Toward Automated Test Generation for Engineering Applications

Songtao Xia [*]
Mail Stop 130
NASA Langley Research Center
Hampton, VA, 23681
s.xia@larc.nasa.gov

Ben Di Vito
Mail Stop 130
NASA Langley Research Center
Hampton, Virginia, 23681
b.divito@nasa.gov

César Muñoz
National Institute of Aerospace 100
Exploration Way
Hampton, Virginia, 23666
munoz@nianet.org

ABSTRACT

In test generation based on model-checking, white-box test criteria are represented as trap conditions written in a temporal logic. A model checker is used to refute trap conditions with counter-examples. From a feasible counter-example test inputs are then generated. Earlier research has demonstrated the usefulness of this approach and revealed its weakness. The major problems of applying this approach to engineering applications derive from the fact that engineering programs have an infinite state space and non-linear numerical computations. Our solution is to combine predicate abstraction (which reduces the state space) with a numerical decision procedure (which supports predicate abstraction by solving non-linear constraints) based on interval analysis. We have developed a prototype and applied it to MC/DC (Modified Condition/Decision Coverage) test case generation. We have used the prototype on a number of C modules taken from a conflict detection and avoidance system and from a Boeing 737 autopilot simulator. The modules range from tens of lines up to thousands of lines in size. Our experience shows that although in theory the inclusion of a decision procedure for non-linear arithmetic may lead to non-terminating behavior and false positives (as abstraction-based model checking already does), our prototype is able to automatically produce feasible counterexamples with only a few exceptions. Furthermore, the process runs with acceptable execution times, without requiring any other knowledge of the specification, and without tampering with the original C programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

[*]National Research Council Post-doctoral Researcher

20th IEEE/ACM International Conference on Automated Software Engineering Long Beach, California, USA

General Terms

Verification, Theory

Keywords

Test case generation, model-checking, predicate abstraction

1. INTRODUCTION

FAA's regulations concerning DO-178B [39] explicitly set test criteria for different levels of safety related software. For example, certification of level A software, one that involves human safety, requires test cases that satisfy the MC/DC (Modified Condition/Decision Coverage) criteria [9, 25]. Efforts in meeting these criteria are highly iterative and labor-intensive. Even small improvements in automated test case generation could help to reduce significantly the development cost of certified software.

In recent years, an active community of researchers on test case generation based on model checking has formed. Several tools have been developed (c.f., [1, 7, 21, 26, 43]). The inputs to these generators are a specification (in a few cases, a program) and a coverage criterion; the outputs are value assignments to the program input parameters. Feeding the value assignments to a test harness will guarantee the coverage demanded by the criteria. Our attention in this paper focuses on generating test cases from a program rather than from a specification. Our tool can be combined with specification-based approaches to accelerate the process of achieving MC/DC coverage.

Using a model checker in test case generation relies on its ability to find a counter-example of an invalid formula. In particular, from coverage criteria one is able to construct a so-called trap condition of the form "P is always not true" for some predicate P. For example, using linear time logic, a trap condition can be: $\Box\neg(pc = L \wedge (x > 0))$, where pc is an artificial variable ranging over program locations. The model checker might find that this trap condition is not true, in which case a counter-example is generated. The counter-example demonstrates how L is reached and $x > 0$ is satisfied. From such a counter-example, one may use a data concretizer to discover test inputs.

Avionics systems are likely to frustrate most existing model checkers because of large state spaces, complicated control structures, and non-linear computation. A large state space

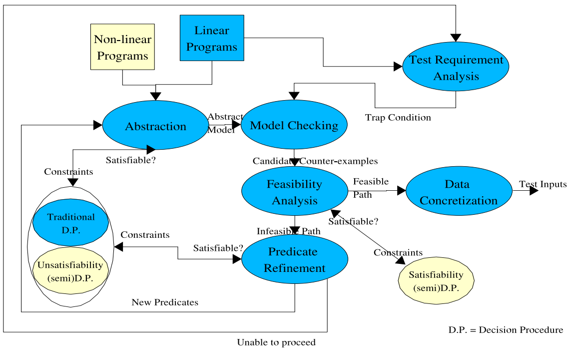

Figure 1: Architecture of Automated Test Case Generation for Engineering Software

