TAP 2009
Short papers

TAP 2009: short papers

ETH Technical Report 630

Preface

This technical report contains the short papers presented at the Third International Conference on Tests and Proofs (TAP 2009) held at ETH Zurich, Switzerland during July 2-3 2009.

The TAP conference is devoted to the convergence of proofs and tests. It combines ideas from both sides for the advancement of software quality. To prove the correctness of a program is to demonstrate, through impeccable mathematical techniques, that it has no bugs; to test a program is to run it with the expectation of discovering bugs. The two techniques seem contradictory: if you have proved your program, it is fruitless to comb it for bugs; and if you are testing it, that is surely a sign that you have given up on any hope of proving its correctness. Accordingly, proofs and tests have, since the onset of software engineering research, been pursued by distinct communities using rather different techniques and tools. And yet the development of both approaches leads to the discovery of common issues and to the realization that each may need the other. The emergence of model checking has been one of the first signs that contradiction may yield to complementarity, but in the past few years an increasing number of research efforts have encountered the need for combining proofs and tests, dropping earlier dogmatic views of incompatibility and taking instead the best of what each of these software engineering domains has to offer.

The first TAP conference (held at ETH Zurich in February 2007) was an attempt to provide a forum for the cross-fertilization of ideas and approaches from the testing and proving communities. The 2008 edition took place in the Monash University Prato Centre near Florence. For the third TAP conference we came back to ETH Zurich. This third edition was co-located with other software conferences, in particular TOOLS Europe.

We wish to sincerely thank all the authors who submitted their work for consideration. We would like also thank the Program Committee members as well as the additional referees for their great effort and work of high quality in the review and selection process. Their names are listed on the following pages.

There were 20 submissions. Each submission was reviewed by at least three persons. The committee decided to accept 10 research papers. The program also included 2 keynote talks. We are grateful to Sriram Rajamani (Microsoft Research, India) and Boutheina Chetali (Gemalto, France) for accepting the invitation to address the conference.

The conference also included some short presentations that were reviewed by at least one program committee member. They are not included in the conference proceedings volume but are part of the present technical ETH report.

The success of the conference resulted from a team effort. We are grateful to the Conference chair and the Steering Committee members for their support at every stage in the conference preparation. We also thank all the members of the Organizing Committee. Finally we gratefully acknowledge the material and financial support provided by Chair of Software Engineering, ETH Zurich.

May 2009 Catherine Dubois

Conference Organization

Conference Chair

Bertrand Meyer	ETH Zurich, Switzerland

Programme Chair

Catherine Dubois	ENSIIE, France

Programme Committee

Bernhard Aichernig	TU Graz, Austria
Bernhard Beckert	University of Koblenz, Germany
Patrice Chalin	Concordia University, Canada
Yoonsik Cheon	University of Texas at El Paso, USA
Koen Claessen	Chalmers University of Technology, Sweden
Gilles Dowek	École Polytechnique, France
Angelo Gargantini	University of Bergamo, Italy
Arnaud Gotlieb	IRISA, France
Yuri Gurevich	Microsoft Research, USA
Bart Jacobs	Katholieke Universiteit Leuven, Belgium
Reiner Hähnle	Chalmers University of Technology, Sweden
Ewen Maclean	Heriot-Watt University, UK
Karl Meinke	KTH Royal Institute of Technology, Sweden
Sam Owre	SRI International, USA
Wolfram Schulte	Microsoft Research, USA
Mark Utting	Waikato University, New Zealand

Additional Reviewers

Bernard Botella	Richard Bubel	Andrea Calvagna
Bruno Dutertre	Frédéric Gervais	Christoph Gladisch
K. Rustan M. Leino	Patricia Mouy	Ulf Norell
David Pichardie	Vlad Rusu	Natarajan Shankar
David Streader	Ashish Tiwari	Margus Veanes
Burkhart Wolff		

Steering Committee

Yuri Gurevich	Microsoft Research, USA
Bertrand Meyer	ETH Zurich, Switzerland

Local Organization

Yi Wei	ETH Zurich, Switzerland
Stephan van Staden	ETH Zurich, Switzerland
Claudia Günthart	ETH Zurich, Switzerland

Sponsoring Institutions

Chair of Software Engineering, ETH Zurich, Switzerland
ENSIIE, Évry, France

Table of Contents

Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder (Extended Abstract)*

Jasmin Christian Blanchette and Tobias Nipkow

Fakultät für Informatik, T. U. München, Garching, Germany
{blanchette,nipkow}@in.tum.de

1 Introduction

Anecdotal evidence suggests that most "theorems" initially given to an interactive theorem prover do not hold, typically because of a typo or a missing assumption, but sometimes because of a deep flaw. Modern proof assistants for higher-order logic (HOL) provide counterexample generators that can be run on putative theorems or on specific subgoals in a proof to spare users the Sisyphean task of trying to prove non-theorems.

