Rightarrow Q$. This substitution is useful as part of various analysis tasks such as completeness and consistency checking. It transforms constraints on the environment stated in terms of quantification into a non-quantified form that can be used in automated analysis. For example, given the following declarations and definitions,

A, B : flight;

$env = \forall(f : \text{flight}).$
$\quad \neg(is\_flying\_level(f) \; \wedge \; is\_climbing(f));$

in a specification, we use the instances of the universally quantified environmental constraint for A and B,

namely:

$$\neg(is\_flying\_level(A) \wedge is\_climbing(A)) \wedge$$
$$\neg(is\_flying\_level(B) \wedge is\_climbing(B))$$

We found this form of substitution very useful for environmental assumptions, which are often stated with universal quantification.

The approach used in test generation is based on a test coverage point of view. The user identifies the type of a quantified variable, treated as a set, as either static or dynamic. A type is *dynamic* if it can be different in different contexts of the specification. For example, quantification over the "flight" type might be dynamic, since there can be different numbers of aircraft within an airspace at any given time. A type is *static* if it is not dynamic, e.g., the set of natural numbers is a static specification element.

When a quantified variable has a type that is a dynamic set, we consider what instances of the type should be analyzed to ensure adequate coverage in testing. This type of simplification can be performed in at least three modes: none, single, or all. In the "single" mode of coverage, for the expression:

$$\forall x : \text{X}. \; P_1(x) \vee P_2(x) \vee \ldots \vee P_n(x)$$

we substitute a single value of type X, because this expression can be satisfied if one value has one of the properties P_i. For example if the type X contains a value c, the quantified expression above would be replaced by $P_1(c)$. In the "all" mode, we substitute n points, each one addressing a different P_i. Any constants introduced must be new, and free in the specification.

4.8 Codifying Domain Knowledge

Domain knowledge, or environmental assumptions, are conditions that must be taken into account during analysis to disregard infeasible combinations of conditions, and simplify expressions. In system-level requirements, we found there are relatively few dependencies between conditions, and therefore these can be expressed concisely using quantified axioms.

For some types of analysis, domain knowledge can be combined with the specification in the analysis. It is the antecedent of the analysis goal, or conjuncted with the symbolic representation of the state set to enforce the constraint. In these cases, the substitution of relevant constants in the quantified expression described in Section 4.7 proved very useful.

In other types of analysis, such as test generation we cannot combine the statements of the domain knowledge with the specification because every part of the output must be traceable to the inputs. For these

cases, we identified three schemata that capture the form of many of the axioms that are often used:

1. $\forall x.G \Rightarrow \text{MutEx}[P_1(x); P_2(x); \ldots P_n(x)]$,

2. $\forall x.G \Rightarrow \text{Subsm}[P_1(x); P_2(x); \ldots P_n(x)]$, and

3. $\forall x.G \Rightarrow \text{States}[P_1(x); P_2(x); \ldots P_n(x)]$.

These schemata map the problem of simplifying an expression containing elements that match the patterns given in the schemata list to the problem of satisfying the guard G for the same instance of x. For example, conditions that form partial orders can be defined using Subsm. Conditions on the right subsume conditions on the left in the Subsm list. The statement $\forall x, y, z.x < y \Rightarrow \text{Subsm}[x < z; y < z]$ captures the information that if $k < i$ then $i < j \Rightarrow k < j$. The optional guard G, in this case $x < y$, provides a means of converting the dependency into a standard domain for which the analysis tool has a decision procedure. An expression such as $5 < x \wedge 10 < x$, is simplified by the schemata to $10 < x$ because it can check $5 < 10$. The MutEx form is used to define dependencies between mutually exclusive conditions. The States form defines conditions that represent a set of states; exactly one is true. These forms, combined with the pattern-matching capabilities provided by the rewrite system, are a powerful method of allowing the user to provide input to the tool as domain knowledge.

Though we found that the above approaches meet our needs, they have certain limitations. First, when there are more dependent relationships dictated by the environment, a formal model of the environment may be more concise than just axioms. Second, for more complex relationships it may be more efficient to provide a specially coded decision procedure, rather than pattern matching and basic evaluation to simplify expressions.

5 Analysis Procedures

The procedures in our toolkit are combined together to form analysis procedures. In this section, we describe the procedures we have applied in examples. Table 1 is a partial list of the commands currently available in our tool.

5.1 Generating a Satisfying Assignment

To further one's understanding of the meaning of a complicated Boolean S expression, it can be useful to examine a satisfying assignment for that expression. This analysis procedure first expands any defined symbols in the expression using symbolic functional evaluation, and then constructs a Boolean abstraction of the expression represented as a BDD. The user chooses the evaluation level for SFE. Using an algorithm found in the Voss system due to Seger, we provide two possible ways of producing a satisfying assignment. One attempts to choose as many true assignments to variables as possible and the other has preference for false assignments.

5.2 Symbolic CTL Model Checking

Our model checking procedure takes constants with definitions that are 1) a CTL formula, 2) a next state relation, 3) an initial condition, and 4) an optional environmental constraint. We have a representation of CTL formula as an S datatype. Internally the expression representing the CTL formula is decomposed to invoke procedures based on the definitions of the component formulae. The next state relation, initial condition, and environmental constraint are all evaluated using SFE, and abstracted to propositional logic as a BDD. The current and next state variables are determined for the next state relation.

We currently have counterexample generation for AG and EF CTL formulae.

5.3 Simulation

For state machine specifications, a non-exhaustive form of configuration space exploration is simulation. The presence of uninterpreted constants in the specification forces our simulation to be symbolic.

Our analysis procedure does simulation based on the BDD representing the next state relation and constraint satisfaction. The user can constrain the set of assignments possible for the initial state and each subsequent state using a list of conditions. A particular assignment to the Boolean variables is chosen at each step. This assignment becomes the previous configuration for the next step. By choosing a particular assignment each time, this form of simulation does not encounter the state space explosion problem as in model checking.

