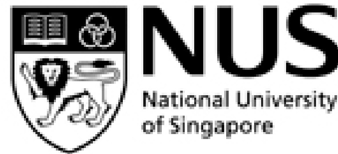
THE NATIONAL UNIVERSITY
of SINGAPORE



School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA5/14

FM2014: Formal Methods Doctor Symposium

Ana Cavalcanti, Francesco Alberti, William Denman, M.S. Conserva Filho, Lin Gui, Nguyen Thien Binh, Quan Thanh Tho, Nguyen Minh Hai, Nguyen Huu Vu, Nguyen Cong Dinh and Maya Retno Ayu Setyautami

May 2014

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

David ROSENBLUM
Dean of School

FM 2014:

Formal Methods

Doctor Symposium

Singapore, May 13, 2014

NUS
National University
of Singapore

School of Computing

NUS SOC Technical Report
Number TRA5/14
May, 2014

School of Computing
National University of Singapore
13 Computing Drive
Singapore 117417
Telephone: +65 6516 2727
Fax: +65 6779 4580
Homepage: http://www.comp.nus.edu.sg

Preface

This volume contains the papers presented at the Doctoral Symposium which was held in Singapore on 13 May 2014 as part of the International Symposium on Formal Methods (FM2014).

The Doctoral Symposium is an important event where students have the opportunity to present their ideas at an international forum. The papers selected for this year's symposium were reviewed carefully by the Programme Committee members:

- Franck Cassez (NICTA, Sydney Australia)
- Wei-Ngan Chin (National Univ of Singapore)
- Christine Choppy (Laboratoire d'Informatique de l'Université Paris Nord)
- Yuan Feng (University of Technology, Sydney, Australia)
- Shang-Wei Lin (Temasek Laboratories, National University of Singapore)
- Graeme Smith (University of Queensland, Australia)
- Elena Troubitsyna (Åbo Akademi, Finland)
- Huibiao Zhu (Software Engineering Institute, East China Normal University)

The symposium invited Prof. Ana Cavalcanti as distinguished speaker, whose abstract of her talk *Can Java ever be safe?* is included in this proceedings.

I would like to thank the authors of the submitted papers, Prof. Cavalcanti and the members of the Programme Committee for their contributions, and finally I thank all symposium participants for making this event fruitful and worthwhile.

Annabelle McIver
FM Doctoral Symposium Chair.

Table of Contents

Can Java Ever be safe? **1**
Ana Cavalcanti

A SMT-based verification framework for software systems handling unbounded arrays **5**
Francesco Alberti

Formal Verification of Nonpolynomial Hybrid Systems by Qualitative Abstraction **11**
William Denman

Livelock Analysis for Component-Based Systems **16**
M. S. Conserva Filho

Reliability Analysis of Non-deterministic Systems **21**
Lin Gui

HOPE: A Framework for Handling Obfuscated Polymorphic Malware **26**
Nguyen Thien Binh, Quan Thanh Tho, Nguyen Minh Hai, Nguyen Huu Vu, Nguyen Cong Dinh

Verification of Refactoring with Delta Representation? **31**
Maya Retno Ayu Setyautami

Can Java ever be safe?
Abstract

Ana Cavalcanti

University of York, UK

For its popularity, both in academia and industry, Java [10] is a language that dispenses introductions. It has a wide base of programmers, an impressive collection of libraries, and continues to evolve with the support of very large number of companies. The real-time community has, however, at first, completely rejected it. Issues included poor support for absolute time, timeouts, management of threads waiting on a lock, and priorities.

Ten years later, the tremendous success of Java lead to a reversal of this situation: the Real-Time Specification for Java (RTSJ) [21] was developed to address the main concerns of the community. It adapts and extends the programming model of Java to support real-time programming abstractions and adequate memory management based on the use of scoped memory areas, which are not subject to garbage collection. With these developments, it is possible to write Java programs with predictable time properties.

This is essential to allow the use of Java in the context of safety-critical applications, since the requirements for many of them involve timed as well as functional properties. RTSJ is not enough in this application domain, though. Some sort of certification is typically needed, and that prescribes a controlled engineering process to obtain programs that are reliable, robust, maintainable, and traceable. For that, the safety-critical industry usually resorts to controlled language subsets [1, 17]. In this context, RTSJ is far too rich, encompassing the whole of Java as well as the novel constructs to support real-time programming.

To address this issue, an international effort has more recently produced an Open Group standard for a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [14]. It is a subset of RTSJ; its execution model is based on missions and event handlers, and it restricts the memory model to prohibit use of the heap and define a policy for the use of memory areas. The SCJ design is organised in Levels (0, 1, and 2), with a decreasing amount of restrictions.

The standardisation work includes the production of a reference implementation, but no particular application-design or verification technique. We have addressed this using the *Circus* [4] family of languages for refinement [7]. They are based on a flexible combination of elements from Z [23] for data modelling, CSP [19] for behavioural specification, and standard imperative commands from Morgan's calculus [16]. Variants and extensions of *Circus* include *Circus Time* [20], which provides facilities for time modelling from Timed CSP [18], and *OhCircus* [5], which is based on the Java model of object-orientation.

Circus has been used for modelling and verification of control systems specified in Simulink [3, 15], including virtualisation software by the US Naval Research Laboratory [9]. The semantics of the *Circus* family of languages is based

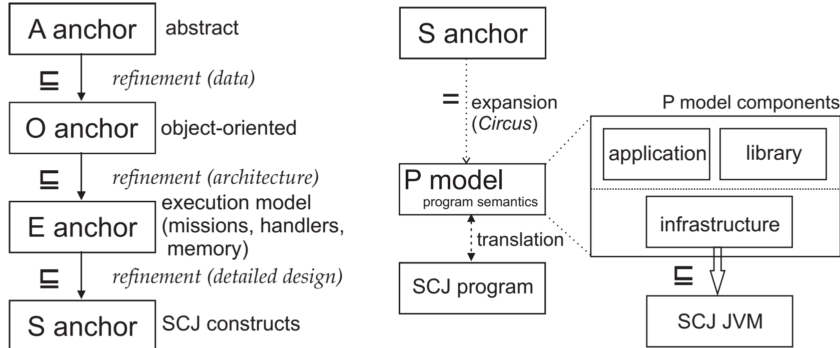


Fig. 1: Our approach to development and verification

on the Unifying Theories of Programming (UTP) [13]. This is a framework that supports refinement-based reasoning in the context of a variety of programming paradigms. It supports the independent treatment of programming theories, with associated techniques for their combination. This makes it possible for us to consider a rich language for refinement that supports the use of object-oriented and SCJ constructs as well as the modelling and verification of time properties.

A *Circus*-based formalisation of the SCJ execution model [24] and a UTP theory for the SCJ memory model [6] are available. In [7], we have presented a refinement strategy for deriving SCJ programs from *Circus* specifications that builds on these results and UTP theories for references [22, 11, 5].

Our refinement strategy supports the stepwise development of SCJ Level 1 programs based on specification models that do not consider the details of either the SCJ mission or memory models. SCJ Level 1 corresponds roughly to the Ravenscar profile for Ada [2]. It is not as restrictive as Level 0, which is based on a cyclic executive programming model, but is controlled enough to impose a reasonable challenge in the development of a programming theory.

As shown in Figure 1, four *Circus* specifications characterise the major development steps of our strategy. We call them anchors, as they identify the (intermediate) targets for refinement and the design aspects treated in each step. Each anchor is written using a different combination of the *Circus* family of notations. The first anchor is the abstract specification model. The last is so close to an SCJ program as to enable automatic code generation. It is written in *SCJ-Circus*, a new version of *Circus* extended with constructs that correspond to the components of the SCJ programming paradigm. They are syntactic abbreviations for definitions introduced in [24] to characterise the SCJ infrastructure and applications; they use a combination of the variants of *Circus* to cater for time, object-orientation, and the SCJ memory model.

By extending *Circus* with SCJ constructs, we can model SCJ programs in the unified framework of a refinement language. We are tackling translation from *SCJ-Circus* model to Java code as separate work in line with what we have

previously achieved for low-level *Circus* models and corresponding Java implementations [8]. Another interesting line of work uses the part of the model of an *SCJ-Circus* program that gives an abstract specification of the SCJ paradigm to support verification of an SCJ virtual machine.

Our development strategy establishes, by construction, that the *SCJ-Circus* model is a refinement of the specification used as the first anchor. This means that safety, liveness, and timing properties are preserved. Safety requires that the sequences of interactions (traces) of the program are possible for the specification. Liveness requires that deadlock or divergence in the program can occur only if allowed in the specification. Finally, preservation of the timing properties requires that the deadlines and budgets defined in the specification are enforced by the deadlines and budgets defined for the components of the program. Our long-term goal is to provide for Safety-Critical Java at least the same level of support that the SPARK tools, for instance, provide for Ada.

Regarding time, our strategy makes use of decomposition via refinement. It is inspired by the work in [12], which introduces time into Morgan's refinement calculus so that derivation of code from specifications is similar to that for untimed specifications. In our approach, the requirements in the first anchor are localised in the SCJ components of the final target anchor. It is, in this way, annotated with the machine-independent timing requirements that every correct implementation (for a specific platform) needs to satisfy. Verifying that they do may require, for instance, schedulability analysis.

So, can Java ever be safe? Can we ever consider the use of Java for programming safety-critical applications? The answer is "a version of Java, namely, SCJ, is already being used in this domain". As explained, it is necessary to perform some essential changes and restrictions to Java. With these, however, it is possible to achieve certification and it is possible to develop a programming theory to support formal development and verification. The work is far from complete: the combination of theories is under development, a catalogue of laws is under consideration, case studies are essential, as are supporting tools.

References

1. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
2. A. Burns. The Ravenscar Profile. *Ada Letters*, XIX:49 – 52, 1999.
3. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 – 181, 2003.
5. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277 – 296, 2005.
6. A. L. .C Cavalcanti, A. Wellings, and J. C. P. Woodcock. The Safety-Critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37 – 57, 2013.
7. A. L. C. Cavalcanti, F. Zeyda, A. Wellings, J. C. P. Woodcock, and K. Wei. Safety-critical Java programs from *Circus* models. *Real-Time Systems*, 2013. Online first. 10.1007/s11241-013-9182-4.

