Lecture Notes in Computer Science 7554

Commenced Publication in 1973
Founding and Former Series Editors: Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison  
*Lancaster University, UK*

Takeo Kanade  
*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler  
*University of Surrey, Guildford, UK*

Jon M. Kleinberg  
*Cornell University, Ithaca, NY, USA*

Alfred Kobsa  
*University of California, Irvine, CA, USA*

Friedemann Mattern  
*ETH Zurich, Switzerland*

John C. Mitchell  
*Stanford University, CA, USA*

Moni Naor  
*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz  
*University of Bern, Switzerland*

C. Pandu Rangan  
*Indian Institute of Technology, Madras, India*

Bernhard Steffen  
*TU Dortmund University, Germany*

Madhu Sudan  
*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos  
*University of California, Los Angeles, CA, USA*

Doug Tygar  
*University of California, Berkeley, CA, USA*

Gerhard Weikum  
*Max Planck Institute for Informatics, Saarbruecken, Germany*
Preface

The Brazilian Symposium on Programming Languages (SBLP) is an annual conference that has been promoted by the Brazilian Computer Society (SBC) since 1996. In the last three years, it has been organized in the context of CBSOFT (Brazilian Conference on Software: Theory and Practice), co-located with a number of other events on computer science and software engineering.


The Program Committee (PC) of SBLP 2012 was formed by 36 members, from 10 countries. The Committee was responsible for selecting 10 full papers and 2 short papers from a total of 27 submissions, with authors from Brazil, Czech Republic, France, Netherlands, Portugal, USA and Uruguay. Each paper was reviewed by at least five reviewers, including 21 reviewers outside the PC. The refereeing reports were discussed by the reviewers, generally leading to a consensus. The final selection was made by the Program Committee Co-chairs, based on the final evaluations but also taking into account the reviewers reports as well as all comments received during the discussion phase. As in previous editions, the authors of the 10 full papers were invited to submit extended versions of their works to be considered for publication in a special issue of a reputed journal in computer science.

The technical program of SBLP 2012 also included keynote talks from Bernhard K. Aichernig (Graz University of Technology, Austria), entitled “The Science of Killing Bugs in a Black Box”, and Luis S. Barbosa (Universidade do Minho, Portugal), entitled “Software Components as Invariant-Typed Arrows.”

Finally, we would like to thank all members of the PC for their efforts, the referees for their reviews and contribution to the final discussion, the invited speakers for accepting our invitation and enriching the technical program with interesting talks, and all the authors, the sponsors and the Organizing Committee of CBSOFT 2012 for contributing to the success of SBLP 2012.

September 2012

Francisco Heron de Carvalho Junior
Luis Soares Barbosa
Organization

SBLP 2012 was organized by the Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, and sponsored by the Brazilian Computer Society (SBC), in the context of CBSOFT 2012 (Third Brazilian Conference on Software: Theory and Practice).

Organizing Committee

Nélio Cacho, UFRN, Brazil
Martin Musicante, UFRN, Brazil

Steering Committee

Christiano Braga, UFF, Brazil
Ricardo Massa Ferreira Lima, UFPE, Brazil
André Luis de Medeiros Santos, UFPE, Brazil
Francisco Carvalho Junior, UFC, Brazil

Program Committee Chairs

Francisco Carvalho Junior, UFC, Brazil
Luis Soares Barbosa, Universidade do Minho, Portugal

Program Committee

Alberto Pardo, Universidad de La Republica, Uruguay
Alex Garcia, IME, Brazil
Alvaro Freitas Moreira, UFRGS, Brazil
André Rauber Du Bois, UFPEl, Brazil
Andre Santos, UFPE, Brazil
Carlos Camarão, UFMG, Brazil
Christiano Braga, UFF, Brazil
Fernando Castor Filho, UFPE, Brazil
Fernando Quintão Pereira, UFMG, Brazil
João Saraiva, Universidade do Minho, Portugal
João F. Ferreira, Teesside University, UK
Jonathan Aldrich, Carnegie Mellon University, USA
José Luiz Fiadeiro, University of Leicester, UK
Lucília Figueiredo, UFOP, Brazil
Manuel António Martins, Universidade de Aveiro, Portugal
VIII Organization