A sequence of steps may not exist that satisfies the listed conditions. An arbitrary choice of a particular state that satisfies the constraints made early in the simulation may result in a satisfying sequence of steps not being found when one does exist. An alternative, slightly more expensive, analysis procedure carries out "one-lookahead". At each step, it chooses a configuration that satisfies the applicable constraint and has a next state that satisfies the next constraint in the list.

Command	Action
`%setorder <const>`	use the BDD variable order given by the expression list `<const>`
`%save_bdd <const> <fname>`	save a BDD associated with a constant in the file
`%load_bdd <const> <fname>`	load a BDD from the file into constant
`%bddsimp <const> <ret_c>`	simplify `<const>` using BDDs; put result in `<ret_c>`
`%bddsatisfies <const>`	using BDDs, provide a satisfying assignment
`%ctlmc <ctl> <nsr> <ic> <env>`	do symbolic CTL model checking given next state relation, initial condition, and environmental assumption
`%simulate <nsr> <c_list>`	simulate the next state relation by satisfying the constraint list in each step
`%comp <const> <env>`	do completeness check of a tabular expression
`%cons <const> <env>`	do consistency check of a tabular expression
`%sym <const> <env>`	do symmetry check of a two-parameter tabular expression
`%tcg <options> <const>`	produce test classes and test frames for `<const>`

Table 1: Analysis Commands

5.4 Completeness, Consistency, and Symmetry Checking

We use the same criteria as Heimdahl and Leveson [19], and Heitmeyer et al. [21] for the completeness and consistency of tabular specifications. Completeness analysis evaluates the expression consisting of the disjunction of the table's rows using SFE. After Boolean abstraction, we check if the expression is a tautology. If not, we reverse the abstraction, and use unexpansion to produce results in a column format, enumerating the cases that are not covered in the table. This presentation is possible because SFE maintains the unevaluated versions of expressions, and it addresses some of the problems identified by Heimdahl in tracing the source of inconsistencies through nested tables where the output is completely expanded [18].

A similar procedure is carried out for consistency checking, where all pairs of columns are checked for overlap.

For symmetry checking, the analysis procedure constructs two versions of a two-parameter table with the parameters swapped, and checks if the two tables are the same.

5.5 Test Generation

System-level requirements-based test generation is an analysis that makes extensive use of rewriting. The rewrite rules used were verified using HOL. The S specification is assumed to be a relation between the stimuli and responses of the system.

After unfolding the specification to the desired level of detail, the resulting formula is transformed into its logically equivalent *Test Class Normal Form* (TCNF) [10, 13]. The TCNF is a conjunction of *test classes*, which describe particular stimulus/response behaviours as implications with the stimuli in the antecedent and responses in the consequent.

The antecedents of the test classes are rewritten further to reduce the size of quantified subexpressions. Choices (disjuncts) within an antecedent represent different test descriptions, referred to as *test frames*. A test frame is a test class that has no choice in the antecedent (other than instantiation). Domain knowledge is applied to simplify the test frames, and remove any that are infeasible.

Test frames are the results of the analysis, and are logical consequences of the given specification. Test frames are selected to cover the Boolean function represented by the test class antecedent using BDDs. The selection of test frames is determined by one of several coverage criteria chosen by the user.

6 Conclusions

We have described a lightweight approach for applying automated analysis techniques to higher-order logic specifications. To support this approach we have created utilities that manipulate higher-order logic expressions in truth-preserving ways. These utilities handle the features of a logic, such as uninterpreted constants and quantification, in evaluation and abstraction.

We have demonstrated that a common core of utilities allows us to implement diverse analysis procedures such as test generation, and model checking. The common toolkit facilitates re-use of code and extension of the suite of analysis procedures with new methods such as symmetry checking and constraint-based simulation. We have also shown methods particular to

embedded notations can be created such as the completeness and consistency analysis of tables.

Two other innovations of our approach are: we allow users to control performance factors such as BDDs in terms of the language of the specification; and through the analysis process we maintain information that produces readable, traceable results in the language of the specification.

Space does not permit us to describe the real-world examples that we have specified and analyzed using our tools. Examples include an aeronautical telecommunications network (ATN) [2, 7], a separation minima for aircraft [9, 12], a small heating system [7], a steam boiler control system [13], and parts of a proprietary air traffic management system [14]. These examples are non-trivial. For example, the parameterized formal ATN statechart specification is approximately 43 pages. The expanded S representation of the ATN next state relation consists of 52 076 nodes in a canonical form.

In the future, we would like to explore how other automated abstraction techniques can be used in our framework. For example, less conservative results can be achieved by abstracting to a variant of first-order logic. We would like to explore decomposition strategies to lessen the state space explosion problem. Our approach, which uses the same specification language as a high-powered tool where these strategies can be verified, allows experts to hard code their verification method to make it accessible to non-experts.

7 Acknowledgments

References

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted–FL: A pragmatic implementation of combined model checking and theorem proving. In *TPHOLs*, number 1690 in LNCS, pages 323–340. Springer, 1999.

[2] J. H. Andrews, N. A. Day, and J. J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *FORTE/PSTV*, 1997.

[3] Myla M. Archer, Constance L. Heitmeyer, and Stever Sims. Tame: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, August 1986.

[5] William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *CAV*, volume 1254 of *LNCS*, pages 316–327, 1997.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] Nancy A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1998.

[8] Nancy A. Day and Jeffrey J. Joyce. Symbolic functional evaluation. In *TPHOLs*, volume 1690 of *LNCS*, pages 341–358. Springer, 1999.

[9] Nancy A. Day, Jeffrey J. Joyce, and Gerry Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Lfm*, pages 35–49. NASA Conference Publication 3356, September 1997.

[10] Michael R. Donat. Automating formal specification-based testing. In *TAPSOFT*, volume 1214 of *LNCS*. Springer, April 1997.

[11] Michael R. Donat. Automatically generated test frames from a Q specification of ICAO flight plan form instructions. Technical Report 98-05, Dept. of Comp. Sci, Univ. of British Columbia, April 1998.