Isabelle/HOL includes two such tools: Quickcheck [1] generates functional code for the HOL formula and evaluates it for random values of the free variables, and Refute [5] searches for finite countermodels of a formula through a reduction to SAT (Boolean satisfiability). Their areas of applicability are almost disjoint: Quickcheck excels at inductive datatypes but is restricted to the executable fragment of HOL (which excludes unbounded quantifiers) and may loop endlessly on inductive predicates. In contrast, Refute copes well with logical symbols, but inductive datatypes and predicates are mostly out of reach due to the state space explosion.

Our new tool, Nitpick [6], is designed to bridge this gap. Instead of using a SAT solver directly, it builds upon the Kodkod first-order relational model finder [4].[1] As a result, it benefits from Kodkod's optimizations (notably its symmetry breaking) and its richer logic. Inductive datatypes are handled following an Alloy idiom [3], and inductive predicates are unrolled as in bounded model checking [2]. Infinite datatypes are approximated by a finite fragment augmented with an undefined value, embedded in a three-valued logic. The current prototype outperforms Refute in nearly all benchmarks while enjoying wider applicability than Quickcheck.

2 Our Approach

Enumeration of domains. Nitpick uses Kodkod to find a finite model (a satisfying assignment to the free variables) of the formula $A \wedge \neg P$, where P is a putative theorem and A is the conjunction of axioms relevant to P. Nitpick, like Refute and Alloy, systematically enumerates the possible domain cardinalities for each type variable, so that if a formula has a finite counterexample, the tool finds it (unless it runs out of memory).

* This work is supported by the DFG grant NI 491/11-1. Thanks to Mark Summerfield and Tjark Weber and to an anonymous reviewer for suggesting textual improvements.

[1] The name Nitpick is shamelessly appropriated from a now retired Kodkod ancestor.

Nonuniform representations. The encoding of HOL terms as Kodkod expressions is designed to take advantage of Kodkod's relational calculus. Functions are normally represented as relations constrained to be left-total and functional. However, predicates and other functions whose range has cardinality 2 are naturally represented as sets. Higher-order quantifiers are skolemized away when practicable. Any remaining quantified function is replaced by a k-element vector of values, where k is the cardinality of the function's domain. Vectors are also used for functions passed as arguments to other functions. Abstractions $\lambda x.\, f(x)$ are encoded as comprehensions $\{(x,y) \mid y = f(x)\}$.

Partiality. Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers prefixes $\{0, 1, \ldots, K\}$ of *nat* and maps numbers beyond K to the undefined value ($\bot$). Formulas of the form $\forall n :: nat.\, P(n)$ become $(\forall n \leq K.\, P(n)) \wedge P(\bot)$. Undefined values lead to a three-valued logic, which is encoded in terms of Kodkod's binary logic. If the formula evaluates to $\bot$, Nitpick reports the model as a potential counterexample and continues looking for a model that makes the formula evaluate to *True*. The user can instruct Nitpick to use Isabelle's automatic tactics to ascertain whether a potential counterexample is genuine.

Inductive datatypes and recursive functions. A typical example of an inductive datatype is α *list*, the type of lists with elements of type α generated by the constructors $Nil :: \alpha\ list$ and $Cons :: \alpha \to \alpha\ list \to \alpha\ list$. Nitpick's approach to inductive datatypes is inspired by the Alloy modeling of infinite datatypes done by Kuncak and Jackson [3]. Conceptually, each constructor is seen as constructing a different type, for which the user can specify a cardinality. For example, using a cardinality of 1 for *Nil* and 10 for *Cons*, Nitpick looks for all counterexamples that can be built using at most 11 different lists. The constructors are constrained relationally to ensure freedom and induction.