Marcelo A. Maia  
Marcello Bonsangue  
Marcelo d’Amorim  
Marco Tulio Valente  
Mariza A. S. Bigonha  
Martin A. Musicante  
Noemi Rodriguez  
Paulo Borba  
Peter Mosses  
Qiu Zongyang  
Rafael Dueire Lins  
Renato Cerqueira  
Ricardo Massa  
Roberto S. Bigonha  
Roberto Ierusalimschy  
Sandro Rigo  
Sérgio Soares  
Simon Thompson  
Varmo Vene

UFU, Brazil
Leiden University/CWI, The Netherlands
UFPE, Brazil
UFMG, Brazil
UFMG, Brazil
UFRN, Brazil
PUC-Rio, Brazil
UFPE, Brazil
Swansea University, UK
Peking University, China
UFPE, Brazil
PUC-Rio, Brazil
UFPE, Brazil
UFMG, Brazil
PUC-Rio, Brazil
UNICAMP, Brazil
UFPE, Brazil
University of Kent, UK
University of Tartu, Estonian

Additional Referees

T. M. Gawlitza  
V. Vojdani  
M. Garcia  
J. Mendes  
J. Barbosa  
V. Rebello  
S. Gouw

L. Santos  
A. Annamaa  
P. Martins  
N. Oliveira  
M. Viera  
H. Yang  
A. Sanchez

G. Albuquerque-Junior  
C. Vieira  
H. Rebêlo  
J. Saraiva  
L. Sierra

Sponsoring Institutions

CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico
http://www.cnpq.br

CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
http://www.capes.gov.br

Ministério da Educação, Brazilian Government
http://www.mec.gov.br

Instituto Nacional de Ciência e Tecnologia para Engenharia de Software
http://www.ines.org.br

Microsoft Research
http://research.microsoft.com

NatalCard
http://www.natalcard.com.br
Table of Contents

Software Components as Invariant-Typed Arrows (Keynote Talk) .......... 1
  Luis Soares Barbosa

The Science of Killing Bugs in a Black Box (Keynote Talk) ............ 6
  Bernhard Klaus Aichernig

Spill Code Placement for SIMD Machines ............................... 12
  Diogo Nunes Sampaio, Elie Gedeon,
  Fernando Magno Quintão Pereira, and Sylvain Collange

Left Recursion in Parsing Expression Grammars .......................... 27
  Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy

Speed and Precision in Range Analysis ................................. 42
  Victor Hugo Sperle Campos, Raphael Ernani Rodrigues,
  Igor Rafael de Assis Costa, and Fernando Magno Quintão Pereira

Parameter Based Constant Propagation .................................. 57
  Péricles Rafael Oliveira Alves, Igor Rafael de Assis Costa,
  Fernando Magno Quintão Pereira, and Eduardo Lage Figueiredo

Adaptable Parsing Expression Grammars ................................. 72
  Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha,
  Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim

Optimizing a Geomodeling Domain Specific Language .................. 87
  Bruno Morais Ferreira, Fernando Magno Quintão Pereira,
  Hermann Rodrigues, and Britaldo Silveira Soares-Filho

A System for Runtime Type Introspection in C++ ....................... 102
  Maximilien de Bayser and Renato Cerqueira

Model-Based Programming Environments for Spreadsheets ............. 117
  Jacome Cunha, Joao Saraiva, and Joost Visser

A Quantitative Assessment of Aspectual Feature Modules for Evolving
Software Product Lines ................................................. 134
  Felipe Nunes Gaia, Gabriel Coutinho Sousa Ferreira,
  Eduardo Figueiredo, and Marcelo de Almeida Maia

Attribute Grammar Macros .............................................. 150
  Marcos Viera and Doaitse Swierstra

Author Index .............................................................. 165
Software Components as Invariant-Typed Arrows
(Keynote Talk)

Luis Soares Barbosa

HASLab - High Assurance Software Laboratory, INESC TEC & Universidade do Minho, Portugal
lsb@di.uminho.pt