[12] Michael R. Donat. Automatically generated test frames from an S specification of separation minima for the North Atlantic Region. Technical Report 98-04, Dept. of Comp. Sci, Univ. of British Columbia, April 1998.

[13] Michael R. Donat. *A Discipline of Specification-Based Test Derivation*. PhD thesis, Department of Computer Science, University of British Columbia, 1998.

[14] Michael R. Donat and Jeffrey J. Joyce. Applying an automated test description tool to testing based on system level requirements. In *INCOSE*, 1998.

[15] Steve Easterbrook and John Callahan. Formal methods for V & V of partial specifications: An experience report. In *RE*, pages 160–168, Annapolis, MD, 1997.

[16] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

[18] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In *ISSTA*, pages 79–83, January 1996.

[19] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. on Soft. Eng.*, 22(6):363–377, June 1996.

[20] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *CAV*, volume 1427 of *LNCS*, pages 526–531. Springer, 1998.

[21] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[22] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *CAV*, volume 697 of *LNCS*. Springer, 1993.

[23] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 285–299. Springer, 1994.

[24] J. Joyce and C-J. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *DAC*. IEEE Computer Press, 1993.

[25] Jeffrey Joyce. *Multi-Level Verification of Microprocessor Based Systems*. PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.

[26] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[27] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV*, volume 1102 of *LNCS*, 1996.

[28] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of *LNCS*, pages 748–752, 1992.

[29] P. Sreeranga Rajan. *Transformations on Data Flow Graphs: Axiomatic Specification and Efficient Mechanical Verification*. PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1995.

[30] Carl-Johan H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Dept. of Comp. Sci, Univ. of British Columbia, December 1993.

[31] N. Shankar, S. Owre, J. M. Rushby, and D.W. J. Stringer-Calvert. PVS prover guide, September 1999. Version 2.3.

[32] Natarajan Shankar. Efficiently executing PVS. Draft Final Report for NASA Contract NAS1-20334, Task 11. Computer Science Laboratory, SRI International, November 30, 1999. (also see http://pvs.csl.sri.com/experimental/eval.html).

[33] J.M. Spivey. *Understanding Z*. Cambridge University Press, Cambridge, 1988.

[34] John P. Van Tassel. A formalization of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications*, pages 359–374. North-Holland, 1993.

[35] P. J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California , Davis, 1990.

An Overview of SAL[*]

Saddek Bensalem[†]   Vijay Ganesh[‡]   Yassine Lakhnech[†]   Cesar Muñoz[§]   Sam Owre[¶]

Harald Rueß[¶]   John Rushby[¶]   Vlad Rusu[‖]   Hassen Saïdi[**]   N. Shankar[¶]

Eli Singerman[††]   Ashish Tiwari[‡‡]

Abstract

To become practical for assurance, automated formal methods must be made more scalable, automatic, and cost-effective. Such an increase in scope, scale, automation, and utility can be derived from an emphasis on a systematic separation of concerns during verification. SAL (Symbolic Analysis Laboratory) attempts to address these issues. It is a framework for combining different tools to calculate properties of concurrent systems. The heart of SAL is a language, developed in collaboration with Stanford, Berkeley, and Verimag, for specifying concurrent systems in a compositional way. Our instantiation of the SAL framework augments PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as calculate properties (i.e., perform symbolic analysis) of concurrent systems. We describe the motivation, the language, the tools, their integration in SAL/PVS, and some preliminary experience of their use.

1 Introduction

To become practical for debugging, assurance, and certification, formal methods must be made more cost-effective. Incremental improvements to individual verification techniques will not suffice. It is our basic premise that a significant advance in the effectiveness and automation of verification of concurrent systems is possible by engineering a systematic separation of concerns through a truly integrated combination of static analysis, model checking, and theorem proving techniques. A key idea is to change the perception (and implementation) of model checkers and theorem provers from tools that perform verifications to ones that calculate *properties* such as slices, abstractions and invariants. In this way, big problems are cut down to manageable size, and properties of big systems emerge from those of reduced subsystems obtained by slicing, abstraction, and composition. By iterating through several such steps, it becomes possible to incrementally accumulate properties that eventually enable computation of a substantial new property—which in turn enables accumulation of further properties. By interacting at the level of properties and abstractions, multiple analysis tools can be used to derive properties that are beyond the capabilities of any individual tool.

SAL (Symbolic Analysis Laboratory) addresses these issues. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of concurrent systems expressed as transition systems. The heart of SAL is an intermediate language, developed in collaboration with Stanford, Berkeley, and Verimag for specifying concurrent systems in a compositional way. This language will serve as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, or Verilog. The intermediate language also serves as a common description from which different analysis tools can be driven by translating the intermediate language to the input format for the tools and translating the output of these tools back to the SAL intermediate language.

This paper is structured as follows. In Section 2 we

[†]VERIMAG, Grenoble, France

[‡]Stanford University, Stanford CA

[§]ICASE, NASA Langley, Hampton VA

[¶]Computer Science Laboratory, SRI International, Menlo Park CA

[‖]IRISA, Rennes, France

[**]Systems Design Laboratory, SRI International, Menlo Park CA

[††]Intel, Haifa, Israel

[‡‡]SUNY Stony Brook, NY

describe the motivation and rationale behind the design of the SAL language and give an overview of its main features. The main part, Section 3, describes SAL components including slicing, invariant generation, abstraction, model checking, simulation, and theorem proving together with their integration into the SAL toolset. Section 4 concludes with some remarks.

2 The SAL Common Intermediate Language

Mechanized formal analysis starts from a description of the problem of interest expressed in the notation of the tool to be employed. Construction of this description often entails considerable work: first to recast the system specification from its native expression in C, Esterel, Java, SCR, UML, Verilog, or whatever, into the notation of the tool concerned, then to extract the part that is relevant to the analysis at hand, and finally to reduce it to a form that the tool can handle. If a second tool is to be employed for a different analysis, then a second description of the problem must be prepared, with considerable duplication of effort. With m source languages and n tools, we need $m*n$ translators. This situation naturally suggests use of a common intermediate language, where the numbers of tools required could be reduced to $m + n$ translators.