8. A. F. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 115 – 130. Springer-Verlag, 2006.
9. L. Freitas and J. P. McDermott. Formal methods for security in the xenon hypervisor. *International Journal on Software Tools for Technology Transfer,*, 13(5):463 – 489, 2011.
10. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
11. W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Theory of Pointers for the UTP. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 141 – 155. Springer-Verlag, 2008.
12. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385 – 448, 2001.
13. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. *Safety Critical Java Specification, First Release 0.76*. The Open Group, UK, 2010. `jcp.org/aboutJava/communityprocess/edr/jsr302/index.html`.
15. A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10 – 11):1151 – 1177, 2012.
16. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
17. Motor Industry Software Reliability Association Guidelines. Guidelines for Use of the C Language in Critical Systems. 2012.
18. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
20. A. Sherif, A. L. C. Cavalcanti, J. He, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153 – 191, 2010.
21. A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
22. J. C. P. Woodcock. The miracle of reactive programming. In *Unifying Theories of Programming 2008*, Lecture Notes in Computer Science, pages 202 – 217. Springer-Verlag, 2009.
23. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.
24. F. Zeyda, A. L. C. Cavalcanti, and A. Wellings. The Safety-critical Java Mission Model: a formal account. In *International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, 2011.

Acknowledgements This work is funded by EPSRC grant EP/H017461/1. Much of what is described here has been developed jointly with Frank Zeyda, Andy Wellings, Jim Woodcock, and Kun Wei. Chris Marriott and Neeraj Singh have contributed with useful discussions.

A SMT-based verification framework for software systems handling unbounded arrays*

Francesco Alberti

University of Lugano, Switzerland
VERIMAG, Grenoble, France

Abstract. We target the formal, SMT-based, safety analysis of software systems handling arrays of unknown length. We show how to lift state-of-the-art formal verification techniques such as Lazy Abstraction with Interpolants and Acceleration to a quantified level suitable for the analysis of systems handling arrays and how to integrate them in a declarative SMT-based framework for the effective analysis of programs handling these unbounded data-structures.

1 Introduction

The aim of this PhD work is to study a new, general and automatic framework for the verification of safety invariant properties of systems handling arrays of unknown length. These are fundamental data structures widely used in computer science. We formally represent programs as triples $\mathcal{S}_T = (\mathbf{v}, I(\mathbf{v}), \tau(\mathbf{v}, \mathbf{v}'))$ where T is a first-order theory, $\mathbf{v}$ is the set of program variables (thus comprising variables of type array), $I(\mathbf{v})$ and $\tau(\mathbf{v}, \mathbf{v}')$ are T-formulæ representing the initial configuration of the analyzed system and how it evolves, respectively[1]. This representation allows several different applications of our techniques, although in this PhD work we will consider only programs handling arrays of unknown length, e.g., the one given in Fig. 1. This program can be formally represented by the transition system identified by $\mathbf{v} := (pc, \mathtt{a}, \mathtt{i}, \mathtt{size}, \mathtt{value})$ where pc is a fresh variable introduced to model the control-flow graph of the procedure, $I(\mathbf{v}) := pc = 1$ and $\tau(\mathbf{v}, \mathbf{v}') := \tau_1 \vee \tau_2 \vee \tau_3$ where[2]

$$\tau_1 := pc = 1 \wedge \mathtt{i}' = 0 \wedge pc' = 2$$
$$\tau_2 := pc = 2 \wedge \mathtt{i} < \mathtt{size} \wedge \forall j. \mathtt{a}'[j] = \mathsf{if}\ (j = \mathtt{i})\ \mathsf{then}\ \mathtt{value}\ \mathsf{else}\ \mathtt{a}[j]$$
$$\tau_3 := pc = 2 \wedge \mathtt{i} \geq \mathtt{size} \wedge pc' = 3$$

Given a T-formula P, the validity of a safety invariant property $\Box P$ for a system $\mathcal{S}_T$ can be proven by showing that $\mathcal{S}_T$ cannot reach a configuration satisfying $\neg P$. The reachability analysis terminates either if it discovers a feasible execution of $\mathcal{S}_T$ reaching $\neg P$ or if it generates a *safe inductive invariant*. This

*Supported by Swiss National Science Foundation under grant no. P1TIP2_152261.

[1] In general $\tau(\mathbf{v}, \mathbf{v}') \equiv \bigvee_{i=1}^{n} \tau_i(\mathbf{v}, \mathbf{v}')$, where each $\tau_i(\mathbf{v}, \mathbf{v}')$ encodes, e.g., a branch of the control-flow graph of the input program.

[2] For any $v' \in \mathbf{v}'$ not explicitly mentioned in τ_1, τ_2, τ_3, we assume that $v = v'$.

```
void array_init ( int a [ ] , int size , int value ) {
  int i = 0;
  while ( i < size ) {
    a[ i ] = value;
    i = i + 1;
  }
}
```

Fig. 1. The `array_init` procedure.

is a logical formula H representing (an over-approximation of) the state-space reachable by $\mathcal{S}_T$ (hence T-entailed by I), closed under post-image computation along τ and T-entailing P. Safe inductive invariants proving the safety of systems handling arrays with respect to array properties can be expressed only by exploiting quantification. This invalidates existing automatic state-of-the-art solutions adopted in verification because of their quantifier-free nature (e.g., [1, 7] to cite a few well-engineered software model-checkers). Consider again the procedure `array_init` of Fig. 1 and let $P(\mathbf{v})$ be the formula

$$pc = 3 \rightarrow \forall x.((0 \leq x \wedge x < \mathtt{size}) \rightarrow \mathtt{a}[x] = \mathtt{value})$$

The safe inductive invariant proving that our formal model of the `array_init` procedure satisfies $\Box P(\mathbf{v})$ is the *quantified* formula

$$(pc = 2 \rightarrow \forall x.((0 \leq x \wedge x < \mathtt{i}) \rightarrow \mathtt{a}[x] = \mathtt{value})) \wedge$$
$$(pc = 3 \rightarrow \forall x.((0 \leq x \wedge x < \mathtt{size}) \rightarrow \mathtt{a}[x] = \mathtt{value}))$$

We plan to achieve our research goal by (i) "lifting" two orthogonal state-of-the-art techniques for the formal analysis of systems, i.e., *Lazy Abstraction with Interpolants* and *Acceleration*, to a quantified level (suitable for arrays) and (ii) devising an integrated SMT-based framework for these two techniques.

The most precise safe inductive invariant showing the safety of a system cannot be computed in general. Abstraction-based solutions target the generation of a safe inductive invariant over-approximating the set of configuration reachable by the input system. The final invariant is expressed as Boolean combination of a set of *predicates* determined by an *abstract domain* [12] or iteratively refined [11]. Abstraction is thus a general solution for the analysis of systems. Several heuristics are usually required in order to increase its practical effectiveness, though. Acceleration instead provides a precise solution (not over-approximated) for the analysis of systems but might be applicable to a smaller set of examples because of well-known theoretical limitations[3]. Our research hypothesis is that abstraction and acceleration are mutually beneficial. Investigating how to successfully combine them will therefore lead to the definition of a verification integrated framework combining their strengths while overcoming their individual limitations, with the ultimate aim of allowing a rigorous verification of systems handling arrays of unknown length.

[3] Acceleration relies on the computation of the transitive closure of relations encoding cyclic actions. Transitive closure, in general, is not definable in first-order logic.

Related work. Abstraction solutions for programs with arrays require quantified predicates. Known solutions targeting their generation either require the manual suggestion of templates (e.g., [14]), or restrict themselves to the generation of formulæ of a certain shape (e.g., [18]). Abstract Interpretation approaches (e.g., [13, 16]) generate invariants not ensured to be safe. First-order theorem provers can also produce quantified array properties [20], once provided the prover with the axioms for handling arithmetic. In the area of acceleration, to the best of our knowledge, the only work dealing with arrays is [8], although such work seems to be limited to the verification of properties with at most one quantified variable.

2 Research Methods

We first need to identify a *quantified fragment* $\mathcal{F}$ of our background theory T allowing the definition of quantified safe inductive invariants and, at the same time, supporting a mechanical and automatic verification procedure.

The Model Checking Modulo Theories [15] framework (MCMT) offers a good starting point for our investigation. It is a fully declarative SMT-based framework based on a *backward* reachability analysis procedure. It suggests a suitable fragment $\mathcal{F}$ meeting our requirements of expressiveness and computability, i.e., the fragment of *existentially flattened formulæ* of T. These are existentially quantified formulæ where arrays are indexed only by quantified variables. Negation[4] of array properties of interest for the analysis of programs with arrays belong to $\mathcal{F}$. Certain classes of formulæ representing T-entailment tests between $\mathcal{F}$-formulæ, required for terminating the reachability analysis, admit decision procedures [6, 10, 17]. The MCMT framework fails when applied to the analysis of the systems we are interested in, though, given its inability in generating predicates for the definition of suitable quantified safe inductive invariants. Our aim is to leverage the MCMT framework to integrating the new abstraction and acceleration techniques.

The *Lazy Abstraction with Interpolants* (LAWI) approach [21] is one of the most efficient frameworks for software analysis. It exploits Craig interpolants for refining the predicates over which the explored state-space is (lazily) abstracted. Interpolants are computed from unsatisfiable formulæ ϕ logically encoding executions $\hat{\pi}$ of an abstraction $\hat{\mathcal{S}}_T$ reaching $\neg P$ but not reproducible by the concrete $\mathcal{S}_T$. Given that the theory of array does not admit quantifier-free interpolation [19], Craig interpolants can be arbitrary quantified formulæ in our context. This kills the regularity with respect to $\mathcal{F}$, preventing practical effectiveness.