Abstract. Invariants are constraints on software components which restrict their behavior in some desirable way, but whose maintenance entails some kind of proof obligation discharge. Such constraints may act not only over the input and output domains, as in a purely functional setting, but also over the underlying state space, as in the case of reactive components. This talk introduces an approach for reasoning about invariants which is both compositional and calculational: compositional because it is based on rules which break the complexity of such proof obligations across the structures involved; calculational because such rules are derived thanks to an algebra of invariants encoded in the language of binary relations. A main tool of this approach is the pointfree transform of the predicate calculus, which opens the possibility of changing the underlying mathematical space so as to enable agile algebraic calculation. The development of a theory of invariant preservation requires a broad, but uniform view of computational processes embodied in software components able to take into account data persistence and continued interaction. Such is the plan for this talk: we first introduce such processes as arrows, and then invariants as their types.

1 Components as Arrows

Probably the most elementary model of a computational process is that of a function \( f : I \rightarrow O \), which specifies a transformation rule between two structures \( I \) and \( O \). In a (metaphorical) sense, this may be dubbed as the ‘engineer’s view’ of reality: here is a recipe to build gnus from gnats. Often, however, reality is not so simple. For example, one may know how to produce ‘gnus’ from ‘gnats’ but not in all cases. This is expressed by observing the output of \( f \) in a more refined context: \( O \) is replaced by \( O + 1 \) and \( f \) is said to be a partial function. In other situations one may recognise that there is some context information about ‘gnats’ that, for some reason, should be hidden from input. It may be the case that such information is huge to be give as a parameter to \( f \), or shared by other functions as well. It might also be the case that building gnus would eventually modify the environment, thus influencing latter production of more ‘gnus’. For \( U \) a denotation of such context information, the signature of \( f \) becomes \( f : I \rightarrow (O \times U)^U \). In both cases \( f \) can be typed as \( f : I \rightarrow TO \)
for $T = \text{id} + 1$ and $T = (\text{id} \times U)^I$, respectively, where, intuitively, $T$ is a type transformer providing a shape for the output of $f$. Technically, $T$ is a functor which, to facilitate composition and manipulation of such functions, is often required to be a monad. In this way, the ‘universe’ in which $f : I \rightarrow T'O$ lives and is reasoned about is the Kleisli category for $T$. In fact, monads in functional programming offer a general technique to smoothly incorporate, and delimit, ‘computational effects’ of this kind without compromising the purely functional semantics of such languages, in particular, referential transparency.

A function computed within a context is often referred to as ‘state-based’, in the sense the word ‘state’ has in automata theory — the memory which both constrains and is constrained by the execution of actions. In fact, the ‘nature’ of $f : I \rightarrow (O \times U)^I$ as a ‘state-based function’ is made more explicit by rewriting its signature as $f : U \rightarrow (O \times U)^I$.

This, in turn, may suggest an alternative model for computations, which (again in a metaphorical sense) one may dub as the ‘natural scientist’s view’. Instead of a recipe to build ‘gnus’ from ‘gnats’, the simple awareness that there exist gnus and gnats and that their evolution can be observed. That observation may entail some form of interference is well known, even from Physics, and thus the underlying notion of computation is not necessarily a passive one.

The able ‘natural scientist’ will equip herself with the right ‘lens’ — that is, a tool to observe with, which necessarily entails a particular shape for observation. Similarly, the engineer will resort to a ‘tool box’ emphasizing the possibility of at least some (essentially finite) things being not only observed, but actually built.

In summary,

\begin{center}
\begin{tabular}{cc}
\text{an observation structure:} & \text{universe} \\ \\
& $\xrightarrow{c} \ igcirc - \bigcirc$ universe \\
\text{an assembly process:} & \text{artifact} \\ \\
& $\xrightarrow{a}$ artifact
\end{tabular}
\end{center}

Assembly processes are specified in a similar (but dual) way to observation structures. Note that in the picture ‘artifact’ has replaced ‘universe’, to stress that one is now dealing with ‘culture’ (as opposed to ‘nature’) and, what is far more relevant, that the arrow has been reversed. Formally, both ‘lenses’ and ‘toolboxes’ are functors. And, therefore, an observation structure is a $\bigcirc - \bigcirc$-coalgebra, and an assembly process is a $\text{□}$-algebra.