The intermediate language must serve as a medium for representing the state transition semantics of a system described in a source language such as Java or Esterel. It must also serve as a common representation for driving a number of back-end tools such as theorem provers and model checkers. A useful intermediate language for describing concurrent systems must attempt to preserve both the structure and meaning of the original specification while supporting a modular analysis of the transition system.

For these reasons, the SAL intermediate language is a rather rich language. In the sequel, we give an overview of the main features of the SAL type language, the expression language, the module language, and the context language. For a precise definition and semantics of the SAL language, including comparisons to related languages for expressing concurrent systems, see [31].

The type system of SAL supports basic types such as booleans, scalars, integers and integer subranges, records, arrays, and abstract datatypes. Expressions are strongly typed. The expressions consist of constants, variables, applications of Boolean, arithmetic, and bit-vector operations (bit-vectors are just arrays of Booleans), and array and record selection and updates. Conditional expressions are also part of the expression

```
mutex : CONTEXT =
BEGIN

PC: TYPE = {trying, criti-
cal, sleeping}

mutex [tval:boolean] : MODULE =
BEGIN
INPUT  pc2: PC, x2: boolean
OUTPUT pc1: PC, x1: boolean

INITIALIZATION
  TRUE -->  pc1 = sleeping;
            x1  = tval

TRANSITION
   pc1 = sleeping
      --> pc1' = trying;
            x1' = (x2=tval)
  []
   pc1 = trying AND
    (pc2=sleeping OR x1= (x2/=tval))
      --> pc1' = critical
  []
   pc1 = critical
      --> pc1' = sleeping;
            x1' = (x2=tval)
END

system: MODULE =
  HIDE x1,x2
    (mutex[FALSE]
     || RENAME pc2 TO pc1,
             x2 TO x1,
             pc1 TO pc2,
             x1 TO x2
          mutex[TRUE])

mutualExclusion: THEOREM
  system |-
     AG(NOT(pc1=critical
            AND pc2=critical))

eventually1: LEMMA
     system |- EF(pc1=critical)
eventually2: LEMMA
     system |- EF(pc2=critical)

END
```

Figure 1. Mutual Exclusion

language and user-defined functions may also be introduced.

A module is a self-contained specification of a transition system in SAL. Usually, several modules are collected in a context. Contexts also include type and constant declarations. A transition system *module* consists of a *state* type, an *initialization condition* on this state type, and a binary *transition relation* of a specific form on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The input and global variables are the *observed* variables of a module and the output, global, and local variables are the *controlled* variables of the module. It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick lookup.

Consider, for example, the SAL specification of a variant of Peterson's mutual exclusion algorithm in Figure 1. Here the state of the module consists of the controlled variables corresponding to its own program counter `pc1` and boolean variable `x1`, and the observed variables are the corresponding `pc2` and `x2` of the other process.

The transitions of a module can be specified variable-wise by means of *definitions* or transition-wise by *guarded commands*. Henceforth, primed variables `X'` denote next-state variables. A definition is of the form `X = f(Y, Z)`. Both the initializations and transitions can also be specified as guarded assignments. Each guarded command consists of a guarded formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current-state and next-state input variables. The assignment part is a list of equalities between a left-hand side next-state variable and a right hand side expression in both current-state and next-state variables.

Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. Module `mutex` in Figure 1, for example, is parametric in the Boolean `tval`. Furthermore, modules in SAL can be combined by either synchronous composition `||`, or asynchronous composition `[]`. Two instances of the `mutex` module, for example, are conjoined synchronously to form a module called `system` in Figure 1. This combination also uses *hiding* and *renaming*. Output and global variables can be made local by the `HIDE` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct.

Besides declaring new types, constants, or modules, SAL also includes constructs for stating module properties and abstractions between modules. CTL formulas

are used, for example, in Figure 1 to state safety and liveness properties about the combined module `system`.

The form of composition in SAL supports a compositional analysis in the sense that any module properties expressed in linear-time temporal logic or in the more expressive universal fragment of CTL* are preserved through composition. A similar claim holds for asynchronous composition with respect to stuttering invariant properties where a stuttering step is one where the local and output variables of the module remain unchanged.

Because SAL is an environment where theorem proving as well as model checking is available, absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods as in other languages. Consider the following definitions:

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

This pair of definitions is acceptable in SAL because we can prove that `X` is *causally dependent* on `Y` only when `A` is *true*, and vice-versa only when it is *false*—hence there is no causal loop. In general, *causality checking* generates proof obligations asserting that the conditions that can trigger a causal loop are unreachable.

3   SAL Components

SAL is built around a blackboard architecture centered around the SAL intermediate language. Different backend tools operate on system descriptions in the intermediate language to generate properties and abstractions. The core of the SAL toolset includes the usual infrastructure for parsing and type-checking. It also allows integration of translators and specialized components for computing and verifying properties of transition systems. These components are loosely coupled and communicate through well-defined interfaces. An invariant generator may expect, for example, various application specific flags and a SAL base module, and it generates a corresponding assertion in the context language together with a justification of the invariant. The SAL toolset keeps track of the dependencies between generated entities, and provides capabilities similar to proof-chain analysis in theorem proving systems like PVS.

The main ingredients of the SAL toolset are specialized components for computing and verifying properties of transition systems. Currently, we have integrated various components providing basic capabilities for analyzing SAL specifications, including

- Validation based on theorem proving, model checking, and animation;

- Abstraction and invariant generation;

- Generation of counterexamples;

- Slicing.

We describe these components in more detail below.