By leveraging the MCMT framework, preimage computation automatically generates quantified predicates (belonging to $\mathcal{F}$) over which the state-space is abstracted [2, 15]. We, therefore, have to investigate a new refinement procedure for systems handling arrays. In presence of array variables, formulæ ϕ represent-

[4]The MCMT approach is "by refutation". That is, given a safe inductive invariant H generated by the MCMT framework, $\neg H$ is the one for the input system.

ing executions $\hat{\pi}$ have quantifiers[5]. We adopt a quantifier-instantiation procedure in charge of generating from ϕ an equisatisfiable quantifier-free formula ϕ'. Counterexamples of transition systems $\mathcal{S}_T = (\mathbf{v}, I(\mathbf{v}), \tau(\mathbf{v}, \mathbf{v}'))$ are generally represented by formulæ of the kind $I(\mathbf{v}^{(0)}) \wedge \tau_{i_1}(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \ldots \wedge \tau_{i_n}(\mathbf{v}^{(n)}, \mathbf{v}^{(n+1)}) \wedge \neg P(\mathbf{v}^{(n+1)})$, where $\mathbf{v}^{(k)}$ represents a copy of the tuple $\mathbf{v}$ with all the variables primed k times. The procedure starts from $\neg P(\mathbf{v}^{(n+1)})$, which can be only a quantifier-free or an existentially quantified formula[6]. This formula is Skolemized and the Skolem constants introduced are saved in a set Δ. Δ represents, intuitively, the positions of the arrays that one has to consider to reproduce the counterexample. At every transition τ_{i_l} the universal quantifier expressing array updates is instantiated over Δ, and Δ is subsequently enriched with new Skolem constants coming from the (flattened) guard of τ_{i_l}[7]. The T-satisfiability of this formula is decidable and ϕ' belongs to a fragment of T admitting quantifier-free interpolation [2]. This preserves the desired regularity with respect to $\mathcal{F}$. Unsatisfiable formulæ admits many interpolants, though, and generating "bad" interpolants may easily lead to divergence. For this we devised a heuristic, *term abstraction*, with the purpose of limiting the presence of certain undesired terms (provided in a *term abstraction list*) in the interpolants. Such terms are guilty of keeping the interpolants too precise and not effective in limiting divergence. Our new LAWI framework showed good results on challenging programs with arrays [3], although it might require a user-defined term abstraction list.

We now turn our attention to *acceleration*. Since this approach is affected by a rather restrictive theoretical limitation, definability of accelerations is usually achieved by identifying syntactic restrictions for the relations encoding cyclic actions allowing the first-order definability of their accelerations (e.g., [9]). Following such reasoning schema, we also seek for a class of relations over arrays encoding cyclic actions (e.g., loops of programs) allowing the first-order definability of the transitive closure. The core of the problem is understanding, given a τ_i encoding a cyclic action, whether a cell z of an array a will be read and/or modified in y consecutive applications of τ_i. This can be understood by exploiting *iterators* and *selectors*, formalisms for expressing how τ_i handles scalar variables for indexing its arrays variables [5]. Based on the notions of iterators and selectors, we defined a class of relations over arrays – called *local ground assignments* (LGAs) – admitting first-order definable acceleration. Of course not all loops of programs with array can be modeled with LGAs[8]. However the motivation for studying acceleration is devising a technique for limiting divergence of our new LAWI framework for arrays. Therefore even though our technique cannot accel-

[5]Quantifiers are needed to express updates of arrays (which are modeled as uninterpreted functions) by exploiting if-then-else SMT-LIB constructs.

[6]We target the verification of "standard" assertions or universally quantified ones.

[7]Consider the transition τ_2 of our `array_init` example. Its flattened form is $\exists x.(pc = 2 \wedge x < \mathtt{size} \wedge x = \mathtt{i} \wedge \forall j.\mathsf{a}'[j] = \mathsf{if}\ (j = x)\ \mathsf{then}\ \mathtt{value}\ \mathsf{else}\ \mathsf{a}[j])$

[8]Iterators and selectors preserve a kind of injectivity. That is, a τ_i that would admit more than one modification of a certain position of an array (or even a scalar variable) in its ideal acceleration cannot be modeled as LGAs.

erate all the loops of a program, it would still provide a schema for alleviating the burden on the LAWI framework and limiting its divergence.

The new acceleration results is template-based: the computation of accelerations is performed based on templates for iterators and selectors and can be adopted as a preprocessing step of the LAWI framework described before. Notably, accelerations of LGAs have alternations of quantifiers. This implies that preimages of accelerations of LGAs will be outside the fragment $\mathcal{F}$. For a successful integration of our two new frameworks, we defined ad-hoc procedures in charge of taking care of such problematic preimages. These are over-approximated with their *monotonic abstractions* [4] which are formulæ belonging to $\mathcal{F}$. Ad-hoc refinement procedures (proposed in [5]) will subsequently take care of spurious path due to such over-approximation.

Beside providing practical advantages when combined with our LAWI framework [5], we recently showed that acceleration allows to identify a class of array-manipulating programs for which the safety is decidable [6].

3 Current stage of research and Future work

So far we addressed the theoretical problems of lifting the LAWI approach and acceleration techniques to a quantified level suitable for arrays. We also achieved promising preliminary experimental results showing that (i) the new LAWI framework is effective on a significant set of programs handling arrays [3] and that (ii) acceleration and abstraction are mutually beneficial [5].

We are still working on our tool. It has been built according to the standard compilers architecture, where the initial parsing phase generates an intermediate representation of the code which is subject to several optimizations before being fed to an engine for checking its safety. From this point of view, acceleration can be viewed as the most important and distinguishing optimization of our approach, while the new LAWI framework acts as the engine performing the analysis. We have implemented and plan to implement other optimization procedures, mainly coming from the compilers or abstract interpretation literature. Techniques able to generate inductive properties of the input code at compile time (as those presented in [13,16]) can definitely alleviate the load on the LAWI framework. They provide an initial set of predicates over which the input system can be abstracted. Such predicates can be of great help for the LAWI framework, as they prevent the analysis of trivially infeasible abstract execution $\hat{\pi}$. Discovering such properties with a refinement procedure might require several heuristics.

In conclusion, this PhD work will present a new SMT-based verification framework for software handling arrays of unknown length. The framework will integrate a new Lazy Abstraction with Interpolants approach with an Acceleration-based procedure, both enhanced to deal with array variables.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *CAV*, volume 7358

of *LNCS*, pages 672–678. Springer, 2012.
2. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.
3. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *CAV*, volume 7358 of *LNCS*, pages 679–685. Springer, 2012.
4. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G.P. Rossi. Universal guards, relativization of quantifiers, and failure models in Model Checking Modulo Theories. *JSAT*, 8(1/2):29–61, 2012.
5. F. Alberti, S. Ghilardi, and N. Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In *FroCos*, volume 8152 of *LNCS*, pages 23–39. Springer, 2013.
6. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for Flat Array Properties. In *TACAS*, volume 8413 of *LNCS*, pages 15–30. Springer, 2014.
7. D. Beyer and M.E. Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.
8. M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, volume 5643 of *LNCS*, pages 157–172. Springer, 2009.
9. M. Bozga, R. Iosif, and F. Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, volume 6174 of *LNCS*, pages 227–242. Springer, 2010.
10. A.R. Bradley, Z. Manna, and H.B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
11. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
13. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
14. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
15. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
16. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.
17. P. Habermehl, R. Iosif, and T. Vojnar. A logic of singly indexed arrays. In *LPAR*, volume 5330 of *LNCS*, pages 558–573. Springer, 2008.
18. R. Jhala and K.L. McMillan. Array abstractions from proofs. In *CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.
19. D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for data structures. In *SIGSOFT FSE*, pages 105–116. ACM, 2006.
20. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
21. K.L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

Formal Verification of Nonpolynomial Hybrid Systems by Qualitative Abstraction

William Denman

Computer Laboratory, University of Cambridge, UK
william.denman@cl.cam.ac.uk

Abstract. Few methods can automatically verify nonlinear hybrid systems that are modelled by nonpolynomial functions. Qualitative abstraction is a potential alternative to numerical reachability methods for formally verifying the safety of these systems. The state-of-the-art qualitative algorithms depend critically on decision procedures that can prove sentences of first order logic over the theory of the Reals. MetiTarski, an automated theorem prover for transcendental functions, fits this role perfectly and has been successfully used for constructing sound abstractions of nonlinear dynamical systems.

1 Problem Outline

Examples of *hybrid systems* include self-driving cars and autonomous drones. As these complex systems are becoming prevalent, fast and efficient methods for safety verification are now more important than ever. The formal verification of hybrid systems is inherently computationally intractable. This is due to the interplay of continuous variables, which vary over the infinite field $\mathbb{R}$, and discrete variables, which introduce nondeterminism. Consequently, determining whether the system can reach an unsafe state is extremely difficult.

Since hybrid systems lie at the interface of the physical world, transcendental and special functions naturally arise in modelling their behaviour. Angular measurements might involve sine, cosine, tangent and related transcendental functions. Several types of friction or drag can involve the exponential function. However, there is no clear choice as to which method is best suited for formally verifying such nonpolynomial hybrid systems.

The contribution of my thesis is the development of an enhanced qualitative abstraction method, which uses the automated theorem prover MetiTarski [1] to discretize nonpolynomial hybrid systems, while performing an on-the-fly reachability analysis. Preliminary results indicate that this verification method is competitive on several nonlinear nonpolynomial hybrid system problems and demonstrates that MetiTarski is powerful enough to discharge theorems arising from the abstraction process.

MetiTarski is an automated theorem prover for arithmetical conjectures involving first-order inequalities that contain transcendental functions. It has been successful in proving theorems arising from the verification of analogue circuits

[2], linear hybrid systems [3] and aircraft stability [4]. Recent improvements to MetiTarski include taking advantage of the nonlinear solver within Z3 [5] for Real Closed Field (RCF) decision calls, enabling proofs of nonlinear systems containing up to 11 continuous variables. The advanced RCF decision procedures within Mathematica, that can handle transcendental functions directly, have also been integrated to aid in proofs containing equalities. The continued improvement of MetiTarski has contributed significantly to the success of the abstraction method developed for my thesis.

2 Literature Survey

Safety, the fact that some bad behaviour will never happen, is perhaps the most important property that should be verified for a system. The reachability computation remains the most common way to check safety of a system. One approach is to iteratively compute over-approximations of reachable states [6], while another is to abstract the state space [7] or transition relation [8]. Though significant advancements have been made, most tools, even state-of-the-art ones, have difficulty dealing with hybrid systems that contain transcendental functions in the definition of the vector field (system of differential equations) or in the transition guards. Moreover, the tools that do support transcendental functions are restricted to techniques such as interval differential equation solving [9], bounded model checking [8] or linearization [10].