Algebras and coalgebras for a functor [13] provide abstract models of essentially construction (or data-oriented) and observation (or behaviour-oriented) computational processes, respectively. Construction compatibility and indistinguishability under observation emerge as the basic notions of equivalence which, moreover, are characterized in a way which is parametric on the particular ‘toolbox’ or ‘lens’ used, respectively. Algebraic compatibility and bisimilarity acquire a shape, which is the source of abstraction such models are proud of. Moreover, it is well known that, if ‘toolboxes’ or ‘lens’ are ‘smooth enough’, there exist canonical representations of all ‘artifacts’ or ‘behaviours into an initial (respectively, final) algebra (respectively, coalgebra).
Both assembly and observation processes, as discussed above, can be modeled by functions, or more generally, by arrows in a suitable category, between the universes-of-interest. Both aspects can be combined in a single arrow

\[
\begin{array}{c}
\begin{array}{c}
\text{U} \\
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
\text{U} \\
\end{array}
\end{array}
\]

formally known as a dialgebra. Initially defined in [14], their theory was developed in [15] and later by [16] in the style of universal algebra. In Computer Science, dialgebras were firstly used in [7] to deal with data types in a purely categorical way and more recently in [11], as a generalization of both algebras and coalgebras. In [12], they are used to specify systems whose states may have an algebraic structure, i.e., as models of evolving algebras [6].

Dialgebras \((d : F U \rightarrow G U)\) generalize many interesting computational structures, among which algebras \((a : F U \rightarrow U)\) and coalgebras \((c : U \rightarrow G U)\) as the simplest instantiations. A basic example is provided by transition systems with specified initial states. If the transition shape is given by \(G\), functor \(Id + 1\) introduces initial states as constants. This makes possible, for example, to introduce initial states on models of automata, as in \(d : Q + 1 \rightarrow Q^{In} \times 2\). Another example are components whose services may have a non deterministic output. If functor \(F\) captures an algebraic signature, \(d : F U \rightarrow P(U)\) caters for non deterministic outcomes.

2 Invariants as Types

If dialgebras provide a very general model for computational processes regarded as arrows between the universes-of-interest, one has also to be precise on what such ‘universes’ really are. A key observation is that, along their lifetime, computer systems are expected to maintain a certain number of properties on which depend their consistency and data integrity. On the other hand, they are subject to the permanent stress of ever changing business rules, which materialise into (either static or dynamic) properties of the underlying code.

Both integrity constraints and domain business rules are examples of invariant properties. The word ‘invariant’ captures the idea that such desirable properties are to be maintained invariant, that is, unharmed across all transactions which are embodied in the system’s functionality.

Invariants are ubiquitous in systems design. Actually, they take several forms and are defined not only over the input and output domains, as in a purely functional setting, but also over the underlying state space, as in imperative programming or reactive systems design. Software evolution and reconfiguration, on the other hand, entails the need for invariant checking whenever running code is upgraded or even dynamically reconfigured. While testing is the most widely used technique for such purpose, it is highly costly and does not ensure correctness. Ideally, one should be able to formally verify that the new invariants are enforced without running the (new) code at all.
This calls for a general theory of invariant preservation upon which one could base such an extended static checking mechanism. This talk sums up a number of steps towards such a theory which is both

- **compositional**: based on rules which break the complexity of the relevant proof obligations across the structures involved
- **calculational**: amenable to agile algebraic manipulation

Our starting point is the explicit use of relational techniques, a body of knowledge often referred to as the *algebra of programming* \cite{5}. In particular an invariant $P \subseteq X$ is represented as a binary relation $y \Phi_P x \equiv y = x \land x \in P$, which is called *coreflexive* because it is a fragment of the identity relation, *i.e.*, $\Phi_P \subseteq \text{id}$. Notice this is one of the standard ways of encoding a set as a binary relation. Since predicates and coreflexives are in one to one correspondence, we will use uppercase Greek letters to denote such coreflexives and will refer to them as ‘invariants’ with no further explanation.