Functions that operate on inductive datatypes are often implemented by recursion. Instead of using the internal representation of functions synthesized by Isabelle's **primrec** and **function** packages, Nitpick relies on the more natural equational specification entered by the user.

Inductive predicates. Besides recursion, Isabelle also supports induction as a definition principle. An inductive predicate corresponds to a least fixed point. If there is only one fixed point (which can usually be proved automatically using Isabelle's termination tactics), Nitpick simply takes the fixed point equation as the specification of the predicate; for example, given the introduction rules $even\ 0$ and $even\ n \Longrightarrow even\ (n+2)$, Nitpick uses the equation $even\ n = (n = 0 \vee (\exists m.\, n = m + 2 \wedge even\ m))$. In the general case, Nitpick unrolls the predicate a given number of times, as in bounded model checking [2]. Because the unrolling is incomplete, the predicate typically evaluates to *True* or $\bot$, rarely *False*.

Function specialization. A simple but effective optimization is to eliminate fixed arguments. Consider the *map* function specified by $map\ f\ Nil = Nil$ and $map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)$. To falsify the formula $map\ h\ [y] = [h\ y]$, Nitpick introduces a new function map' defined by $map'\ Nil = Nil$ and $map'\ (Cons\ x\ xs) = Cons\ (h\ x)\ (map'\ xs)$ and attempts to falsify $map'\ [y] = [h\ y]$ instead. This drastically reduces the search space, especially for higher-order functions.

3 Evaluation

To assess Nitpick, we used a database of mutated formulas consisting mostly of non-theorems, as was done for Quickcheck [1]. Figure 1 summarizes the results of running Nitpick, Refute, and Quickcheck on 3 000 formulas derived from three theories: *List* is the standard theory of lists, *AVL* is a theory of AVL trees, and *POPLmark* is a formalization of System $F_{<:}$. Many of the formulas were not executable, in which case Quickcheck was not applicable. Refute's three-valued logic is unsound, so all counterexamples for formulas that involve an infinite type are potentially spurious. In contrast, Nitpick fared well on all kinds of formulas.

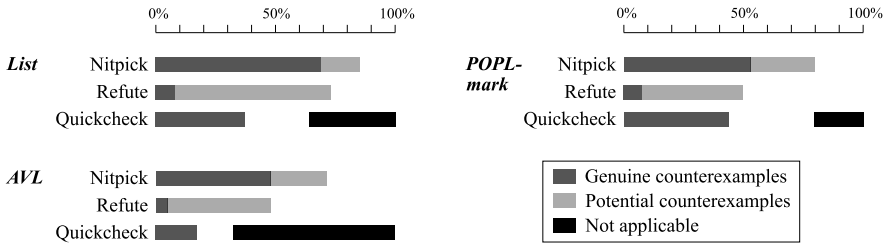

Fig. 1. Success rates of the counterexample generators on three theories

4 Conclusion and Future Work

Thanks to Kodkod and a nonuniform encoding scheme, Nitpick generates more counterexamples than similar tools, without restrictions on the form of the formulas. This means that Isabelle users can avoid a lot of wasted time spent trying to prove non-theorems. In addition, using Isabelle and Nitpick together provides a viable higher-order alternative to Alloy. Future work includes looking for encodings that would make it possible to handle much larger states, which arise when formalizing programming language semantics, and finding ways to rule out certain domain sizes through static analysis of the formula.

References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) *SEFM 2004*, pp. 230–239. IEEE C.S. (2004)
2. Biere, A., Cimatti, A., Clarke, E. M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) *TACAS 1999*, LNCS vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H. (ed.) *Proc. ESEC/FSE 2005*, pp. 207–216 (2005)
4. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*, LNCS vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
5. Weber, T.: *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T. U. München (2008)
6. Software: Nitpick. `http://isabelle.in.tum.de/~blanchet/#nitpick`

Tool Demonstration: Euclide *

Benjamin Cama, Arnaud Gotlieb, Guillermo Andrade-Barroso

INRIA Research Center of Rennes
35042 Rennes Cedex, France
{Benjamin.Cama, Arnaud.Gotlieb, Guillermo.Andrade-Barroso}@irisa.fr

Abstract. Euclide is an open source prototype tool that can help testing and verifying critical C programs. The prototype takes a C program as input, optionally annotated with assertions or post-conditions, and generates input test data that can reach specified locations within the code. Additionally, it can either prove that the assertions or post-conditions are verified, or proposes counter-examples to these properties. Euclide is mainly developed in Prolog and is accessible online through a web interface[1]. The demonstration of the tool will show the main functionalities of the tool on a small but realistic example.