implies that abstraction is necessary. The most successful abstraction technique in recent years is predicate abstraction (c.f., [3,6,8,14,18,20,22,30,40]). Model checking based on predicate abstraction can be automated through counter-example driven predicate discovery (c.f., [4,10,11,15,32,41]). The abstraction relation computation, the feasibility test of a candidate counter-example, and the automated predicate discovery, all depend on cooperative decision procedures. State-of-the-art cooperative decision procedures (c.f., [5,16, 19,33]) do not handle non-linear arithmetic due to undecidability[1].

In automating test case generation, imprecision (from the point of view of rigorous verification) and incompleteness are tolerated and numerical solutions are acceptable (if not sought). In addition to the current decision procedures, we adopt techniques from the constraint programming community, where numerical methods based on interval analysis [36] can decide (un)satisfiability, and compute either mathematical or numerical solutions of a non-linear formula (c.f., [23,27,38]). The contribution of our research is the use of a numerical satisfiability solver in place of the decision procedures in model checking based on predicate abstraction. Used as the engine for test case generation, such a model checker will find feasible counter-examples that cannot be found by other tools to the best of our knowledge. A general architecture of our framework is illustrated in Figure 1. Further details will be explained in Section 2.

Care must be taken when adopting such decision procedures. Various theoretical limitations (such as termination) and practical considerations (such as the trade-off between speed and precision) will cause the checker to be either unsound (producing false positives) or incomplete (producing false negatives). Indeed, under certain assumptions, our approach of integrating these decision procedures into predicate abstraction is considered sound. That our approach is incomplete is obvious.

We implement LAST (Logic-based Automation for Software Testing) as a prototype of our method. LAST is based on BLAST (Berkeley Lazy Abstraction and Software verification Toolkit) [29], which analyzes C programs. It extends BLAST's predicate abstraction with a numerical non-linear constraint unsatisfiability checker based on Realpaver [23], an experimental interval analysis tool developed at the University of Nantes. We have applied our prototype implementation to the MC/DC test case generation for a number of C modules, including those from a conflict detection and resolution program [17] and a Boeing 737 autopilot simulator. The largest module contains about 2000 lines of code. Our prototype is able to automatically produce feasible counter-examples for MC/DC-like criteria, with only a few exceptions, within acceptable execution time, without any other knowledge of the specification, and without tampering with the original C programs.

2. THEORETICAL FRAMEWORK

2.1 Definitions

Throughout the paper, our use of terms such as "decisions" (except in the term "decision procedure") and "conditions" is in accordance with DO-178B. For a (procedural) program[2], a *decision* refers to a (Boolean) expression based on which the choice of the next target is made. A *condition* is an atomic Boolean expression (a relational expression) where there are no logical connectives. A *state* is a type preserving value assignment to program variables, including artificial ones such as *pc*. We denote by $E(s)$ the value of expression E at state s. We write $s \models P$ if the predicate P holds at state s. A set of test inputs is said to achieve *MC/DC coverage* only if[3] for every condition C_i in decision D at location L, there are at least two test inputs T_1 and T_2, where executing the program with T_1 and T_2 will reach states s_1 and s_2 respectively, such that: a) $pc(s_1) = pc(s_2) = L$, b) $s_1 \models C_i$ and $s_2 \models \neg C_i$, c) for every other condition C that appears in D, $s_1 \models C$ iff $s_2 \models C$, and d) $D(s_1) = \neg D(s_2)$.

Coverage criteria can be characterized as logic assignments to conditions. A *trap condition* is a temporal logic formula R of the form $\Box\neg(pc = L \wedge P)$, where P is a conjunction of conditions or their negations. A *counter-example* to a trap condition $R = \Box\neg Er$ is a sequence of states $s_i, i = 1, \ldots, n$, such that $s_n \models Er$.

We assume that a sound computation of weakest precondition in the language exists and the weakest precondition is expressible in the syntax in Figure 2. We further assume that there is only one machine number format throughout the development process.