Then, we resort to such relations to model types for arrows representing computational processes. Actually, if one regards invariants as *types*, the computational processes they type are *arrows*:

$$F \Phi_P \overset{d}{\longrightarrow} G \Phi_P$$  \hspace{1cm} (1)

where $F \Phi$ and $G \Phi$ represent invariant $\Phi$ placed in a *context* abstracted by functors $F$ and $G$, in the sense discussed above.

Typing computational processes (modelled as dialgebras) by invariants encodes a *proof obligation*. Actually the meaning of arrow (1) is

$$d \cdot F \Phi_P \subseteq G \Phi_P \cdot d$$  \hspace{1cm} (2)

which is computed as the relational counterpart to the following first-order formula $\langle \forall u :: u \in F(P) \implies d(u) \in G(P) \rangle$.

The intuition behind this move is that a dialgebra typed by a predicate is a structure for which such a predicate is to be maintained along its evolution. We will show how this can generalised in the context of a category whose objects are predicates and arrows encode proof obligations, *cf*,

- for general functions: $\Phi \overset{f}{\longrightarrow} \Psi$
- for *reactive processes* modelled as dialgebras $F \Phi \overset{d}{\longrightarrow} G \Phi$
- for *imperative programs*: $\Phi_{\text{pre}} \overset{R}{\longrightarrow} \Phi_{\text{post}}$ corresponding to *Hoare triples* $\{\text{post}\}R\{\text{pre}\}$. This requires a generalization of the invariant calculus to *relations*, to capture the calculus of weakest pre-conditions.

In each case, a calculus of invariants’ proof obligation discharge is developed, generalising our previous work. References \cite{2,8,3,9} provide a roadmap through our research on (coalgebraic) calculi for components-as-arrows. Most results on typing such arrows by predicates first appeared in \cite{4}, with further developments in \cite{10}.
Acknowledgments. Long time collaboration with J. N. Oliveira (Minho), Alexandra Silva (Nijmegen) and Manuel A. Martins (Aveiro), is deeply acknowledged. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - the Portuguese Foundation for Science and Technology within project FCOMP-01-0124-FEDER-010047.

References

The Science of Killing Bugs in a Black Box
(Keynote Talk)

Bernhard Klaus Aichernig

Institute for Software Technology,
Graz University of Technology, Austria
aichernig@ist.tugraz.at

Abstract. In this talk I will discuss the combination of model-based testing and mutation testing. Model-based testing is a black-box testing technique that avoids the labour of manually writing hundreds of test cases, but instead advocates the capturing of the expected behaviour in a model of the system under test. The test cases are automatically generated from this model. The technique is receiving growing interest in the embedded-systems domain, where models are the rule rather than the exception.

Mutation testing is a technique for assessing and improving a test suite. A number of faulty versions of a program under test are produced by injecting bugs into its source code. These faulty programs are called mutants. A tester analyses if his test suite can ”kill” all mutants. We say that a test kills a mutant if it is able to distinguish it from the original. The tester improves his test suite until all faulty mutants get killed.

In model-based mutation testing, we combine the central ideas of model-based testing and mutation testing: we inject bugs in a model and generate a test suite that will kill these bugs. In this talk, I will discuss its scientific foundations, tools, and results. The foundations include semantics and conformance relations; the supporting tools involve model checkers, constraint solvers and SMT solvers; our experimental results are taken from two European projects on embedded-systems. I will conclude with a proposal how model-based mutation testing can be integrated into an agile, iterative development process.

1 Combining Model-Based and Mutation Testing

In this keynote talk I discuss the results of our ongoing research on model-based mutation testing. Mutation testing is a fault-based white-box testing technique. Model-based testing is a black-box testing technique. Their combination leads to a fault-based black-box testing technique that we call model-based mutation testing. Similar to the Science of Programming [12], we build our automated testing approach on formal semantics and refinement techniques.