3.1 Backend translations

We have developed translators from the SAL intermediate language to PVS, SMV, and Java for validating SAL specifications by means of theorem proving (in PVS), model checking (in SMV), and animation (in Java). These compilers implement *shallow structural embeddings* [26] of the SAL language; that is, SAL types and expressions are given a semantics with respect to a model defined by the logic of the target language. The compilers performs a limited set of semantic checks. These checks mainly concern the use of state variables. More complex checks, as for example type checking, are left to the verification tools.

3.1.1 Theorem Proving: SAL to PVS

PVS is a specification and verification environment based on higher-order logic [27]. SAL contexts containing definitions of types, constants, and modules, are translated into PVS theories. This translation yields a semantics for SAL transition systems. Modules are translated as parametric theories containing a record type to represent the state type, a predicate over states to represent the initialization condition, and a relation over states to represent the transition relation. Figure 2 describes a typical translation of a SAL module in PVS. Notice that initializations as well as transitions may be nondeterministic.

Compositions of modules are embedded as logical operations on the transition relations of the corresponding modules: disjunction for the case of asynchronous composition, conjunction for the case of synchronous composition. Hiding and renaming operations are modeled as morphisms on the state types of the modules. Logical properties are encoded via the temporal logic of the PVS specification language.

3.1.2 Model Checking: SAL to SMV

SMV is a popular model checker with its own system description language [25]. SAL modules are mapped to SMV modules. Type and constant definitions appearing

```
module[para:Parameters] : THEORY
BEGIN
  State : TYPE = [#
    input  : InputVars,
    output : OutputVars,
    local  : LocalVars
  #]

  state,next : VAR State

  initialization(state):boolean =
    (guard_init_1 AND
     output(state) = ... AND
     local(state) = ...)
    OR ... OR (guard_init_n AND ...)

  transitions(state, next):boolean =
    (guard_trans_1 AND
     output(next) =
         output(state) WITH [...]
     local(next) =
         local(state) WITH [... ])
    OR ... OR
    (guard_trans_m AND ...)
    OR
    (NOT guard_trans_1 AND ... AND
     NOT guard_trans_m AND
     output(next) = output(state)
     local(next) = local(state))
```

Figure 2. A SAL module in PVS

in SAL contexts are directly expanded in the SMV specifications. Output and local variables are translated to variables in SMV. Input variables are encoded as parameters of SMV modules.

The nondeterministic assignment of SMV is used to capture the arbitrary choice of an enabled SAL transition. Roughly speaking, two extra variables are introduced. The first is assigned nondeterministically with a value representing a SAL transition. The guard of the transition represented by this variable is the first guard to be evaluated. The second variable loops over all transitions starting from the chosen one until it finds a transition which is enabled. This mechanism assures that every transition satisfying the guard has an equal chance to being fired in the first place. Composition of SAL modules and logical properties are directly translated via the specification language of SMV.

3.1.3 Animation: SAL to Java

Animation of SAL specifications is possible via compilation to Java. However, not all the features of the SAL

language are supported by the compiler. In particular, the expression language that is supported is limited to that of Java. For example, only integers and booleans are accepted as basic types. Elements of enumeration types are translated as constants and record types are represented by classes.

The state type of a SAL module is represented by a class containing fields for the input, output, and local variables. In order to simulate the nondeterminism of the initialization conditions, we have implemented a random function that arbitrary chooses one of the initialization transition satisfying the guard.

Each transition is translated as a Java thread class. At execution time, all the threads share the same state object. We assume that the Java virtual Machine is non-deterministic with respect to execution of threads. The main function of the Java translation creates one state object and passes the object as an argument to the thread object constructors. It then starts all the threads. Safety properties are encoded by using the exception mechanism of Java, and are checked at run time.

3.1.4 Case Study: Flight Guidance System

Mode confusion is a concern in aviation safety. It occurs when pilots get confused about the actual states of the flight deck automation. NASA Langley conducts research to formally characterize mode confusion situations in avionics systems. In particular, a prototype of a Flight Guidance System (FGS) has been selected a case study for the application of formal techniques to identify mode confusion problems. FGS has been specified in various formalisms (see [23] for a comprehensive list of related work). Based on work by Lüttgen and Carreño, we have developed a complete specification of FGS in SAL. The specification has been automatically translated to SMV and PVS, where it has been analyzed. We did not experience any significant overhead in model checking translated SAL models compared to hand-coded SMV models. This case study is available at `http://www.icase.edu./~munoz/sources.html`.

3.2 Invariant Generation

An *invariant* of a transition system is an assertion—a predicate on the state—that holds of every reachable state of the transition system. An *inductive invariant* is a assertion that holds of the initial states and is preserved by each transition of the transition system. An inductive invariant is also an invariant but not every invariant is inductive.

Let $\text{SP}(\mathcal{T}, \phi)$ denote the formula that represents the set of all states that can be reached from any state in ϕ

via a single transition of the system $\mathcal{T}$, and Θ denote the formula that denotes the initial states. A formula ϕ is an inductive invariant for the transition system $\mathcal{T}$ if (i) $\Theta \rightarrow \phi$; (ii) $\text{SP}(\mathcal{T}, \phi) \rightarrow \phi$.

We recall that for a given transition system $\mathcal{T}$ and a set of states described by formula ϕ, the notation $\text{SP}(\mathcal{T}, \phi)$ denotes the formula that characterizes all states reachable from states ϕ using exactly one transition from $\mathcal{T}$. If Θ denotes the initial state, then it follows from the definition of invariants that any fixed-point of the operator $F(\phi) = \text{SP}(\mathcal{T}, \phi) \vee \Theta$ is an invariant.

Notice that the computation of strongest postconditions introduces existentially quantified formulas. Due to novel theorem proving techniques in PVS2.3 that are based on the combination of a set of ground decision procedures and quantifier elimination we are able to effectively reason about these formulas in many interesting cases.