The focus of my thesis is on abstraction techniques, which reduce the complexity of the system but at the same time preserve the properties under verification. Sloth and Wisniewski [11] developed a method for creating a sound and complete abstraction of continuous system using Lyapunov functions. By using the Lyapunov function as a predicate for partitioning, they were able to convert the infinite state space of a continuous system into timed automata. The motivation was to use tools, such as UPPAAL [12] and KRONOS [13], that can automatically check properties of timed automata [12]. One issue with this work is that Lyapunov functions are generally difficult to find, although SOSTOOLS [14] can help in the search. As well, to ensure soundness and completeness they focus on a very restricted class of linear systems.

There are several restricted classes of hybrid systems and continuous systems that have been shown to have decidable reachability properties but most are too weak for practical applications. Another method of abstraction borrows from the domain of qualitative reasoning. Qualitative reasoning is motivated by the idea that numerical simulation is limited when not all the parameters of the system are known. Instead of trying to compute a solution, it is sufficient to look at how the vector field itself changes with time. Tiwari [15] uses predicates that evaluate over the three symbols $\{+, -, 0\}$ to split up the infinite state space. This construction of the abstraction uses the decidability of the first order theory of real closed fields [16] to compute the transitions between abstract states. Once the abstraction is created then a model checker is used to evaluate CTL properties on the abstract system. For linear systems it is easy to choose the

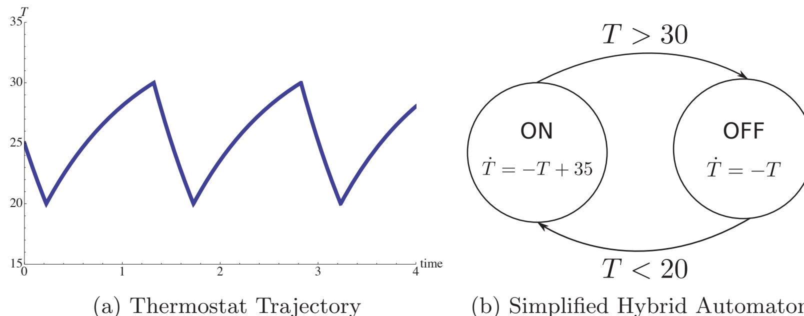
predicates, but nonlinear systems still require a search that relies mostly on heuristics.

3 Progress

A common formalism used for modelling hybrid systems is the *hybrid automaton*. An example hybrid automaton modelling a thermostat is shown in Figure 1b. Each state defines the continuous behaviour of the system governed by a set of differential equations and the conditions (guards) on the edges define the discrete behaviour (jumps) of the system.

Example 1 (Hybrid System). A thermostat has two modes of operation, heating (on) and cooling (off). In Figure 1a the trajectory of a thermostat is plotted, with temperature vs time. The initial temperature is set to 25°C. When the temperature rises above 30°C the thermostat shuts off, when the temperature decreases below 20°C degrees, the thermostat turns on. In each mode, continuous evolution is described by a separate differential equation.

Verifying hybrid automata is quite difficult. A well known theoretical result is that the reachability question, *"Does there exist a run of the hybrid automaton that can reach an unsafe state?"*, is undecidable [17]. Standard transition system verification techniques, such as model checking, will therefore not work.



(a) Thermostat Trajectory (b) Simplified Hybrid Automaton

Fig. 1: Hybrid System

One alternative approach to verifying reachability properties of hybrid automata is based on discretizing the continuous state space into distinct cells. Then, using properties of the underlying vector field, identify transitions between cells. This abstraction process produces a finite-state automaton that admits efficient formal verification algorithms. There are many flavours of this methodology that differ in how the cells are constructed, including: hyper-rectangles, Maler [18], Carter [19], piecewise-linear functions, Asarin et al. [20], linearisations, Dang et al. [10], qualitative abstractions, Tiwari [7], Lyapunov functions,

Sloth et al. [11] and many others. In my dissertation, I develop a similar discretizing approach but with two distinguishing modifications.

- The cell partitioning functions are allowed to contain nonpolynomial terms.
- The feasibility of cells and the transitions between them are determined and verified using automated theorem proving.

The initial work on using this type of qualitative abstraction was implemented by Tiwari in the original HybridSAL tool. However, the problems analysable by HybridSAL were restricted to linear and nonlinear polynomial differential equations. The QUANTUM abstracter[1] that has been developed as part of my thesis implements a modified algorithm for performing qualitative abstraction.

The hybrid system's state space is first discretized by a series of nonpolynomial functions that are chosen from the definition of the hybrid automaton. This includes the differential equations defining the vector fields and the functions in the state guards. Any safety properties of the system are also added to the set of discretization functions.

MetiTarski is then used to remove infeasible abstract states and to validate abstract transitions through analysis of the nonpolynomial vector field. QUANTUM performs a *lazy* abstraction that will immediately terminate if a predefined safety property is invalidated (by transitioning into an unsafe state).

The results of experiments using QUANTUM indicate that it is at least competitive at verifying benchmark nonpolynomial hybrid system problems [21]. This result is promising. Qualitative reasoning has generally not been considered powerful enough to reason about complex dynamical systems. The combination of qualitative reasoning and the automated theorem prover MetiTarski has the potential to prove this notion wrong.

References

1. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning **44** (2010) 175–205
2. Denman, W., Akbarpour, B., Tahar, S., Zaki, M.H., Paulson, L.C.: Formal verification of analog designs using MetiTarski. In: Formal Methods in Computer-Aided Design. FMCAD 2009. (November 2009) 93 –100
3. Akbarpour, B., Paulson, L.C.: Applications of MetiTarski in the verification of control and hybrid systems. In: Hybrid Systems: Computation and Control. Volume 5469 of Lecture Notes in Computer Science. (2009) 1–15
4. Denman, W., Zaki, M.H., Tahar, S., Rodrigues, L.: Towards flight control verification using automated theorem proving. In: NASA Formal Methods. (2011) 89–100
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag (2008) 337–340

[1] available for download at http://www-dyn.cl.cam.ac.uk/~wd239/quantum

6. Frehse, G., et al.: SpaceEx: Scalable verification of hybrid systems. In: Computer Aided Verification (CAV). LNCS, Springer (2011)
7. Tiwari, A.: Abstractions for hybrid systems. Form. Methods Syst. Des. **32**(1) (February 2008) 57–83
8. Tiwari, A.: HybridSAL relational abstracter. In: CAV. (2012) 725–731
9. Eggers, A., et al.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In: Software engineering and formal methods. SEFM'11, Berlin, Heidelberg, Springer-Verlag (2011) 172–187
10. Dang, T., Maler, O., Testylier, R.: Accurate hybridization of nonlinear systems. In: Hybrid systems: computation and control, ACM (2010) 11–20
11. Sloth, C., Wisniewski, R.: Abstraction of continuous dynamical systems utilizing lyapunov functions. In: Decision and Control (CDC), 2010 49th IEEE Conference on. (December 2010) 3760–3765
12. Larsen, K.G., Bengtsson, J., Bengtsson, J., Larsen, K., Larsson, F., Larsson, F., Pettersson, P., Pettersson, P., Yi, W., Yi, W.: Uppaal - a tool suite for automatic verification of real-time systems. In: Hybrid Systems III, LNCS 1066, Springer-Verlag (1995) 232–243
13. Yovine, S.: Kronos: A verification tool for real-time systems. (kronos user's manual release 2.2). International Journal on Software Tools for Technology Transfer **1** (1997) 123–133
14. Prajna, S., Papachristodoulou, A., Seiler, P., Parrilo, P.A.: SOSTOOLS: Sum of squares optimization toolbox for MATLAB (2004)
15. Tiwari, A., Khanna, G.: Series of abstractions for hybrid automata. In: Hybrid Systems: Computation and Control. LNCS 2289, Springer (2002) 465–478
16. Tarski, A.: A decision method for elementary algebra and geometry. Technical report, RAND Corp. (1948)
17. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Journal of Computer and System Sciences, ACM Press (1995) 373–382
18. Maler, O., Batt, G.: Approximating continuous systems by timed automata. In: Formal methods in systems biology. Springer (2008) 77–89
19. Carter, R., Navarro-Lpez, E.: Dynamically-driven timed automaton abstractions for proving liveness of continuous systems. In Jurdziski, M., Nikovi, D., eds.: Formal Modeling and Analysis of Timed Systems. Volume 7595 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 59–74
20. Asarin, E., Dang, T., Girard, A.: Reachability analysis of nonlinear systems using conservative approximation. In: In Oded Maler and Amir Pnueli, editors, Hybrid Systems: Computation and Control, LNCS 2623, Springer-Verlag (2003) 20–35
21. Denman, W.: Verifying nonpolynomial hybrid systems by qualitative abstraction and automated theorem proving. In: NASA Formal Methods 2014. (April 2014)

Livelock Analysis for Component-Based Systems

M. S. Conserva Filho

Federal University of Rio Grande do Norte (UFRN) – Natal, RN, Brazil
madiel@ppgsc.ufrn.br

Abstract. The use of increasingly complex software applications is demanding a greater investment of resources in software development to ensure that applications are safe. Therefore, several techniques are being used in Software Engineering in order to make the development process more agile and effective. Component-based development (CBD) emphasizes the reuse of software and building systems by integrating existing components. However, in order to ensure trustworthy systems, a formal verification must be carefully performed when components are being integrated, especially in critical applications, during which classical problems can arise, such as livelock. In this paper, we propose a systematic approach to build livelock-free systems in $\mathcal{BRIC}$, which is a component model that defines components and how they interact with each other.

Keywords: Component-based systems, Livelock, $\mathcal{BRIC}$.

1 Introduction

The use of increasingly complex software applications is demanding a greater investment of resources in software development to ensure that applications are safe, robust and error-free. With this concern, new techniques, methodologies and tools are been used in order to enhance quality and safety in software development.