Model-based testing is a black-box testing technique focusing on the external behaviour of a system under test (SUT). Hence, we assume that we have no access to the internals of the SUT, like e.g., the source code. The test stimuli are
automatically generated from an abstract model of the SUT. This test model is usually derived from the requirements. The model serves also as a test oracle providing the verdict (pass or fail) of a test case execution. The models are expressed in special modelling languages that support the abstract specification of the central properties to be tested. A detailed introduction to model-based testing can be found in [18,19].

Why should practitioners accept the efforts to learn new modelling languages and create models along their implementations? The answer is cost reduction. Testing consumes up to 50% of the development costs in a mission-critical project. Once the models and adaptors are created, the test cases come for free, i.e. they are automatically generated. Furthermore, when requirements change, it is much easier to change an abstract model compared to updating hundreds of hand-written test cases. Similarly, when the interface changes, only the test adaptor, mapping abstract test cases to the concrete implementation level, needs an update. Hence, test automation for saving costs is the major motivation from a practitioner’s point of view.

Mutation testing is a way of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code following patterns of typical programming errors. These deviations in the code are called mutations. The resulting faulty versions of the program are called mutants. Usually, each mutant includes only one mutation. Examples of typical mutations include renaming of variables, replacing operators, e.g., an assignment for an equivalence operator, and slightly changing Boolean and arithmetic expressions. The number and kind of mutations depend on the programming language and are defined as so-called mutation operators.

A mutation operator is a rewrite rule that defines how certain terms in the programming language are replaced by mutations. For every occurrence of the term the mutation operator rewrites the original program into a new mutant. After a set of mutants has been generated, the test cases are run both on the original and on each mutant. If a test case can distinguish a mutant from the original program, i.e. a different output behaviour can be observed, we say that this test case kills a mutant. The goal is to develop a test suite that kills all mutants, if possible (some mutants are behaviourally equivalent). This technique is known since the 1970ies and receives growing interest [14]. However, “most work on Mutation Testing has been concerned with the generation of mutants. Comparatively less work has concentrated on the generation of test cases to kill mutants.” [14] In our work we address this, by focusing on test case generation.

Model-based mutation testing uses the model for both, generating test vectors and as a test oracle. Hence, we generate test cases from a model in order to test the conformance of a SUT. In contrast to classical model-based testing, only those test cases are generated that would kill a set of mutated models. The generated tests are then executed on the SUT and will detect if a mutated model has been implemented. Hence, model-based mutation testing rather tests against
non-conformance, than for conformance. In terms of epistemology, we are rather aiming for falsification than for verification. It is a complementary fault-centred testing approach.

2 From Semantics to Automated Test-Case Generation

Contracts are pre-postcondition specifications added to the source code of a program. Contracts abstract away from the internals of an algorithm. Semantically, they represent relations between the program’s state before and after execution.

Our first work on model-based mutation testing was purely theoretical [14]. The idea was to mutate the contracts and to derive test cases that would kill implementations of the mutated contract. We exploited the negated refinement laws of the refinement calculus. The result was a condition for a mutation test case for non-deterministic contracts: the input should cover the case where the mutant allows behaviour that is forbidden by the specification. In addition, the tests should cover valid inputs with undefined behaviour in the mutated specification. The insights gained were the key to our following more applied results.

We implemented a tool that took an UML-OCL contract and its mutant, translated it to a constraint solving problem, and generated a test case covering the fault in the mutant [8]. Later we applied this concept also to contracts in the C# language [15].

Communication protocols. More recently, we applied model-based mutation testing to several implementations of communication protocols. In this domain we are interested in sequences of observable communication events. Hence, the generated test cases have the form of event sequences in the deterministic case, or they have a tree-like shape in the non-deterministic case. This is in contrast to the work on contracts, where we only generated test cases as input-output vectors.

Our first work in this domain was the model-based testing of the Apache web-server[3]. In this project we modelled parts of the HTTP-protocol in a process algebra. We used counter-examples from conformance checks in the CADP toolbox as test-purposes for the test-case generator TGV [13]. The hundred generated mutation tests did find some unexpected behaviour in the conditional page requests to Apache.