1 Introduction

Modern program verification can be understood in some ways as the convergence of tests and proofs. Proving the correctness of a program offers guarantees on the functional correctness of an idealized semantics of the source code, while testing a program offers guarantees on the functional correctness of the binary code into its environment. Hence, these two activities, albeit often considered as opposite, should be considered as complementary as both serve the automatic verification of programs. Euclide is a prototype tool that features three applications in a single framework, namely automatic test data generation, counter-example generation and partial program proving [4]. The core algorithm of the tool takes as input a C program that respects some constraints, and a point to reach somewhere in the code. As a result, it outcomes either a test datum that reaches the selected point, or a certificate showing that the selected point is unreachable. Optionally, the tool takes as input additional safety properties that can be given under the form of pre/post conditions or assertions directly written in the code. In this case, Euclide can either prove that these properties or assertions are verified or find a counter-example when there is one. As these problems are undecidable in the general case, Euclide only provides a semi-correct procedure (when it terminates, it provides the right answer) for them and then it has to deal with non-termination problems. Euclide also implements several procedures that combine atomic calls to the core algorithm. For example, by selecting appropriate points to reach in the source code, the tool can generate a complete

* This work is partially supported by ANR through the CAVERN project under ref. ANR-07-SESUR-003

[1] http://euclide.gforge.inria.fr/

test suite able to cover the all_decisions criteria. It can also monitor the path coverage information related to a given test data.

2 Innovations of the Approach

2.1 History

The development of Euclide results from a long-term effort that had the objective to demonstrate the potential of constraint programming techniques to automate parts of the software testing process. More than ten years ago, we proposed to generate automatically test data for C programs by using Constraint Logic Programming [6,7]. We developed a goal-oriented approach for this problem while other works in this area were restricted to be path-oriented. Since then, we extended the scope of the approach by dealing with C programs containing floating-point computations [1], pointers towards named locations of the memory [8], dynamically allocated structures [2]. In 2002, we started working on the *oracle problem* [10] and proposed an extension of the approach for dealing with specification properties [5]. This openned the road for us to the usage of constraint programming techniques for verifying C programs against safety properties. And recently, an experiment on a real-world example (TCAS implementation) convinced us to the potential of these techniques for software verification too [4].

2.2 Scope

Euclide is dedicated to the testing and verification of safety-critical (and ISO/IEC compliant) C programs. These programs share some characteristics such as being written in a restricted subset of the C language that excludes recursion and dynamic memory allocation among other things. The C language, as defined by the ISO/IEC standard [9], has also the considerable advantage to be well defined in terms of syntax and semantics, even if several operations have still an undefined behavior[2] or a behavior defined by the implementation (in particular for floating-point computations). Euclide handles a subset of C that includes integer and floating-point computations, pointers towards named locations, arrays of statically-allocated size, structures, function calls, bit-to-bit operations such as masks, all control structures (including loops) and almost all operators (34 of 42). But, it has also many restrictions: it does not deal accurately with unstructured statements such as gotos, unconstrained pointer arithmetic (such as using a physical address of a memory segment or adding two unrelated addresses as if they were integers), function pointers, functions with a unknown number of parameters, volatiles, unions, memory type casting (such as reading an integer as it was an address), library and external function calls (unavailable source code). The tool has also intrinsic restrictions due to its usage of existing

[2] Exact behavior which arises is not specified by the standard, and exactly what will happen does not have to be documented by the C implementation.

constraint solving libraries. For example, it cannot deal accurately with integers encoded with more than 25 bits. We expect to leverage these restrictions in a near future.