Component-based system development (CBSD) has been widely used to deal with the increasing complexity of software systems. The main idea of this approach is to develop systems by integrating existing independent parts, called components. Furthermore, CBSD emphasizes the reuse of reliable software components in order to build high-quality applications. Consequently, the reduction in development costs due to reusability is another advantage of this approach.

However, in order to ensure that systems based on CBSD are reliable, a crucial issue is how the components are connected. Problems may arise when two or more components are integrated for the first time. This concern is even more relevant when a group of components is put together in order to perform certain tasks.

Therefore, only using the CBSD approach is not enough to ensure that a system is error-free. Besides CBSD, Formal Methods is another approach that research has used to improve quality when developing systems. They provide techniques and tools to specify and verify the properties of a system. In addition, they enable a system to be verified with mathematical accuracy. Therefore, Formal Methods and CBSD are techniques which may complement each other.

In order to ensure the success of CBSD, it is essential to perform a thorough verification when components are being integrated, especially in concurrent systems. This integration may cause some well-known problems, such as, deadlock and livelock.

Some authors have been conducting research on how to identify problems in compositions and the correctness of component-based systems, such as [[1], [3], [4], [6]. [2], [5]]. However, as far as we are aware, these researchers have not provided a trustworthy composition strategy to ensure that a system is livelock-free.

In [7] a correct-by-construction strategy is proposed so as to develop reliable systems. It deals with preserving deadlock-freedom of a system based on the deadlock-freedom of its constituents. This approach is based on the CSP process algebra [8], which focuses on examining the specifications of concurrent systems and uses semantic models to perform process verification. Furthermore, this work provides a generic component model, $\mathcal{BRIC}$, that imposes some constraints on the components and how they interact with each other. In $\mathcal{BRIC}$ [7], each component is represented by a contract (Ctr : $\mathcal{B}$,$\mathcal{R}$,$\mathcal{I}$,$\mathcal{C}$), which describes the dynamic behaviour $\mathcal{B}$ (represented as a CSP process), a set of communication channels $\mathcal{C}$, which is used to compose two components, a set of interfaces $\mathcal{I}$, and a total function between channels and interfaces $\mathcal{R}$.

Even though this strategy ensures the preservation of deadlock-freedom in $\mathcal{BRIC}$, this is not enough to avoid all problems in CSBD. In some cases, a system may be free of deadlock and yet it makes no progress in executing its tasks. This situation is referred to as livelock, which is regarded as even worse than deadlock.

Both deadlock and livelock result in a lack of progress in the system. Sometimes, they are considered as the same problem. However, these two problems are used to refer to two different kinds of non-progress. Thus, the techniques used to check these two properties are totally different.

Ensuring freedom from livelock is a challenging task and it is highly desirable to ensure that systems are reliable. In systems based on CSBD, livelock can arise when components infinitely communicate on internal channels, which are not externally observable.

Given this important concern, the main contribution of this article, which is drawn from research in progress, is to develop a systematic approach to ensure livelock-free systems by using component-based development, more specifically, for $\mathcal{BRIC}$ components.

The remainder of this paper is structured as follows. The next section gives more details about the methodology and the current development. Finally, Section 3 makes concluding remarks and outline the expected results.

2 Methodology and Current Stage

This section briefly describes the methodology that has been used in this study. Furthermore, the current stage of our research which tackles on how to build safe systems in a compositional way is also presented.

2.1 Methodology

The first task of this work was to conduct research on component-based development in order to find how it deals with the problem of livelock. This task provided us with good feedback on the state of the art and showed that there are few papers that address the problem of livelock.

Thereafter, an in-depth study on livelock in concurrent systems was performed. This analysis helped us to understand how this problem arises and how it can be solved. With this in mind, we are ready to start to propose solutions for this problem.

2.2 Current Stage

Currently, two important steps are being performed. The first consists of identifying infinite behaviours. This is a crucial issue in this work because one way to avoid livelock is to ensure that a process never computes internally through an infinite sequence without communicating with its environment. This occurrence is not detectable or controllable by the environment

As the dynamic behaviour in $\mathcal{BRIC}$ is represented as a CSP process, we must be aware as livelock may arise in CSP. The possibility of writing down a divergent CSP process may arise from the presence of hiding. It converts visible actions into invisible ones and is used as a key device for abstraction. By way of ilustration, let us consider the process $P = a \to P \setminus \{a\}$, where P converts the external event a into an internal action. Therefore, P indefinitely performs internal actions, causing a divergence.

So, reasoning about livelock requires reasoning about infinite behaviours. Therefore, in our approach to ensure livelock-freedom, one major concern consists of identifying what the infinite sequences of a given component contract are. This represents a finite sequence of traces that leads the process to its initial state. In other words, these are the sequences of traces which are repeatedly offered to the environment.

By identifying these sequences, it is possible to discover which channels may be used in the composition and which ones can introduce divergences if they are used. In order to make this possible, we are formally defining some sets and functions.

The second step of the current development consists of defining the conditions needed to compose two components, ensuring that this composition will not introduce livelocks.

To achieve this goal, a set of compositional rules is being developed, which will make it possible to build reliable systems in a compositional way. Furthermore, this work is using metadata to make some calculations during the composition process. The metadata stores useful information that can be used in future compositions.

These rules represent two kinds of composition: the binary composition involves two components via two communication channels, and the unary composition assembles two channels of the same component.

In this approach, when a composition is performed via the communications channels, they are removed from the set of communications channels of the component contract, $\mathcal{C}$. In others words, they are not more externally observed and not offered to the environment in further compositions. With this process, these channels will be transformed into internal channels, which may introduce livelock.

So far, in order to ensure livelock in $\mathcal{BRIC}$ compositions, the conditions required to compose are governed by two separate rules, *Simple Composition* and *Problematic Composition*. The former represents the case of the simplest binary composition and requires less complex verifications. It deals with the cases in which the communication channels used in the composition do not introduce divergence. However, this composition is not applicable in all cases.

As a result of this concern, the second rule, *Problematic Composition*, deals with cases which may introduce livelock, and this verification requires a careful analysis. Through this rule, we can ensure livelock-freedom in the problematic composition and prevent the composition from being performed.

To make this composition possible, two conditions must be satisfied. The first condition deals with the possibility of performing a composition via problematic channels. However, one proviso must be true. If the communication channels used to compose do not interact with each other, the resulting composition is livelock-free.

On the other hand, there are cases when this proviso may fail. For these compositions, by means of a formal verification, we ensure that, if the composition is performed, it will introduce livelock. Consequently, this composition must be avoided.

Thus, by applying these rules, we can ensure that the resulting component of the composition is livelock-free. Furthermore, these compositions can predict when a specific composition may have divergence; in other words, we can identify this problem early.

3 Conclusion and Expected Results

This paper presented a brief overview of the approach that has been developed for building trustworthy component-based systems, focusing on livelock freedom when integrating components.

At the end of this doctoral we expect to contribute with of the state-of-the art of software engineering in order to remedy the shortcomings of CBD with regard to the composition of components by proposing several constraints that must be used to ensure, by construction, the absence of livelock. Furthermore, this work intends to develop a framework that implements this strategy. This will improve the sucess of the component-based system development.

4 Acknowledgments

The EU FP7 Integrated Project COMPASS (Grant Agreement 287829) ?nanced most of the work presented here. INES and CNPq partially support the work of Madiel Conserva Filho: 573964/2008-4 and 483329/2012-6. CAPES also supports this work. My supervisors Marcel Oliveira and Augusto Sampaio have provided relevant insights related to this work. Pedro Antonino and Jose Dihego also contributed to this work.

References

1. Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.
2. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997.
3. Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.
4. Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, October 1996.
5. X. Li J. He and Z. Liu. A theory of reactive components. *Electronic Notes in Theoretical Computer Science.*, pages 173–195, 2006.
6. Mubarak Sami Mohammad. *A Formal Component-Based Software Engineering Approach For Developing Trustworty Systems*. PhD thesis, Concordia University, April 2009.
7. R. T. Ramos. *Systematic Development of Trustworthy Component-based Systems*. PhD thesis, Centro de Informtica, Universidade Federal de Pernambuco, 2011.
8. A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

Reliability Analysis of Non-deterministic Systems

Lin Gui
Supervisor: Jin Song Dong

NUS Graduate School for Integrative Sciences and Engineering
National University of Singapore
lin.gui@nus.edu.sg

Abstract. As software becomes more complex and often operates in a dynamic environment, the usage of certain software components is hard to be determined prior to their deployment, and thus, better to be modeled nondeterministically. However, traditional reliability analysis methods are only suitable for deterministic system. In this work, we propose to analysis software reliability via probabilistic model checking on Markov decision processes, a well-known automatic verification technique dealing with both probabilistic and non-deterministic behavior. On the top of that, we integrate various techniques, e.g., statistical, numerical and graphical methods, to make the reliability analysis much more scalable and efficient. Reliability analysis on various real-world systems has been conducted including a stock trading system and an ambient assisted living system.

1 Introduction

System quality, such as reliability, performance and security, is the key to the success of modern software systems. In particular, reliability and fault tolerance are central issues to many systems, including cloud-based web services, industrial control systems, wireless sensor networks, etc. The term reliability refers to the probability of failure-free software operation for a specified period and environment [5,2]. Being reliable is important for these systems, as a breakdown would potentially lead to significant impacts to users together with financial and reputational damages.

Existing Approaches Existing approaches on the reliability analysis problem fall into two categories: black-box approaches [4,10] and white-box approaches [2,5]. The black-box approaches treat a system as a monolith and evaluate its reliability using testing techniques. They use the observed failure information to predict the reliability of software based on several mathematical models. On the contrary, the white-box approaches assume reliability of system components are known and evaluate software reliability analytically based on the model of the system architecture including Discrete Time Markov Chains (DTMCs) [2], or

Continuous Time Markov Chains (CTMCs) [5]. In these approaches, the probabilistic transfer of control among components is assumed to be known. For instance, the probability is assumed to be a constant in DTMC-based approaches or a function of time in white-box approaches.

Non-deterministic Systems As software becomes more complex and often operates in a dynamic environment, the execution orders among or the usage of certain software components are hard to be measured prior to its deployment. We consider that as a *non-deterministic system*. The reliability analysis of such a non-deterministic system become highly non-trivial. The requirements of reliability analysis approaches for such non-deterministic system are summarized as follows.