It is a simple observation that not only is the greatest fixed point of the above operator an invariant, but every intermediate ϕ_i generated in an iterated computation procedure of greatest fixed point also is an invariant.

$$
\begin{array}{rcl}
\phi_0 & : & \texttt{true} \\
\phi_{i+1} & : & \text{SP}(\mathcal{T}, \phi_i) \vee \Theta
\end{array}
$$

A consequence of the above observation is that we do not need to detect when we have reached a fixed point in order to output an invariant.

As a technical point about implementation of the above greatest fixed point computation in SAL, we mention that we break up the (possibly infinite) state space of the system into finitely many (disjoint) control states. Thereafter, rather than working with the global invariants ϕ_i, we work with local invariants that hold at particular control states. The iterative greatest fixed point computation can now be seen as a method of generating invariants based on *affirmation* and *propagation* [6].

Note that rather than computing the greatest fixed point, if we performed the least fixed point computation, we would get the strongest invariant for any given system. The problem with least fixed points is that their computation does not converge as easily as those of greatest fixed points. Unlike greatest fixed points, the intermediate predicates in the computation of the least fixed point are not invariants. We are currently investigating approaches based on widening to compute invariants in a convergent manner using least fixed points [8].

The techniques described so far are noncompositional since they examine all the transitions of the given system. We use a novel composition rule defined in [29] allowing local invariants of each of the modules to be composed into global invariants for the whole system.

This composition rule allows us to generate stronger invariants than the invariants generated by the techniques described in [6,7]. The generated invariants allows us to obtain boolean abstractions of the analyzed system using the incremental analysis techniques presented in [29].

3.3 Slicing

Program analyses like slicing can help remove code irrelevant to the property under consideration from the input transition system which may result in a reduced state-space, thus easing the computational needs of subsequent formal analysis efforts. Our slicing tool [18] accepts an input transition system which may be synchronously or asynchronously composed of multiple modules written in SAL and the property under verification. The property under verification is converted into a slicing criterion and the input transition system is sliced with respect to this slicing criterion. The slicing criterion is merely a set of local/output variables of a subset of the modules in the input SAL program that are not relevant to the property. The output of the slicing algorithm is another SAL program similarly composed of modules wherein irrelevant code manipulating irrelevant variables from each module has been sliced out. For every input module there will be an output module, empty or otherwise. In a nutshell the slicing algorithm does a dependency analysis of each module and computes backward transitive closure of the dependencies. This transitive closure would take into consideration only a subset of all transitions in the module. We call these transitions observable and the remaining transitions are called τ or silent transitions. We replace silent transitions with skips.

We are currently investigating reduction techniques that are simpler than slicing and also ones that are more aggressive. One example is the cone-of-influence reduction where the slicing criterion is a set of variables V, and the reduction computes a transition system that includes all the variables in the transitive closure of V given by the dependencies between variables [21]. In comparison with slicing, the cone-of-influence reduction is insensitive to control and is therefore easier to compute but generally not as efficient at pruning irrelevance. Slicing preserves program behavior with respect to the slicing criterion. One could obtain a more dramatic reduction by admitting slices that admitted more behaviors by introducing nondeterminism. Such aggressive slicing would be needed for example to abstract away from the internal behavior of a transition system within its critical section for the purpose of verifying mutual exclusion. Slicing for concurrent systems with respect to temporal properties has been investigated by

Dwyer and Hatcliff [16].

3.4 Connecting InVeSt with SAL

So far we have described specialized SAL components that provide core features for the analysis of concurrent systems, but we have also integrated the standalone InVeSt [5] into the SAL framework. Besides compositional techniques for constructing abstraction and features for generating counterexamples from failed verification attempts, InVeSt introduces alternative methods for invariant generation to SAL. InVeSt not only serves as a backend tool for SAL but also has been connected to the IF laboratory [10], Aldebaran [9], TGV [17] and Kronos [15].

The salient feature of InVeSt is that it combines the algorithmic with the deductive approaches to program verification in two different ways. First, it integrates the principles underlying the algorithmic (e.g. [11, 28]) and the deductive methods (e.g. [24]) in the sense that it uses fixed point calculation as in the algorithmic approach but also the reduction of the invariance problem to a set of first-order formulas as in the deductive approach. Second, it integrates the theorem prover PVS [27] with the model checker SMV [25] through the automatic computation of finite abstractions. That is, it provides the ability to automatically compute finite abstractions of infinite state systems which are then analyzed by SMV or, alternatively, by the model checker of PVS. Furthermore, InVeSt supports the proof of invariance properties using the method based on induction and auxiliary invariants (e.g. [24]) as well as a method based on abstraction techniques [2, 12–14, 21, 22]. InVeSt uses PVS as a backend tool and depends heavily on its theorem proving capabilities for deciding the myriad verification conditions.

3.4.1 Abstraction

InVeSt provides also a capability that computes an abstract system from a given concrete system and an abstraction function. The method underlying this technique is presented in [4]. The main features of this method is that it is automatic and compositional. It computes an abstract system $S^a = S_\alpha^1 \parallel \cdots \parallel S_\alpha^n$, for a given system $S = S^1 \parallel \cdots \parallel S^n$ and abstraction function α, such that S simulates S_α is guaranteed by the construction. Hence, by known preservation results, if S_α satisfies an invariant φ then S satisfies the invariant $\alpha^{-1}(\varphi)$. Since the produced abstract system is not given by a graph but in a programming language, one still can apply all the known methods for avoiding the state explosion problem while analyzing S_α. Moreover,

it generates an abstract system which has the same structure as the concrete one. This gives the ability to apply further abstractions and techniques to reduce the state explosion problem and facilitates the debugging of the concrete system. The computed abstract system is optionally represented in the specification language of PVS or in that of SMV.