2.3 An Online Prototype Tool

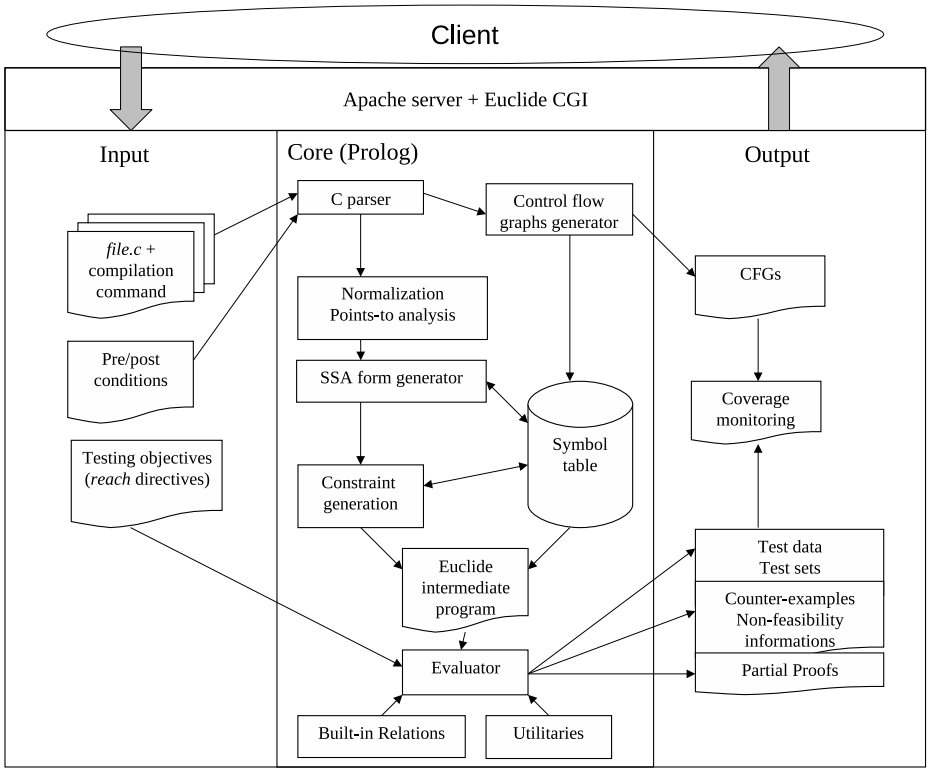

Fig. 1. Euclide's architecture

Euclide is accessible online through a web interface, meaning that C annotated programs can be uploaded and automatically verified online. The website generates test data that can be used to test the program after being compiled and linked within its own development chain. Note that the website does not offer facilities for compiling or executing these programs. We chose this implementation a few months ago in order to facilitate our demonstrations and experimentations. Note that offering access to Euclide through a web interface has the advantage of being independant for the user of third-party libraries such as those required to solve constraints. Anyone can use the tool without having

to install any component on its machine, and profit from the CPU power of the server platform. In addition, the development team of the tool can continuously enhance the core and interface of the tool without having to set up a release versions policy. Of course, this comes with some overhead on time required by the processing of the requests and communication with the server. Nevertheless, Euclide uses the quite unique ability of Prolog to save and restore its execution context between requests, needed because of the disconnected nature of the HTTP protocol. This way, the tool can be used interactively, like any client-side tool would be, without too much overhead for the server, and thus the user. Fig. 1 shows the components of the tool and its architecture. Details can be found in [4].

2.4 Proving Properties

From a source code, it is possible to derive a constraint model P of the program under test that not only captures its semantics but also extends it for backward reasoning. The model represents a potentially infinite set of (unbounded) behaviours. For loops, the model proposes a constraint reasoning operator that has no termination guarantee [3], but that does not require loop invariants to be specified. Any additional assertion or property S that should hold over the variables of the program can be added to the constraint model under negative form. Solving the constraint system $P \wedge \neg S$ yields two possible results: either a solution to the constraint system is found, meaning that there is a counter-example (a test datum) to property S, or the constraint system is shown as being unsatisfiable, meaning that property S is satisfied by any behaviour of P. In the latter case, we hold a proof of correction of P w.r.t. property S under both the hypothesis of the correction of our constraint model and the correction of the constraint solver.

3 Perspectives

There are still a number of indispensable improvments to perform on the core constraint solver of the tool. Good results highly depend on good optimizations in this matter. In addition, many C programs are currently simply rejected due to the narrow scope of the prototype that has to be extended. As of today, the web interface only exposes a small fraction of Euclide's capabilities. It could be expanded to add new ways of specifying goals and constraints, generates other forms of results, and better integrates with existing testing frameworks and tools. We hope that the way the tool can be enhanced and tested, through the web, we will be able to get quicker feedback from the users and that they will be able to try new features more easily.

