Charles University
Faculty of Mathematics and Physics

HABILITATION THESIS

Jan Kofroň

Verification of Software

Computer Science, Software Systems

Prague, Czech Republic 2018

Contents

Preface

The thesis presents selected results in the area of specification and verification of software properties. The work has been carried out during my stay at the Department of Distributed and Dependable Systems (formerly Distributed Systems Research Group) of the Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, and Forschungszentrum Informatik, Karlsruhe, Germany.

The selected topics include two main directions—first, it is the problem of semantic specification of software behavior, with a focus on component software. Second, we address the problem of efficient verification of software in general, in particular improving scaling of explicit and symbolic verification methods as well as providing a scalable yet precise static analysis algorithms for dynamic languages. The thesis consists of published research papers (selecting those that summarize the achieved results in particular topics) and connecting comments to make the text seamless as much as possible.

Apart from the papers, there are also research results in the form of taking part in international and national projects and organization of international conferences and workshops. The international projects include a bilateral project with France Telecom "Component Reliability Extensions for Fractal component model", FP7 European project "Q-ImPrESS", FP7 Marie Curie ITN project "Relate", and FP7 FET Proactive Initiative project "Ascens". The national projects include several ones funded by the Czech Science Foundation, in particular 102/03/0672, 201/03/0911, 201/06/0770, 201/08/0266, P103/11/1489, 14-11384S, and 17-12465S.

Major partners in the aforementioned collaborative research projects include Orange S.A., formerly France Télécom S.A., France, Universität Karlsruhe, Germany, Univerzità Svizzera della italiana, Lugano, Switzerland, and Vysoké učení technické v Brně, Czech Republic.

The research presented in the thesis is of a collective rather than individual nature. The software prototypes mentioned in the thesis are large piece of software; it is beyond abilities of an individual researcher to bring them to a working state in a reasonable amount of time. The published research papers were included with a list of all the contributing authors, while the connecting comments are mine.

I am grateful to my colleagues from the Department of Distributed and Dependable Systems (formerly Distributed Systems Research Group). Jiří Adámek, Rima Al Ali,

Paolo Arcaini, Lubomír Bulej, Jakub Daniel, Ilias Gerostatopoulos, David Hauzar, Petr Hnětynka, Viliam Holub, Pavel Jančík, Pavel Ježek, Tomáš Kalibera, Lucia Kapová, Michał Kit, Michal Malohlava, Vladimír Mencl, Pavel Parízek, Tomáš Poch, Tomáš Pop, Ondřej Šerý, Viliam Šimko, and Jiří Vinárek have all participated in the research activities relevant to the thesis and therefore have made this work possible. A special thank belongs to František Plášil, Petr Tůma, and Tomáš Bureš for not only participating in research but also leading the department (group).

I am also grateful to my colleagues from Forschungszentrum Informatik, Karlsruhe, in particular Steffen Becker and Mircea Trifu, who have welcomed me during my visit and led the research activities.

Jan Kofroň

CHAPTER 1

Introduction

Software is ubiquitous. Its reliability has become an important aspect of everyday lives and the errors within it can cause not only inconvenience at the user side, but also represent a significant danger to people's health and lives (e.g., [50]). A number of techniques thus have been developed to reduce the number of errors inside software. On the one hand, they include modern programming concepts and languages, practically eliminating some types of errors, such as type mismatch. On the other hand, testing and verification procedures can automatically reveal errors beyond syntax and type system of the used programming language. Unfortunately, the techniques of the latter group suffer from the theoretical complexity of the task; the required time usually grows exponentially with the size of the input, more often, the task is even undecidable. Testing, including both traditional application testing by human testers and more sophisticated methods such as unit testing, brings an additional burden in terms of effort to prepare and perform the tests. Testing definitely improves quality of software in most common use cases and scenarios; despite those come on mind of both testers and developers, they are particularly weak in covering the corner cases. Verification methods can successfully address this weakness; here, the effort is largely moved from humans to computers. Still, humans have to specify the desired properties of the software to be verified.