2.2 Predicate Abstraction

We give an operational definition of predicate abstraction partially following the account in Ball's paper [2]. Predicate abstraction accepts as input a statement c and a set Φ of predicates and outputs a function (called *abstraction transition*) between two abstract states represented as two

[1]What we vaguely refer to as non-linear arithmetic here includes non-linear multiplication, transcendental functions and mixed real and integer arithmetic, all of which contribute to undecidability.

[2]Throughout the paper, we use the term "program" to refer to the unit being tested.

[3]We list only necessary conditions of the coverage criteria relevant to our development here.

Constraint P	::=	$f = f \mid f >= f \mid f <= f$
Expression f	::=	$f + f \mid f - f \mid f * f \mid f/f$
		$\mid (f) \mid -f \mid +f$
		constants $\mid$ vars $\mid \mathcal{F}(\overrightarrow{f})$
Interpreted Funs $\mathcal{F}$	::=	min | max | pow | sqrt | log
		| exp | cos | sin | tan | acos | asin...

Figure 2: Syntax for Constraints

vectors. Every bit[4] in the vector represents the truth value (plus possibly another value $*$) of a predicate in Φ.

The computation can be understood as observing the program's behavior over Φ in a piecewise manner. Informally, the effect of a statement c over a predicate $P_i \in \Phi$ can be written as an assignment:

$$b_i \quad = \quad \mathcal{WP}(c, P_i)$$

where we use b_i for the bit corresponding to P_i. Because the domain of the abstraction has to be an abstract state too, that is, every control has to be made solely based on an abstract state, we have to compute an approximation of $\mathcal{WP}(c, P)$ using the bit vector. We write such an approximation of $\mathcal{WP}(c, P)$ as $\mathcal{WP}_E(c, P)$, which can be computed by checking for every conjunction Q_i in Φ if $Q_i \Rightarrow \mathcal{WP}(c, P)$. This is implemented by calling a satisfiability checker on the formula $Q_i \wedge \neg\mathcal{WP}(c, P)$. Thus, $\mathcal{WP}_E(c, P)$ is defined as the disjunction of all Q_i's that pass the check. Note that, in general, $WP_E(c, P) \neq \neg WP_E(c, \neg P)$, thus the assignment to b_i can be written as a program (called *Boolean program*, or BP) statement of the form:

$$\begin{aligned} b_i \quad = \quad & \text{if } (\mathcal{WP}_E(c, P_i)) \text{ then true} \\ & \text{elseif } (\mathcal{WP}_E(c, \neg P_i)) \text{ then false else } * \end{aligned}$$

where $*$ represents a non-deterministic choice. Note that a BP is only used as the input to a model checker, where a test of * means both branches based on this test must be searched by the model checker.

An abstract state (bit-vector) s_a can also be represented as a proposition. Suppose the residue of the vector after filtering out the $*$ bits is v. Then the proposition is:

$$\gamma(s_a) \quad = \quad \bigwedge_{i=1}^{n} \text{if } (v(i)) \text{ then } P_i \text{ else } \neg P_i.$$

Function γ is called a concretization function.

From an initial abstract state s_0 and a set of predicates Φ, a model checker can repeatedly apply the abstract transition to compute a set of reachable states until a fixed point is reached, or an error state is met. Every possible move is followed; when more than one move is eligible, each one is followed and the other is retained for future backtracking.

2.3 Predicate Discovery

Based on counter-example feasibility testing, counter-example driven predicate discovery allows the model checker to incrementally discover a suitable set of predicates, starting with an initial value of Φ that includes the atomic predicates in Er of the form $(pc = L) \wedge \phi$. This procedure is also known as *predicate refinement* and is in general incomplete (c.f., [2]) because it is possible that the path is not feasible and we could not find new predicates. Given an error path $\beta = s_0, \ldots, s_n$, we compute a sequence of statements $tr(\beta) = c_1, \ldots, c_n$ (less by one than the number of states in the error path). Iteratively, we compute the weakest preconditions $P_1, \ldots, P_n$ using the equations below:

$$\begin{aligned} P_1 \quad &= \quad \mathcal{WP}(c_n, \phi) \\ P_{i+1} \quad &= \quad \mathcal{WP}(c_{n-i+1}, P_i) \end{aligned}$$

We check whether P_i is satisfiable. If, for some j, P_j is not satisfiable, we attempt to find new predicates from the path from s_j to s_n. One way to find new predicates is to collect all the predicates involved or use certain heuristics to select the new predicates. A better approach is to use Craig interpolation ([28, 35]).