Later, we optimised the technique for testing SIP registrars used in voice-over-IP applications, see e.g. [20]. Here, we developed our own input-output conformance checker (ioco) for generating mutation test cases. In one experiment the mutation tests detected one additional fault in the commercial implementation that was not revealed by other model-based testing techniques.

An interesting alternative to process algebras like LOTOS is the coordination language REO [9]. It is a visual modelling language for expressing a network coordinating the communication between a set of components. The coordination is exogenous, which means that the network is responsible for connecting and synchronising the communication. This new language for protocols opens new
opportunities for mutation. For example, exchanging one type of a connector by another, changes the coordination pattern of a network. Hence, new fault models can be expressed by single (first order) mutations. The basis for test case generation was a new relational REO semantics [17]. This formulation made it possible to adopt our earlier theoretical results [4].

*Embedded systems* are another line of research. We developed a tool chain comprising a translator from UML to a version of Back’s Action Systems [16] and a newly developed conformance checker for Action System models [11]. The tool can also handle the mutation testing of hybrid systems. Action systems are a kind of guarded command language for modelling concurrent reactive systems [10].

Our test case generator is an *ioco* checker for Action Systems. It takes two Action Systems, an original and a mutated one, and generates a test case that kills the mutant. It expects the actions being labelled as input, output and internal actions. For non-deterministic models a tree-like adaptive test case is generated. The tool was implemented in Sicstus Prolog exploiting the backtracking facilities during the model explorations.

Different strategies for selecting the test cases are supported: linear test cases to each fault, adaptive test cases to each fault, adaptive test cases to one fault. The test-case generator also checks if a given or previously generated test case is able to kill a mutant. Only if none of the test cases in a directory can kill a new mutant, a new test case is generated. Furthermore, the tool is able to generate test cases randomly. Our experiments showed that for complex models it is beneficial to generate first a number of long random tests for killing the most trivial mutants. Only when the randomly generated tests cannot kill a mutant, the computationally more expensive conformance check is started. The different strategies for generating test cases are reported in [2].

3 Symbolic Mutation Testing

We currently investigate different symbolic analysis techniques to address state space explosion: constraint solving and SMT solving are promising candidates. However, for reactive systems with long input-output traces, we cannot simply translate the non-conformance problem to one big formula and let the solvers do the job. A clever combination of normal form transformation, directed search and solving is necessary. Note that the solving of non-deterministic models is complex, since the formula includes negation [7].

First experiments with our tool based on a constraint solver have shown promising results [6]. By now, our symbolic tool has been applied to a car alarm system. The refinement checks for 207 mutants require 19 seconds, whereas our previous explicit *ioco* checker, spends 68 seconds for the same set of mutants [5]. Another implementation using the SMT solver Z3 shows similar good performance.
Generating test cases from mutants is computationally costly. This might be a reason for the limited amount of research in this direction. However, recent results show that for many systems under test this can be put into practice.

**Acknowledgement.** The recent research has received funding from the ARTEMIS Joint Undertaking under grant agreement No 269335 and from the Austrian Research Promotion Agency (FFG) under grant agreement No 829817 for the implementation of the project MBAT, Combined Model-based Analysis and Testing of Embedded Systems. The work was also funded by the Austrian Research Promotion Agency (FFG), program line “Trust in IT Systems”, project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL).

**References**

Spill Code Placement for SIMD Machines

Diogo Nunes Sampaio, Elie Gedeon, Fernando Magno Quintão Pereira, and Sylvain Collange

Abstract. The Single Instruction, Multiple Data (SIMD) execution model has been receiving renewed attention recently. This awareness stems from the rise of graphics processing units (GPUs) as a powerful alternative for parallel computing. Many compiler optimizations have been recently proposed for this hardware, but register allocation is a field yet to be explored. In this context, this paper describes a register spiller for SIMD machines that capitalizes on the opportunity to share identical data between threads. It provides two different benefits: first, it uses less memory, as more spilled values are shared among threads. Second, it improves the access times to spilled values. We have implemented our proposed allocator in the Ocelot open source compiler, and have been able to speedup the code produced by this framework by 21%. Although we have designed our algorithm on top of a linear scan register allocator, we claim that our ideas can be easily adapted to fit the necessities of other register allocators.