R0: The approaches should be able to handle non-deterministic behavior. That is, even for a system operating in complex and dynamic environments, the reliability can still be analyzed as close to the 'true' reliability as possible, before software deployment.
R1: To handle large scale software system, the approaches should have good scalability.
R2: Even for the large and complex software system, the approach should be efficient, i.e., the computation should be as fast as possible.

Although there is a lot related work on improving reliability analysis accuracy, scalability (R1) and efficiency (R2), insofar, none of them can work for non-deterministic system (R0). In this work, we are motivated to propose an approach to meet R0. On the top of that, R1, and R2 should also be satisfied.

Our Approach A potential candidate is probabilistic model checking [1,8,9] based on Markov Decision Processes (MDPs), which is designed to deal with both probabilistic and non-deterministic behavior. As a popular verification method, probabilistic model checking provides a set of automatic techniques to analysis quantitative properties of a system. In this work, we propose to perform reliability analysis via probabilistic model checking on MDPs. As a result, the unknown usages of system components can be easily modeled non-deterministically. To foster a better reliability analysis in terms of scalability and computation speed, we are further exploring feasibilities of integrating various statistical, numerical and graphical methods into probabilistic model checking techniques. The research progress is summarized as follows :

- Up to the present, we have proposed to a framework that combine hypothesis testing with probabilistic model checking [3]. Specifically, the deterministic system components are tested independently, and overall results are lifted through non-determinism via probabilistic model checking.
- We have developed a toolkit RaPiD to support automatic software reliability analysis including reliability assessment, reliability distribution and sensitivity analysis. Case studies have been carried out on real world systems including a stock trading system and an ambient assisted living system [6].

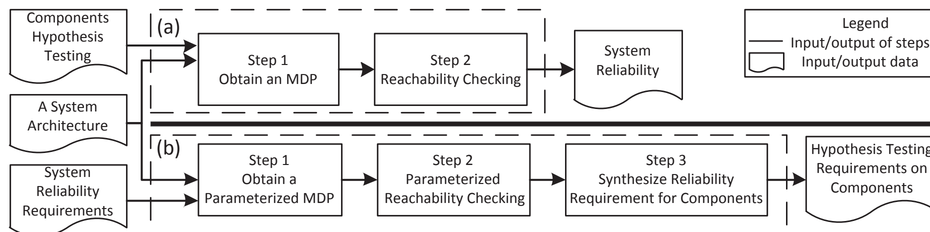


Fig. 1. Workflow: (a) reliability prediction; (b) reliability distribution

– For the ongoing research activities, we continue to improve the scalability and computation speed of the reliability analysis. As the size global state space exponentially increases w.r.t sizes of local state spaces, we propose to shrink the local state spaces by abstracting and refining the communication events. Further, we plan to speed up the computation by removing the loops inside the MDPs, as the loops impede fix point calculations.

2 Towards Non-deterministic Systems Analysis

To perform reliability analysis on non-deterministic systems (R0), we have proposed to use probabilistic model checking methods and build the high level interactions among software components in Markov Decision Processes (MDPs). Research efforts are also putted into improving the scalability and computation time of probabilistic model checking (R1, R2), by incorporating with other statistical, numerical and graphical methods.

2.1 Combining Testing and Probabilistic Model Checking

We have proposed a framework to combine testing and probabilistic model checking. This framework can be applied to the reliability analysis activities including *reliability assessment, reliability distribution* and *sensitivity analysis*. The idea is to apply testing to deterministic system components, and use probabilistic model checking techniques to lift the results through non-determinism.

The overall framework is shown in Figure 1. Figure 1-a shows our workflow of solving the reliability prediction problem. Firstly, hypothesis testing is applied to obtain the reliability of system components. The result is a probability, i.e., the reliability of a component being larger or equal to this probability, with error bounds defined by users. Next, MDP-based reachability analysis is used to compute the overall system reliability, which is the probability that the system reaches the success state. Notice that existing algorithms on probabilistic reachability checking must be extended to handle the error bounds obtained with hypothesis testing. Figure 1-b shows the workflow of solving the reliability distribution problem. Given a reliability requirement for the system with error bounds, we solve the problem in three steps. Firstly, we construct a parameterized MDP

model within which each component is associated with variables representing its reliability measurement. Next, we develop a parameterized probabilistic reachability checking algorithm to obtain the minimum constraints on the variables. Lastly, we synthesize concrete reliability requirement for each component based on the constraints. Noted that the approach for reliability distribution can be easily extended to sensitivity analysis by taking differentiation of the parameterized expression. A paper on this work has been published in [3].

2.2 A Toolkit RaPiD and Case Studies

Our approach has been realized in a toolkit named RaPiD, which has been applied to investigate several real-world systems, including a stock trading system (Cross Call System), a hospital system (Therapy Control System), and a pervasive system (Smart House). In particular, a case study on an ambient assisted living system will be presented in [6]. In that work, we have reported our experience from analyzing reliability of a smart healthcare system named AMUPADH for elderly people with dementia in a Singapore-based nursing home. Using MDPs, we have performed reliability analysis in three aspects. First, we have assessed the overall system reliability based on the reliability value of each component. Second, given a required system reliability, we have performed the reliability distribution to calculate the reliability requirement for each component. Last, sensitivity analysis has been performed to identify the component affecting the system reliability most significantly. As a result, our evaluations have provided insightful results on overall system reliability and critical components that affect the reliability most.

3 Ongoing Research Activities

3.1 Improved Probabilistic Model Checking for Distributed Systems

For a distributed system, the size of the global state space are exponential to the size of local state spaces. Our ongoing work is to improve the scalability of reliability analysis for such systems. We improve the probabilistic model checking through a method of abstraction and reduction, which controls the communications among different system components and actively reduces the size of each system component.

More specifically, we assume that a distributed system consists of a set of system components, each of which has its local state space and interfaces for communication. Our key concept is to shrink the global state space by reducing the sizes of its local state spaces, which is achieved via controlling the communications among different system components and actively reducing the size of each. More specifically, we start with an abstract system by turning a subset of communication events into local (i.e., non-communication) events, which can be effectively removed afterwards by re-calculating the local probabilistic distributions for the rest of communication events. We then perform probabilistic model

checking to calculate the reliabilities (here referred to as 'approximations'). If the resulting approximations are not precise enough, the refinement step can be performed by incrementally enlarging the set of communication events, which eventually yield an 'actual result' based on the complete model.

3.2 Speedup via Loops Removal

Reliability analysis can be transformed into a probability reachability analysis based on an MDP. Value iteration method is popular used in probabilistic model checkers. It works by finding a better approximation iteratively until certain stopping criteria are satisfied. However, this approach has the issue of slow convergence such that the algorithm requires many iterations before it converges to a certain value when there are many loops in the probabilistic systems. We have accomplished a preliminary study that eliminates strong connected components using the divide and conquer algorithm [7]. However, this only works for DTMCs. My future work is to extend the algorithm to MDPs. The plan is to find an efficient method to reduce the redundant memoryless schedulers so as to transform the problem into solving minimal number of the induced DTMCs.

Acknowledgements

I sincerely thank Dr. Jin Song Dong, Dr. Jun Sun and Dr. Yang Liu, for their great guidance and support during my Ph.D. study.

References

1. C. Baier and J. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
2. R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Software Engineering*, SE-6(2):118–125, 1980.
3. L. Gui, J. Sun, Y. Liu, Y. J. Si, J. S. Dong, and X. Y. Wang. Combining model checking and testing with an application to reliability prediction and distribution. In *ISSTA*, pages 101–111. ACM, 2013.
4. A. Immonen and E. Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.
5. J. C. Laprie and K. Kanoun. *Handbook of software Reliability Enginerring*, chapter Software Reliability and System Reliability, pages 27–69. McGraw-Hill, New York, NY, 1996.
6. Y. Liu, L. Gui, and Y. Liu. Mdp-based reliability analysis of an ambient assisted living system. In *FM Industry Track*, Singapore, May 2014 (Accepted).
7. S. Song, L. Gui, J. Sun, Y. Liu, and J. S. Dong. Improved reachability analysis in dtmc via divide and conquer. In *IFM*, pages 162–176. Springer, 2013.
8. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.
9. J. Sun, S. Song, and Y. Liu. Model checking hierarchical probabilistic systems. In *FMSE*, pages 388–403. Springer, 2010.
10. D. M. Woit. *Estimating software reliability with hypothesis testing.* Citeseer, 1993.

HOPE: A Framework for Handling Obfuscated Polymorphic Malware

Nguyen Thien Binh[1], Quan Thanh Tho[1], Nguyen Minh Hai[1], Nguyen Huu Vu[2], Nguyen Cong Dinh[1]

[1]HoChiMinh City University of Technology, Vietnam
[2]Pôles Universitaire Francais Ho Chi Minh Ville (PUF-HCM), Vietnam

Abstract. To overcome the limitations of commercial anti-virus programs, model checking has been proposed since this technique can use an underlying logic to capture malicious behavior. However, obfuscation techniques employed by sophisticated malware make it virtually impossible to form an one-size-fits-all logic for obfuscated virus. To tackle this problem, we propose a framework, known as HOPE (Handling Obfuscated Polymorphic malwarE) whose principal idea is to separate the deobfuscation step from the virus detection process performed by the model checking. The novel part of our work is that we make an empirical assumption that popular obfuscations are free of dynamic jump. This allows us to isolate possible obfuscated code segments and efficiently handle them. We attempt to prove that this approach enables us to overcome almost popular obfuscation techniques. Initial experiment with a data set of real viruses reveals promising results, especially in comparison to state-of-the-art tools in this field.

Keywords: Polymorphic virus, Binary code analysis, Abstract Interpretation, Obfuscation, Deobfuscation, Model Checking.