2.4 System Diagram

Constraint satisfiability is studied by the constraint logic programming community. Modern constraint solvers often combines interval arithmetic [36] with local inconsistency checking [34]. Practical considerations such as speed or time-out contribute to the imprecision of constraint solvers. We divide them into satisfiability checkers, whose "yes" decision is accurate, and unsatisfiability ones, whose "no solution" answer is accurate[5]. Unsatisfiability constraint solvers can be used to construct a (semi-)decision procedure to be used in predicate abstraction. An unsatisfiability solver may fail to identify that an implication holds thus introduce further overapproximation. A satisfiability constraint solver is suitable for feasibility test [6]. In addition, the data concretizer itself should be constructed from a satisfiability solver.

The discussion above is reflected in Figure 1. A test requirement analyzer analyzes the input program and generates trap conditions. The trap conditions and source program are then taken into an iterative abstraction-model checking-refinement process. This process depends on different decision procedures. Traditional ones are colored blue (darker gray-scale) while numerical ones are colored pale yellow (lighter gray-scale). This iterative process may not terminate. When it does, it either generates test inputs, or declares the trap condition can never happen, or complains that new predicates cannot be found (the latter two are combined as "unable to proceed" in the figure). Finally, we have the soundness theorem under the assumptions above.

CLAIM 1 (SOUNDNESS). *If the process described in Section 2 returns a feasible error path β, then there exists a test input, the execution of which will satisfy the coverage criterion.*

2.5 Example

Figure 3 is a code snippet from a Boeing 737 autopilot simulator. Suppose the decision we are interested in is $a \neq 0.0$

[4]Strictly speaking not a bit, but a variable ranging over values from a lattice induced from {true, false}.

[5]We refrain from using the words "sound" and "complete" because they are potentially confusing in this context. For example, what is not sound to the decision problem could be sound for predicate abstraction.

[6]Because a satisfiability solver is often slow, we use an unsatisfiability checker as an approximation in our prototype

```
1 : /* COMPUTE PARTIAL G WITH RESPECT TO X */
2 : if ( (*lastout > deflmt[0])   &&
3 :      (*lastout < deflmt[1]) )
4 :     pGwrx = 1.0;
5 : else
6 :      pGwrx = 0.0;
7 : a = -gain * pHwre * pGwrx;
8 : if ( a != 0.0 ) {
9 :     /* something else */
10: } else {
11:     /* something else */
12: }
```

Figure 3: Code snippet from autopilot simulator

at line 8. We feed a trap condition $\Box\neg(pc = 8 \wedge a \neq 0.0)$ to the model checker. A counter-example driven approach will start from the predicates that appear in the trap condition ($\{a = 0.0\}$). Because a is not relevant to the decision in Line 2 and 3, the decision will be abstracted to $*$. Suppose the else branch is taken first. Line 6 does not affect a. The abstract state after Line 7 is still $b_1 = *$. Now Line 8 is reached, and the conjunction of $*$ and $a \neq 0.0$ is indeed satisfiable. The trace leading to this error state will be returned as a counter-example, which will be analyzed to see if it is feasible. For example, the trace corresponding to lines 2-3-6-7-8-9 is not feasible because it requires both $\texttt{pGwrx} = 0$ and $\texttt{a} \neq 0$, which is impossible due to the assignment at Line 7. This conclusion is drawn by inquiring the decision procedure about the following constraints, which are computed using weakest precondition:

$$\begin{aligned} \texttt{a} &\neq 0.0 \\ 0.0 &\neq -\texttt{gain} * \texttt{pHwre} * \texttt{pGwrx} \ \wedge \\ 0.0 &\neq -\texttt{gain} * \texttt{pHwre} * 0.0 \ \wedge \end{aligned}$$

Because the constraint set is non-linear, traditional decision procedures cannot solve it. However, the numerical decision procedure will have no difficulty in deciding that the constraints are not satisfiable. Thus the error trace is not feasible.