In order to produce reliable and dependable software, it is necessary to create a specification capturing semantics of the software and verify the desired properties thereof. This requires using an appropriate specification platform featuring verification tools. In the design, this can help to form the software architecture that enables the implementation to satisfy the properties. Later in the development process, the properties of the code have to be verified as well. Here, as the recent research shows, code level specification, usually in the form of annotations, is more appropriate than re-using the design specification and extracting information from it [29]. Nonetheless, maintaining consistency between the design and code level is a challenging task.

In this thesis, we focus on the methods improving software design and verification. This spans from various approaches to capturing the desired properties and modeling software behavior (semantics) to techniques of verifying the validity of properties at the code level.

First, we focus on **software behavior specification**. The challenge here is to capture the desired properties of the system, while still keeping the specification reasonably simple to be able to maintain it, communicate it, and analyze its properties. This especially employs using an appropriate specification language. With a sufficient expressiveness of the language at one hand, one has to keep in mind the complexity of the verification process on the other hand. This means that a compact yet expressive specification language, model checking (or another kind of formal analysis) of which is infeasible, is practically not very useful. In Chapter 2, we describe our research in this area, applied in the domain of software components. In our research, we focused on both development of a suitable specification language for software component behavior and development of algorithms allowing for implementation and application of the verification tools on real-live component applications. Our contribution is summed up in Chapters 4–7.

The second part of this thesis is devoted to **code verification**. This does not necessarily imply direct analysis of source code, but usually employs a pre-processing compilation phase, such as compiling Java code into Java bytecode and transforming C code into a formula in propositional logic. In this part, we first focus on explicit code model checking and verification of properties of Java programs. Here, we address the state explosion problem [21], which is the major obstacle in application of explicit model checking in practice. By means of dead-heap-variable analyses, we propose optimization of state space that reduces both the representation of particular states and the number of states to be explored. Our achievements are described in Chapter 8.

While being theoretically undecidable, code model checking can, in many cases, decide on validity of software properties; however, its inherent complexity, exponential in the size of the input program, limits its practical usability. Often, approximate results on property violations are of great value. The imprecision of such verification results includes not covering all the issues and reporting spurious ones. It is then up to the developer to investigate them and decide about their relevance. Such an approximate piece of information can be computed by means of **static analysis**. Its advantage over model checking is a lower complexity—while model checking works upon the state space of the input program, which can grow exponentially due to non-deterministic user input and thread interleavings, static analysis uses directly the code representation, without generating its state space. The challenge of designing a static-analysis algorithm is to balance its precision with its performance. Low precision of the algorithm results in many spuriously reported issues, while low performance of a precise algorithm hinders practical usability of the corresponding tool in the development process. In this area, we focus on static analysis of dynamic languages, such as PHP and JavaScript, to reveal vulnerabilities (potential security problems) of web applications. In particular, we aim at reasonably precise representation of heap data structures to reduce the imprecision (i.e., over-approximation) of the analysis. The results are described in Chapters 9 and 10.

The final part of the thesis forming Chapters 11 and 12 is devoted to the area of **symbolic verification**. In particular, we focus on improving efficiency of verification methods employing Craig interpolants. The interpolants are used for capturing semantics of programs (functions) in an over-approximate, i.e., simpler way, since a precise representation is practically useless due to its high complexity and size. Similarly to the previous part, one of the most burning issues here is the efficiency of the verification process; while it works well for simple programs, scaling to larger real-life applications is still difficult to achieve. Therefore, we address the problem of efficiency of the interpolation procedure. The smaller representation of interpolants and the faster way they are computed, the more efficient the overall verification is. In particular, we have extended the interpolation systems by the option to specify a partial variable assignment, thus focusing the computed function interpolant (*function summary*) on a specific context in which the corresponding function is used. This not only helps to generate a more compact interpolant representation, but also to make the computation procedure more efficient, both in terms of time and memory.