The basic idea behind our method of computing abstractions is simple. In order to construct an abstraction of S, we construct for each concrete transition τ_c an abstract transition τ_a. To construct τ_a we proceed by elimination starting from the universal relation, which relates every abstract state to every abstract state, and eliminate pairs of abstract states in a conservative way, that is, it is guaranteed that after elimination of a pair the obtained transition is still an abstraction of τ_c. To check whether a pair (a, a') of abstract states can be eliminated we have to check that the concrete transition τ_c does not lead from any state c with $\alpha(c) = a$ to any state c' with $\alpha(c') = a'$. This amounts to proving a Hoare triple. The elimination method is in general too complex. Therefore, we combine it with three techniques that allow many fewer Hoare triples to be checked. These techniques are based on partitioning the set of abstract variables, using substitutions, and a new preservation result which allows to use the invariant to be proved during the construction process of the abstract system.

We implemented our method using the theorem prover PVS [27] to check the Hoare triples generated by the elimination method. The first-order formulas corresponding to these Hoare triples are constructed automatically and a strategy that is given by the user is applied. In [1] we developed also a general analysis methodology for *heterogeneous* infinite-state models, extended automata operating on variables which may range over several different domains, based on combining abstraction and symbolic reachability analysis.

3.4.2 Generation of Invariants

There are two different way to generate invariants in InVeSt. First, we use calculation of pre-fixed points by applying the body of the backward procedure a finite number of times and use techniques for the automatic generation of invariants (cf. [3]) to support the search for auxiliary invariants. The tool provides strategies which allow derivation of *local invariants*, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control point. InVeSt includes strategies for deriving local invariants for sequential systems as well as a composition principle that allows combination of invariants generated for sequential systems to obtain in-

variants of a composed system. Consider a composed system $S_1 \parallel S_2$ and control locations l_1 and l_2 of S_1 and S_2, respectively. Suppose that we generated the local invariants P_1 and P_2 at l_1 and l_2, respectively. Let us call P_i *interference independent*, if P_i does not contain a free variable that is written by S_j with $j \neq i$. Then, depending on whether P_i is interference independent we compose the local invariants P_1 and P_2 to obtain a local invariant at (l_1, l_2) as follows: if P_i is interference independent, then we can affirm that P_i is an invariant at (l_1, l_2) and if both P_1 and P_2 are interference dependent, then $P_1 \vee P_2$ is an invariant at (l_1, l_2). This composition principle proved to be useful in the examples we considered. However, examples showed that predicates obtained by this composition principle can become very large. Therefore, we also consider the alternative option where local invariants are not composed until they are needed in a verification condition. Thus, we assign to each component of the system two lists of local invariants. The first corresponds to interference independent local invariants and the second to interference dependent ones. Then, when a verification condition is considered, we use heuristics to determine which local invariants are useful when discharging the verification condition. A useful heuristic concerns the case when the verification condition is of the form $(pc(1) = l_1 \wedge pc(2) = l_2) \Rightarrow \phi$, where $pc(1) = l_1 \wedge pc(2) = l_2$ asserts that computation is at the local control locations l_1 and l_2. In this case, we combine the local invariants associated to l_1 and l_2 and add the result to the left hand side of the implication.

Second, we use abstraction generating invariants at the concrete level: Let $\mathcal{S}_{\alpha_1}$ the result of the abstraction of a concrete system $\mathcal{S}$, the set of reachable states denoted by $Reach(\mathcal{S}_{\alpha_1})$ is an invariant of $\mathcal{S}_{\alpha_1}$ (the strongest one including the initial configurations in fact). We developed a method that extract the formula which characterizes the reachable states from the BDD. Hence, $\alpha_1^{-1}(Reach(\mathcal{S}_{\alpha_1}))$ is an invariant of the concrete model $\mathcal{S}$. This invariant can be used to strengthen φ and show that it is an invariant of $\mathcal{S}$.

3.4.3 Analysis of Counterexamples

The generation of the abstract system is *completely automatic* and compositional as we consider transition by transition. Thus, for each concrete transition we obtain an abstract transition (which might be nondeterministic). This is a very important property of our method, since it enables the debugging of the concrete system or alternatively enhancing the abstraction function. Indeed, the constructed abstract system may not satisfy the desired property, for three possible reasons:

1. The concrete system does not satisfy the invariant,

2. The abstraction function is not suitable for proving the invariant, or

3. The proof strategies provided are too weak.

Now, a model checker such as SMV provides a trace as a counterexample, if the abstract system does not satisfy the abstract invariant. Since we have a clear correspondence between abstract and concrete transitions, we can examine the trace and find out which of the three reasons listed above is the case. In particular if the concrete system does not satisfy the invariant then we can transform the trace given by SMV to a concrete trace, thus generating a concrete counterexample.

3.5 Predicate/Boolean Abstraction

In addition to the InVeSt abstraction mechanisms, we implemented boolean abstraction of SAL specifications. We use the boolean abstraction scheme defined in [19] that uses predicates over concrete variables as abstract variables to abstract infinite or large state systems into finite state systems analyzable by model checking. The advantage of using boolean abstractions can be summarized as follows:

- Any abstraction to a finite state system can be expressed as a boolean abstraction.

- The abstract transition relation can be represented symbolically using Binary Decision Diagram (BDDs). Thus, efficient symbolic model checking [25] can be effectively applied.

- We have defined in [30] an efficient algorithm for the construction of boolean abstractions. We also designed an efficient refinement technique that allows us to refine automatically an already constructed abstraction until the property of interest is proved or a counter-example is generated.

- Abstraction followed by model checking and successive refinement is an efficient and more powerful alternative to invariant generation techniques such as the ones presented in [6, 7].

3.5.1 Automatic Construction of Boolean Abstractions

The automatic abstraction module takes as input a SAL basemodule and a set of predicates defining the boolean abstraction. Using the algorithm in [30] we automatically construct the corresponding abstract transition system. This process relies heavily on the PVS decision procedures.

```
...
INPUT   x: integer
OUTPUT  y, z: integer

INITIALIZATION
  TRUE -->  INIT(x) = 0;
            INIT(y) = 0;
            INIT(z) = y;

TRANSITION
  NOT(x > 0)  -->  y' = y + 1
  []   z > 0  -->  z' = y - 1, y' = 0
...
```

Figure 3. Concrete Module.