From such an analysis, the tool realizes that it is the execution of Line 6 that causes the constraints to be unsatisfiable. Therefore, Φ is updated to $\{a = 0.0, \texttt{pGwrx} = 0.0\}$. Subsequent search based on the new set of predicates should notice that if Line 6 is reached then $\texttt{a} = 0.0$ will be true after executing Line 7. Then the error state is not reached because $a = 0.0 \wedge a \neq 0.0$ is false. The model checker backtracks and selects Line 4 instead of Line 6. This time, the error state is reached at Line 8 and the counter-example is feasible.

3. IMPLEMENTATION

We implement our prototype system based on two existing systems, BLAST from Berkeley and Realpaver from University of Nantes. BLAST provides the reachability test of a C program; Realpaver can be used to determine the satisfiability of a set of non-linear constraints. We extend Realpaver to a decision procedure of the Nelson and Oppen flavor. We then plug the new decision procedure into BLAST, replacing the decision procedures used there (one of Vampyre, Simplify, or CVC-lite, though Simplify is favored and sometimes hard coded). Some amount of reprogramming is required to tailor BLAST for our purpose. Realpaver is claimed to satisfy the following property [23]:

Proposition 1 (Reliability). *Realpaver computes a union of boxes that contains all the solutions of the original constraint satisfaction problem. Therefore, if no box is computed by Realpaver, the constraint satisfaction problem has no solutions.*

We apply our prototype to two programs. One is an experimental air traffic conflict detection and resolution system, called KB3D [17]. The other is a linear dynamics simulator for a Boeing 737 autopilot system (called simulator below). Both programs contains non-linear computations that involve non-linear terms as well as exponential and trigonometric functions. The former is a small program containing several modules, while the latter is a larger program that contains 20 modules. The sizes of the modules range from tens of lines to thousands of lines. The number of decisions are 40 for the KB3D program and 220 for the simulator.

We first generate test cases for each of the modules. With a few exceptions (4 cases for the simulator), we are able to generate feasible examples. Note that certain for-loops are taken special care of in predicate abstraction based approaches. The running time of our prototype is short, typically seconds, with the longest one more than 20 minutes on a commodity computer. The Realpaver-based decision procedure handles the smaller problems generated during abstraction quickly. More complicated problems generated during feasibility testing, which often contain more than 100 variables and hundreds of constraints, take as long as 10 minutes for Realpaver to respond. Our ongoing research includes experimenting with rsolver [38] to generate test cases from the feasible paths.

4. RELATED WORK

- *Test Case Generation* It has been long recognized that test cases can be generated from symbolic execution of the program [31]. Early work in the area of model checking based test case generation [1,21,26] generates test cases from specification. A few others [7,43] take program as input but do not handle non-linear computation. Using a constraint solver in test case generation is built into commercial tools (for example,Reactis[7]). Test case generation for real time system is explored by Hamon et al [24]. Some non-linear decisions are supported by the underlying ICS decision [19] procedure. This work also suggests that for test case generation, a combination of different model checking techniques is needed.

- *Decision Procedures* Tarski [42] shows the first order theory of real numbers with addition and multiplication is decidable through quantifier elimination. Collins shows that quantifier elimination can be done through Cylindrical Algebraic Decomposition [12]. Adding periodic functions will cause the theory to be undecidable, while adding *exp* is decidable if Schanuel's conjecture holds.

[7] `http://www.reactis.com`

- *Data Flow Analysis* ASTREE [13], a dataflower analyzer for safety critical systems, is based on such abstraction domains as octagon, ellipsoid and decision trees.

5. REFERENCES

[1] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM)*, pages 46–54, 1998.

[2] T. Ball. Formalizing counter-example driven predicate refinement with weakest preconditions. Technical Report MSR-TR-2004-134, Microsoft Research, 2004.

[3] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstraction for Model-checking C Programs. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) 2001, Lecture Notes in Computer Science 2031*, pages 268–283. Springer-Verlag, 2001.