1 Introduction

Polymorphic viruses are particularly hard to identify with simple mechanisms because they can change their *signatures* by using *obfuscation techniques*. Industrial anti-virus software which checks precise signatures would therefore fail to check adaptations of the original virus. For example, in Figure 1(b), the author intentionally performed *push* and *pop* on *ebx* register meaninglessly for obfuscation. To overcome the limitation of industrial anti-virus methods, CTL logic and its improved versions were proposed for describing virus behaviors. However, since the concrete actions that can be made to perform obfuscation are virtually infinite, it seems to be a hopeless direction to use an enhanced version of temporal logic to capture all of possible obfuscation actions. For instance, in Figure 1(c) is another method to change the stack content which cannot be captured by the existing methods.

Moreover, all approaches above for polymorphic virus detection assume the existence of *external Oracle* which can reconstruct the Control Flow Graph CFG

$l_1 : mov\ eax, 0$	$l_1 : mov\ eax, 1$	$l_1 : mov\ eax, 0$	$l_1 : assign\ (eax, 0)$
$l_2 : push\ eax$	$l_2 : dec\ eax$	$l_2 : mov\ [esp], eax$	$l_2 : push\ eax$
$l_3 : call$	$l_3 : push\ eax$	$l_3 : push\ ebx$	$l_3 : call$
$ds : GetModuleHandleA$	$l_4 : push\ ebx$	$l_4 : pop\ ebx$	$ds : GetModuleHandleA$
	$l_5 : pop\ ebx$	$l_5 : call$	
	$l_6 : call$	$ds : GetModuleHandleA$	
	$ds : GetModuleHandleA$		
(a)	(b)	(c)	(d)

Fig. 1: Avron virus in various formal representations

from original binary code. This practically encounters several difficulties, especially with the problem of dynamic jump instructions, i.e., instruction such as *jmp ebx*, whose target are identified at runtime. However, in the context of virus infection, virus maker suffers from similar obstacles if he wants to infect his malicious codes into a piece of code. By analyzing common obfuscation techniques presented in [2], we fortunately realize the following assumption.

Assumption 1 *Obfuscation techniques used by real-life virus appear to infect only basic blocks and do not rely on dynamic jumps.*

2 Related Works

2.1 Model-checking-based Approaches for Detecting Virus

Back to 2005, the idea of representing binary code as a model and the malicious behavior as properties to be verified was already proposed [9]. In order to describe properties to be verified, LTL is suggested first [7]. However, the major difficulty of this approach is that one needs to specify the path to be verified, which is simply infeasible in when dealing with real programs. To overcome the difficulty of LTL-based approach, several CTL based approaches were proposed such as [7]. CTPL [9], SCTPL [11] and SCTPL\X [10] to tackle the general description of virus behavior and they can handle some certain obfuscation techniques, as illustrated in Section 1.

2.2 CFG Construction for Binary Code

To construct CFG from binary code, Gogul Balakrishnan and Thomas Reps introduced value-set analysis (VSA) [4]. This analysis technique was implemented in a tool called CodeSurfer [3], which is an extension of IDAPro [1]. Meanwhile, *under-appromixation* approach usually combines static and dynamic analysis [8] ,[5]. Similarly, a refinement-based method is proposed based on k-sets [6].

3 Our Proposal

3.1 The Framework

In Figure 2 is our proposed framework, known as HOPE (Handling Obfuscated Polymorphic malwarE). Our main idea is that we separate the deobfuscation step

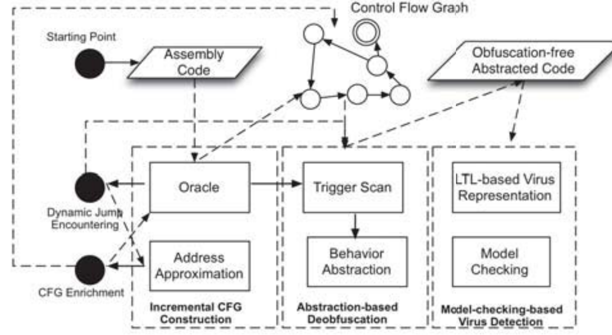


Fig. 2: The proposed framework

with the virus detection process performed by model checking. Similar to other model-checking-based approaches, an *external Oracle* is used in our framework to reconstruct CFG from binary code. However, instead of reconstructing the entire CFG at the first place, we only proceed the CFG reconstruction until *first dynamic jump* is found. From Assumption 1 we split the program into a number of static blocks, which are separated by dynamic jumps. In other words, each static block is a *dynamic-jump-free segment.* Then, we only perform deobfuscation on each static block. In deobfuscation process, first of all, *Trigger Scan* is performed to detect some predefined signals of malware. If a signal is found, *Behavior Abstraction* will be performed, resulting an obfuscation-free *abstracted program.* A specific Model Checking tool then analyzes this abstracted program to detect viral behavior. If no viral behavior is found, the CFG is continuously constructed from the pending point of dynamic jump until finished.

Compared to Other Existing Works In comparison with other approaches using model checking to detect polymorphic viruses, our contributions are as follows. Firstly, by reconstructing CFG incrementally from each found dynamic jump, we can isolate obfuscation code blocks if they exist. Next, we recommend an abstract language, and show that the corresponding abstracted program are neutralized with popular obfuscation techniques. Finally, the separation of deobfuscation and virus detection by model checking into two independent processes allow us to not have to change the descriptive logic when dealing with process of new obfuscation techniques.

3.2 Behavior Abstraction for Deobfuscation

Behavior Abstraction basically translates the original binary code into an abstract language, which is presented in Figure 3. Despite its simplicity, our abstract language is capable of capturing the real behavior of a binary code. Generally, obfuscation techniques aim at changing program states in various ways,

then recovering the old states again. However, since our Behavior Abstraction only focuses on the value of the last **assign** instruction, all of inserted obfuscation code will be virtually negated. For example, the problematic program given in Figure 1(c),can be abstracted using our technique as illustrated in Figure 1(d). Then, the abstracted code can be represented using a CTPL formula as $\exists r_1 EF(assign(r_1, 0) \wedge push(r_1) \wedge call(GetModuleHandleA))$

- ***assign*** *l-value= expression* : assign the value of a given *expression* to a *l-value*.
- ***jump*** *expression* : jump to the address given by the value of the *expression*.
- ***push/pop*** *register* : push or pop value of *register* into/from the stack.
- ***call*** *module* : call a subroutine *module*.
- ***skip*** : do nothing.
- ***guard*** *expression* : stop if the *expression* evaluated as false, and continue otherwise.

Fig. 3: The abstract language

Proposition 1 *If a virus can be detected by a model checker M, HOPE can detect all of its obfuscated variations using M.*

Proof. According to Assumption 1, the obfuscation technique does not involve dynamic jumps. Therefore, whole obfuscation code will be completely included in the CFG whenever we encounter the dynamic jump. As above discussed, all existing obfuscations were eliminated when we produce the corresponding abstracted program, rendering the original virus detectable by M □

3.3 Small Experiments

We test our deobfuscation performance with some real virus samples obtained from http://vxheaven.org. To evaluate performance, we compared our results with other well-known virus-scanners including Avira Avast, Kaspersky and Bit Defender. We have applied 5 popular kinds of obfuscation as presented in Table 1, in which the successful rate of obfuscated virus detection by well-known virus-scanners are compared to our approach.

Table 1: Experimental results

	Avira	Avast	Kapersky	Bit Defender	Our Approach
Nop-insertion	70%	72%	85%	80%	100%
Code-reordering	63%	60%	70%	75%	100%
Register-renaming	45%	40%	60%	55%	100%
Stack-operation	30%	32%	56%	60%	100%
Procedure-split	25%	20%	40%	45%	100%
Other obfuscation techniques	10%	9%	30%	40%	90%

As observed from Table 1, popular commercial virus scanners suffer from remarkable failure percentage when tackling popular obfuscations, which are

successfully handled by HOPE. However, HOPE still fails to deal with some complex cases of obfuscation, mostly due to self-encryption technique, which leaves a direction to pursue for our future work.

4 Conclusion

In this paper, we address the weakness of current model-checking-based approaches when dealing with obfuscation techniques employed by polymorphic virus. Hence, we provide a novel approach whose main idea is the separation of deobfuscation with virus detection step, combined with an incremental strategy of CFG construction. The major contributions of our abstraction-based method is the nullification of all popular obfuscation techniques which leads to very positive results, in comparison with commercial virus scanners.

References

1. IDAPro disassembler http://www.datarescue.com/idabase/.
2. A. Balakrishnan and C. Schulze. Code obfuscation literature survey http://pages.cs.wisc.edu/ arinib/writeup.pdf.
3. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Compiler Construction*, Lecture Notes in Computer Science, pages 250–254. Springer Berlin Heidelberg, 2005.
4. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer Berlin Heidelberg, 2004.
5. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The bincoa framework for binary code analysis. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 165–170. Springer Berlin Heidelberg, 2011.
6. S. Bardin, P. Herrmann, and F. Védrine. Refinement-based cfg reconstruction from unstructured programs. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer Berlin Heidelberg, 2011.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. pages 52–71. Springer Berlin Heidelberg, 1982.
8. T. Izumida, K. Futatsugi, and A. Mori. A generic binary analysis method for malware. In *Advances in Information and Computer Security*, volume 6434 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.
9. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Kruegel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 174–187. Springer Berlin Heidelberg, 2005.
10. F. Song and T. Touili. Efficient malware detection using model-checking. In *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 418–433. Springer Berlin Heidelberg, 2012.
11. F. Song and T. Touili. Pushdown model checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, 2012.

Verification of Refactoring with Delta Representation

Maya Retno Ayu Setyautami

Fakultas Ilmu Komputer, Universitas Indonesia
mayaretno@cs.ui.ac.id

Abstract. *In software development, changes are generally inevitable. Refactoring is a common method to change the structure of systems to improve the efficiency without altering the external behavior. However, refactoring mechanisms do not ensure that the process preserves the final results. Accordingly, the refactoring process is not verifiable yet. In this research, we intend to represent refactoring in Delta Oriented Programming (DOP) to manage the verification mechanism. Delta in DOP is also used to manage the changes. Therefore, we attempt to create delta representation of refactoring. This is supported by the translation mechanism from UML class diagram to DOP. Verified refactoring can help the developers convince that refactoring maintains the external behavior.*

Keywords: change, delta, dop, refactoring, uml class diagram, verification

1 Introduction

Refactoring is a method to restructure a model or program code by changing the internal structure without altering the external behaviors [1]. There are many kinds of refactoring that have been defined by Fowler in [1] and [2]. It can be related to code program or design model represented in UML class diagram. Refactoring can enhance the effectiveness of a program execution without entirely changing the code. Thus, it can improve the existing design without tampering with behaviors resulted from the model.