The main part of the thesis consists of the following papers and articles published at international conferences or in international journals:

Martin Mach, František Plášil, and Jan Kofroň: *Behavior Protocols Verification: Fighting State Explosion*, International Journal of Computer and Information Science, Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, March 2005

Jan Kofroň: *Checking Software Component Behavior Using Behavior Protocols and Spin*, Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, Seoul, Korea, ISBN 1-59593-480-4, pp. 1513-1517, March 2007

Jan Kofroň, František Plášil, and Ondřej Šerý: *Modes in component behavior specification via EBP and their application in product lines*, Information and Software Technology 51/1, pp. 31-41, Elsevier, January 2009

Tomáš Poch, Ondřej Šerý, František Plášil, and Jan Kofroň: *Threaded Behavior Protocols*, Formal Aspects of Computing, Volume 25, Issue 4 , pp 543-572, ISSN 0934-5043, Springer-Verlag, July 2013

Pavel Jančík and Jan Kofroň: *On Partial State Matching*, Formal Aspects of Computing, ISSN: 1433-299X, pp. 1–27, Springer Verlag, January 2017

David Hauzar and Jan Kofroň: *Framework for Static Analysis of PHP Applications*, Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015), July 2015

David Hauzar and Jan Kofroň: *WeVerca: Web Applications Verification for PHP*, Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM'14), Grenoble, France. LNCS, September 2014

Pavel Jančík, Jan Kofroň, Simone Fulvio Rollini, and Natasha Sharygina: *On Interpolants and Variable Assignments*, Proceedings of Formal Methods in Computer-Aided Design 2014, Lausanne, Switzerland, October 2014

Pavel Jančík, Leonardo Alt, Grigory Fedyukovich, Antti E.J. Hyvärinen, Jan Kofroň, and Natasha Sharygina: *PVAIR: Partial Variable Assignment InterpolatoR*, Proceedings of FASE'16, Eindhoven, Netherlands, April 2016

CHAPTER 2

Specification of Software Behavior

Specification of software behavior and its properties is an important part of the development process. Without a specification, one can hardly decide upon software correctness and whether it fulfills the original expectations. Moreover, correctness, or error freedom, of software is an aspect that is parametrized by the property or properties of interest; what can be perceived as correct behavior in some cases, might be erroneous in other ones. Therefore, the properties of interest are also to be captured.

2.1 Software components and services

Challenges and approaches. Software components have become a widely used mean of software construction, both in industry and academia. A plenty of component systems have been introduced, each one focusing on different aspects of the systems [11, 13, 14, 17, 18, 52]. Particular software components can be specified in terms of modeling (simulating) their behavior and expressing their (both functional and extra-functional) properties. It is worth mentioning that properties of a software component include not only its provided properties, but also those required by other components, usually communicating with this one (its environment).

Specification of software components is important for many reasons. First, for complex software, composing components together is a non-trivial task. One has to pay attention to fulfilling all the components requirements and achieve the intended functionality. Here, the specification not only serves for checking the composition correctness, but also provides the developer with a formal and precise description of the component functionality. In other words, it can be seen as a form of developer documentation.

Second, for mission-critical software, which is usually not that large, absence of errors and adherence to the specification is of particular importance. This includes software in areas such as avionics, medical devices, and military devices. Here, more than elsewhere, deviation from the specification can have tragic consequences. Specification of particular components helps to not only assemble the systems together and prove properties of particular parts, but also to devise the validity of the overall system specification out of these.

Specification of a software component does not involve just its type information, i.e., type specification of provided (and required) interfaces. An important part of the specification is also semantic information, i.e., description of the behavior of the component. This piece of information can take a form of a temporal logic formulae, such as LTL and CTL [21], or a model of abstract behavior of the component in the form of an automaton or generally a type of state-transition system [31, 33, 36].

The semantic specification of a software component can capture both its functionality and the assumed ways of using it. While the assumed way of usage is something one can imagine as a set of allowed sequences of method calls or messages issued on the component, the meaning of component's functionality varies across the component systems. Since components are usually understood as black or gray boxes, the functional specification usually narrows to contracts or rules relating the component's inputs and outputs. This includes pre- and post-conditions of particular provided methods (or services) [10, 38] and dependencies of usage of the required interfaces on particular provided ones [6, 8] and [44].

Contribution. Chapters 4–7 describe our development of a specification platform called *behavior protocols family*. Here, particular specification languages model behavior of particular components by means of communication protocols; this involves, for each component, specification of allowed sequences of provided method calls and for each provided method, a set of possible reactions of the component in terms of calling particular required methods (i.e., those of required interfaces). Having all the components forming a particular application specified in terms of behavior protocols, it is possible to check compatibility of the components, i.e., the correctness of their composition. This involves compatibility of the protocols of communicating components, but also, in hierarchical systems, correctness of realization of each composed component by its sub-components. We refer to those as *horizontal* resp. *vertical compliance* [8]. Since validation of the compliance relations by hand becomes practically impossible for applications consisting of tens and more components, (semi-)automatic tools performing these tasks become a necessity. Along the development, tools verifying correctness of component composition in terms of both horizontal and vertical compliance [6, 9] were implemented for each specification language in the context of the SOFA component system [18].

Verification of correctness of the communication among particular components significantly helps during the design phase. Nonetheless, adhering to the specification when implementing the system (which means implementing both *primitive* components in a programming language and *composite* components by composition of other ones) is a non-trivial task, too. Whereas the correctness of composition in case of the composite components is already established at the design phase, correspondence of behavior of primitive components with their specification is definitely not guaranteed. This problem is undecidable in general, usually even after (reasonable) limiting both the specification language and the programming one. Fortunately, methods for checking the correspondence working in most practical cases are available. In particular, for behavior protocols, these include [39].

Hereby, we have naturally stepped to the topic of the following chapter—verification of software properties at the code level.

CHAPTER 3

Verification of Source Code

Creating a detailed specification of software semantics and consequently maintaining its correspondence with the implementation is tedious and can be, by some, even perceived as superfluous. Currently, the trend in this area is to specify the properties directly in source code, usually by means of annotations [22, 26, 34]. Alternatively, the required properties can be defined generally, that is independently of actual code, usually just reflecting specifics of a particular domain [2] and [53]. The notion of a domain includes a particular programming language and its specific issues (absence of null-pointer de-references in C/C++ and Java) and particular software kinds, such as device drivers.

Verification of source code introduces a second-level check following the syntax and type checks performed by a compiler. It is desirable that this semantic check discovers any technical issue that may arise at runtime. Of course, this idea has its limits in terms of what is the intended behavior of the software piece—things that are correct and intended in one case can be wrong in another. This justifies the need for explicit specification in cases where software reliability is of particular importance.

3.1 Explicit model checking

Challenges and approaches. The idea of model checking dates back to early 1980s. Originally formulated for finite-state systems [19, 20, 25, 43], it allowed one to systematically and automatically verify properties of computational systems, if their model in the form of a finite state graph was available. The state space of complex software is often infinite (or so large that it is considered infinite from the analysis point of view), thus disallowing a straightforward application of model checking in general. Moreover, even for finite-state software, constructing its state space results in large transition systems, whose traversal is practically infeasible, anyway. Despite this, a lot of attention has been paid to developing appropriate methods to face these issues and as for today, several explicit code model checkers are available and even used outside academia in industry [30, 49].

Success of a particular explicit model checking method and the corresponding tool crucially depends on its practical usability. This means both its performance and the set of properties it is able to verify. As to the supported properties, most of the tools in this area are able to verify reachability properties, usually materialized as assertions inside the code. This allows for simplification of the overall model checking process, focusing on reduction of the state space needed to explore, and efficient traversal thereof. The reduction techniques are in particular important in case of multi-threaded programs, where the state explosion problem arises in a huge extent.

Partial Order Reduction (POR) [21] is a reduction technique exploiting the fact that two or more sequences of actions can result in the same state. Then, just a single sequence from such set needs to be explored, while the other ones can be omitted. In the context of code model checking, this corresponds to different thread schedulings when there is no race condition in the code. This reduction is implemented in a form in all explicit state model checkers today [30, 49] and significantly improves performance of these tools.

Other techniques focus on reduction of the state sizes, such as Dead Variables Reduction (DVR). Based on the information which variables are accessed during a future execution, i.e., the live variables, the representation of a state can be significantly reduced. The problem here is to identify the live variables at particular program states efficiently. Our research in this area is devoted to finding methods that identify future accesses to variables and objects on the heap, given a program state. Even though several results in this direction have been published so far [16, 35, 47], they usually restrict themselves just to local variables, or miss some important properties, such as sound support for multi-threaded programs. Successful reduction of state representation by removing their dead parts results not only in a more compact representation, but also decreases the number of explored states, since more states are considered as equal; in particular those differing just in the dead parts.

Contribution. Our results in this direction are described in Chapter 8. We address the problem of dead variable analysis for data stored at the heap. In particular, this involves fields of dynamically allocated objects, which are the most common type of objects in Java programs. We have developed and implemented two types of analysis, one aiming at speed and simplicity, while the other at precision and maximal state space reduction. The methods are based on tracking live fields during state space traversal and identification of states being equivalent in the values of these fields, i.e., omitting the dead ones. Our experiments prove the technique useful; it has the potential to significantly decrease not only the size of program state space, but also the size of particular state representation.

3.2 Static analysis

Challenges and approaches. In many cases, precise formal analysis of software properties is (computationally and sometimes even theoretically) infeasible. Here, static analysis [15] can be applied and provide very useful results. Static analysis works at the level of code representation rather than at the level of the associated state space, which results in better scaling and a wider set of programs that can be handled; the price paid is a lower precision of the results in terms of over-approximation. When aiming at not missing a violation of the specification, the method can yield false negatives; in other words, it can report spurious specification violations. Static analysis can be also used as a means of bug hunting. In such a case, it is more desirable that the reported specification violations are real, with the possibility of not discovering all of them. Both cases can be covered by static analysis, being set up different ways.

In our work, we focus on the first settings, i.e., we aim at discovering all potential issues; the decision if a reported problem is real or spurious is a task for the user/developer. The goal of static analysis can differ a lot in different cases, which also implies different kind of information that is computed by it. We are concerned with information about data types and values, based on which more specific information can be deduced; this can include information about what variables can be influenced by user input and thus are subject to security checks. This type of static analysis is called *data-flow program analysis*.

The high-level view on the data-flow-analysis algorithm is a cycle extending the set of possible values (or types) of each program variable, based on the possible values of variables influencing this one. The sets of possible values are extended until a fixed point is reached. Since the fixed-point computation can take very long, i.e., many iterations of the main cycle can be needed to reach it, *widening* of those sets of possible values that have met a threshold size is made. Generally, widening extends the set of possible values by adding new values without being a direct consequence of values of other variables. In particular, this can be realized by assuming that a variable can take any value of its domain. Widening thus becomes a source of over-approximation and, in turn, of reporting spurious issues. Even without widening, spurious issues can be reported because some combination of computed variable values might be infeasible in the given program.

To mitigate the impact of the over-approximation, several steps to improve the result precision have been made. In general, the algorithm can take into account various aspects of the program that is by default disregarded for the sake of analysis performance. *Flow-sensitive analysis* takes into account the ordering of particular statements, i.e., their mutual position in the program. Possible values of a variable can then be narrower. *Path-sensitive analysis* computes several versions of possible value sets for each variable parametrized by the conditional branches taken in the past. This is usually realized as adding the conditions that determine the particular branches. *Context-sensitive analysis* takes into account the program point from which a particular function or method is called and computes several versions of the possible value sets parametrized by this context. This kind of sensitiveness make sense just for inter-procedural analyses, which is not always the case. Static analysis can be made sensitive in any combination of the aforementioned dimensions, which usually improves the precision, but lowers its performance.