[4] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science 2057*, pages 103–122. Springer-Verlag, May 2001.

[5] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[6] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proceedings of Conference on Computer Aided Verification (CAV) 98, Lecture Notes in Computer Science 1427*, pages 319–331, June 1998.

[7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generate tests from counter-examples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Portland, Oregon, 2004. IEEE.

[8] S. Chaki, E. Clarke, A. Groce, and S. Jha. Modular verification of software components in c. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[9] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of Conference on Computer Aided Verification (CAV) 00*. Springer-Verlag, 2000.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pages 154–169, 2000.

[12] G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, 1975.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astre analyzer. In *ESOP'05*, 2005.

[14] S. Das, D. Dill, and S. J. Park. Experience with Predicate Abstraction. In *Proceedings of Conference on Computer Aided Verification(CAV) 99, Lecture Notes in Computer Science 1633*, pages 160–171, Trento, Italy, July 1999.

[15] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proceedings of Conference on Formal Methods in Computer-Aided Design*, Portland, Oregon, November 2002.

[16] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.

[17] G. Dowek, A. Geser, and C. Muñoz. Tactical conflict detection and resolution in a 3-D airspace. In *Proceedings of the 4th USA/Europe Air Traffic Management R&DSeminar, ATM 2001*, Santa Fe, New Mexico, 2001. A long version appears as report NASA/CR-2001-210853 ICASE Report No. 2001-7.

[18] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, R. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification.

[19] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (Paris, France)*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, July 2001.

[20] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of Symposium on Principles of Programming Languages (POPL) 2002*, pages 191–202, 2002.

[21] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146–162, London, UK, 1999. Springer-Verlag.

[22] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of Conference on Computer Aided Verification (CAV) 97, Lecture Notes in Computer Science 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[23] L. Granvilliers. On the combination of interval constraint solvers. *Reliable Computing*, 7(6):467–483, 2001.

[24] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[25] K. Hayhurst, D. Veerhusen, J. Chilenski, and L. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA Langley Research Center, May 2001.

[26] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[27] P. V. Hentenryck, L. Michel, and Y. Deville. *Numerica, A Modeling Language for Global Optimization*. The MIT Press, 1997.

[28] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstraction from Proofs. In *Proceedings of ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL)*, pages 232–244, 2004.

[29] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *Proceedings of Conference on Computer-Aided Verification (CAV)*, pages 526–538, 2002.

[30] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[32] R. Kurshan. Models Whose Checks Don't Explode. In *Proceedings of Computer-Aided Verification (CAV)*, pages 222–233, 1994.

[33] S. K. Lahiri and S. A. Seshia. The uclid decision procedure. In *CAV*, pages 475–478, 2004.

[34] O. Lhomme. Consistency techniques for numerical csps. In *Proceedings of International Joint Conference on Artificial Inteligence (IJCAI)*, pages 232–238, 1993.

[35] K. L. McMillan. Craig interpolation and reachability analysis. In *SAS*, page 336, 2003.

[36] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[37] S. Ratschan. Slides, available at http://www.mpi-sb.mpg.de/r̃atschan/decproc1.pdf.

[38] S. Ratschan. Continuous first-order constraint satisfaction. In *Proceedings of Artificial Intelligence and Symbolic Computation*, LNCS. Springer, 2002.

[39] RTCA SC-167 and EUROCAE WG-12. Software considerations in airborne systems and equipment certification, December 1992.

[40] H. Saidi. Model-checking Guided Abstraction and Analysis. In *Proceedings of Static Analysis Symposium (SAS) 00, Lecture Notes in Computer Science 1824*, pages 377–389, Santa Barbara, CA, USA, July 2000. Springer-Verlag.

[41] H. Saidi and N. Shankar. Abstract and Model-check While You Prove. In *Proceedings of Conference on Computer Aided Verification (CAV) 99, Lecture Notes in Computer Science 1633*, pages 443–454. Springer-Verlag, July 1999.

[42] A. Tarski. *Logic, Semantics, Metamathematics, papers from 1923 to 1938*. Hackett Publishing Company, 1983. English Version, original in Polish.

[43] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.