Figure 3 and 4 display a simple SAL module and its abstraction where the boolean variables B1, B2 and B3 correspond to the predicates $x > 0$, $y > 0$, and $z > 0$. Notice that the assignment to B3 is nondeterministically chosen from the set {TRUE, FALSE}.

```
...
INPUT  B1: boolean
OUTPUT B2,B3: boolean

INITIALIZATION
  TRUE -->  INIT(B1) = FALSE;
            INIT(B2) = FALSE;
            INIT(B3) = FALSE;

TRANSITION
    NOT(B1) --> B2'=F
 [] B3 --> B2'=T, B3'= { TRUE, FALSE }
...
```

Figure 4. Abstract Module.

3.5.2 Explicit Model Checking

Finite-state SAL modules can be translated to SMV for model checking as explained above. However, model checkers usually do not allow to access their internal data structures where intermediate computation steps of the model-checking process can be exploited. For this reason, we implemented an efficient explicit-state model checker for SAL systems obtained by boolean abstraction. The abstract SAL description is translated into an executable Lisp code that performs the explicit state model checking procedure allowing us to explore about

twenty thousand states a second. This procedure builds an abstract state graph that can be exploited for further analysis. Furthermore, additional abstractions can be applied on the fly while the abstract state graph is being built.

3.5.3 Automatic Refinement of Abstractions

When model checking fails to establish the property of interest, we use the results developed in [29, 30] to decide whether the constructed abstraction is too coarse and needs to be refined, or that the property is violated in the concrete system and that the ge nerated counter-example corresponds indeed to an execution of the concrete system violating the property. This is done by examining the generated abstract state graph. The refinement technique computes the precondition to a transition where nondeterministic assignments occur. The preconditions corresponding to the cases where the variables get either `TRUE` or `FALSE` define two predicates that are used as new abstract variables. The following transition from the example

```
B3 --> B2'=TRUE, B3'= {TRUE, FALSE}
```

can be automatically refined to

```
B3 --> B2'=TRUE, B3'=B4 ,
       B4'=FALSE, B5' = FALSE
```

where `B4` and `B5` correspond to the predicates `y=1` and `y>1`, respectively.

4 Conclusions

SAL is a tool that combines techniques from static analysis, model checking, and theorem proving in a truly integrated environment. Currently, its core is realized as an extension of the PVS system and has a well-defined interface for coupling specialized analysis tools. So far, we have been focusing on developing and connecting back-end tools for validating SAL specifications by means of animation, theorem proving, and model checking, and also for computing abstractions, slices, and invariants of SAL modules. There are as yet no automated translators into the SAL language. Primary candidates are translators for source languages such as Java, Verilog, Esterel, Statecharts, or SDL. Since SAL is an open system with well-defined interfaces, however, we hope others will write those if the rest of the system proves effective.

We are currently completing the implementation of the SAL prototype which includes a parser, typechecker, a slicer, an invariant generator, the connection to InVeSt,

and translators to SMV and PVS. We expect to release the prototype SAL system in mid-2000.

Although our experience with the combined power of several forms of mechanized formal analysis in the SAL system is still rather limited, we predict that proofs and refutations of concurrent systems that currently require significant human effort will soon become routine calculations.

References

[1] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [20], pages 146–159.

[2] S. Bensalem, A. Bouajjani, C. Loiseau, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification'92*, volume 663 of *LNCS*, pages 260–273. Springer-Verlag, 1992.

[3] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[4] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 319–331. Springer-Verlag, 1998.

[5] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 505–510. Springer-Verlag, 1998.

[6] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[7] N. Bjørner, I. A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993. Vol. 735 of *Lecture Notes in Computer Science*, Springer-Verlag.

[9] M. Bozga, J. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the Aldebaran toolset. *Software Tools and Technology Transfer journal*, 1998.

[10] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of FM'99, Toulouse, France*, LNCS, 1999.

[11] E. Clarke, E. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symp. of Prog. Lang.* ACM Press, 1983.

[12] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[13] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technical University of Eindhoven, 1996.

[14] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 (PROCOMET)*. IFIP Transactions, North-Holland/Elsevier, 1994.

[15] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. FORTE'94*, Berne, Switzerland, Oct. 1994.

[16] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Jan. 1999.

[17] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *LNCS*. Springer Verlag, 1996. Also available as INRIA Research Report RR-2987.

[18] V. Ganesh, H. Saïdi, and N. Shankar. Slicing SAL. Draft, 1999.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[20] N. Halbwachs and D. Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[21] R. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[22] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.

[23] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99)*, volume 1680 of *Lecture Notes in Computer Science*, pages 120–135, Toulouse, France, September 1999. Springer-Verlag.

[24] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[25] K. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, 1993.

[26] C. Muñoz and J. Rushby. Structural embeddings: Mechanization with method. In *Proceedings of the World Congress on Formal Methods FM 99*, volume 1708 of *LNCS*, pages 452–471, 1999.

[27] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[28] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[29] H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, Oct. 1999.

[30] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [20], pages 443–454.

[31] The SAL Group. The SAL intermediate language. Available at: http://sal.csl.sri.com/, 1999.

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE June 2000	3. REPORT TYPE AND DATES COVERED Conference Publication

4. TITLE AND SUBTITLE
Lfm2000: Fifth NASA Langley Formal Methods Workshop

5. FUNDING NUMBERS
WU 519-50-11-01

6. AUTHOR(S)
C. Michael Holloway, Compiler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-17985

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA/CP-2000-210100

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61   Distribution: Standard
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This is the proceedings of Lfm2000: Fifth NASA Langley Formal Methods Workshop. The workshop was held June 13-15, 2000, in Williamsburg, Virginia. See the web site <http://shemesh.larc.nasa.gov/lfm2000/> for complete information about the event.

14. SUBJECT TERMS
Formal Methods, Software Engineering, Proof, Model Checking, Safety

15. NUMBER OF PAGES
209

16. PRICE CODE
A10

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL