NASA/CP-2008-215309

Proceedings of the Sixth NASA Langley Formal Methods Workshop

Edited by

Kristin Yvonne Rozier
Langley Research Center, Hampton, Virginia

May 2008

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
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CP-2008-215309

Proceedings of the Sixth NASA Langley Formal Methods Workshop

Edited by

Kristin Yvonne Rozier
Langley Research Center, Hampton, Virginia

Proceedings of a workshop sponsored by
the National Aeronautics and Space Administration
and held in Newport News, Virginia
April 30-May 2, 2008

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

May 2008

Trade names and trademarks are used in this report for identification only. Their usage does not constitute an official endorsement, either expressed or implied, by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Preface

Today's verification techniques are hard-pressed to scale with the ever-increasing complexity of safety critical systems. Within the field of aeronautics alone, we find the need for verification of algorithms for separation assurance, air traffic control, auto-pilot, Unmanned Aerial Vehicles (UAVs), adaptive avionics, automated decision authority, and much more. Recent advances in formal methods have made verifying more of these problems realistic. Thus we need to continually re-assess what we can solve now and identify the next barriers to overcome. Only through an exchange of ideas between theoreticians and practitioners from academia to industry can we extend formal methods for the verification of ever more challenging problem domains.

The goal of this workshop on formal methods for verification is to examine formal verification techniques, their theory, application areas, current capabilities, and limitations. This format is designed to introduce researchers, graduate students, and partners in industry to those topics that are of fundamental interest and importance, to survey current research, and to discuss major unsolved problems and directions for future research.

This volume contains the extended abstracts of the talks presented at LFM 2008: The Sixth NASA Langley Formal Methods Workshop held on April 30 - May 2, 2008 in Newport News, Virginia, USA. The LFM Workshop series was incepted in 1990 as a local meeting centered around NASA Langley's formal methods projects. It was held sporadically in the years 1992, 1995, 1997, and 2000 and gradually expanded into an international venue for the presentation of a broad range of formal methods research topics. The topics of interest that were listed in the call for abstracts were: advances in formal verification techniques; formal models of distributed computing; planning and scheduling; automated air traffic management; fault tolerance; hybrid systems/hybrid automata; embedded systems; safety critical applications; safety cases; accident/safety analysis.

The committee decided to accept 24 submissions to be presented at the workshop and included in the proceedings. Each submitted abstract was reviewed and voted on by the entire programme committee with ties broken by the vote of the PC chair. Following the programme committee decision on each submission, one member of the PC was elected to summarize the thoughts of the entire programme committee and send this composite review to the authors. The LFM 2008 programme also includes five absolutely stellar invited talks, spanning the range of topics addressed by LFM. Gerard J. Holzmann, Amy R. Pritchett, John Rushby, Moshe Y. Vardi delivered the four keynote talks. I also invited Ricky W. Butler, the leader and founder of the Langley Formal Methods research group, to give a talk on FM research at LaRC. LFM 2008 was well-attended by a range of participants from academia, industry, and government; there were a total of 74 registered participants.

LFM 2008 is proudly sponsored by the NASA Integrated Vehicle Health Management (IVHM) and Airspace Systems Programs and by the National Institute of Aerospace (NIA). In particular, I would like to thank Brian T. Baxley, Raymond S. Calloway, Eric G. Cooper, and Michael C. Lightfoot for their advocation and financial support. I would like to thank all of the members of the programme committee for their help in composing a strong program for LFM 2008, for serving as session chairs, and for the other support and helpful suggestions they lent to ensure the workshop ran smoothly. I am grateful to Deborah L. Ford and Marie W. Hamann for procurement

services and to Charles A. "Pete" Polen for legal consultation and for helping me navigate NASA's legal framework to accomplish everything I wanted for this workshop. I would also like to thank Raymond V. Meyer for designing our logo, posters, and other artwork associated with LFM, and Lisa F. Peckham and Eric W. D. Rozier for invaluable help and advice along the way.

April 2008 Kristin Yvonne Rozier

Conference Organization

General Chair

Kristin Yvonne Rozier

Programme Committee

Ricky W. Butler
Eric G. Cooper
Benedetto L. Di Vito
Jeffrey M. Maddalon
Mahyar R. Malekpour
Paul S. Miner
César A. Muñoz
Kristin Y. Rozier
Radu L. Siminiceanu

Table of Contents

Session 1. Logic Into Practice

NASA Langley's Formal Methods Research in Support of the Next Generation Air Transportation System *(invited talk)* 3
Ricky W. Butler and César A. Muñoz

From Philosophical to Industrial Logics *(invited talk)* 6
Moshe Y. Vardi

From Informal Safety-Critical Requirements to Property-Driven Formal Validation 7
Alessandro Cimatti, Marco Roveri, Angelo Susi, Stefano Tonetta

Verification and Planning Based on Coinductive Logic Programming 9
Ajay Bansal, Richard Min, Luke Simon, Ajay Mallya, Gopal Gupta

Assessing Requirements Quality Through Requirements Coverage 12
Ajitha Rajan, Mats Heimdahl, Kurt Woodham

Session 2. Verification Under Constraints

Monitoring IVHM Systems Using a Monitor-Oriented Programming Framework 17
Sudipto Ghoshal, Solaiappan Manimaran, Grigore Rosu, Traian Florin Serbanuta, Gheorghe Stefanescu

Self-* Programming Run-Time Parallel Control Search for Reflection Box 20
Olga Brukman, Shlomi Dolev

Getting Somewhat Formal with CSP and C++ 23
William B. Gardner

Challenges and Demands on Automated Software Revision 26
Borzoo Bonakdarpour, Sandeep S. Kulkarni

Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications 29
Jorge Navas, Mario Méndez-Lojo, Manuel V. Hermenegildo

Toward a Formal Evaluation of Refactorings 33
John Paul, Nadya Kuzmina, Ruben Gamboa, James Caldwell

The Nation's Needs in Aviation Formal Methods *(invited talk)* 36
Amy R. Pritchett

Session 3. Certification and Practical Formal Methods

Formal Methods and Certification *(invited talk)* 39
John Rushby

Certifying Auto-Generated Flight Code 40
Ewen Denney

Aeronautical Regulations Should Be Rigorously Developed Too! 41
Eduardo Rafael López Ruiz, Yves Ledru, Michel Lemoine

Use of Intelligent Assistants in Practical Theorem Proving 44
David L. Barton

Combining Predicate and Numeric Abstraction for Software Model Checking 47
Arie Gurfinkel, Sagar Chaki

Session 4. Component-Based Verification

Reuse versus Reinvention: How Will Formal Methods Deal with Composable Systems? 53
Mary Ann Malloy

Automating System Assembly of Aerospace Systems 55
Panagiotis Manolios

Formal Verification of Gate-Level Computer Systems 56
Sergey Tverdyshev, Andrey Shadrin

Proving Correctness for Pointer Programs in a Verifying Compiler 59
Gregory Kulczycki, Amrinder Singh

Formal Modeling of Erroneous Human Behavior and its Implications for Model Checking 62
Matthew L. Bolton, Ellen J. Bass

A Framework for Stability Analysis of Control Systems Software at the Source Code Level 65
Fernando Alegre, Eric Feron

Mise en Scene : A Scenario-Based Medium Supporting Formal Software Development 66
John Douglas Carter

Session 5. The Future of Tools for Verification

On Limits *(invited talk)* 69
Gerard J. Holzmann

An Update on Yices 70
Bruno Dutertre

Distributing Formal Verification: The Evidential Tool Bus 71
Florent Kirchner

Model Checking for the Practical Verificationist: A User's Perspective on SAL 74
Lee Pike

An Overview of Starfish: A Table-Centric Tool for Interactive Synthesis . . 76
Alex Tsow

Session 1: Logic Into Practice

NASA Langley's Formal Methods Research in Support of the Next Generation Air Transportation System

Ricky W. Butler[1], *César A. Muñoz*[2]

[1] *NASA Langley Research Center, Hampton, Virginia 23681, USA*

[2] *National Institute of Aerospace, Hampton, Virginia 23666, USA*

R.W.Butler@nasa.gov, munoz@nianet.org

http://www.shemesh.larc.nasa.gov/fm

Extended Abstract

This talk will provide a brief introduction to the formal methods developed at NASA Langley and the National Institute for Aerospace (NIA) for air traffic management applications. NASA Langley's formal methods research supports the Interagency Joint Planning and Development Office (JPDO) effort to define and develop the 2025 Next Generation Air Transportation System (NGATS). The JPDO was created by the passage of the Vision 100 Century of Aviation Reauthorization Act in Dec 2003. The NGATS vision calls for a major transformation of the nation's air transportation system that will enable growth to 3 times the traffic of the current system. The transformation will require an unprecedented level of safety-critical automation used in complex procedural operations based on 4-dimensional (4D) trajectories that enable dynamic reconfiguration of airspace scalable to geographic and temporal demand.

The goal of our formal methods research is to provide verification methods that can be used to insure the safety of the NGATS system. Our work has focused on the safety assessment of concepts of operation and fundamental algorithms for conflict detection and resolution (CD&R) and self-spacing in the terminal area. Formal analysis of a concept of operations is a novel area of application of formal methods. Here one must establish that a system concept involving aircraft, pilots, and ground resources is safe. The formal analysis of algorithms is a more traditional endeavor. However, the formal analysis of ATM algorithms involves reasoning about the interaction of algorithmic logic and aircraft trajectories defined over an airspace. These trajectories are described using 2D and 3D vectors and are often constrained by trigonometric relations. Thus, in many cases it has been necessary to unload the full power of an advanced theorem prover. The verification challenge is to establish that the safety-critical algorithms produce valid solutions that are guaranteed to maintain separation under all possible scenarios. Current research has assumed perfect knowledge of the location of other aircraft in the vicinity so absolute guarantees are possible, but increasingly we are relaxing the assumptions to allow incomplete, inaccurate, and/or faulty information from communication sources.

The following is a list of the projects that the Langley/NIA formal methods team have been involved with:

- Airborne Information for LateralSpacing (AILS)
- CD3D and KB3D Conflict Detection and Resolution algorithms
- Runway Incursion Prevention System (RIPS)

- Small Aircraft Transportation System (SATS)
- Enhanced Oceanic Operations (EOO)
- Loss of Separation (LoS) Recovery Algorithms

In this talk we will look at three of these: SATS, KB3D, and LoS.

The goal of the SATS program was to significantly increase the capacity of regional airports. One of the most revolutionary aspects of the SATS approach is the use of a software system to sequence aircraft into the SATS airspace with no air traffic controller present. Obviously, there are serious safety issues associated with these software systems and their underlying key algorithms. A formal finite-state machine model of the SATS operational procedures using 24 transition rules was developed. This enabled an exhaustive analysis of the entire state space of the concept of operations and the proof of six safety properties. Nine issues were identified during the formal analysis. Two issues required changes to the rules of the ConOps, five issues were due to implicit or explicit omissions, and two were clarifications. All recommendations from formal methods team were adopted by SATS Conops Team.

The KB3D project developed and formally verified a new algorithm for conflict detection and resolution. The KB3D algorithm is a generalization of Karl Bilimoria's CD&R algorithm to 3 dimensions. The algorithm (KB3D) produces multiple solutions that only require a change in only one state parameter (i.e. heading, ground speed, or vertical speed). The algorithm has been formally verified to produce correct solutions when either one or both aircraft use the algorithm. KB3D is guaranteed to generate at least one valid solution for two aircraft with arbitrary trajectories. Usually the algorithm generates six different solutions. For two aircraft executing the CD&R algorithm, a proof has been completed that shows that the algorithm is implicitly coordinated. That is the algorithm produces solutions that send the two aircraft in opposite directions without any explicit communication between the aircraft. For the perfectly symmetric situation, KB3D uses a symmetry breaking mechanism. All of the proofs were accomplished using the Prototype Verification System (PVS) developed by SRI International.

Recent work at Langley has been developing a formal framework for the mathematical analysis of conflict resolution algorithms that recover from loss of separation. This work is motivated by some recent TMX simulation studies of the KB3D algorithm. The TMX studies explored the capabilities of KB3D to deal with multiple aircraft in complex traffic situations. The traffic density was approximately 3 times today's traffic and was generated by extrapolation from existing traffic patterns. There were almost no situations where a loss of separation occurred. But, it became clear to us that the KB3D algorithm should be generalized to recover from those situations. In this work we have developed a rigorous definition of correctness for vertical and horizontal maneuvers and simple criteria for loss of separation recovery algorithms that are sufficient to guarantee correctness. We have sought to make the criteria simple so that algorithms can be checked against the criteria in a straight-forward way. The criteria only uses information available to the local aircraft, but are powerful enough to prove distributed system properties. In particular, we propose rigorous definitions of horizontal and vertical maneuver correctness that yield horizontal and vertical separation, respectively, in a bounded amount of time. We also provide sufficient conditions for independent correctness, e.g., separation under the assumption that only one aircraft maneuvers, and for implicitly coordinated correctness, e.g., separation under the assumption that both aircraft maneuver. An important benefit of this approach is that different aircraft can execute different algorithms and implicit coordination will still be achieved, as long as they all meet the explicit

criteria of the framework. The mathematical framework has been formalized and mechanically verified using the Prototype Verification System (PVS) developed by SRI International.

References

[1] SATS project publications, http://research.nianet.org/fm-at-nia/SATS/

[2] KB3D project publications, http://research.nianet.org/fm-at-nia/KB3D/

[3] FM publications, http://shemesh.larc.nasa.gov/fm/fm-main-research.html

Moshe Y. Vardi: From Philosophical to Industrial Logics, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.6–6

From Philosophical to Industrial Logics

Moshe Y. Vardi

Rice University, Houston, Texas 77005, USA
vardi@cs.rice.edu

Invited Talk

One of the surprising developments in the area of program verification is how several ideas introduced by logicians in the first part of the 20th century ended up yielding at the start of the 21st century industry-standard property-specification languages called PSL and SVA. This development was enabled by the equally unlikely transformation of the mathematical machinery of automata on infinite words, introduced in the early 1960s for second-order arithmetics, into effective algorithms for industrial model-checking tools. This talk attempts to trace the tangled threads of this development.

Alessandro Cimatti et al.: From Informal Safety-Critical Requirements to Property-Driven Formal Validation, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.7–8

From Informal Safety-Critical Requirements to Property-Driven Formal Validation

Alessandro Cimatti, Marco Roveri, Angelo Susi, Stefano Tonetta

Fondazione Bruno Kessler - Istituto per la Ricerca Scientifica e Tecnologica, Trento, Italy

{cimatti,roveri,susi,tonettas}@fbk.eu

Extended Abstract

Most of the efforts in formal methods have historically been devoted to comparing a design against a set of requirements. The validation of the requirements themselves, however, has often been disregarded, and it can be considered a largely open problem, which poses several challenges.

The first challenge is given by the fact that requirements are often written in natural language, and may thus contain a high degree of ambiguity. Despite the progresses in Natural Language Processing techniques, the task of understanding a set of requirements cannot be automatized, and must be carried out by domain experts, who are typically not familiar with formal languages. Furthermore, in order to retain a direct connection with the informal requirements, the formalization cannot follow standard model-based approaches.

The second challenge lies in the formal validation of requirements. On one hand, it is not even clear which are the correctness criteria or the high-level properties that the requirements must fulfill. On the other hand, the expressivity of the language used in the formalization may go beyond the theoretical and/or practical capacity of state-of-the-art formal verification.

In order to solve these issues, we propose a new methodology that comprises of a chain of steps, each supported by a specific tool. The main steps are the following. First, the informal requirements are split into basic fragments, which are classified into categories, and dependency and generalization relationships among them are identified. Second, the fragments are modeled using a visual language such as UML. The UML diagrams are both syntactically restricted (in order to guarantee a formal semantics), and enriched with a highly controlled natural language (to allow for modeling static and temporal constraints). Third, an automatic formal analysis phase iterates over the modeled requirements, by combining several, complementary techniques: checking consistency; verifying whether the requirements entail some desirable properties; verify whether the requirements are consistent with selected scenarios; diagnosing inconsistencies by identifying inconsistent cores; identifying vacuous requirements; constructing multiple explanations by enabling the fault-tree analysis related to particular fault models; verifying whether the specification is realizable.

The methodology aims at increasing the confidence in the correctness of the requirements. On one hand, with the adoption of a property-based approach, every requirement is associated with a formal counterpart; on the other hand, a semi-formal language is exploited to narrow the gap with the natural language. The verification techniques are optimized in order to deal with large sets of requirements. The granularity of the formalization allows to focus on different types and levels of abstraction based on the hierarchy and on the modularity of the requirements; furthermore, it makes it possible to perform what-if analysis, based on hypothetical changes to the

specification; finally, the diagnostic information helps in localizing the formalization mistakes and the corresponding specification ambiguities.

This methodology has been proposed in response to the call to tender ERA/2007/ERTMS/OP/01 "Feasibility study for the formal specification of ETCS functions". The European Train Control System (ETCS) is a huge set of requirements that defines a control system to guarantee the interoperability between the European rail system and trains. Due to its complexity, ETCS presents the mentioned issues at a high level of magnitude.

Ajay Bansal et al.: Verification and Planning Based on Coinductive Logic Programming, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.9–11

Verification and Planning Based on Coinductive Logic Programming

Ajay Bansal, Richard Min, Luke Simon, Ajay Mallya, Gopal Gupta

Department of Computer Science, University of Texas at Dallas, USA
Contact author: Gopal Gupta, e-mail: `gupta@utdallas.edu`

Extended Abstract

Coinduction is a powerful technique for reasoning about unfounded sets, unbounded structures, infinite automata, and interactive computations [6]. Where induction corresponds to least fixed points semantics, coinduction corresponds to greatest fixed point semantics. Recently coinduction has been incorporated into logic programming and an elegant operational semantics developed for it [11, 12]. This operational semantics is the greatest fix point counterpart of SLD resolution (SLD resolution imparts operational semantics to least fix point based computations) and is termed co-SLD resolution. In co-SLD resolution, a predicate goal `p(`$\bar{t}$`)` succeeds if it unifies with one of its ancestor calls. In addition, rational infinite terms are allowed as arguments of predicates. Infinite terms are represented as solutions to unification equations and the occurs check is omitted during the unification process: for example, `X = [1 | X]` represents the binding of `X` to an infinite list of `1`'s. Thus, in co-SLD resolution, given a single clause

```
p([ 1 | X ]) :- p(X).
```

the query `?- p(A)` will succeed with the (infinite) answer:

```
A = [1 | A]
```

Coinductive Logic Programming (Co-LP) and Co-SLD resolution can be used to elegantly perform *model checking* and *planning*. A combined SLD and Co-SLD resolution based LP system forms the common basis for planning, scheduling, verification, model checking, and constraint solving [9, 4]. This is achieved by amalgamating SLD resolution, co-SLD resolution, and constraint logic programming [13] in a single logic programming system. Given that parallelism in logic programs can be implicitly exploited [8], complex, compute-intensive applications (planning, scheduling, model checking, etc.) can be executed in parallel on multi-core machines. Parallel execution can result in speed-ups as well as in larger instances of the problems being solved.

In the remainder we elaborate on (i) how planning can be elegantly and efficiently performed under real-time constraints, (ii) how real-time systems can be elegantly and efficiently model-checked, as well as (iii) how hybrid systems can be verified in a combined system with both co-SLD and SLD resolution. Implementations of co-SLD resolution as well as preliminary implementations of the planning and verification applications have been developed [4].

Co-LP and Model Checking: The vast majority of properties that are to be verified can be classified into *safety* properties and *liveness* properties. It is well known within model checking that safety properties can be verified by reachability analysis, i.e, if a counter-example to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Checking for reachability amounts to finding the least fixed-point, which is relatively straightforward to compute (for example, using *tabled logic programming* [2]). Verification of

liveness properties is however problematic because counterexamples to liveness properties take the form of infinite traces, which are semantically expressed as greatest fixed-points. Co-LP can be directly used to verify liveness properties by constructing counterexamples using greatest fixed-point temporal logic formulae. Intuitively, a state S is not live if there is an infinite loop (cycle) intervening between the current state and S. In a coinductive formulation, liveness also reduces to the reachability problem. Liveness counterexamples are elegantly found by (coinductively) enumerating all possible states that can be "reached" via infinite loops.

Co-LP and Planning: Coinduction can also be used to develop methods for goal-directed execution of non-monotonic logics, traditionally used for developing planners. In particular, top-down, goal-directed execution methods can be designed for *answer set programs*, a popular formalism for non-monotonic reasoning [1, 5]. Developing such a goal-directed reasoner has been an open problem for some time. It turns out that one can use co-SLD resolution to solve this problem [3]. In planning, a domain description D is given along with a set of observations about the initial state O and a collection of fluent literals $G = \{g_1, \ldots, g_l\}$, which is referred to as a goal. The problem is to find a sequence of actions $a_1, \ldots, a_n$ such that $\forall i,\ 1 \leq i \leq l$, D entails g_i from initial state O, after actions $a_1, \ldots, a_n$. The sequence of actions $a_1, \ldots, a_n$ is called a plan for goal G w.r.t. *(D,O)* [5]. Action Description Languages have been designed to encode the domain descriptions [1]. These Action Description Languages are implemented through rules of non-monotonic logic, in particular, answer set programming; these languages can be elegantly and efficiently implemented using co-LP and used for solving planning problems [4].

Timed Planning and Verification: Within logic programming, continuous time and time-deadlines can be modeled as constraints over reals [7]. Together with co-induction, this ability can be used to perform verification of timed-systems as well as perform planning under time constraints. Elsewhere we illustrate the verification of timed-systems by considering a formulation of the dining philosophers problem with stop-watches and proving that it is deadlock free (safety) and starvation free (liveness) [4]. We also illustrate the solution of time constrained planning problems using the soccer-playing planning domain extended with real-time constraints [4]. Note that a combination of generalized constraints and coinduction leads to a general framework for verifying hybrid systems as well as performing *hybrid planning*, i.e., planning under discrete and continous constraints.

References

[1] M. Balduccini, M. Gelfond, et al. An A-Prolog Decision Support System for the Space Shuttle-I. In Lecture Notes in Computer Science: Proceedings of Practical Aspects of Declarative Languages '01, Vol. 1990, pp 169-183, 2001.

[2] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. Proc. CAV'97, Haifa, Israel, Lecture Notes in Computer Science, Vol. 1243.

[3] A. Bansal, R. Min, G. Gupta. Goal-directed Execution of Answer Set Programs. UTD Technical Report. Feb. 2008.

[4] A. Bansal. Towards next generation logic programming systems. Ph.D. thesis. University of Texas at Dallas. Dec. 2007.

[5] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.

[6] J. Barwise, L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, 1996.

[7] G. Gupta, E. Pontelli. Constraint-based Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium* (RTSS'97). pp. 230-239.

[8] G. Gupta et al. Parallel Execution of Prolog Programs: A Survey. In *ACM Transactions on Programming Languages and Systems*, Vol 23, No. 4, pp. 472-602.

[9] G. Gupta, A. Bansal, R. Min, L. Simon, A. Mallya. Coinductive Logic Programming and Its Applications. Invited Tutorial. Proc. Int'l Conf. on Logic Programming (ICLP'07). pp. 27-44.

[10] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-Logic Programming. Proc. Int'l Conf. on Automata, Languages and Programming (ICALP'07), pp. 472-483.

[11] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming. Int'l Conf. on Logic Prog. (ICLP'06). Springer Verlag LNCS 4079. pp. 330-344.

[12] Luke Simon. Extending Logic Programming with Coinduction. University of Texas at Dallas. Ph.D. Thesis. 2006.

[13] K. Marriott and P. Stuckey. Prog. with Constraints: An Introduction. *MIT Press, 1998*.

Ajitha Rajan et al.: Assessing Requirements Quality Through Requirements Coverage, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.12–14

Assessing Requirements Quality Through Requirements Coverage*

Ajitha Rajan[1], *Mats Heimdahl*[1], *Kurt Woodham*[2]

[1] *University of Minnesota*

[2] *L-3 Communications*

arajan@cs.umn.edu, heimdahl@cs.umn.edu, kurt.woodham@l-3com.com

Extended Abstract

In model-based development, the development effort is centered around a formal description of the proposed software system—the "model". This model is derived from some high-level requirements describing the expected behavior of the software. For validation and verification purposes, this model can then be subjected to various types of analysis, for example, completeness and consistency analysis [6], model checking [3], theorem proving [1], and test-case generation [4, 7]. This development paradigm is making rapid inroads in certain industries, e.g., automotive, avionics, space applications, and medical technology. This shift towards model-based development naturally leads to changes in the verification and validation (V&V) process. The *model validation* problem—determining that the model accurately captures the customers' high-level requirements—has received little attention and the sufficiency of the validation activities has been largely determined through ad-hoc methods. Since the model serves as the central artifact, its correctness with respect to the users' needs is absolutely crucial. In our investigation, we attempt to answer the following two questions with respect to validation (1) *Are the requirements sufficiently defined for the system?* and (2) *How well does the model implement the behaviors specified by the requirements?* The second question can be addressed using formal verification. Nevertheless, the size and complexity of many industrial systems make formal verification infeasible even if we have a formal model and formalized requirements. Thus, presently, there is no objective way of answering these two questions. To this end, we propose an approach based on testing that—when given a set of formal requirements—explores the relationship between *requirements-based* structural test-adequacy coverage and *model-based* structural test-adequacy coverage.

The proposed technique uses requirements coverage metrics defined in [9] on formal high-level software requirements and existing model coverage metrics such as the Modified Condition and Decision Coverage (MC/DC) used when testing highly critical software in the avionics industry [8]. Our work is related to Chockler et al. [2], but we base our work on traditional testing techniques as opposed to verification techniques.

To objectively assess whether the high-level requirements have been sufficiently defined for the system, we produce a set of test cases that achieve a certain level of structural coverage of the high-level requirements, and then measure coverage achieved by the test suite over the model. If a test suite provides high requirements coverage but yields poor coverage of a model, it may be

* This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

due to one or more of the following: (a) there are missing or implicit requirements, (b) there is behavior in the model that is not derived from the requirements, or (c) the set of tests derived from the requirements was inadequate. On the other hand, to objectively assess how well the model implements the behaviors specified in the requirements, we generate a set of test cases that achieve structural coverage of the model, and then measure requirements coverage achieved. Poor requirements coverage is an indicator of either (a) the model does not adequately implement the behaviors specified in the requirements, or (b) the model is correct and the requirements are poorly written.

To illustrate the technique, we use a rigorous requirements coverage metric *Unique First Cause* (UFC) coverage defined in over requirements formalized as Linear Temporal Logic (LTL) properties [9]. We use the Modified Condition/Decision Coverage (MC/DC) criterion [5] to measure structural coverage over the model. In a preliminary study, we use five industrial case examples from the civil avionics domain. For each of these systems, we perform two kinds of assessment—(1) generate test suites to provide UFC coverage over the requirements and measure MC/DC achieved over the model, and (2) generate test suites to provide MC/DC over the model and measure UFC coverage achieved over the formal requirements. We analyze the relationship between requirements coverage and model coverage to make an assessment of the quality of the sets of requirements as well as the models.

On three of the five case examples, test suites generated to provide UFC coverage of the requirements provided reasonably good MC/DC of the models. This indicates that for these case examples, the requirements are well defined. Nevertheless, the test suites provided 10%-20% less than achievable MC/DC over the models. This is somewhat expected since the requirements (representing DO-178B high-level requirements) are typically less detailed than the model (representing DO-178B low-level requirements). Another reason may be that the UFC metric used for requirements coverage is not sufficiently rigorous and we thus have an inadequate set of requirements-based tests. On the remaining two case examples, test suites providing requirements UFC coverage gave very poor MC/DC on the model. Closer investigation revealed that on one example, there were many missing requirements. In the final case example, the requirements were good, however, their structure was so that the complexity of conditions in the requirements were hidden. For such requirements, the UFC metric that we use is not effective since the structure of the formalized requirements effectively "cheated" the UFC metric. One solution to this would be to restructure the requirements to reveal condition complexity. Another possible solution is to use a requirements coverage metric that is not as sensitive to the structure of the requirements. We hope to investigate this issue further in our future work.

We found that on all but one of the industrial systems, test suites providing MC/DC over the model achieved close to achievable requirements UFC coverage. This implies that the model exercises almost all the behaviors specified by the requirements for these systems. Nevertheless, on one model the MC/DC test suites did poorly, only achieving 30% of the achievable requirements coverage. This may either be because the model does not implement all the behaviors or the MC/DC metric is not rigorous enough. At this time we have not been able to determine the cause more closely, but we hope to do so in our future work.

To summarize, we found that analyzing the relationship between requirements coverage and model coverage provides a promising means of assessing requirements quality. Nevertheless, the effectiveness of this approach is highly dependent on the rigor and effectiveness of the coverage metrics used, and awareness of the pitfalls of structural coverage metrics is essential. For instance,

in this experiment we found that the UFC metric was surprisingly sensitive to the structure of the requirements, and one has to ensure that the requirements structure does not hide the complexity of conditions for the metric to be effective.

References

[1] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, pages 89–107, San Jose, CA, January 1999. IEEE Computer Society.

[2] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, volume 2860 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, October 2003.

[3] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[4] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[5] K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[6] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[7] A. Jefferson Offutt, Yiwie Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.

[8] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[9] Michael Whalen, Ajitha Rajan, Mats Heimdahl, and Steven Miller. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.

Session 2: Verification Under Constraints

Sudipto Ghoshal et al.: Monitoring IVHM Systems Using a Monitor-Oriented Programming Framework, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.17–19

Monitoring IVHM Systems Using a Monitor-Oriented Programming Framework

Sudipto Ghoshal[1], *Solaiappan Manimaran*[1], *Grigore Rosu*[2], *Traian Florin Serbanuta*[2], *Gheorghe Stefanescu*[2]

[1] *Qualtech Systems Inc., Wethersfield, CT, USA*

[2] *Department of Computer Science, University of Illinois at Urbana-Champaign, USA*

{sudipto,mani}@teamqsi.com, {grosu,tserban2,stefanes}@cs.uiuc.edu

Abstract

We describe a runtime verification approach to increase the safety of IVHM systems by an integration of TEAMS models and MOP (Monitor-Oriented Programming). The TEAMS model is used to automatically extract relevant runtime information from the controlled system by means of events. This information is passed on-line to the MOP engine, allowing to verify complex temporal properties and to discover running patterns which are of interest in detecting and preventing faulty behaviors.

1. Monitor-Oriented Programming (MOP)

MOP [2, 1] has its roots in a runtime verification system, PathExplorer (PAX) [6, 5], developed jointly with former NASA colleagues. PAX has found mission critical errors in NASA software. In a recent OOPSLA'07 paper [2], it was shown that the MOP framework can monitor large programs against complex parametric temporal specifications at a typically unnoticeable runtime overhead.

Many properties can be monitored in parallel in MOP. The execution trace against which the various properties are checked is extracted via automatic code instrumentation from the running program as a sequence of events – state snapshots. Events produce sufficient information about the concrete program state in order for the monitors to correctly check their properties.

In MOP, the runtime monitoring of each property consists of two orthogonal mechanisms: *observation* and *verification*. The observation mechanism extracts property-relevant and filtered system states at designated points, e.g., when property-specific events happen. The verification mechanism checks the obtained abstract trace against the property and triggers desired actions in case of violations or validations. Observation and verification are therefore independent: the algorithm used within the monitor does not affect how the execution is observed, and vice versa. MOP is a highly configurable and extensible runtime verification framework. Depending upon configuration, the monitors can be separate programs reading events from a log file, from a socket or from a buffer, or can be in-lined within the program at the event observation points.

Properties can be specified in MOP by means of *logic plugins* which essentially encapsulate and standardize monitor synthesis algorithms for various formalisms of interest. Here are several logic plugins currently provided by MOP:

— Design by Contract: A JAVA logic plugin for JASS has been implemented in MOP. JASS supports the following types of assertions: method pre-conditions and post-conditions, loop variants and invariants, and class invariants.

— Temporal Logics: Temporal logics proved to be indispensable expressive formalisms in the field of formal specification and verification of systems. Many practical safety properties can be naturally expressed in temporal logics, making them desirable specification formalisms in the MOP framework. Login plugins for both future and past time temporal logics are available.

— Extended Regular Expressions: Software engineers and programmers understand easily regular patterns, as shown by the interest in and the success of scripting languages like PERL. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is a string of states. Extended regular expressions (ERE) add complementation to regular expressions, allowing one to specify patterns that must not occur. An ERE logic plugin is available in MOP.

2. TEAMS Models

TEAMS [4] is a model-based diagnosis system. The TEAMS model of a system is a dependency model capturing relationships between failure modes of the system and their observable effects.

QSI's TEAMS Tool Set [4] consists of four software applications: TEAMS Designer, TEAMS-RT, TEAMATE and TEAMS-RDS, using a diagnostic data knowledge base called TEAMS-KB. The model is created in TEAMS Designer, or imported into TEAMS Designer from other data capture environments, and then analyzed and converted into run-time versions for export to the run-time reasoners TEAMATE and TEAMS-RT. The TEAMS Designer application provides a user-friendly graphical environment for developing dependency models of systems while allowing the specification of several additional practical aspects about the system that are required by the run-time inference engines to provide efficient diagnosis. It does so by allowing the modeler to specify cause-effect dependencies using a hierarchical, multi-layered, directed graph representation of the system. In this representation, the system's physical elements are represented as module nodes; the physical locations, where the measurements of the system's performance or other attributes are made are represented as test-point nodes; and the dependency relationships are represented as directed links.

Once a TEAMS model specification is complete, a reachability analysis can be performed in TEAMS to internally generate the dependency matrix model of the system subject to analysis constraints specified by the user. When the dependency-matrix model is available, diagnosis becomes the process of using the dependency relationships and the observed failures or anomalies to infer their possible causes. The TEAMS-RT inference engine processes failure events (exceedances, built-in test failures, performance anomalies, etc.), as they become available. It uses the data to infer the status of the root causes (the identification of one or more component faults). Thus, TEAMS-RT is appropriate for processing on-board data that is either received in real time or downloaded post-mission/operation. The TEAMATE diagnosis reasoner not only performs inference of component health status, but also computes an optimal sequence of tests that needs to be performed for fault isolation, given the current inferred health status, the allowable set of tests, and any precedence constraints on the tests. Thus, TEAMATE is appropriate for ground-based deployment where troubleshooting is performed interactively.

3. Monitoring TEAMS Specifications using MOP

We report partial work on developing a TEAMS logic plugin for MOP, which will automatically generate monitoring code from the TEAMS temporal specifications.

In a system where requirements are monitored and recovery code is executed when violations are detected, the correctness of the entire system relies only on the guarantees provided by the monitor and the recovery code. Verification of the entire system can be decomposed into checking the correctness of the monitor and of the recovery code, which are expected to be much simpler and cheaper than verifying the original program. Currently, we are working on developing an integrated framework for IVHM system monitoring, control and verification. In this framework, the TEAMS tools will be used to capture the requirements specifications of the flight system. The MOP framework, extended with a TEAMS logic plugin, will process the captured system specifications and generate monitoring code automatically. The generated monitors will check the flight system at runtime via the monitoring mechanism provided by TEAMS, steering the system if failures are detected. This way, system models and runtime verification together are expected to form a solid foundation for developing reliable aviation systems.

References

[1] F. Chen, M. D'Amorim, and G. Rosu. A formal monitoring-based framework for software development and analysis. In: *Proc. ICFEM 04*, volume 3308 of LNCS, 2004, pp. 357–373.

[2] F. Chen and G. Rosu. Mop: An efficient and generic runtime verification framework. In *Proc. OOPSLA'07*, ACM Press, 2007, pp. 569-588.

[3] F. Chen, T.F. Serbanuta and G. Rosu. jPredictor: A predictive runtime analysis tool for Java In: *Proc. ICSE'08*, to appear.

[4] S. Deb, S. Ghoshal, V. N. Malepati, and D. L. Kleinman. Tele-diagnosis: Remote monitoring of large-scale systems. In: *Proc. The IEEE Aerospace Conference*, 2000.

[5] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In: *Proc. 1st Workshop on Runtime Verification (RV 01)*, ENTCS, Vol. 55, 2001.

[6] K. Havelund and G. Rosu. Monitoring programs using rewriting. In: *Proc. International Conference on Automated Software Engineering (ASE 01)*, IEEE, 2001, pp. 135-143.

Olga Brukman et al.: Self-* Programming: Run-Time Parallel Control Search for Reflection Box, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.20–22

Self-* Programming: Run-Time Parallel Control Search for Reflection Box*

Olga Brukman, Shlomi Dolev

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{brukman, dolev}@cs.bgu.ac.il

Extended Abstract

In the early attempts to reach supersonic speeds, flight pilots experienced a strange phenomenon that made their control surfaces useless, and their aircraft uncontrollable. The airplanes were saved by either reducing the speed or changing the usual control procedure. Flying an airplane in a volcano ash cloud may stop the operation of the airplane. The airplane can still be saved if the pilots direct it out of the ash cloud, let the engines cool and then restart them. These two examples demonstrate the type of dramatic control changes that are sometimes required to be made on-line without prior experience, when the environment changes unexpectedly.

Today, when a programmer creates a program, he/she designs the program for a certain environment. When the program encounters unanticipated environmental behavior, program performance may degrade drastically, it may continue to execute while producing a faulty (unexpected) output, or it may crash. Programmers and system administrators use their accumulated knowledge of the system and of the environment to investigate and solve problems by patching up the system each time a new problem is detected. In many cases, the solution is post mortem and off-line. Ideally, systems would be autonomous, i.e., the systems would be able to cope with unexpected situations dynamically and independently, without human intervention.

In the example of the plane in the volcano ash cloud, imagine that the plane is able to release miniature replicas of itself into the air. Each replica is set to try a different control program. The replicas that manage to successfully get out of the ash cloud report back to the plane. The plane uses the obtained successful control to overcome the problem. This is an example of an autonomic system that is able to deal with unexpected changes in the environment.

Our contribution. We assume that an environment is very large, sophisticated and dynamic. We do not make any assumptions on changes the environment may undergo. On-line learning and modeling of the typically unbounded environment automaton for every change is impossible. We choose to learn a control for a *plant* only, where the plant is part of the environment with which the control interacts. A plant can be either a *black box* or an *rs-box* (reflection and set box). In case of a *black box* plant, only the plant inputs and output are observable. Otherwise, when the plant is a *rs-box* plant we are able to observe the plant state and/or set the plant to a certain state.

The environment is *reentrant* and *history oblivious* for long enough periods. A *reentrant* environment allows several copies of a plant to interact with the environment independently; A *history*

* Partially supported by the Lynne and William Frankel Center for Computer Sciences, by a Deutsche Telecom grant, the Israeli Ministry of Science, and the Rita Altura Trust Chair in Computer Sciences

oblivious environment ensures a repetition of a plant-environment interaction (in the probabilistic case with the same probability) when the plant is set to the same beginning state.

Our settings differ from the common approach where the whole environment is considered with no distinction between the machinery the control interacts with and the rest of the universe [3]. When the whole environment is considered, the environment can only be modeled by a non-deterministic infinite automaton. Making a distinction between a plant and an environment allows us to assume that a plant can be modeled by a deterministic or probabilistic finite automaton. We use testing techniques to obtain a control for the plant dynamically in an efficient manner.

The control search engine receives specifications as an input. No *realizable* specifications [1] exist for unpredictable dynamic environments. The control specifications are *unrealizable*. We assume that unrealizable specifications are potentially more abstract and short than realizable specifications, and, therefore, less prone to human mistakes. Thus, the inherit human-specifications interface is more robust than the human-program interface.

We search for a control that satisfies unrealizable specifications [1] by on line experimentation on the plant replicas. During the on line experimentation, we implicitly check whether the unrealizable specifications define *weakly realizable* specifications, given the behavior restriction on the current environment. A successful search for a control implicitly identifies the *weakly realizable* specifications, and explicitly identifies the implementation that respect the specifications.

We suggest a control search engine that finds a supervisory control dynamically and automatically by experimenting on plant replicas. The program search engine continuesly produces a control which respects a set of desired specifications in the presence of dynamic changes in an environment. In order to detect the deterioration of an executed control due to a change in the environment, the control search engine constantly monitors the current control execution by obtaining a reliable record of the control-plant-environment interaction from a dependable entity called an *Observer* and evaluating the quality of the interaction. A search for a new control is initiated if performance of the existing control is not satisfactory.

We design control search algorithms for various settings and capabilities of the plant state observation and manipulation. In particular, the plant state manipulation capabilities are: (*i*) *plant state reflection* which allows a control search algorithm to learn a connected component of a current plant state in the plant automaton graph (*ii*) *plant state set* that generalizes the reset capability, allows setting the plant to each of its states and exploring all connected components of the plant automaton graph, and (*iii*) (static or dynamic) *plant replication* capability that allows instantiation of new replicas, or use of preexisting replicas for parallelizing testing algorithms. Figure 1 summarizes the complexity of the algorithms we have designed for different settings: the total number of steps in all experiments and the length of the longest experiment (i.e., the longest execution of a plant replica). N_{max} is the upper bound on the number of plant automaton states N, Σ_{in}^{pc} is an alphabet of values of the plant input variables in the plant-control interaction, and P is the length (number of steps in a system execution) of a period in which the system repeats once a certain behavior (achieves some goal).

We consider two cases. In the first case the plant and the environment (at every given moment) are deterministic (algorithms $I - IV$). In the second case probabilistic transition functions for the environment and for the plant are considered (algorithm *Probabilistic*). See [2] for more details.

In our work, we concentrate on showing how parallelization and the capabilities for observing and manipulating the plant state allow us to improve the control search complexity. The use of parallelization makes the search time reasonable for on line systems, trading off (possibly exponential

Algorithm	**Reflection**	**Set**	**Total Number Of Steps In All Experiments**	**Longest Experiment**
I	¬Available	¬Available	$O((PN_{max})\lvert\Sigma_{in}^{pc}\rvert^{N_{max}+P+1})$	$O(PN_{max})$
II	Available	¬Available	$O(PN)$	$O(P)$
III	¬Available	Available	$O(N\lvert\Sigma_{in}^{pc}\rvert)$	$O(N)$
IV	Available	Available	$O(N\lvert\Sigma_{in}^{pc}\rvert)$	$O(1)$
Probabilistic	Available	Available	$O(N^3\lvert\Sigma_{in}^{pc}\rvert)$	$O(1)$

Figure 1. Properties of the control search algorithms.

or polynomial) time with a (exponential or polynomial) number of plant replicas. The plant state capabilities allow reduction of the number of experiments on the plant replicas from exponential (for a *black box* plant) to polynomial (for *rs*-box plant) in the number of plant automaton states.

Acknowledgments. We thank Moshe Vardi and Doron Peled for useful inputs and discussions.

References

[1] M. Abadi, L. Lamport, P. Wolper. "Realizable and Unrealizable Specifications of Reactive Systems". *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, pp. 1-17, Stresa, Italy, July 1989.

[2] O. Brukman, S. Dolev. "Self-* Programming Run-Time Reflection&Set&Replication-Box Control Synthesis". *Technical Report #08-08*, Ben-Gurion University of the Negev, Beer-Sheva, Israel, February, 2008.

[3] A. Pnueli, R. Rosner. "On the Synthesis of a Reactive Module". *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pp. 179-190, Austin, Texas, USA, January 1989.

William B. Gardner: Getting Somewhat Formal with CSP and C++, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.23–25

Getting *Somewhat* Formal with CSP and C++

William B. Gardner

Modeling & Design Automation Group, Dept. of Computing & Information Science, University of Guelph, Ontario, Canada

wgardner@cis.uoguelph.ca

1. Introduction

Concurrent systems present special design challenges due to their complex interactions, both with their environment, and internally in terms of synchronization and communication among their constituent processes. This is the case whether they are single-host systems, distributed systems, or embedded systems with hardware and software components. Formal methods have been advocated as a way to verify system properties at the design stage, but industry practitioners have not been eager to adopt abstruse mathematical notations, uncommon programming languages, or additional costly engineering process steps. Thus concurrent systems often continue to be designed and tested on an ad hoc basis.

There is a "spectrum of formality" in system development that ranges from (1) largely ad hoc vs. (2) mature, repeatable development processes at the minimal end, through (3) the use of formal specifications vs. (4) full formal development processes at the maximal end, with correspondingly greater development costs moving along the spectrum. The goal of our approach is to occupy ground between points (3) and (4): utilizing verifiable formal specifications written in CSP (Communicating Sequential Processes) for *selected portions* of a system-particularly the control backbone where interprocess synchronization and communication take place-and proceeding to an implementation via automated software synthesis instead of via hand translation. Modules to perform computation and I/O may be written in ordinary C++ and linked to the control backbone through CSP events and channels. This approach, based on the tool called CSP++, is described in [3] as a method of bridging the typically separate worlds of formal methods and conventional programming.

Advantages of this approach, compared to hand implementation of formal specifications, include code that embodies the specification's verified properties; and, compared to full formal development, reduced cost and development time due to less reliance on formal methods "gurus," and a role for widely-available C++ programmers.

2. Outline of approach

The design flow will be described, starting from tools (from Formal Systems Europe, Ltd.) used for checking CSPm specifications, exploring their state space, and formally verifying their properties, through the automatic translation step, and execution via the CSP++ object-oriented application framework (OOAF), to checking actual system traces for trace refinement against the specification. CSPm is a commonly-used machine-readable form of CSP, which our tools now support for translation [4], thus providing a "straight through" design flow from commercial

verification tools to executable code. Run-time execution is based on Gnu Pth (Portable Threads). Recently, support for operators from Timed CSP-timed prefix, timeout, and interrupt-has been added to CSP++. This approach is targeted at soft real-time systems, but not currently suitable for hard real-time, due to the inability to guarantee maximum latencies.

2.1. Technical challenges

Making a direct translation from CSP to a conventional programming language is difficult because a language like C++ lacks all the key elements of the CSP computational model: concurrent processes (or threads), interprocess communication, and interprocess synchronization. These elements are typically obtained from third-party packages, such as POSIX Pthreads, and introducing them in an informal fashion is what gets programmers into trouble. Rather than attempting a direct translation, an OO framework was constructed to be a high-level translation target with classes providing the execution semantics of CSP processes, events, and channels, including deterministic choice. (The most challenging aspect was multiprocess synchronization, which has barrier semantics, combined with choice operators active in any or all processes.) The translator tool of CSP++ inputs a user's specification written in CSPm, and outputs a C++ program that is an "instantiation" of the OOAF. When compiled with the framework's header files and linked with its library, the resulting program is an executable form of the CSPm specification.

At the above stage, the program is useful for simulating the CSP specification. It is intended to form the control backbone for an application, and, by itself, is only capable of limited interaction with a user at a console. For example, the CSP channel input operation `option?x` would prompt the user to enter an integer for channel "option" and store the result in the specification's variable x. To go beyond this stage, a programmer would prepare C++ functions to link with selected channels and events in the CSP control backbone. If the programmer links a user-coded function (UCF) to channel *option*, then the framework will call the function at run time. It will in turn obtain actual input-from a console, sensor, file, etc.-and pass it back to the framework.

UCFs must be under some restrictions in order to avoid violating the verified properties of the CSP specification: they cannot perform interprocess communication, nor maintain state between invocations that is not provided by the framework (because a UCF may be called from multiple threads). Furthermore, any given CSP event or channel can be used internally within the specification, or for linking to a UCF, but not both. Currently, UCF-linked events and channels can only participate in choice operations to a limited extent, because they do not implement the necessary try-and-backout capability. This restriction may be lifted in the future.

UCFs are intended to perform computations that would be awkward or inefficient to express in CSPm, to carry out I/O with the application's environment, or, in general, to do any processing that a designer considers not worthwhile to formally specify.

In terms of formal verification, the CSP++ tool chain is designed to leverage third-party verification tools. CSP++ is used to make an already-verified specification executable; it does not do verification itself, other than to help prove trace refinement.

2.2. Alternative approaches

The main alternative for combining CSP-style formal interprocess communication semantics with conventional programming languages is the family of xCSP libraries: JCSP [6] for Java, C++CSP for C++, CCSP for C. The programmer codes in the target language, utilizing library

objects to construct channels, launch processes, etc. Thus concurrency is provided for C and C++, while in Java, its native concurrency mechanisms are overlaid with primitives having formal semantics. This approach can be considered more readily accessible than CSP++, in that programmers need not know any CSP at all (though, in that case, they may misuse the components). To start with a verified specification, it would have to be hand-translated into, say, JCSP, with the risk of incorrect construction, or utilize some other complex steps of formal refinement [5].

3. Case studies

Several case studies created using CSP++ will be outlined: a point-of-sale register [1] developed to run on a Xilinx Microblaze core with uClinux; an automated teller [2] containing 22 UCFs with socket connections to a MySQL simulated bank database; and an automated vacuum cleaner (demonstrating new timed operators).

4. Future work

CSP++ is being enhanced for hardware/software codesign with the ability to communicate using CSP channels from the software control backbone to hardware "IP" blocks. The objective is to use CSP++ for system-on-programmable-chip applications.

References

[1] J. Carter, M. Xu, and W. B. Gardner. Rapid prototyping of embedded software using selective formalism. In *Proc. 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 99-104, Washington, DC, USA, 2005. IEEE Computer Society.

[2] S. Doxsee and W. Gardner. Synthesis of C++ Software from Verifiable CSPm Specifications. In *Proc. 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05.*, pages 193-201, 2005.

[3] W. B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302-330, 2005.

[4] W. B. Gardner. CSP++: How Faithful to CSPm? In *Proc. Communicating Process Architectures 2005 (WoTUG-27)*, pages 129-146, Eindhoven, Sept. 2005. IOS Press.

[5] A. A. McEwan. A Calculated Implementation of a Control System. In I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch, editors, *Communicating Process Architectures 2004*, pages 265-280, 2004.

[6] P. H. Welch, N. C. C. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh. Integrating and Extending JCSP. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 349-370. IOS Press, July 2007.

Borzoo Bonakdarpour et al.: Challenges and Demands on Automated Software Revision, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.26–28

Challenges and Demands on Automated Software Revision

Borzoo Bonakdarpour, Sandeep S. Kulkarni

Department of Computer Science and Engineering,
Michigan State University, East Lansing, Michigan 48824, USA
{borzoo, sandeep}@cse.msu.edu

1. Motivation

In the past three decades, automated program verification has undoubtedly been one of the most successful contributions of formal methods to software development. However, when verification of a program against a logical specification discovers bugs in the program, manual manipulation of the program is needed in order to repair it. Thus, in the face of existence of numerous unverified and uncertified legacy software in virtually any organization, tools that enable engineers to automatically verify and subsequently *fix* existing programs are highly desirable. In addition, since requirements of software systems often evolve during the software life cycle, the issue of *incomplete specification* has become a customary fact in many design and development teams. Thus, automated techniques that *revise* existing programs according to new specifications are of great assistance to designers, developers, and maintenance engineers. As a result, incorporating *program synthesis* techniques where an algorithm generates a program, that is *correct-by-construction*, seems to be a necessity.

The notion of manual program repair described above turns out to be even more complex when programs are integrated with large collections of sensors and actuators in hostile physical environments in the so-called *cyber-physical systems*. When such systems are safety/mission-critical (e.g., in avionics systems), it is essential that the system reacts to physical events such as faults, delays, signals, attacks, etc, so that the system specification is not violated. In fact, since it is impossible to anticipate all possible such physical events at design time, it is highly desirable to have automated techniques that revise programs with respect to newly identified physical events according to the system specification. Thus, one can observe that while formal software verification plays an important role in ensuring the correctness of systems, it is equally important to address the following fundamental question:

> *In the face of constant evolution of existing computing systems and their physical environment, how should we revise them according to their specification and how should we cure their vulnerabilities (e.g., failures, time unpredictability, insecurity, etc) in an incremental and automated fashion?*

2. Current Results

The notion of program revision (repair) was independently introduced by Bonakdarpour, Ebnenasir, and Kulkarni [Fmics'06, Opodis'05] and Jobstmann, Griesmayer, and Bloem [Cav'05]. In our work, we have focused on developing a theory of automated program revision from different perspectives such as time-predictability, fault-tolerance, and distribution. The main focus of this theory is to identify instances where sound and complete automated revision

of programs can be achieved in polynomial-time, and, where it is hard in some class of complexity. Complexity analysis identifies cases where program revision is (1) likely to be successful via developing efficient algorithms and heuristics, or (2) unlikely to have an impact. Completeness of a revision algorithm is important in the sense that if the algorithm fails to revise a program with respect to a property, it implies that the program in its current form is not *fixable* and, hence, a more comprehensive approach (e.g., synthesis from specification) must be applied. Thus far, the theory has been established in the following contexts:

1. We concentrated on automatic addition of untimed (respectively, real-time) UNITY properties to programs in the form of a finite state automata (respectively, timed automata) such that revised programs continue to satisfy universally quantified properties of the original program [FMICS'06, OPODIS'05].
2. We have extended the basic theory by considering systems where programs are subject to a set of uncontrollable *faults* [SSS'06]. We considered synthesizing three levels of fault-tolerance, namely *failsafe*, *nonmasking*, and *masking*, based on satisfaction of safety and liveness properties in the presence of faults. For failsafe and masking fault-tolerance, we considered two additional levels, namely *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. In our case studies, besides the factual benefits of automated addition of fault-tolerance, we observed that our synthesis methods can be potentially used to determine incompleteness of specification as well. We also introduced the notion of *bounded-time phased recovery* [FM'08] where simple recovery to the program's normal behavior is necessary, but not sufficient. For such programs, it is necessary to accomplish recovery in a sequence of phases, each ensuring certain constraints.
3. In order to make synthesis algorithms efficient so that they can be used in tools in practice, we have developed a set of symbolic heuristics for automatic synthesis of fault-tolerant distributed untimed programs [ICDCS'07]. Our experimental results on synthesis of classic fault-tolerant distributed problems showed that synthesis for these problems is feasible for state space of size 10^{30} and beyond. The tool SYCRAFT (*SYmboliC synthesizeR and Adder of Fault-Tolerance*) implements the aforementioned heuristics.

The correctness of a selection of our synthesis algorithms is verified by the theorem prover PVS [AFM'06, LOPSTR'04]. This verification essentially shows that any program synthesized by our algorithms is indeed correct-by-construction.

3. Related Work

Other well-known paradigms that have applications in program revision include *controller synthesis*, where program and fault transitions may be modeled as controllable and uncontrollable actions, and *game theory*, where program and fault transitions may be modeled in terms of two players. In controller synthesis (respectively, game theory) the objective is to restrict a *plant* (respectively, an *adversary*) at each state through synthesizing a controller (respectively, a wining strategy) such that the behavior of the entire system always meets some safety and/or reachability conditions. Note, however, that there are several distinctions. First, in addition to safety and reachability constraints, our notion of fault-tolerance is also concerned with adding new *recovery* behaviors to the given program as well, which is normally not a concern in controller synthesis

and game theory. Secondly, we model distributed systems by imposing read-write restrictions over variables of each process in a shared-memory model. Finally, rather than addressing any arbitrary specification, we concentrate on properties typically used in specifying systems.

4. Future Research Directions

A grand challenge in dealing with formal analysis of cyber-physical systems is to develop abstractions, models of computation, formal frameworks, and efficient automated techniques to specify and reason about such systems.

Formal specification of cyber-physical systems. This direction includes (1) structural specification, which models how components work and how they are interconnected, and (2) behavioral specification, which models how each component responds to an internal or external event.

Bridging the gap between specification and implementation. Another direction is to explore mechanisms for ensuring that implementation of cyber-physical systems refines their specification. To this end, one may generalize our existing synthesis/revision algorithms and tools to bridge the gap between formal specification and implementation of *multi-tolerant hybrid* cyber-physical systems.

Establishing interfaces between components operating in different contexts. As recognized by the research community, cyber-physical systems must be reliable, secure, safe, efficient, distributed, and operate in real-time. We plan to study how to express and reason about multiple (and often conflicting) concerns by considering the state of knowledge of agents in a distributed system using *epistemic logic*.

Making the developed methods scalable. The main challenge in developing verification and synthesis algorithms is scalability. Thus, we plan to accommodate *model checking techniques* in the context of program synthesis so that synthesis tools can be exploited by engineers and designers in practice.

Jorge Navas et al.: Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.29–32

Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications

Jorge Navas[1], *Mario Méndez-Lojo*[1], *Manuel V. Hermenegildo*[1,2]

[1] *Dept. of Computer Science, University of New Mexico, USA*

[2] *Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software*

jorge@cs.unm.edu, mario@cs.unm.edu, herme@fi.upm.es

1. Introduction

Many space applications such as sensor networks, on-board satellite-based platforms, on-board vehicle monitoring systems, etc. handle large amounts of data and analysis of such data is often critical for the scientific mission. Transmitting such large amounts of data to the remote control station for analysis is usually too expensive for time-critical applications. Instead, modern space applications are increasingly relying on autonomous on-board data analysis.

All these applications face many resource constraints. A key requirement is to minimize energy consumption. Several approaches have been developed for estimating the energy consumption of such applications (e.g. [3, 1]) based on measuring actual consumption at run-time for large sets of random inputs. However, this approach has the limitation that it is in general not possible to cover all possible inputs. Using formal techniques offers the potential for inferring *safe* energy consumption bounds, thus being specially interesting for space exploration and safety-critical systems.

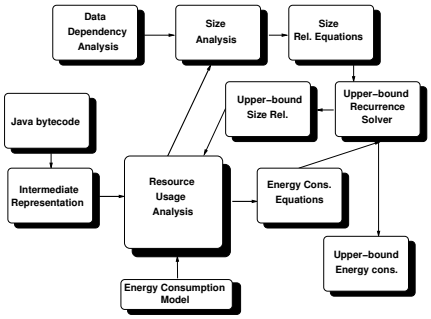


Figure 1. Energy Consumption Framework

We have proposed and implemented a general framework for resource usage analysis of Java bytecode [2]. The user defines a set of resource(s) of interest to be tracked and some annotations that describe the cost of some elementary elements of the program for those resources. These values can be constants or, more generally, *functions of the input data sizes*. The analysis then statically derives an upper bound on the amount of those resources that the program as a whole will consume or provide, also as functions of the input data sizes. This article develops a novel application of the analysis of [2] to inferring safe upper bounds on the energy consumption of Java bytecode applications. We first use a resource model that describes the cost of each bytecode instruction in terms of the joules it consumes. With this resource model, we then generate energy consumption cost relations, which are then used to infer safe upper bounds. How energy consumption for each bytecode instruction is measured is beyond the scope of this paper. Instead, this paper is about how to infer safe energy consumption estimations assuming that those energy consumption costs are provided. For concreteness, we use a simplified version of an existing resource model [1] in which an energy consumption cost for individual Java opcodes is defined.

```
import java.lang.Stream;
public class SensorNet {
 public StringBuffer collectData(
                        Sensor sensors[]){
   int i;
   int n= sensors.length;
   StringBuffer buf = new StringBuffer();
   for (i=n; i > 0;i--){
     String data = sensors[i].read();
     buf.append(data);
   }
   return buf; }
 interface Sensor { String read();}
 class TempSensor implements Sensor {
   @Cost("10*size(ret)")
   public native String read(); }
 class SeismicSensor implements Sensor {
   @Cost("20*size(ret)")
   public native String read(); }
}
```

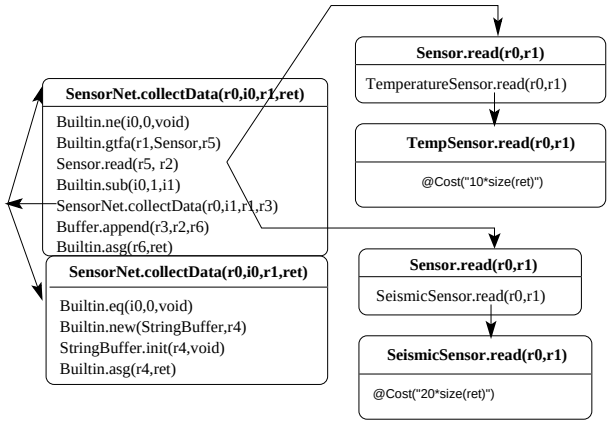


Figure 2. Motivating example (Java source code and CFG)

2. Energy Consumption Analyzer

For space reasons, we will illustrate the overall energy consumption analyzer through a working example. The Java program in Fig. 2 emulates the process of collecting data from an array of sensors within a sensor network for further processing and sending to a remote control station. For simplicity, we only show the collecting process of the sensor network. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The sensor network is implemented by class `SensorNet` and defines the method `collectData` that receives an array of sensors (`Sensor`), reads from each one the data observed, and stores it in a buffer (`StringBuffer`) for further processing. There are two types of sensors: `TempSensor`, which takes simple temperature measurements, and `SeismicSensor` which records motions of the ground. The length of the buffer which the method ultimately produces depends on the size of data measured by the sensors.

Library methods including builtins (assignment `asg`, field dereference `gtf`, method invocations `invokevirtual`, etc.) have been annotated such that our analyzer can associate energy consumption costs with them using the cost model of [1]. The objective of the analysis is then to approximate the energy consumption of the whole program. Additionally, Java programmers can define *native* methods to represent methods with absence of any callee code to analyze. In the example, the energy consumption of reading data from `TempSensor` and `SeismicSensor` sensors is proportional (10 and 20 μJ/character, respectively) to the number of characters read. This domain knowledge is reflected by the programmer in the native methods that are ultimately responsible for reading (`TempSensor.read` and `SeismicSensor.read`), by adding the annotations `@Cost("10*size(ret)")` and `@Cost("20*size(ret)")`. The rest of this section describes the different steps applied by the analyzer to approximate the energy consumption of the program depicted in Fig. 2. The main components of the framework are shown in Fig. 1.

Step 1: Constructing the Control Flow Graph. The analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception

handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 2. The `for` loop has been transformed into a recursion and the original `collectData` method has been compiled into two block methods that share the same signature: class where declared, name (`SensorNet.collectData`), and number and type of the formal parameters. The bottommost box represents the base case and the sibling corresponds to the recursive case. The virtual invocation of `read` has been transformed into a static call to a block method named `Sensor.read`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TempSensor.read` and `SeismicSensor.read`. The annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. In this example, the size of (the contents of) a variable is its value. Note that for other type of variables we have also defined different ways of measuring its size. The following equations are inferred by the analysis for the two `SensorNet.collectData` block methods:

$$size_{ret}(s_{r_0}, s_{i_0}, s_{r_1}) \leq \begin{cases} 0 & \text{if } s_{i_0} = 0 \\ 1 + size_{ret}(s_{r_0}, s_{i_0} - 1, s_{r_1}) & \text{if } s_{i_0} > 0 \end{cases}$$

The size of the returned value *ret* is independent from the sizes of the input parameter *this* (s_{r_0}) but not from the length s_{i_0} of the array *sensors* (i_0 and r_1 respectively in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. The equation system must be approximated by a recurrence solver in order to obtain a closed form solution. In this case, our analysis yields the solution $size_{ret}(s_{r_0}, s_{i_0}, s_{r1}) \leq s_{i_0}$.

Step 3: Energy Consumption Analysis. In this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps in order to infer energy consumption equations for each block method in the CFG and further simplify the resulting obtaining closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the analysis is to statically derive safe upper bounds on the energy that each of the block methods in the CFG consumes. The result given by our analysis for the energy consumption of reading the array of sensors (`SensorNet.collectData`) is

$$cost_{collectData}(s_{r_0}, s_{i_0}, s_{r_1}) \leq \begin{cases} 241 & \text{if } s_{i_0} = 0 \\ 20 \times s_{r_2} + 487 + & \text{if } s_{i_0} > 0 \\ cost_{collectData}(s_{r_0}, s_{i_0} - 1, s_{r_1}) & \end{cases}$$

i.e., the energy consumption is proportional to the length of the array of sensors (`sensors` in the source, i_0 in the CFG), and the size s_{r_2} of observed data (r_2 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $cost_{collectData}(s_{r_0}, s_{i_0}, s_{r_1})$- $\leq 20 \times s_{r_2} \times s_{i_0} + 487 \times s_{i_0} + 241$.

References

[1] S. Lafond and J. Lilius. Energy Consumption Analysis for Two Embedded Java Virtual Machines. *J. Syst. Archit.'07*.

[2] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. *TR-CS-2008-02*, UNM.

[3] C. Seo, S. Malek, and N. Medvidovic. An Energy Consumption Framework for Distributed Java-based Systems. *ASE '07*.

John Paul et al.: Toward a Formal Evaluation of Refactorings, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.33–35

Toward a Formal Evaluation of Refactorings

John Paul, Nadya Kuzmina, Ruben Gamboa, James Caldwell⋆

University of Wyoming, Laramie, Wyoming 82071-3315, USA

{jpaul, nadya, ruben, jlc}@cs.uwyc.edu

1. Introduction

Refactoring is a software development strategy that characteristically alters the syntactic structure of a program without changing its external behavior [2]. In this talk we present a methodology for extracting formal models from programs in order to evaluate how incremental refactorings affect the verifiability of their structural specifications. We envision that this same technique may be applicable to other types of properties such as those that concern the design and maintenance of safety-critical systems.

2. Formal Methodology

An object-oriented design D consists of a set of classes expressed in Java or another class-based object-oriented language. For the purposes of reasoning about D and formally comparing it to its refactorings we model D as a first-order theory of the form $\langle \Sigma, \mathcal{R} \rangle$ where Σ is a relational signature extracted from D's structural features, and $\mathcal{R}$ is a finite set of Σ-sentences expressing facts or axioms that partially capture D's class-level behavior [5]. $\mathcal{R}$ may result from the direct study of D or its documentation. We will consider the case when $\mathcal{R}$ is the output of a particular program analysis.

An additional set of Σ-sentences, $\mathcal{S}$, provides an abstract specification of what it means for D to be correct. The extent to which the set of facts $\mathcal{R}$, that we hold about D, implies $\mathcal{S}$, is indicative of how verifiable the design is by us. Any refactoring, D', of D will have the same correctness criteria as D. To compare the verifiability of the two designs we assume that there is a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ between the original and the refactored designs[1], but σ only needs to be defined on the subset of Σ used to express $\mathcal{S}$. Letting $\mathcal{R}$ and $\mathcal{R}'$ be the two sets of facts that we hold with respect to each design, we say that D' is better verifiable than D under the following conditions.

1. For every $\psi \in \mathcal{S}$, if $\mathcal{R}$ implies ψ, then $\mathcal{R}'$ implies the translated formula $\sigma(\psi)$.
2. For some $\psi \in \mathcal{S}$, $\mathcal{R}'$ implies $\sigma(\psi)$, but $\mathcal{R}$ does not imply ψ.

The first requirement merely states that D' is a behavior-preserving refactoring of D with respect to the verifiable behavior of D. While the second requirement states that we are able to verify D' more thoroughly than we are D.

⋆ This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-0613919.

[1] More precisely the notion of a *derivor*[3] can be used.

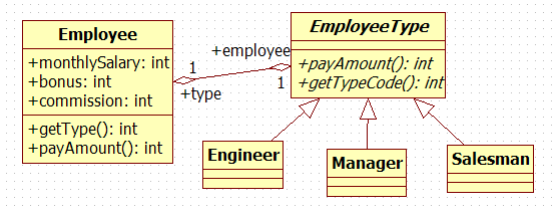
```
public class Employee {
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    private int type;
    private int monthlySalary;
    private int commission;
    private int bonus;

    ...

    public int payAmount() {
        switch (type) {
        case ENGINEER:
            return monthlySalary;
        case SALESMAN:
            return monthlySalary + commission;
        case MANAGER:
            return monthlySalary + bonus;
        }
    }
}
```

Initial Design D



Refactored Design D′

Figure 1. Initial design D and refactored design D' of the *Employee* class.

Alloy [4] is a relational language based on first-order logic that allows us to express theories about designs. When combined with the Alloy Analyzer it offers a practical way to implement our methodology and to check 1) and 2) in practice. Once $\langle \Sigma, \mathcal{R} \rangle$ is presented as an Alloy theory, the Alloy Analyzer enumerates its models up to a user specified depth and reports any counterexample that it finds for a particular $\psi \in \mathcal{S}$, each of which is encoded as an Alloy assertion. The same is done for $\langle \Sigma', \mathcal{R}' \rangle$ and each $\sigma(\psi)$. While not a proof, the absence of a counterexample serves as evidence that an assertion may be valid within a theory and hence verifiable about a particular design.

3. Implementation

Next, we describe how we used Alloy and two different automatic constraint detectors to apply this methodology to the *Employee* example of the 'replace conditional with state' refactoring from Fowler's book [2]. Figure 1 presents the initial design D as a single class, `Employee`, providing a `payAmount` method which uses a `switch`-statement to compute the monthly earnings of an employee based on his or her occupation (`type`). In the refactored design D', however, `Employee` delegates the earnings computation to a polymorphic state object, `type`, which is no longer just a simple `int`, and the computation is now distributed over three different kinds of `EmployeeType` objects. The right side of Figure 1 depicts this design.

The extraction of Σ and Σ' is based on the class structures of D and D' and is virtually automatic. Each class or datatype is presented as its own disjoint set of atoms, while attributes and methods are presented as relations on these sets. The signature morphism $\sigma : \Sigma \to \Sigma'$ must be constructed by hand. In our example, σ is the identity mapping for all sorts and relations except for `type`. The assertions standing for the specification $\mathcal{S}$ must also be constructed by hand. For instance, one assertion about the `payAmount` method is that an *Engineer*'s monthly earnings are

equal to his or her salary.

Finally, $\mathcal{R}$ and $\mathcal{R}'$ consist of the machine translated invariants output by one of the two program analysis tools, Daikon [1] or ContExt. One fact that Daikon recovered from the design D' states that the `payAmount` method for the `Engineer` class returns the `monthlySalary` of its `Employee` attribute. After the two respective Alloy theories, $\langle \Sigma, \mathcal{R} \rangle$ and $\langle \Sigma', \mathcal{R}' \rangle$ have been created for designs D and D' by either Daikon or ContExt, we use the Alloy Analyzer to check whether the set of assertions that hold in the refactored theory is a proper superset of the assertions that hold in the unrefactored theory. If so, then we conclude that there is evidence to suggest that D' may be better verifiable than D in light of the facts obtainable by either tool.

In this example the Alloy analyzer considered all possible models consisting of two `Employee` entities, two `EmployeeType` entities and each `int` from -16 to 15. Our study suggests that the verifiability of the refactored design improves with respect to the facts obtained by Daikon, while the verifiability of either design is sufficiently good in light of the facts obtained by ConText.

4. Conclusions

Insofar as some structural properties of programs *are* safety-critical, the methodology presented here already applies to them. For instance, a specification for a controller may contain a safety-critical class invariant that states which configurations are reachable. Our methodology allows a way to monitor the verifiability of such properties as refactorings are applied throughout the software lifecycle. More investigation is needed to evaluate our approach on a real world example.

References

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] J. A. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4. Prentice Hall, 1978.

[4] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[5] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In *Proceedings of Foundations of Software Egineering*, pages 207–216, Lisbon, Portugal, 2005.

Amy R. Pritchett: The Nation's Needs in Aviation Formal Methods, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.36–36

The Nation's Needs in Aviation Formal Methods

Amy R. Pritchett

NASA Headquarters, Washington, DC, USA
Amy.R.Pritchett@nasa.gov

Invited Talk

Aviation – both on-board systems and the National Airspace System – can be transformed by many current and future technical capabilities. To name but a few, improved efficiency may be achieved by integrating functions; robustness may be improved by distributing functions; and safety may be improved by building in risk mitigation through not only redundant, independent systems but also through operational concepts and effective interaction with human operators. This talk will review the key aspects of verification, validation and certification for which formal methods will provide a critical function in enabling truly revolutionary designs to enter the operational community, illustrating successes in formal modeling to date and posing further questions for the formal modeling community.

Session 3: Certification and Practical Formal Methods

John Rushby: Formal Methods and Certification, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.39–39

Formal Methods and Certification

John Rushby

SRI International, Menlo Park, California 94025, USA
rushby@csl.sri.com

Invited Talk

The great strength of formal methods is that they allow consideration of all possible behaviors, and this could be of immense value in certification. However, automated formal analysis of large and complex artifacts is computationally infeasible, so compromises have to be made. These include use of interactive human guidance rather than full automation, analysis of models and abstractions rather than the real artifact, and analysis of weak properties (e.g., by static analysis) rather than full requirements. I will consider how these and other practical limitations affect the potential role of formal methods in certification. I will also outline weaknesses in current standards-based approaches to certification and sketch how multi-legged safety cases might provide a way to incorporate formal methods into certification processes.

Ewen Denney: Certifying Auto-Generated Flight Code, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.40–40

Certifying Auto-Generated Flight Code

Ewen Denney

RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
Ewen.W.Denney@nasa.gov

Model-based design and automated code generation are being used increasingly at NASA. Many NASA projects now use MathWorks Simulink and Real-Time Workshop for at least some of their modeling and code development. However, there are substantial obstacles to more widespread adoption of code generators in safety-critical domains. Since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified. Moreover, the regeneration of code can require complete recertification, which offsets many of the advantages of using a generator. Indeed, manual review of autocode can be more challenging than for hand-written code. Since the direct V&V of code generators is too laborious and complicated due to their complex (and often proprietary) nature, we have developed a generator plug-in to support the certification of the auto-generated code. Specifically, the AutoCert tool supports certification by formally verifying that the generated code is free of different safety violations, by constructing an independently verifiable certificate, and by explaining its analysis in a textual form suitable for code reviews. The generated documentation also contains substantial tracing information, allowing users to trace between model, code, documentation, and V&V artifacts. This enables missions to obtain assurance about the safety and reliability of the code without excessive manual V&V effort and, as a consequence, eases the acceptance of code generators in safety-critical contexts. The generation of explicit certificates and textual reports is particularly well-suited to supporting independent V&V. The primary contribution of this approach is the combination of human-friendly documentation with formal analysis.

The key technical idea is to exploit the idiomatic nature of auto-generated code in order to automatically infer logical annotations. The annotation inference algorithm itself is generic, and parametrized with respect to a library of coding patterns that depend on the safety policies and the code generator. The patterns characterize the notions of definitions and uses that are specific to the given safety property. For example, for initialization safety, definitions correspond to variable initializations while uses are statements which read a variable, whereas for array bounds safety, definitions are the array declarations, while uses are statements which access an array variable. The inferred annotations are thus highly dependent on the actual program and the properties being proven.

The annotations, themselves, need not be trusted, but are crucial to obtain the automatic formal verification of the safety properties without requiring access to the internals of the code generator.

The approach has been applied to both in-house and commercial code generators, but is independent of the particular generator used. It is currently being adapted to flight code generated using MathWorks Real-Time Workshop, an automatic code generator that translates from Simulink/Stateflow models into embedded C code.

Eduardo R. López Ruiz et al.: Aeronautical Regulations Should Be Rigorously Developed Too!, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.41–43

Aeronautical Regulations Should Be Rigorously Developed Too!

Eduardo R. López Ruiz[1], *Yves Ledru*[2], *Michel Lemoine*[1]

[1] *ONERA, Toulouse, France*

[2] *Laboratoire d'Informatique de Grenoble, Saint Martin d'Hères, France*

eduardo.lopez-ruiz@onera.fr, yves.ledru@imag.fr, michel.lemoine@onera.fr

Abstract

We propose the use of formal techniques, complementary to semi-formal models, to improve the contemporary rulemaking process that is used to develop aeronautical safety and security regulations. The two main contributions of this approach are: (1) the use of rigorous methods and tools to help improve the regulation's validation process, and (2) the capacity to help identify the impact of proposed amendments on enacting regulation (while helping mitigate regressions).

1. Introduction

New paradigms in civil aviation (e.g. the reduction or even the suppression of flight crew) require the use of formal development techniques for on-board software. But these advances in civil aviation also require evolutions of the related regulations. In this paper we argue for the use of similar formal techniques to design these evolutions of the regulation.

Indeed, regulations are natural language documents, and hence inherently ambiguous. The act of translating them into a semiformal model would help reduce the use of innately ambiguous terms. On top of that, associating their trickiest parts to formal versions of them will help provide a sound basis for their detailed understanding, analysis and implementation.

Furthermore, regulations are rarely built from scratch. New regulations are evolutions of existing ones, and it often makes sense to study their non-regression. Formal models allow the use of animation and proof techniques to compare successive versions of the regulation and detect regressions.

Formal methods have already been used within the context of modeling regulations in order to ease their validation and verification [1]. They have also been used within other domains of aeronautics, "to enhance the current practice of procedures development" [2] and for the analysis and verification of aeronautical safety critical systems. What's more, EDEMOI [3] proposed the mixed implementation of semi-formal (UML models) and rigorous methods (formal models using the B and Z notations [4]) to assist in the specification, design and validation of aviation security regulations. So, building upon the *positive results of the EDEMOI experience, we proposed a twofold expansion of the methodology* (See Figure 1) *by (1) broadening its scope to include aviation safety regulations and (2) by extending its usability throughout the regulation's "lifecycle".*

2. The Chosen Approach

As was done in our appraisal [5] of EU Directive 2320/2002, an interpretation of the regulation is captured using (part of) the UML language *(Figure 1, Step 2.a)*. The use of this graphical

notation helps tackle the text's innate ambiguity, while proposing a conceptual layout that can be validated/invalidated by officials from the certification authority.

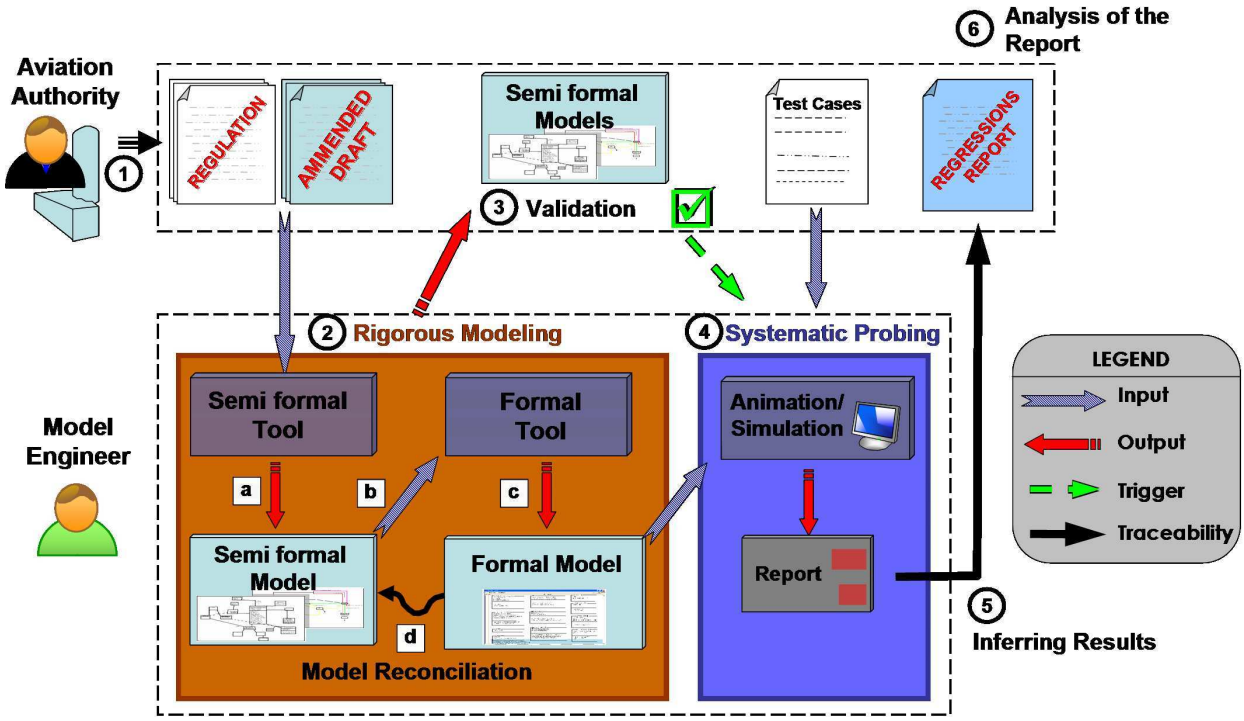


Figure 1. The extended EDEMOI methodology

The validation of this conceptual layout *(Figure 1, Step 3)* helps establish the adhesion of the semiformal model to the convened international standard. For this, the appraisal and feedbacks provided by aviation authority officials are integrated into the model by way of amendments. Once validated, the semiformal model is translated *(Figure 1, Step 2.b)* to a rigorous formal model using translation rules between the semiformal and formal notations (UML → Z [6]).

This ensures that our methodology benefits fully from the integration of both approaches: the intuitive structured notation of the semiformal approach and the precise semantics of the formal approach.

Finally, when the models have been deemed mature enough (both in their notation and their faithfulness to the regulation) an animation or verification tool *(Figure 1, Step 4)* is used to test the formal model's consistency (through simulation) and robustness (through counterexample checking). The results of the tests and simulation are stored to enable regression analysis after further evolutions of the regulation and the models *(Figure 1, Step 5)*. Currently, two formal method targets are being considered: RoZ + Jaza Animator [4] and Alloy Analyzer [7].

3. Appraising the Methodology

The introduction of new paradigms in civil aviation entails the need to adapt the regulatory framework dictating the behavior and the interactions of airplanes. For this reason, the stakeholders to this undertaking -such as the aircraft manufacturers, service providers and safety regulators- are concerned with determining the regulatory enhancements that will be required to ensure safe

operations under new states of affairs.

Rigorous formal models of the affected aeronautical regulations will help (1) identify the scope of impact that the flight-crew reduction would have on the regulations, and (2) determine if the proposed regulatory evolutions will have the desired effect.

References

[1] International Workshop on Regulations Modelling and their Validation & Verification (REMO2V'2006), http://lacl.univ-paris12.fr//REMO2V/

[2] Degani, A, Heymann, M. and Barshi, I.: A formal methodology, tools and algorithm for the analysis, verification and design of emergency procedures and recovery sequences. NASA Internal white paper. (2005)

[3] Laleau, R. et al. Adopting a situational requirements engineering approach for the analysis of civil aviation security standards. The Journal of Software Process: Improvement and Practice (SPIP), Vol. 11, Issue 5, Pages 487-503. July 2006. DOI= http://dx.doi.org/10.1002/spip.291

[4] Ledru, Y. Using Jaza to animate RoZ specifications of UML class diagrams. In Proceedings of the 16th International Z User Meeting. IEEE. (Columbia, USA,April 24-28, 2006). ZUM 2006. IEEE Press. DOI= http://doi.acm.org/10.1109/SEW.2006.39

[5] López Ruiz, E.R. Formal Specification of Security Regulations: The Modeling of European Civil Aviation Security. Master Thesis. SUPAERO. Toulouse, France. 2006.

[6] Dupuy, S., Ledru, Y. and Chabre-Peccoud, M.: An overview of RoZ : A Tool for Integrating UML and Z. In: 12th Conference on Advanced Information Systems Engineering (Stockholm, Sweden, June 05-09, 2000). CAiSE 2000. Springer Press. DOI= http://doi.acm.org/10.1007/s10515-006-0273-5

[7] Jackson, D. Dependable Software by Design. In Scientific American. (June, 2006). Volume 294, Number 6. Page 68-75.

David L. Barton: Use of Intelligent Assistants in Practical Theorem Proving, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.44–46

Use of Intelligent Assistants in Practical Theorem Proving

David L. Barton

EDAptive Computing, Inc., Dayton, Ohio 45458, USA
d.barton@edaptive.com
http://www.edaptive.com

Extended Abstract

At present, the greatest barrier to the full scale adoption of theorem proving in industrial practice is its intractability to the average engineering user. Model checking can be taught, or at least hidden, by fairly understandable tools. Theorem proving is another matter; intensive and time consuming training is needed before the average user can approach the use of theorem proving tools. Worse, what might be termed the "style of thinking" of models that are useful for theorem proving is fundamentally different from models useful in most engineering disciplines. This requires either extensive translation or a different approach to most engineering problems, a different approach that has the effect of divorcing the verification engineer from the development engineer. Such a separation is never desirable.

By far the most common tool used by engineers today is MATLAB, and its graphical interface Simulink. Theorem provers that can work with MATLAB and Simulink would be most desirable; however, MATLAB and Simulink are designed for simulation and mathematical execution in ways that are fundamentally different from the kinds of manipulation required by, say, PVS. What is needed is a means of moving problems from the engineering space into the theorem proving space, proving properties of interest, and taking not only the fact of the proof, but information from the proof back into the engineering space so that it can be used appropriately.

The mathematics of theorem proving and logic are such that this problem is impossible to automate generally (or if not impossible, then beyond current technology). However, for a narrow class of similar problems, automation can be provided by the use of *proof assistants* that consist of lemmas, proof strategies, and a friendly graphical interface. The production of these lemmas, strategies, and interfaces (hereafter grouped together and called "assistants" or "the assistant") is complex; the use of the assistant is simple, and brings theorem proving within that narrow range of problems into the grasp of working engineers.

Figure 1 represents the entire process of shifting from the engineering space to the theorem proving space, using the theorem proving, and shifting back. A MATLAB / Simulink representation is translated into an internal representation which allows both graphical access (using Syscape^{TM}, a proprietary tool to EDAptive Computing, Inc.) and theorem proving via PVS. The automated assistance is provided via a series of PVS lemmas and proof strategies which have been provided in the field of control law development; in fact, a subset of control laws which were of interest to the customer in this case and the project, the Automated Aerial Refueling project. Access to the lemmas and strategies was given via Syscape^{TM}, which has extensive customization capabilities as seen in step 4. This allows the engineer to operate the theorem prover as a tool from the Syscape^{TM} interface with a limited set of choices. If these choices fail, the engineer's recourse is to a consultant or theorem proving expert.

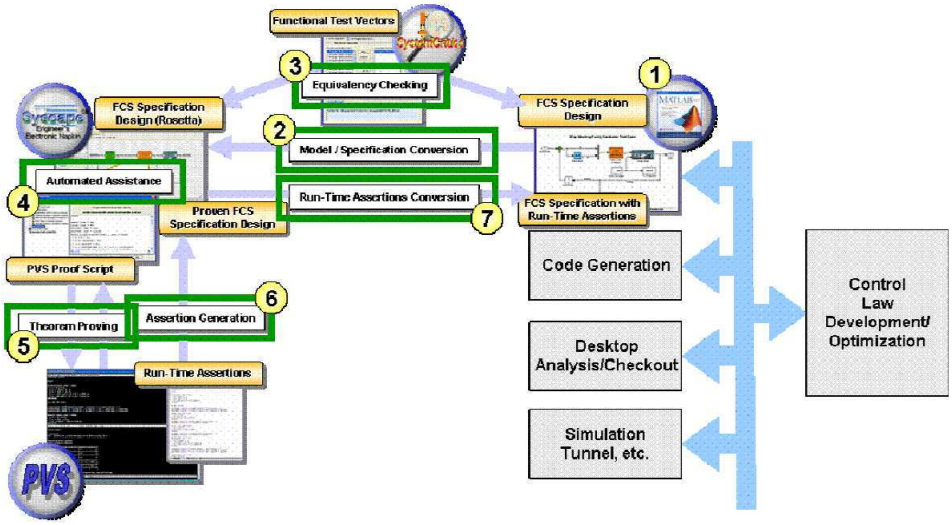


Figure 1: Shifting from MATLAB to Theorem Proving and back again.

The specific lemmas generated by PVS during the proof process are turned into run time assertions, as shown in step 6 of the figure. These are turned into executable assertions which can be inserted into the MATLAB code, as shown in the *return* code loop shown in step 7 of the figure. This can be simulated in MATLAB / Simulink and, if desired, inserted into the code for code generation as a part of the code of the control law as part of the normal code generation process from Simulink to C or C++.

Because the code has been proven correct, the assertions do not test for correctness during execution. Rather, the assertions test for those conditions which the proofs assume are true in order for the proofs to be valid. If equipment fails or the environmental conditions change, the assertions will be violated and an error condition will be flagged before a catastrophic reaction can take place. In the case of the AAR, an operator on board the refueling aircraft can take action to avoid collision or loss of life.

The automated assistant in this case provides very little "artificial intelligence" compared to some in the literature. In to Figure 2, for the AAR, the pre-prepared lemmas consisted of 589 that were generated by the replacement, many of which were repetitive. The templates were prepared beforehand, so that when a user "drags and drops" a gain block into his diagram, the lemmas *for that block* are generated from the template and inserted into the proof script for the system. Only certain configurations are supported for overall proof.

Lemma Templates

System	Lemmas	Breakdown
Longitudinal FCS	126	Constant, Discrete Time Integrator, Gain, Product, Sum, Zero-Order Hold
Lateral FCS	130	Discrete Time Integrator, Gain, Sum, Zero-Order Hold
Axial FCS	48	Gain, Sum, Transfer Function, Zero-Order Hold
Directional FCS	130	Discrete Time Integrator, Gain, Sum, Zero-Order Hold
Mixer	155	Gain, Sum, Zero-Order Hold
FCS (Total)	589	+Bus Creators

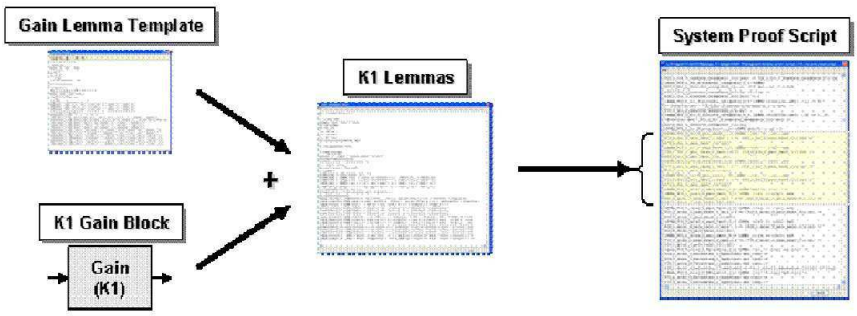


Figure 2. Lemmas for the AAR example.

In summary, the translation and back translation of the MATLAB model gives the engineering user access to the power of theorem proving in a narrow field. Moving outside this field requires a theorem proving expert to extend the assistant, which is a complex task.

Arie Gurfinkel et al.: Combining Predicate and Numeric Abstraction for Software Model Checking, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.47–49

Combining Predicate and Numeric Abstraction for Software Model Checking*

Arie Gurfinkel, Sagar Chaki

Software Engineering Institute, Carnegie Mellon University, USA

{arie, chaki}@sei.cmu.edu

Extended Abstract

Predicate abstraction [6] (PA) and Abstract Interpretation [4] (AI) with numeric abstract domains, called Numeric abstraction (NA), are two mainstream techniques for automatic program verification. Although it is sometimes assumed that the difference between the two is that of precision versus efficiency, experience of projects based on PA (such as SLAM [1]) and those based on NA (such as ASTRÉE [3]) indicate that both techniques can balance efficiency and precision when applied to problems in a particular domain. However, the two techniques have complimentary strengths and weaknesses.

Predicate abstraction reduces program verification to propositional reasoning via an automated decision procedure, and then uses a model checker for analysis. This makes PA well-suited for verifying programs and properties that are control driven and (mostly) data-independent. An example of such a program is the code fragment in Fig. 1(a). However, in the worst case, reduction to propositional reasoning is exponential in the number of predicates. Hence, PA is not as effective for data-driven and (mostly) control-independent programs and properties, such as the code fragment shown in Fig. 1(b) In summary, PA is works best for propositional reasoning, and performs poorly for arithmetic.

On the other hand, Numeric abstraction restricts all reasoning to conjunction of linear constraints. For instance, NA with Intervals is limited to conjunctions of inequalities of the form $c_1 \leq x \leq c_2$, where x is a variable and c_1, and c_2 are constants. Instead of relying on a general-purpose decision procedure, NA leverages a special data structure – Numeric Abstract Domain. The data structure is designed to represent and manipulate sets of numeric constraints efficiently; and provides algorithms to encode statements as transformers of numeric constraints. Thus, in contrast to PA, NA is appropriate for verifying properties that are (mostly) control-independent, but require arithmetic reasoning. One example of such a program is the code fragment in Fig. 1(b). On the flip side, NA performs poorly when propositional reasoning (i.e., precisely representing disjunctions and negations) is required. For example, the code fragment in Fig. 1(a) is hard for NA.

In practice, precise, efficient, and scalable program analysis requires the strengths of both predicate and numeric abstraction. Consider the problem of verifying the code fragment in Fig. 1(c). In this case, propositional reasoning is needed to distinguish between different program paths, and arithmetic reasoning is needed to efficiently compute strong enough invariant to discharge the assertion. More importantly, the propositional and numeric reasoning must interact in non-trivial

* An article reporting on this research is currently under submission to a conference.

```
assume(i==1 || i==2);
switch(i)
 case 1: a1=3; break;
 case 2: a2=-4; break;
switch (i)
 case 1: assert(a1>0);
 case 2: assert(a2<0);
 default: assert(0);
```

(a)

```
if(3 <= y1 <= 4)
  x1 = y1 - 2;
  x2 = y2 + 2;
else if(3 <= y2 <= 4)
  x1 = y2 - 2;
  x2 = y2 + 2;
assert(5 <= (x1+x2) <= 10);
```

(b)

```
assume(x1==x2);
if (A[y1 + y2] == 3)
  x1 = y1 - 2;
  x2 = y2 + 2;
else
  A[x1 + x2] = 5;
if (A [x1 + x2] == 3)
  x1 = x1 + x2;
  x2 = x2 + y1;
assert(x1==x2);
```

(c)

$assume(x_1 = x_2);$
$((assume(p);$
$\quad x_1 := y_1 - 2 \wedge q := choice(\mathtt{f}, \mathtt{f});$
$\quad x_2 := y_2 + 2 \wedge q := choice(x_1 + 2 = y_1 \wedge p, \mathtt{f})) \vee$
$(assume(\neg p);$
$\quad q := choice(\mathtt{f}, \mathtt{t})));$
$((assume(q);$
$\quad x_1 := x_1 + x_2;$
$\quad x_2 := x_2 + y_1) \vee assume(\neg q));$
$assert(x_1 = x_2)$

(d)

Figure 1. Example programs (a), (b), (c). Part (d) is an abstraction of (c) with $V_P = \{p, q\}$, $V_N = \{x_1, x_2, y_1, y_2\}$, where $p \triangleq ((A[y_1 + y_2] = 3)$, and $q \triangleq (A[x_1 + x_2] = 3)$.

ways. Therefore, a combination of PA and NA is more powerful and efficient than either technique alone.

Any meaningful combination of PA and NA must have at least two features: (a) propositional predicates are interpreted as numeric constraints where appropriate, and (b) abstract transfer functions respect the numeric nature of predicates. The first requirement means that, unlike most AI-based combinations, the combined abstract domain cannot treat predicates as uninterpreted Boolean variables. The second requirement implies that the combination must support abstract transformers that allow the numeric information to affect the update of the predicate information, and vice versa.

Against this background we make the following contributions. We present the interface of an abstract domain, called NumPredDom, that combines both PA and NA, and supports a rich set of abstract transfer functions that enables the updates of numeric and predicate state information to be influenced by each other. For example, an NumPredDom-based abstraction of the code fragment in Fig. 1(c) is shown in Fig. 1(d). Here, two predicates are used to relate reasoning about conditions in control-flow statements with reasoning about numeric variables. Note that the value of the predicates depends on values of numeric constraints.

We propose four data-structures — NEXPoint, NEX, MTNDD and NDD — that implement NumPredDom. The data structures (summarized in Table 1) differ in their expressiveness and in the choice of representation for the numeric part of the domain. Our target is PA-based software analysis. Thus, all of the data-structures use BDDs for efficient (symbolic) propositional reasoning.

Related work. A typical way to combine PA and NA in AI is to use a direct, or reduced [4] product, possibly extending it with disjunctions (or unions) using a disjunctive completion [4]. The domains we develop in this paper are variants of (disjunctive completion of) reduced product

between PA and NA. One practical approach for combining these domains is to combine results of the analyses [7], e.g., by using light-weight data-flow analyses, such as alias analysis and constant propagation, to simplify a program prior to applying predicate abstraction. Thus, the invariants discovered by one analysis are assumed by the other. Another approach is to run the analyses over different abstract domains in parallel within a single analysis framework, using the abstract transfer functions of each domain as is [5, 2]. The analyses may influence each other, but only through conditionals of the program.

The contribution of our work is in adapting, extending, and evaluating existing work on combining propositional and arithmetic reasoning to the needs of software model-checking. We have implemented a general framework for reachability analysis of C programs on top of our four data structures. Our experiments on non-trivial examples show that our proposed combination of PA and NA is more powerful and more efficient than either technique alone. Finally, by coupling PA and NA tightly, our approach opens up new research directions toward automated abstraction refinement techniques that are more efficient that existing solutions.

Name	**Value**	**Example**	**Num.**
NEXPoint	$2^{2^P} \times N$	$(p \vee q) \wedge (0 \leq x \leq 5)$	EXP
NEX	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	EXP
MTNDD	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	SYM
NDD	$2^P \mapsto 2^N$	$(p \wedge (x = 0 \vee x = 3) \vee (q \wedge (x = 1 \vee x = 5)))$	SYM

Table 1. Summary of implementations of NumPredDom; P = predicates; N = numerical abstract values; **Value** = type of an abstract element; **Example** = example of allowed abstract value; **Num** = numeric part representation (explicit or symbolic).

References

[1] T. Ball and S.K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces". In *Proc. of SPIN*, 2001.

[2] Dirk Beyer, Thomas A. Henzienger, and Gregory Theoduloz. 'Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis"'. In *CAV*, 2007.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software". In *PLDI*, 2003.

[4] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *JLC*, (4), 1992.

[5] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. "Joining dataflow with predicates". In *FSE*, 2005.

[6] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *CAV*, 1997.

[7] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop". In *CAV*, 2006.

Session 4: Component-Based Verification

Mary Ann Malloy, Ph.D.: Reuse versus Reinvention: How Will Formal Methods Deal with Composable Systems?, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.53–54

Reuse versus Reinvention: How Will Formal Methods Deal with Composable Systems?

Mary Ann Malloy, Ph.D.

The MITRE Corporation, Hampton, Virginia 23666, USA
mmalloy@mitre.org

Extended Abstract

Traditional software development techniques have emphasized programming in terms of procedures and abstract data types using simple module interconnections. That paradigm supported formal reasoning about and analysis of (sequential) programs made up of primitive constituents, and provided the basis for tools and environments for system design, development and testing.

But system implementations increasingly are centered around building an appropriate workflow as part of a multi-faceted orchestration, as opposed to configuring or customizing self-contained software modules. Thus contemporary development approaches focus on automated solutions built from more sophisticated components described using interface specifications and from connectors described via protocol specifications. Researchers are seeking ways to express high-level abstractions for and special properties of these so-called *composable* systems / applications (a.k.a. *composite* systems / applications). Such formalisms are needed as a basis for the kinds of tools and environments that will provide a *lateral arabesque* for development and testing communities in this slightly unsettling new world, where reuse and integration take precedence over reinvention.

Composable systems represent an emergent problem domain that poses special challenges to verification and testing practitioners. Composability is a design principle dealing with the interrelationships of components. A composable system provides recombinant components that can be selected and assembled in various ways to satisfy specific requirements; it is highly desirable that little or no change is needed to the composable components for them to interoperate. Composable applications represent the desired end state of a full-scale service-oriented architecture implementation. They are connected, process-based sets of independent services existing inside or outside the enterprise (e.g., service providers, outsourced functions). They are applied to a set of requirements much like a custom software solution would be, except without the hard-coded integration logic.

This talk will survey current efforts to establish models and principles for supporting the composable system lifecycle in systematic – rather than *ad hoc* – ways, with a particular focus on developments in the field of testing composable systems / applications. Relevant questions to be addressed include: How are we coping when building a whole with parts that make incompatible assumptions about their mutual interactions? What kinds of checking and analysis can we support? What does reliability mean for a solution that will continuously evolve vice being developed, tested, configured and launched? Pointers will be provided to educational materials as well as proof-of-concept notations, models and tools available for further experimentation on the World Wide Web.

Composable solutions is the direction in which the U.S. Department of Defense, federal stakeholders and commercial enterprises are evolving their information sharing and processing infrastructures. This talk will cite areas where progress is being made, as well as gaps where additional research is recommendable. Such insights are needed to ensure the establishment of appropriate performance criteria and mensuration techniques for applying formal methods, particularly for verification and testing, to composable systems / applications.

Panagiotis Manolios: Automating System Assembly of Aerospace Systems, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.55–55

Automating System Assembly of Aerospace Systems*

Panagiotis Manolios

College of Computer and Information Science, Northeastern University, Boston, MA, 02115, USA
pete@ccs.neu.edu
http://www.ccs.neu.edu/home/pete

Abstract

One of the major challenges in modern aerospace designs is the integration and assembly of independently developed components. We have formalized this as the system assembly problem: from a sea of available components, which should be selected and how should they be connected, integrated, and assembled so that the overall system requirements are satisfied in a certifiable way? We present a powerful framework for automatically solving the system assembly problem directly from system requirements by using formal verification technology. We also present a case study where we applied our work to large-scale industrial examples from the Boeing Dreamliner.

* This research was funded in part by NASA and Boeing.

Sergey Tverdyshev et al.: Formal Verification of Gate-Level Computer Systems, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.56–58

Formal Verification of Gate-Level Computer Systems

Sergey Tverdyshev⋆, *Andrey Shadrin*⋆

Saarland University, Germany

{deru, shavez}@wjpserver.cs.uni-sb.de

Extended Abstract

Modern computer systems are used in many safety-critical applications. In order to guarantee an error-free behavior of such a system one often employs formal methods, e.g. model checking, and theorem proving. However, most of the times formal methods are only applied to stand-alone components or to abstract models. Thus, a pervasive correctness of computer systems, with all their details, can not be guaranteed.

In the Verisoft project [1] we show that it is possible to build and to verify a computer system, which consists of a hardware platform with devices, a compiler for a C-like language, a micro-kernel, an operating system, and user applications. The goal is to verify these components and to guarantee formally that they can be combined into one computer system stack. In this paper, we present the verification of a complex *gate-level* computer system. This hardware is formally verified against an assembly-level model, i.e. a model as seen by an assembly programmer.

The integrity of the results in the Verisoft project requires that all models are defined/verified in one environment. Therefore, we have chosen Isabelle/HOL "work-horse", which is an interactive theorem prover for higher-order logic. To reduce the user work, we developed an environment IHaVeIt [2, 3]. It couples Isabelle/HOL with external tools, such as model checkers (NuMSV [4] and SMV [5]) and SAT solvers. It also provides several reduction and abstraction techniques which increase the application efficiency of the external tools. Thus, we develop and verify the hardware in Isabelle/HOL and then automatically translate it into Verilog (via IHaVeIt) and run it on an FPGA.

The verified computer system consists of a VAMP processor [6] and a generic device model for memory mapped devices.

The VAMP processor features the DLX instruction set, an out-of-order execution, precise interrupts, a delayed branch, and support for virtual memory. We verified the gate-level processor against a model as seen by an assembly programmer, i.e. a model which executes a complete instruction with every step.

The device model on the gate level is modelled as an I/O automaton. It can contain arbitrary devices which run in parallel and communicate with external environment, e.g. with a network. This model is verified against a sequential device model where the devices progress one after another.

The gate-level computer system is built by a parallel composition of the VAMP and the gate-level device model, which communicate via a bus with a common clock. This system is verified

⋆ The authors were supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38

against an assembly model which executes with every transition either a device step or the processor step, with or without device access. The correctness criterion states that every run of the gate-level model can be simulated by a run of the assembly-level model. The criterion is proved via a non-trivial combination of the proofs for the processor and the generic device model. The proof is carried out interactively in Isabelle/HOL with the help of IHaVeIt.

Finally, we instantiated the verified computer system, on the gate and assembly levels, with an automotive bus controller. Thus, we built a verified electronic control unit for a distributed automotive system [7]. This unit has been synthesised and run on a Xilink FPGA. The size of the unit is 5,180,002 gates.

For the first time, we report on the formal verification of a gate-level computer system, which consists of a non-trivial processor and devices. Our results outperform the outcomes for the hardware platform of the famous CLI stack [8], where the verification of a small computer system stack is attempted. Our approach also covers the gap between verification of the devices [9, 10, 11, 12] and the processors [13, 14, 15, 16, 17, 18] as stand-alone components.

References

[1] The Verisoft Consortium: The Verisoft Project. `http://www.verisoft.de/` (2003)

[2] Tverdyshev, S.: Combination of Isabelle/HOL with automatic tools. In Gramlich, B., ed.: FroCoS 2005. Volume 3717 of Lecture Notes in Computer Science., Springer (2005) 302–309

[3] Tverdyshev, S., Alkassar, E.: Efficient bit-level model reductions for automated hardware verification. In: 15th International Symposium on Temporal Representation and Reasoning: TIME 2008, to appear, IEEE (2008)

[4] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open source tool for symbolic model checking. In: CAV '02, Springer (2002) 359–364

[5] McMillan, K.L.: Getting started with SMV. Technical report, Berkeley Labs (1999)

[6] Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In Borrione, D., Paul, W., eds.: CHARME 2005. LNCS, Springer (2005) 301–316

[7] Knapp, S.: Distributed Automotive Systems (draft). PhD thesis, Saarland University, Saarbrcken (2008)

[8] Bevier, W.R., Jr., W.A.H., Moore, J.S., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning **5** (1989) 411–428

[9] Cohen, B.: Component design by example: A step-by-step process using VHDL with UART as vehicle. VhdlCohen (2000)

[10] Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD. (2003) 433–440

[11] ALDEC – The Design Verification Company: UART nVS. `http://www.aldec.com/products/ipcores/_datasheets/nSys/UART_nVS.pdf` (2006)

[12] Rashinkar, P., Paterson, P., Singh, L.: System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell, MA, USA (2001)

[13] Manolios, P., Srinivasan, S.K.: A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures. Journal of Automated Reasoning **37** (2006) 93–116

[14] Velev, M.N.: Automatic formal verification of liveness for pipelined processors with multicycle functional units. In: CHARME. (2005) 97–113

[15] Aagaard, M., Day, N.A., Jones, R.B.: Synchronization-at-retirement for pipeline verification. In: FMCAD. (2004) 113–127

[16] Sawada, J., Jr., W.A.H.: Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. Formal Methods in System Design **20** (2002) 187–222

[17] Hosabettu, R., Gopalakrishnan, G., Srivas, M.K.: Formal verification of a complex pipelined processor. Formal Methods in System Design **23** (2003) 171–213

[18] Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Putting it all together - formal verification of the VAMP. STTT Journal, Special Issue on Recent Advances in Hardware Verification (2005)

Gregory Kulczycki et al.: Proving Correctness for Pointer Programs in a Verifying Compiler, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.59–61

Proving Correctness for Pointer Programs in a Verifying Compiler

Gregory Kulczycki, Amrinder Singh

Virginia Tech, Department of Computer Science, Falls Church, VA 22043, USA

{gregwk, amrinder}@vt.edu

Extended Abstract

This research describes a component-based approach to proving the correctness of programs involving pointer behavior. The approach supports modular reasoning and is designed to be used within the larger context of a verifying compiler. The approach consists of two parts. When a system component requires the direct manipulation of pointer operations in its implementation, we implement it using a built-in component specifically designed to capture the functional and performance behavior of pointers. When a system component requires pointer behavior via a linked data structure, we ensure that the complexities of the pointer operations are encapsulated within the data structure and are hidden to the client component. In this way, programs that rely on pointers can be verified modularly, without requiring special rules for pointers.

The ultimate objective of a verifying compiler is to prove-with as little human intervention as possible-that proposed program code is correct with respect to a full behavioral specification. Full verification for software is especially important for an agency like NASA that is routinely involved in the development of mission critical systems.

There are at least two fundamental problems in developing a verifying compiler that is scalable. One of these is modularity. There is near universal agreement that to be scalable, the verification system must be modular. In other words, it must be possible for the verifying compiler to take just the specifications of components used by a piece of code and to establish that the proposed implementation is correct with respect to its specification [1].

The other problem concerns the complexity of the assertions that are involved in the verification process. A variety of techniques have been explored in the literature to mitigate this. Most recent work involves "lightweight" formal methods, meaning a focus on specification-independent or easy-to-state ad hoc properties of an implementation. Such a system may be used, for example, to establish the absence of null dereferences [2] or to demonstrate the absence of cycles in a pointer-based data structure [3]. Lightweight methods offer two benefits: they relieve programmers from the need to write full behavioral specifications and internal assertions such as loop invariants, and they show that progress toward the goal of full verification can be incremental.

While the lightweight approach is necessary and useful in the immediate term, verifiers ultimately need to get beyond the point of showing merely that blatant error conditions do not exist, and to establish that programs actually achieve a full specification of desired behavior. A verifying compiler eventually must be able to deal with nontrivial assertions such as those needed to prove correctness of implementations that use pointers and references. Researchers who focus their attention on pointer verification problems tend to fall into two categories. Those who extend prior research in lightweight methods, and those who focusing on more general verification. Our work

falls into the second category.

From experience, we know that full verification is especially difficult for programs that involve pointers or references and linked data structures [4]. In some situations, pointers are unavoidable; in others, verification can be simplified by using suitable abstractions. Regardless, a verifying compiler should be able to handle both cases, preferably using the same set of rules. To enable this, we use a two-part approach to full verification of typical pointer programs.

The first part is used when pointer behavior is necessary to satisfy performance requirements. It replaces language-supplied pointers or references with a software component-which we typically refer to as the pointer component-whose specification abstracts pointers and pointer-manipulation operations [5]. The specification does not require any special techniques or constructs. The pointer component has a specification that allows programmers and, importantly, the verification system to view it as any other component. However, to achieve the same performance benefits of pointers, the compiler cannot implement it the same way it implements other components. For example, the call Relocate(p, q) assigns p to q's location, effectively resulting in p and q being aliases. Although the programmer can reason about the statement as a method call, the compiler will implement it by copying a single reference. By using the pointer component, we can ensure that the complications associated with indirection arise only from the specification of the pointer component itself. They occur only where the required behavior of the program to be verified relies on the need for indirection, rather than permeating all proofs of correctness.

We have used the pointer component in the verification of correctness for non-trivial algorithms such as the Schorr-Waite graph-marking algorithm [6]. In the case of the Schorr-Waite algorithm, we wrote the specification for it, implemented it with the pointer component, and showed that the implementation was correct with respect to the specification. This process led us to propose additional operations for the pointer component that simplified certain verification tasks. In particular, we can define and specify pointer component operations that do not create memory leaks. Using these operations in place of traditional ones in the implementation can ease the burden on the verification system.

The second part of our approach uses abstract specifications to encapsulate data structures, such as lists and trees, whose implementations typically require pointer behavior. This limits the pointer-associated verification complexity to the verification of the data structures themselves, and simplifies the verification of components that use these data structures. Since the data structures used in the second part of the approach often require pointer behavior, the pointer component described in the first part of our approach can be used to implement them. Regardless of whether we are verifying components implemented with the pointer component or component implemented with linked data structures, their full verification can be handled similarly. There is neither a need to focus on selected pointer properties, such as the absence of null references or cycles, nor a need for special rules to handle pointers.

Primarily, the language we have used in this research is Resolve, which is a combined programming and specification language designed to support modular verification [7]. It also supports full alias-avoidance at the component level. Efforts are being made to incorporate the programming aspects of our approach to other languages. For example, the Tako language is an object-oriented language with Java-like syntax that employs a pointer component and avoids many common sources of aliasing [8].

1. References

[1] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S., Hollingsworth, J.E.: Reasoning About Software-Component Behavior. Procs. Sixth Int. Conf. on Software Reuse. Springer-Verlag (2000) 266-283

[2] Leino, K.R.M., Nelson, G., Saxe, J.B.: ESC/Java User's Manual. Compaq Systems Research Center (2000)

[3] Hackett, B., Rugina, R.: Region-Based Shape Analysis with Tracked Locations. POPL '05 (2005)

[4] Kulczycki, G., Sitaraman, M., Keown, H., Weide, B.: Abstracting Pointers for a Verifying Compiler. 31st Annual Software Engineering Workshop, Baltimore, MD (2007)

[5] Kulczycki, G.W., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: Capturing the Behavior of Linked Data Structures. Proceedings RESOLVE 2002 Workshop (2002)

[6] Singh, A.: A component-based approach to proving the correctness of the Schorr-Waite algorithm. Computer Science, Vol. Master of Science. Virginia Tech, Falls Church (2007)

[7] Sitaraman, M., Weide, B.W.: Component-Based Software Using RESOLVE. ACM Software Engineering Notes 19 (1994) 21-67

[8] Kulczycki, G., Vasudeo, J.: Simplifying Reasoning about Objects with Tako. Proceedings of the Specification and Verification of Component-Based Systems Workshop, Portland, OR (2006)

Matthew L. Bolton et al.: Formal Modeling of Erroneous Human Behavior and its Implications for Model Checking, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.62–64

Formal Modeling of Erroneous Human Behavior and its Implications for Model Checking

Matthew L. Bolton, Ellen J. Bass

University of Virginia, Department of Systems and Information Engineering ,
Charlottesville, Virginia 22904, USA
Mlb4b@Virginia.edu Ejb4n@Virginia.edu

Extended Abstract

Modern, safety-critical systems are inherently complex as multiple interacting subsystems and people (operators, maintenance crews, etc.) attempt to achieve multiple, often conflicting, goals. While the majority of the sub-systems (including the humanmachine interfaces to control them) are well engineered, system failures still occur: airplane crashes, air-traffic conflicts, power plant failures, defense system false alarms, etc. [1]. Such failures are often due not to the breakdown of a single component, but to a series of minor events that occur at separate times, ultimately leading to dangerous outcomes. Further, more of the pre-cursor events that lead to such outcomes are the result of human error (the error resulting from the interaction between human operators and the system) rather than equipment or component failure [2].

Formal methods, and particularly model checking, have proven useful in detecting design errors that produce system failure in computer hardware and software systems. A number of techniques also exist for modeling human behavior using formal computational structure such as Goals, Operators, Methods, and Selection rules (GOMS) [4], ConcurTaskTrees (CTT) [5], and the Operator Function Model (OFM) [6]. In addition, efforts have also been made to classify human error based on its formal characteristics. While there are a number of reasons why humans may perform an erroneous act (a sequence of activities that do not produce the intended result during human-system interaction), there are very limited formal characteristics for the way that errors can manifest themselves [2]. To address this, Hollnagel [7] classified human error based on a hierarchy of phenotypes, the formal characteristics of observable erroneous behavior. Hollnagel showed that all human errors were composed of one or more of the following errors (all observable for a single act): premature start of an action, delayed start of an action, premature finishing of an action, delayed finishing of an action, omitting an action, skipping an action, reperforming a previously performed action, repeating an action, and performing an unplanned action (an intrusion).

A variety of work has investigated the use of formal system and human behavior models in order to predict and model human error (an overview can be found in [3]). However, the majority of this work has focused on discovering mode confusion and automation surprise (preconditions for a subset of human errors), or have relied on human factors experts to incorporate erroneous behavior into human-behavior models. None of these methods have integrated model checking, human behavior modeling, and human error phenotype classification to automatically model erroneous behavior and use it to predict its contribution to system failure.

To address this, we are developing an extension of the model checking verification process [3] (Fig. 1). This framework includes three automatic processes: human error prediction, translation,

and model checking. The human error prediction process examines a normative human behavior model and a human-system interface model in order to determine what erroneous human behavior patterns are likely. It produces a modified version of the human behavior model with both the normative and erroneous behavior. The translation process uses the system model and the modified human behavior model and produces a single model that is readable by the model checker. The model checking process verifies that the system properties from the specification are true in the system model. If verification fails, the process will generate a counterexample showing how the failure condition occurred. This framework has been instantiated using the SMART and SAL modeling checking programs and used successfully with a simplified model of the Therac-25, a piece of radiological medical equipment for which human error played an important role in a fatal system failure (see [3]). The work discussed here focuses on the human error prediction process, the erroneous human behavior model it produces, and its implication for model checking.

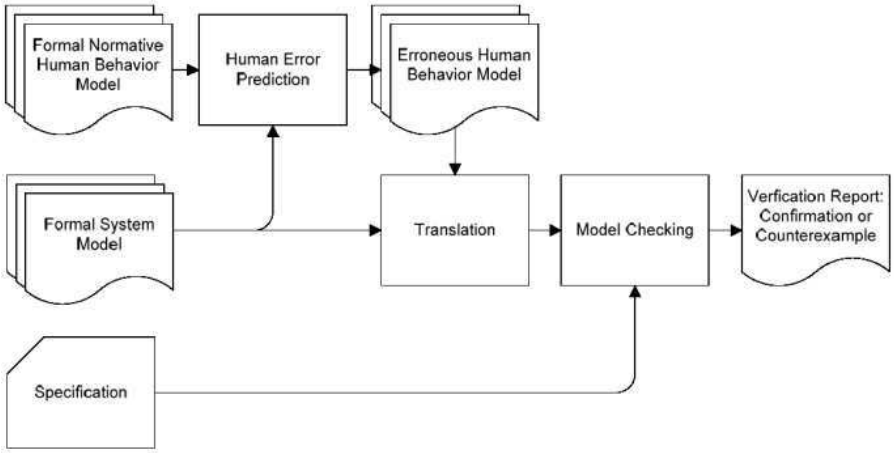


Figure 1. Human error and system failure prediction framework

This work discusses a systematic means of modifying a normative human behavior model specified in the OFM paradigm (decomposing higher level activities into atomic actions) in order to incorporate the observable erroneous behavior identified by Hollnagel (all of which can be constructed from errors at the atomic action level). Given the hierarchal nature of the OFMs and Hollnagel s error phenotypes, this process can be done by replacing each of an OFM s atomic actions with a set of erroneous acts that may occur at that action. Given that the framework being employed in this work assumes a formal model of the human-computer interface and full normative human behavior models, both can be used to determine which of Hollnagel s phenotypes can manifest themselves at a particular action.

Human behavior models used with the proposed framework (Fig. 1) have two important implications for model checking. First, given the nature of model checking, any system containing human-system interaction that is evaluated via model checking will encompass a superset of human behavior beyond what is likely. In this context, the erroneous behavior model can be viewed as a filter for the system model as it limits the human behavior possibilities the model checker needs to evaluate. Thus, we may be able to reduce the system model s state space during the translation process in Fig. 1, potentially alleviating the state explosion problem. Second, the behavior models can be used to explain how human error may have contributed to a system failure identified in a counterexample. This is useful as it may suggest interface or other design changes that prevent the error from occurring.

1. References

[1] Perrow, C., Normal Accidents: Living with High-Risk Technologies, Basic Books, New York (1984)

[2] Reason, J., Human Error, Cambridge University Press, Cambridge, England (1990)

[3] Bolton, M.L., Bass, E.J., Siminiceanu, R.I., Using Formal Methods to Predict Human Error and System Failures, In: Applied Human Factors and Ergonomics International 2008 (In Press)

[4] Kieras, D., John, B., The GOMS family of analysis techniques: tools for design and evaluation, CMU-HCII-94-106 (1994)

[5] Mori, G., Patern, F., Santoro, C., CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, IEEE Transactions on Software Engineering, 28, 8, 797–813 (2002)

[6] Thurman, D.A., Chappell, A.R., Mitchell, C.M., An enhanced architecture for OFMspert: A Domain-independent System for Intent Inferencing. IEEE International Conference on Systems Man and Cybernetics, pp. 955–960. IEEE Press, New York (1998)

[7] Hollnagel, E., 1993, The Phenotype of Erroneous Actions, International Journal of Man-Machine Studies, 39, 1–32 (1993).

Fernando Alegre et al.: A Framework for Stability Analysis of Control Systems Software at the Source Code Level, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.65–65

A Framework for Stability Analysis of Control Systems Software at the Source Code Level

Fernando Alegre, Eric Feron

Georgia Institute of Techology, Atlanta, GA 30332, USA

{fernando, feron}@gatech.edu

Abstract

In this paper, we show how to apply the Floyd-Hoare formalism to analyze C programs implementing feedback control systems. In particular, we show that the well-known Lyapunov, non-expansivity and passivity theories can not only be applied at the specification level but also carried over to the implementation level. We demonstrate how some of those properties, such as bounded input bounded state stability, can be embedded as pre- and post-conditions of each statement in the source code and illustrate how to use this methodology in linear controllers, either subject to bounded input or to parametric uncertainties, and also in controllers with sector-bounded non-linearities.

We also explain how an automatic static analyzer can propagate invariants and produce a proof of stability at the source code level. This proof, basically in the form of a Matlab program, could be independently validated. Therefore, proof generation and proof validation can be performed independently and without the need to trust each other. Similarities and differences of our framework with proof-carrying-code frameworks are also discussed.

John Douglas Carter: Mise en Scene: A Scenario-Based Medium Supporting Formal Software Development, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.66–66

Mise en Scene: A Scenario-Based Medium Supporting Formal Software Development

John Douglas Carter

Dept. of Computing & Information Science, University of Guelph, Ontario, Canada
jcarter@uoguelph.ca

Extended Abstract

The design and engineering of reliable software systems present many technical and managerial challenges. Software engineers come to the proverbial 'drawing board' with a technical understanding of how systems are created but often have difficulties interfacing with customers to accurately elicit requirements and prioritize stakeholder needs. Project stakeholders are experts in the problem domain of the system under design; however, many times they cannot describe a satisfactory software design ("I'll know it when I see it...") or may not be able to identify features lacking in a preliminary design.

Scenarios bridge communication between engineers and project stakeholders. Scenarios describe the system in terms of steps its components perform to meet requirements. Scenario-based approaches provide concrete ways for engineers and stakeholders to discuss and reason about the system without premature commitment to a specific implementation. Scenarios are an excellent starting point for describing intended behavior of a system being designed, and when formalized, they can serve as the input to an automated software engineering approach, such as R2D2C, discussed next.

The "Requirements to Design to Code" (R2D2C) project of NASA's Software Engineering Laboratory is based on inferring a formal, provably-correct specification expressed in Communicating Sequential Processes (CSP) from system requirements supplied in the form of CSP traces. From such a CSP specification, software can be automatically synthesized. R2D2C is a multiinstitution, collaborative effort, including contributors from industry, NASA's Goddard Space Flight Center (GSFC), Virginia Tech and the University of Guelph.

Mise en Scene contains three components. First, a scenario medium designed to be amenable to conversion to CSP traces, to be represented using Mise en Scene's trace medium (Mise en Scene's second component). The trace medium is designed for conveyance to the inference stage of R2D2C. The third component of Mise en Scene is a process for the automatic translation from scenarios to traces.

I present a brief overview of the R2D2C project, the Mise en Scene scenario medium and recent work toward the automatic translation of Mise en Scene scenarios to a CSP specification.

Session 5: The Future of Tools for Verification

On Limits

Gerard J. Holzmann

NASA/JPL Laboratory for Reliable Software, Pasadena, CA 91109, USA
Gerard.J.Holzmann@jpl.nasa.gov

Invited Talk

In the last 3 decades or so, the size of systems we have been able to verify formally with automated tools has increased dramatically. At each point in this development, we encountered a different set of limits – many of which we were eventually able to overcome. Today, we may have reached some limits that may be much harder to conquer. The problem I will discuss is the following: given a hypothetical machine with infinite memory that is seamlessly shared among infinitely many CPUs (or CPU cores), what is the largest problem size that we could solve?

An Update on Yices

Bruno Dutertre

Computer Science Laboratory, SRI International,
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
bruno@csl.sri.com

Extended Abstract

Recent breakthroughs in Boolean satisfiability solving have enabled new approaches to software and hardware verification. Existing SAT solvers can handle problems with millions of clauses and variables that are encountered in bounded model checking, test-case generation, and certain types of planning problems. SAT solving has thus become a major tool in automated analysis of hardware and other finite systems. Satisfiability modulo theories (SMT) generalizes SAT by adding equality reasoning, arithmetic, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in these theories. SMT solvers enable bounded model checking of infinite systems. They have applications in theorem proving, software verification and other domains such as scheduling, temporal or metric planning, and test-case generation.

Yices is an SMT solver developed at SRI International. It is capable of handling large and propositionally complex formulas in a rich combination of theories. Yices formulas can mix uninterpreted functions, linear real and integer arithmetic, bit vectors, scalar and recursive data types, and quantifiers. An important application of Yices is as a back-end solver for the SAL system. In this role, Yices supports verification of finite or infinite state-transition systems using bounded-model checking techniques. Yices is also integrated with the PVS interactive theorem prover, where is complements the existing PVS decision procedures. Other application areas include static analysis of software and software testing.

We give an overview of the architecture and algorithms employed by the new Yices 2 solver planned to be released this year. We describe the logic supported by Yices 2, and the new functionality Yices 2 provides through an improved API.

The core of Yices 2 is a modern Boolean satisfiability solver similar to state-of-the-art tools such as MiniSat, with additional functionality to interact with different theory solvers. The second major module implements a decision procedure for the theory of equality with uninterpreted symbols with extensions for reasoning about tuples and other constructs. The other parts of Yices 2 are specialized decision procedures for bit vectors and arithmetic, and a module that implements quantifier instantiation. We present the major features of these different modules and describe their interaction.