Refactoring is accomplished by doing a combination of several small transformations that change the existing structure but preserve its final result. However, the developers cannot verify and guarantee whether the process is changing the result and external behavior of the system or not. On the other hand, there is a novel programming approach, called, Delta Oriented Programming (DOP) [3], that can change the program without changing the code itself.

In DOP, the main code is implemented in the core module. The changing mechanism is maintained in the delta modules (delta). Delta defines the modification of the core module. For example, delta can modify, remove, or add method from the core module without changing the code in the core module. The developers handle the changes in the system without altering the original code. Therefore, the changes can be maintained systematically in delta modules.

Both refactoring and delta are related to changes in the system. Refactoring is used to change the code or model of systems and delta is used to change the code from the core module. Different from refactoring, delta has verification mechanism to guarantee that the system is trustworthy [4]. Therefore, assuming that refactoring can be represented in the delta, one of the benefits is that refactoring can be verified. As a contribution, verified refactoring can help developers ensure that refactoring maintains the external behavior and preserves its final results.

We utilize delta from Abstract Behavioral Specification (ABS), modeling language with DOP approach, described in Section 2. At this stage, we converge to create delta representation of refactoring UML class diagram. We have defined the translation mechanism from UML class diagram to ABS as a support for delta representation of refactoring. The summary of translation mechanism is described in Section 3. Last section describes the brief explanation about the proposed approach to create delta representation. We have attempted to implement the approach to simple refactoring case study.

2 Abstract Behavioral Specification

One of the researches that implements DOP approach is HATS (Highly Adaptable and Trustworthy Software using Formal Models) project [5]. It produced a language based on DOP called Abstract Behavioral Specification (ABS) [6]. ABS is an object-oriented modeling language that can be executed. Since it used DOP approach, ABS also has mechanism that can help the developer overcome requirement changes called delta modeling.

ABS is also supplied with a tool built on Eclipse plug-in. ABS tool[1] can automatically compile ABS models into other programming languages, such as Java and Maude. ABS tool can also give a visualization of ABS execution in Java language through Sequence Diagram. Moreover, ABS tool can help the developers verify the product's validity based on the feature constraints.

The complete specification of ABS syntax and semantics can be seen in ABS Language Specification [7]. ABS is comprised of several layers of language that model a system, including Core ABS, Micro Textual Variablity Language (TVL), Delta Modeling Language (DML), Product Line Configuration Language (CL), and Product Selection Language (PSL) [6]. Core ABS language defines the core module and DML defines the delta modules.

The following code is an example of delta module `DFee` that modifies core ABS module `Account` without changing the original code directly [8]. Class `AccountImpl` in core ABS module `Account` has method `deposit` that adds balance with a certain value. Delta `DFee` is created to modify method `deposit` by subtracting the value of balance with a specific fee.

```
delta DFee (Int fee);
uses Account;
```

[1] http://tools.hats-project.eu/

```
modifies class AccountImpl {
   modifies Int deposit(Int x) {
     Int result = x;
     if (x>=fee) result = original(x-fee);
     return result;
     }
}
```

3 UML Class Diagram Translation to ABS

At the current stage, we have created the translation mechanism from UML class diagram to ABS. Both UML class diagram and ABS based on object-oriented, but there are several differences between them. For example, UML class diagram has inheritance concept, but ABS does not have. Hence, inheritance cannot be modeled in ABS and it is required to change the structure of UML class diagram without changing its behavior. It can be done by refactoring that result UML class diagram with elements conform to ABS elements.

We have defined two new rules of refactoring to get UML class diagram with ABS element. Those are refactoring inheritance into delta and refactoring class to ABS class. The result of refactoring is UML class diagram with ABS profile. As first, we have defined UML profile for ABS (ABS profile) as a standardization for refactoring result. UML Profile is the mechanism from Object Management Group (OMG) to extend UML diagram [9]. By creating ABS profile, all elements from ABS are mapped to UML elements in the stereotypes, tagged values, and constraints. After creating the profile, it can be applied to UML class diagram that produce UML class diagram with ABS profile that may contain ABS elements.

If UML class diagram already represents ABS elements, the translation process is straightforward. Each element of UML class diagram can be mapped to ABS element with simple translation rules. For example, class with stereotype `<<absClass>>` in UML class diagram is mapped to ABS class in ABS, class with stereotype `<<absDelta>>` is mapped to delta in ABS. We have implemented the rules in Eclipse Acceleo Model-to-Text Transformation [10] to help the user run the translation process automatically. The translation process can be run in the Eclipse with UML class diagram with ABS profile as an input and the output is ABS model.

4 Delta Representation of Refactoring

As mentioned above, provided that refactoring can be represented in delta, it can be verified. Thus, we attempted to create delta in ABS that have similar behavior with refactoring in UML class diagram. Fig. 1 shows the connection between refactoring in UML class diagram and delta modeling in ABS. In UML class diagram, UML version 1 transforms into UML version 2 by Refactoring A.

The translation mechanism from UML class diagram to ABS is used to support creation of delta representation for refactoring.

We can get ABS 1 that models the problem similar to UML version 1 by doing UML translation to ABS. In this research, we attempted to create delta A, that modified ABS 1 to ABS 2, which equivalent to the process done by Refactoring A. To verify the result, we translate UML version 2 to ABS 2 and check whether delta A can produce similar ABS 2 when it applied to ABS 1. If the result ABS 2 from translation process and delta application is similar, it means that the delta can represent the refactoring process.

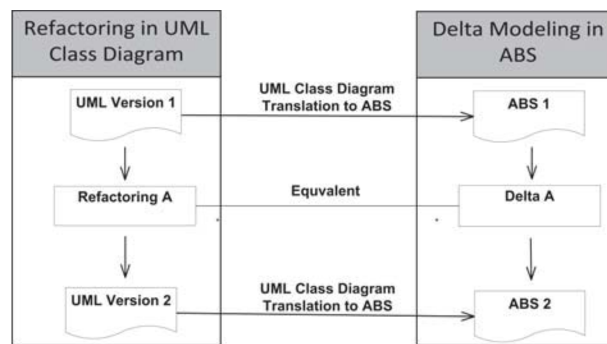


Fig. 1. Refactoring and Delta Modeling

For example, we have attempted to create delta representation for refactoring 'Move Field'. It happened when a field is, or will be, used by another class more than the class on which it is defined [11]. The procedure is creating a new field in the target class and change all its users. Fig. 2 below shows the illustration of Refactoring 'Move Field'. On the left side, there is UML class diagram with class Account, AccountWithFee and AccountWithOverdraft. Class Account has attributes aid, balance, debit, credit, and fee. Supposed that field fee is used by class AccountWithFee more than class Account. Thus, we should apply refactoring 'Move Field' to improve the efficiency. We move field fee from the class Account to class AccountWithFee.

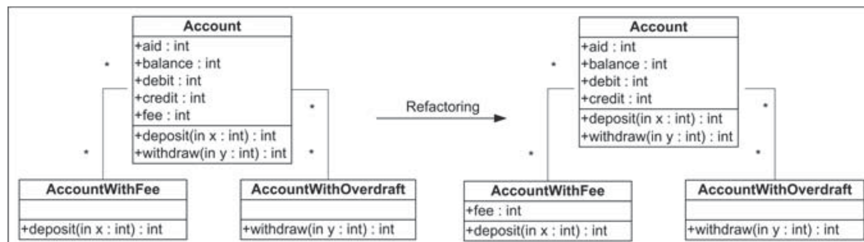


Fig. 2. Refactoring Move Field

Refactoring 'Move Field' fee can be represented in delta modeling by creating delta `DRemFee` that removes field fee from class Account and delta `DAddFee` that adds field fee to class AccountWithFee. The following code is the implementation of refactoring 'Move Field in delta DRemFee and delta DAddFee.

1. Delta `DRemFee`: removes fee from class Account	2. Delta `DAddFee`: adds fee to class AccountWithFee

`delta DRemFee();`
`uses ModAccount;`
`modifies class Account {`
`  removes Int fee;`
`}`
	`delta DAddFee();`
	`uses ModAccountWithFee;`
	`modifies class AccountWithFee {`
	`  adds Int fee;`
	`}`

Based on the experiment above, delta `DRemFee` and delta `DAddFee` are able to represent refactoring 'Move Field'. ABS model that used those deltas have similar behavior with the result of refactoring. There are many kinds of refactoring defined by Fowler in [1]. Thus, we need to explore the similar characteristic and classify the mechanism to devise the delta representation. Afterwards, we can analyze whether the delta can represent all kinds of refactoring or not.

References

1. Fowler, Martin: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1994)
2. Refactoring.com, `http://www.refactoring.com/catalog/`
3. Schaefer, Ina, et al: Delta-oriented Programming of Software Product Lines. In: 14th Software Product Line Conference, pp.77-91. Springer (2010)
4. Damiani, Ferruccio, et al: A Transformational Proof System for Delta-Oriented Programming. In: 16th International Software Product Line Conference, pp.53-60. ACM (2012)
5. HATS (Highly Adaptable and Trustworthy Software using Formal Models), `http://www.hats-project.eu`
6. Johnsen, Einar B, et al.: ABS: A Core Langage for Abstract Behavioral Specification. In: Proceedings of the Formal Methods for Components and Objects. Springer (2011)
7. HATS Project: The ABS Language Specification for ABS Version 1.2.0, `http://tools.hats-project.eu/download/absrefmanual.pdf`
8. Hahnle, Reiner: The Abstract Behavioral Specification Language: A Tutorial Introduction, `http://www.hats-project.eu/sites/default/files/Hahnle.pdf`
9. Object Management Group: OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.4.1, OMG document number: formal/2011-08-05, `http://www.omg.org/spec/UML/2.4.1/Infrastructure`
10. Eclipse: Acceleo, `http://wiki.eclipse.org/Acceleo`
11. Refactoring.com, `http://refactoring.com/catalog/moveField.html`