1 Introduction

The increasing programmability, allied to the decreasing costs of graphics processing units (GPUs), is boosting the interest of the industry and the academia in this hardware. Today it is possible to acquire, for a few hundred dollars GPUs with a thousand processing units on the same board. This possibility is bringing together academics, engineers and enthusiasts, who join efforts to develop new programming models that fit the subtleties of the graphics hardware. The compiler community is taking active part in such efforts. Each day novel analyses and code generation techniques that specifically target GPUs are designed and implemented. Examples of this new breed include back-end optimizations such as Branch Fusion [10], thread reallocation [29], iteration delaying [7] and branch distribution [17]. Nevertheless, register allocation, which is arguably the most important compiler optimization, has still to be revisited under the light of graphics processing units.

Register allocation is the problem of finding locations for the values manipulated by a program. These values can be stored either in registers, few but fast, or in memory, plenty but slow. Values mapped to memory are called spills. A good allocator keeps the most used values in registers. Register allocation was already an important issue when the first compilers where designed, sixty years ago [2]. Since then, this problem has been explored in a plethora of ways, and today an industrial-strength compiler is as good as a seasoned assembly programmer at assigning registers to variables. However, GPUs, with their Single Instruction, Multiple Data (SIMD) execution model, pose new
challenges to traditional register allocators. By taking advantage of explicit data-level parallelism, GPUs provide about ten times the computational throughput of comparable CPUs [19]. They run tens of thousands of instances (or threads) of a program at the same time. Such massive parallelism causes intense register pressure, because the register bank is partitioned between all threads. For instance, the GeForce 8800 has 8,192 registers per multiprocessor. This number might seem large at first, but it must be shared with up to 768 threads, leaving each thread with at most 10 registers. It is our goal, in this paper, to describe a register allocator that explores the opportunity to share identical data between threads to relieve register pressure.

In this paper we propose a Divergence Aware Spilling Strategy. This algorithm is specifically tailored for SIMD machines. In such model we have many threads, also called processing elements (PEs), executing in lock-step. All these PEs see the same set of virtual variable names; however, these names are mapped into different physical locations. Some of these variables, which we call uniform, always hold the same value for all the threads at a given point during the program execution. Our register allocator is able to place this common data into fast-access locations that can be shared among many threads. When compared to a traditional allocator, the gains that we can obtain with our divergence aware design are remarkable. We have implemented the register allocator proposed in this paper in the Ocelot open source CUDA compiler [12], and have used it to compile 46 well-known benchmarks to a high-end GPU. The code that we produce outperforms the code produced by Ocelot’s original allocator by almost 21%. Notice that we are not comparing against a straw-man: Ocelot is an industrial quality compiler, able to process the whole PTX instruction set, i.e., the intermediate format that NVIDIA uses to represent CUDA programs. The divergence aware capabilities of our allocator have been implemented as re-writing rules on top of Ocelot’s allocator. In other words, both register allocators that we empirically evaluate use the same algorithm. Thus, we claim in this paper that most of the traditional register allocation algorithms used in compilers today can be easily adapted to be divergence aware.

2 Background

C for CUDA is a programming language that allows programmers to develop applications to NVIDIA’s graphics processing units. This language has a syntax similar to standard C; however, its semantics is substantially different. This language follows the so called Single Instruction, Multiple Thread (SIMT) execution model [14][15][20][21]. In this model, the same program is executed by many virtual threads. Each virtual thread is instantiated to a physical thread, and the maximum number of physical threads simultaneously in execution depends on the capacity of the parallel hardware. In order to keep the hardware cost low, GPUs resort to SIMD execution. Threads are bundled together into groups called warps in NVIDIA’s jargon, or wavefronts in ATI’s. Threads in a warp execute in lockstep, which allows them to share a common instruction control logic. As an example, the GeForce GTX 580 has 16 Streaming Multiprocessors, and each of them can run 48 warps of 32 threads. Thus, each warp might perform 32 instances of the same instruction in lockstep mode.

Regular applications, such as scalar vector multiplication, fare very well in GPUs, as we have the same operation being independently performed on different chunks of data.