To address some limitations of the current Yices system, we have thoroughly redesigned Yices 2's API to facilitate its integration into other systems. We discuss several API enhancements and simplifications, and we present the new features that we are currently developing.

Florent Kirchner: Distributing Formal Verification: The Evidential Tool Bus, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.71–73

Distributing Formal Verification: The Evidential Tool Bus

Florent Kirchner

SRI International, Menlo Park, CA 94025, USA
florent.kirchner@sri.com

Extended Abstract

The rising diversity of verification tools—proof assistants, model checkers, satisfiability solvers, predicate abstractors, to name a few—can be seen as both a testament to the health of formal methods, and as an impediment to their widespread adoption. In particular, in such a rich ecosystem, the process of making an enlightened choice about the best combination of tools for a given verification task can, in itself, be fairly problematic. This choice bears even more weight considering that there is no guarantee that formal developments in a given system can be later ported to other systems. Solutions to this problem often come as *adhoc* implementations: mainly, translators between proof assistants [9, 14, 15, 5], and integration of solvers, model-checkers, and decision procedures into proof assistants [18, 8]. These approaches all have in common the fact that they are at the same time fragile, because any change in the source or target implementation will break the translation; and expensive to establish and maintain, since they require in-depth expertise of the systems involved.

The novel concept of a *formal tool bus* takes a different approach towards composition and interoperability, by relying on asynchronous message passing between standalone formal verification tools. The tools behave as *distributed agents* that can either publish a formula they wish to see proved, or answer such a request with some evidence attesting of their success. Agents register the services they provide, as well as the syntax and semantics of their logical language, to a *facilitator*, that takes care of the lower-level parts of the connection. Since 2007, work has started at SRI International on a formal tool bus called Evidential Tool Bus (ETB) [17], basing it on the industrial-strength distributed framework Open Agent Architecture [13], and starting with the connection of the Yices SMT-solver [7], the SAL model-checking suite [2], and the PVS proof assistant [16].

Initial investigations about distributed interoperation structures have turned up a number of questions, which in turn opened new research perspectives and contributions. First is the problem of deciding logical compatibility between the components of the system. This is tackled using a first-order metalogical framework and a paradigm that extends Avron's work on consequence relations [1]. However, the strong computational capabilities of solvers such as Yices are poorly represented in this exclusively deductive setting: this leads us to investigate systems that allow the combination of logical inferences and term rewriting steps, such as *deduction modulo* [6]. Second is the problem of managing the coordination between distributed agents, a fundamental feature in order to be able to guide non-trivial agent interactions. For instance, scenarios such as the counter-example guided abstraction and refinement (CEGAR) loop [3] rely on a precise coordination between predicate abstractors, model-checkers, and SAT-solvers. This takes us beyond current work on proof languages [11, 12], which only provide sequencing control over proof development, and into the realm of specialized architecture definition and coordination languages [10, 4].

The ETB currently provides basic functionality for SAL-to-Yices communications, and its development is progressing quickly. By combining formal verification tools in a distributed framework, the formal tool bus architecture aims at facilitating the elaboration of powerful, flexible, and interoperable tool chains. The logical and coordination aspects are at the heart of the design of the tool bus, and we believe the choices that are being investigated and implemented will provide the solid foundations required for the success of this project.

References

[1] Arnon Avron. Simple consequence relations. *Information and Computation*, 92(1):105–139, 1991.

[2] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Natarajan Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[4] Charles Consel, Wilfried Jouve, Julien Lancia, and Nicolas Palix. Ontology-directed generation of frameworks for pervasive service development. In *Proceedings of The 4th IEEE Workshop on Middleware Support for Pervasive Computing*, White Plains, NY, USA, March 2007.

[5] Ewen Denney. A prototype proof translator from HOL to Coq. In Mark Aagaard and John Harrison, editors, *Proc. 13th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000.

[6] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

[7] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006.

[8] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006.

[9] Douglas Howe. Importing mathematics from HOL into Nuprl. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proc. 9th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 1996.

[10] Wilfried Jouve, Julien Lancia, Nicolas Palix, Charles Consel, and Julia Lawall. High-level programming support for robust pervasive computing applications. In *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications, TO APPEAR*, Hong Kong, China, March 2008.

[11] Florent Kirchner and César Muñoz. PVS#: Streamlined tacticals for PVS. In *Proc. 6th Int. Workshop on Strategies in Automated Deduction*, volume 174/11 of *Electronic Notes in Theoretical Computer Science*, pages 47–58. Elsevier Science, 2007.

[12] Florent Kirchner and César Muñoz. The proof monad. Draft, 2008.

[13] David Martin, Adam Cheyer, and Douglas Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, January 1999.

[14] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In Richard Boulton and Paul Jackson, editors, *Proc. 14th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer-Verlag, 2001.

[15] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. 3rd Int. Joint Conf. on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2006.

[16] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[17] John M. Rushby. Harnessing disruptive innovation in formal verification. In *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods*, volume 0, pages 21–30. IEEE Comp. Soc. Press, 2006.

[18] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, 1999.

Lee Pike: Model Checking for the Practical Verificationist: A User's Perspective on SAL, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.74–75

Model Checking for the Practical Verificationist A User's Perspective on SAL*

*Lee Pike***

Galois, Inc., Beaverton, OR 97005, USA
leepike@galois.com

SRI's Symbolic Analysis Laboratory (SAL)[1] is bad news for interactive mechanical theorem provers. SAL is so automated yet expressive that for many of the verification endeavors I might have previously used a mechanical theorem prover, I would now use SAL. The purpose of this brief report is to persuade you to do the same.

To convince the reader, I highlight SAL's features that are especially useful or novel from a practitioner's perspective. With my coauthors, I have had the opportunity to use SAL in a number of applied verifications [1, 2, 3, 5, 6, 7].[2] These works draw from the domains of distributed systems, fault-tolerant protocols, and asynchronous hardware protocols (the most notable omission is the domain of software, although the techniques reported are not domain-specific).

Specifically, in this talk, I will cover using higher-order functions in model-checking, how to use infinite-state bounded model checking (*inf-bmc*) to verify real-time systems, synchronous and asynchronous composition for inf-bmc, and integrating model checking in industrial projects.

References

[1] Geoffrey Brown and Lee Pike. Temporal refinement using smt and model checking with an application to physical-layer protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'2007)*, pages 171–180. OmniPress, 2007. Available at http://www.cs.indiana.edu/~lepike/pub_pages/refinement.html.

[2] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, pages 58–72, 2006. Available at http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html.

[3] Geoffrey M. Brown and Lee Pike. "easy" parameterized verificaton of cross domain clock protocols. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006. Satellite Event of ETAPS. Available at http://www.cs.indiana.edu/~lepike/pub_pages/dcc.html.

[4] Lee Pike. Model checking for the practical verificationist: a user's perspective on SAL. In *Proceedings of the Automated Formal Methods Workshop (AFM07)*. ACM Press, 2007. Available at http://www.cs.indiana.edu/~lepike/pub_pages/afm07.html.

* This abstract and associated talk is derived from a previously-published paper [4].

** Much of the research cited herein was completed while I was a member of the NASA Langley Research Center Formal Methods Group, working on the SPIDER project, under the Vehicle Systems Program. This support is gratefully acknolwedged.

[1] SAL is open source under a GPL license and the tool, documentation, a user-community wiki, etc. are all available at http://sal.csl.sri.com.

[2] My coathors for these works include Geoffrey Brown, Steve Johnson, Paul Miner, and Wilfredo Torres-Pomales. The specifications associated with these works are all available from http://www.cs.indiana.edu/~lepike.

[5] Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 231–238. IEEE, 2007. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/fmcad.html`. Best Paper Award.

[6] Lee Pike and Steven D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html`.

[7] Lee Pike, Paul Miner, and Wilfredo Torres. Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM–2004–213278, NASA Langley Research Center, November 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html`.

Alex Tsow: An Overview of Starfish: A Table-Centric Tool for Interactive Synthesis, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.76–78

An Overview of *Starfish*: A Table-Centric Tool for Interactive Synthesis*

Alex Tsow

*The MITRE Corporation**, Mclean, VA 22102, USA*
atsow@mitre.org

Extended Abstract

Engineering is an interactive process that requires intelligent interaction at many levels. My thesis [1] advances an engineering discipline for high-level synthesis and architectural decomposition that integrates perspicuous representation, designer interaction, and mathematical rigor. *Starfish*, the software prototype for the design method, implements a table-centric transformation system for reorganizing control-dominated system expressions into high-level architectures. Based on the *digital design derivation (DDD)* system—a designer-guided synthesis technique that applies correctness preserving transformations to synchronous data flow specifications expressed as co-recursive stream equations—Starfish enhances user interaction and extends the reachable design space by incorporating four innovations: *behavior tables*, *serialization tables*, *data refinement*, and *operator retiming*.

Behavior tables express systems of co-recursive stream equations as a table of guarded signal updates. Developers and users of the DDD system used manually constructed behavior tables to help them decide which transformations to apply and how to specify them. These design exercises produced several formally constructed hardware implementations: the FM9001 microprocessor, an SECD machine for evaluating LISP, and the *SchemEngine*, garbage collected machine for interpreting a byte-code representation of compiled Scheme programs. Bose and Tuna, two of DDD's developers, have subsequently commercialized the design derivation methodology at *Derivation Systems, Inc. (DSI)*. DSI has formally derived and validated PCI bus interfaces and a Java byte-code processor; they further executed a contract to prototype SPIDER—NASA's ultra-reliable communications bus.

To date, most derivations from DDD and DRS have targeted hardware due to its synchronous design paradigm. However, Starfish expressions are independent of the synchronization mechanism; there is no commitment to hardware or globally broadcast clocks. Though software back-ends for design derivation are limited to the DDD stream-interpreter, targeting synchronous or real-time software is not substantively different from targeting hardware.

The separation of concerns—e.g., architecture, behavior, data representation, and interface coordination—is standard engineering doctrine. In particular, it is futile to expose all aspects

* Many thanks to the NASA Langley Research Center's generous sponsorship of this work through their Graduate Student Researcher's Program, NGT-1-010009. This extended abstract is a revised excerpt from the author's doctoral dissertation accepted by the Indiana University Computer Science Department.[1]

** The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

equally well with a single language. Behavior tables represent a compromise between behavior and architecture: its rows roughly characterize a specification's control oriented aspects, while the columns represent its architectural, or structural, aspects.

The behavior table transformations—among other things—allow designers to trade between these two axes, thereby balancing between the two aspects. It is no surprise, then, that behavior tables are well suited for deriving architectural components from behaviorally oriented expressions.

Type Management and Data Refinement: Behavior tables operate on arbitrarily abstract data-types, not just bit-vectors and bounded integers. In this respect, they are far more expressive than standard hardware description languages. Starfish implements an explicit type system and a framework for data refinement to support high-level specification with abstract data types.

Demand for explicit typing arose from several areas: the need to limit decision expressions to finitely branching guards, the need to prevent incompatible signal merging opportunities among unused slots in table columns, and the desire to increase feedback by disallowing unsound transformations at earlier stages. The type system, which is based on multisorted structures, takes on a second responsibility: it forms a database of term-level identities. One of the core transformations applies algebraic identities (e.g., operator commutativity) to terms. While many term rewrites in DDD are combinator expansions, each algebraic term rewrite requires external validation. Starfish leverages the type system's identity database to confirm algebraic rewrites—only the identity pattern needs external verification.

Since the type system declares function symbols, signatures and identities, it provides a foundation for *data refinement*. At the simplest level, a system of identities can express one-to-one homomorphisms between types. While such an identity system transforms abstract *terms* into representation terms, the architectural algebra preceding Starfish could not transform abstract *signals* into representation signals in a general way. The first attempts to impose signal-level refinement were *ad hoc*, but the current refinement process follows from retiming and recursive identity expansion. In addition to refinement by one-to-one homomorphism, Starfish supports *one-to-many refinements*, where there are multiple representations for each abstract type, and *stateful refinements*, which represent multiple signals with references to a shared store.

While behavior tables are not useful for defining data refinements, they are useful for exploiting and managing their consequences. Data refinements lead to more detailed specifications and consequently a wider transformation space. System decompositions, the problem for which behavior tables were developed, may "cut across" a representation that implements an abstract type with a collection of signals. For instance, suppose a refinement simulates abstract stacks with a pointer and array; subsequent architectural organization may separate the array from the pointer. In another case, a stateful refinement may impose serial access on the previously unconstrained concurrency of abstract operations. Behavior tables and their scheduling aid, serialization tables, provide an interactive method for integrating the serial requirements into a system's control and architecture by scheduling access before and after stateful refinements.

Scheduling and serialization: Starfish introduces *serialization tables* for scheduling the evaluation of complex action terms over several steps. Like behavior tables, columns represent signals and rows represent simultaneous actions which update the signals. Serialization tables are an organizational aid that helps designers solve the NP-hard problems involved in high-level synthesis; e.g., how to fit an evaluation sequence within a specified number of registers and execution units. Serialization tables help specify evaluation order and intermediate resource usage for compound actions in a behavior table. As the schedule develops, the serialization tables display partial

symbolic-evaluations of the intermediate actions. This feedback mechanism helps designers specify actions in the subsequent steps. Starfish validates correctness before integrating the actions into behavior table expressions.

The DDD algebra views serialization as primarily a behavioral problem. Yet, the goal of scheduling is often architecturally dictated. One must use a limited set of resources. Register allocation, functional allocation, and timing are not fully exposed in DDD's behavioral representations. Serialization tables display these aspects more clearly than DDD's co-recursive stream equations, making them a better suited medium for the schedule specification process.

Retiming: Starfish supports retiming in two ways. One is with serialization tables and local re-serialization, or adjustment of schedules. The other is with a transformation that converts combinational signals to sequential signals and vice versa. In schematic terms, the transformation pushes a functional unit from one side of a register to the other. Although retiming is the critical step in transforming abstract signals to representation signals, the motivating example in Starfish was a stack-calculator derivation. The original specification used a combinational `top` accessor for the output signal. Any plausible implementation would store the top value in a register. The exercise of hand-specifying a stack-calculator with a registered `top` signal was enough to see a generalizable pattern. Indeed, equivalent transformations have been used in formal synthesis and microarchitecture algebra.

This talk surveys Starfish's incorporation of behavior tables, data refinement, serialization, and retiming into design derivation. Please see my thesis [1] for an in-depth presentation of these techniques; full text is freely available on the Web.

References

[1] Alex Tsow. *Starfish: A Table Centric Tool for Design Derivation.* PhD thesis, Indiana University Computer Science Department, Bloomington, IN, July 2007. Technical Report 650, 272 pages, `http://www.cs.indiana.edu/pub/techreports/TR650.pdf`.

Author Index

Alegre, Fernando 65

Bansal, Ajay 9
Barton, David L. 44
Bass, Ellen J. 62
Bolton, Matthew L. 62
Bonakdarpour, Borzoo 26
Brukman, Olga 20
Butler, Ricky W. 3

Caldwell, James 33
Carter, John Douglas 66
Chaki, Sagar 47
Cimatti, Alessandro 7

Denney, Ewen 40
Dolev, Shlomi 20
Dutertre, Bruno 70

Feron, Eric 65

Gamboa, Ruben 33
Gardner, William B. 23
Ghoshal, Sudipto 17
Gupta, Gopal 9
Gurfinkel, Arie 47

Heimdahl, Mats 12
Hermenegildo, Manuel V. 29
Holzmann, Gerard 69

Kirchner, Florent 71
Kulczycki, Gregory 59
Kulkarni, Sandeep S. 26
Kuzmina, Nadya 33

López Ruiz, Eduardo Rafael 41
Ledru, Yves 41
Lemoine, Michel 41

Malloy, Mary Ann 53
Mallya, Ajay 9
Manimaran, Solaiappan 17
Manolios, Panagiotis 55
Méndez-Lojo, Mario 29

Min, Richard 9
Muñoz, César A. 3

Navas, Jorge 29

Paul, John 33
Pike, Lee 74
Pritchett, Amy R. 36

Rajan, Ajitha 12
Rosu, Grigore 17
Roveri, Marco 7
Rushby, John 39

Serbanuta, Traian Florin 17
Shadrin, Andrey 56
Simon, Luke 9
Singh, Amrinder 59
Stefanescu, Gheorghe 17
Susi, Angelo 7

Tonetta, Stefano 7
Tsow, Alex 76
Tverdyshev, Sergey 56

Vardi, Moshe Y. 6

Woodham, Kurt 12

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The 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 Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
01- 05 - 2008

2. REPORT TYPE
Conference Publication

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Proceedings of the Sixth NASA Langley Formal Methods Workshop

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Rozier, Kristin Yvonne (Editor)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
645846.02.07.07.07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19476

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CP-2008-215309

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 59
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
Proceedings of a workshop sponsored by the National Aeronautics and Space Administration and held in Newport News, Virginia, April 30-May 2, 2008

14. ABSTRACT

Today's verification techniques are hard-pressed to scale with the ever-increasing complexity of safety critical systems. Within the field of aeronautics alone, we find the need for verification of algorithms for separation assurance, air traffic control, auto-pilot, Unmanned Aerial Vehicles (UAVs), adaptive avionics, automated decision authority, and much more. Recent advances in formal methods have made verifying more of these problems realistic. Thus we need to continually re-assess what we can solve now and identify the next barriers to overcome. Only through an exchange of ideas between theoreticians and practitioners from academia to industry can we extend formal methods for the verification of ever more challenging problem domains. This volume contains the extended abstracts of the talks presented at LFM 2008: The Sixth NASA Langley Formal Methods Workshop held on April 30 - May 2, 2008 in Newport News, Virginia, USA. The topics of interest that were listed in the call for abstracts were: advances in formal verification techniques; formal models of distributed computing; planning and scheduling; automated air traffic management; fault tolerance; hybrid systems/hybrid automata; embedded systems; safety critical applications; safety cases; accident/safety analysis.

15. SUBJECT TERMS
Formal methods; Formal model; Hardware assurance; Life-critical; Model checking; Safety-critical; Software assurance; Theorem proving; Verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	94	19b. TELEPHONE NUMBER *(Include area code)* (301) 621-0390

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18