References

1. B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):97121, 2006.

2. F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *TAIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.
3. T. Denmat, A. Gotlieb, and M. Ducasse. An abstract interpretation based combinator for modeling while loops in constraint programming. In *Proceedings of Principles and Practices of Constraint Programming (CP'07)*, Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.
4. A. Gotlieb. Euclide: A constraint-based testing platform for critical c programs. In *2th International Conference on Software Testing, Validation and Verification (ICST'09)*, Denver, CO, Apr. 2009.
5. A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proc. of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, Dallas, TX, USA, 2003. 3th to 7th November.
6. A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.
7. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *In First International Conference on Computational Logic*, pages 399–413. Springer Verlag, 2000.
8. A. Gotlieb, T. Denmat, and B. Botella. Constraint-based test data generation in the presence of stack-directed pointers. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, USA, Nov. 2005. 4 pages, short paper.
9. ISO/IEC 9899. Programming languages - C. Available at `www.open-std.org/JTC1/SC22/WG14/www/standards`, 1999.
10. E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, November 1982.

Incremental, two-level deadlock analysis for incomplete Java Card 3.0 programs

Rebekka Neumann, Michael Thies, and Uwe Kastens

University of Paderborn,
Fuerstenallee 11, 33102 Paderborn, Germany
rneumann@s-lab.upb.de
{mthies,uwe}@upb.de

Abstract. We present an approach that combines static program analysis and model checking to verify software that resides on smart cards. The Java Card 3.0 standard (JC 3.0) introduces concurrency as a new language feature in Java for smart cards. Hence, we have chosen deadlock detection as our first goal for verification.
A web server with its libraries burned on JC 3.0 smart cards is a realistic object for our analysis. From the viewpoint of analysis it is to be considered as an incomplete program that can be completed by JC 3.0 web applications loaded on the smart card. The costs of our deadlock analysis will be reduced by analysing the web server separately and by reusing the results when the completed program is analysed.
Since results of static program analysis are conservative, some of them indicate only potential deadlocks. We will further explore these cases by model checking to get definite answers. The presented analysis approach cannot substitute testing completely in the quality assurance process, but it can simplify and guide the testing activities for deadlock detection in JC 3.0 programs.
As a first result, we show that 50% of the static analysis results for the JC 3.0 web server will be reusable when analysing the completed program.

Key words: Deadlock detection, static program analysis, software model checking, testing process, incomplete programs

Motivating the detection of deadlocks in JC 3.0 programs: JC 3.0 is Sun's new standard for Java on smart cards. JC 3.0 smart cards are an internet-ready interface that contain a web server to manage mobile users' personal trusted data. Moreover, concurrency is a new language feature in Java for smart cards. Consequently, first, the quality assurance activities for JC 3.0 programs have to take concurrency properties like deadlocks into account. Second, they have to be able to handle the increased complexity of JC 3.0 programs e.g. by considering them as incomplete programs. The focus on deadlock detection in JC 3.0 programs allows for gaining more precise analysis results. First, we can exploit application-specific knowledge about the JC 3.0 web server, our analysis target. Second, the JC 3.0 setting can be handled more easily because of the limited JC 3.0 features, e.g. reflection is missing in JC 3.0. In our work, we

concentrate on detecting deadlocks caused by conflicting mutual exclusions and lock orderings. Speaking in terms of Java, we consider `synchronized` and `wait` statements as potential causes of deadlocks (this is as in [4]).

Motivating the development of a combined formal method for detecting deadlocks in incomplete JC 3.0 programs: Several methods for detecting deadlocks in Java exist. Williams et al. [4] propose a static program analysis for detecting deadlocks in Java libraries, which forbids common constructs, like callbacks from library to library clients and specialisation of library classes within library clients. The method is incomplete, thus produces false alarms. In contrast, software model checkers like Java PathFinder [3] (JPF) are complete, but less efficient, since they suffer from the state explosion problem. Moreover, JPF cannot detect *local* deadlocks (deadlocks that cause only *some* system threads to block), but only *global* ones. Havelund [1] uses runtime trace information to overcome this limitation. The reuse of static analysis results motivates the development of static analyses for incomplete programs. Thies [2] introduces a static program analysis for incomplete programs implemented in a program analysis framework named SAILDOWN that can be applied to arbitrary incomplete program slices.

Our combined formal method for detecting deadlocks in incomplete JC 3.0 programs: To our knowledge, deadlock detection for incomplete programs using a combination of static program analysis and model checking to improve the analysis result's precision has not been investigated yet. We propose an incremental, two-level combined formal analysis method to detect deadlocks in incomplete JC 3.0 programs (see figure 1). The incomplete JC 3.0 web server

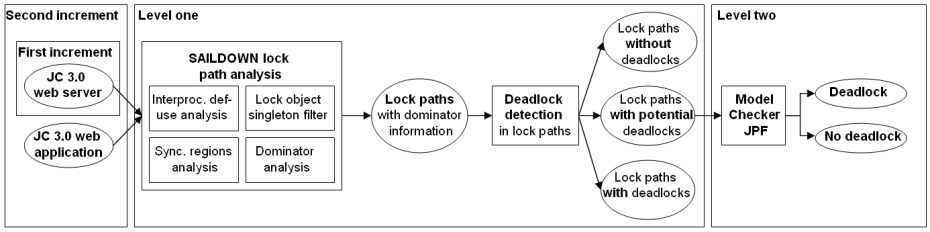

Fig. 1. Overview over incremental, two-level deadlock analysis

and the completed JC 3.0 web server containing a JC 3.0 web application serve us as analysis and evaluation targets for the incremental application of our deadlock analysis. Our deadlock analysis consists of the following levels. In level one, a static SAILDOWN program analysis will determine all lock paths completely contained in the closed (unaffected by JC 3.0 web applications) parts of the incomplete JC 3.0 web server. A lock path consists of all objects used for hierarchical locking starting at a JC 3.0 web server program entry point. All deadlocks contained in all possible combinations of these lock paths will be calculated. The same steps will be conducted for the JC 3.0 web server completed by a JC 3.0 web application in the second increment, but under reuse of the JC 3.0 web server's lock paths. In level two, the software model checker JPF will check all

deadlocks for which the static program analysis could not determine definite answers. It will use the lock paths leading to potential deadlocks to search the program's state space in a goal-orientied way. We will have to answer the following questions about our analysis method. First, is it worthwhile to reuse static analysis results? We have answered this question qualitatively (see next paragraph), and still have to answer it quantitatively (with respect to memory usage and runtime). Second, does the static analysis information given to the model checker allow for faster checking of local and global deadlocks?

Evaluating the efficiency of our deadlock analysis: The goal of the simple SAILDOWN analysis we conducted is to determine the percentage of reusable analysis results of the JC 3.0 web server (including the JC 3.0 API). Therefore we have modeled a partial analysis system consisting of four data flow elements ordered in a linear chain which constitute the constant part of the methods' locking behaviour. In order to solve the partial analysis system, a single aspect model (SAM) is calculated for each method of the JC 3.0 web server reflecting the direct effect of the method's body on its locking behavior as a constant term and the effects of the callees as applications of further SAMs. The effects of a dynamically bound call site are expressed as a method family single aspect model (MF-SAM) in the solution of the partial analysis system. All SAMs and MF-SAMs are simplified as far as possible while calculating the analysis result. For the JC 3.0 web server about 50% of the analysis results are fixed and hence can be reused when analysing the JC 3.0 web server together with a JC 3.0 web application. 13% of the SAMs and MF-SAMs lock objects, whereof 25% are fixed, so that the incremental application of our deadlock analysis to an extended analysis context seems reasonable and effective.

Conclusions and Perspectives: We have introduced an incremental, two-level deadlock analysis combining static program analysis and model checking. The reusability of 50% of the static program analysis results warrants further exploration of this approach. As a first step, we have already expanded the basic analysis to calculate synchronisation regions and associated sets of lock objects for all methods. The next steps of our work consist of modeling the SAILDOWN analysis to calculate the JC 3.0 web server's and a JC 3.0 web application's lock paths, to analyse these lock paths for deadlocks and to model check all potential deadlocks. We will then be able to integrate our deadlock analysis tool into the testing process of JC 3.0 programs in order to simplify it.

References

1. K. Havelund. Using runtime analysis to guide model checking of Java programs. pages 245–264. Springer, 2000.
2. M. Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization.* PhD thesis, University of Paderborn, 2001.
3. W. Visser and K. Havelund. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12. Press, 2000.
4. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.