Contribution. In our work, we focus on security analysis of dynamic languages, especially PHP. We are interested in detecting *vulnerabilities*, i.e., possibilities of leaking and damaging data by means of passing malicious user input. The most famous types of vulnerabilities are SQL injection (SQLi) and Cross-site scripting (XSS) attacks [48].

It is not too difficult to design and implement fast data-flow analysis; the drawback is usually its low precision. On the other hand, it is not too difficult to come up with a precise analysis algorithm; the analysis then usually runs out of computational resources—memory and time. A tool being very fast but imprecise in terms of reporting many spurious warnings (next to the real ones) is not of much practical use. Similarly, a tool producing precise results, but being too slow or even running out of memory in most cases would not be more useful. Balancing these two aspects is a basic assumption for a success of an analysis tool.

To achieve a reasonable precision of the analysis algorithm, it is necessary to represent the data in a precise and easy-to-process way. In contrast to other state-of-the-art tools for security analysis of PHP, we decided to support also the heap data structures and their interconnections in terms of references with no particular nesting limit and most of the PHP5 constructs such as classes, the *eval* function, and dynamic includes [40].

We have created an analysis framework for dynamic languages (PHP, JavaScript) with a PHP front-end that demonstrates its usefulness. It provides the developers with an easy way to implement a custom kind of data-flow analysis. The framework processes the input program in two phases; in the first phase, the AST representation of the code is created. This is not an easy task, since in dynamic languages, names of included files can be computed at runtime, making the problem undecidable in general. Fortunately, constructing filenames is often limited to using basic string operations, so in most cases, this piece of information can be computed by means of static analysis. Consequently, the basic information about data types and values are computed. Providing a second-phase analysis is up to the developer. We have implemented a security analysis for PHP that was able to find a previously unknown real vulnerability inside real code. The results of our work are described in Chapters 9 and 10.

3.3 Symbolic verification methods

Challenges and approaches. Despite the success of explicit verification methods, they still have to face several issues hindering its practical usability. While the approach of state space traversal in an explicit way is not very complex in principle, the complexity and practical time (and often also memory) requirements stemming from the fact that the number of different thread schedulings grows exponentially in number of threads and the program size significantly limits scaling of these methods. Symbolic verification methods, on contrary, can handle the state explosion problem in much better way. Even though usually being of the same theoretical complexity as the explicit methods, symbolic methods can perform better in practice. However, they have their drawbacks, too. It is usually principally difficult to support different aspects of programs, such as dynamic heap allocation, and multi-threading, that are commonly used. Therefore, the available approaches and tools are often limited and their application in industrial settings is not easy. Nonetheless, significant advances have been recently made that contribute to practical usability of the related tools [27, 32, 42, 51].

Symbolic model checking, proposed by K. L. McMillan in his doctoral thesis [36] in 1992, employs binary decision diagrams for representation of set of states. For symbolic methods in code verification, different approaches are used. Some of them employ static analysis and abstract interpretation, while others exploit SAT and SMT solvers. In the latter case, the program is transformed to a propositional or a first-order-theory formula; consequently, a SAT or SMT solver is called to decide on satisfiability of the formula, corresponding to reachability of an error state. The hard part of the problem is thus yielded to a solver, while the verification tool itself is responsible for preparing the solver input and interpreting the solver results. Since for large programs, precise formula representations are impractical due to their sizes, an abstraction method is to be employed. Here, Craig interpolation plays a central role.

Given an unsatisfiable formula in the form $A \wedge B$, *Craig interpolant* [24] is a formula I such that (i) $A \rightarrow I$, (ii) $B \wedge I \rightarrow \bot$, and (iii) I contains only variables common to both A and B. Interpolants can be used for over-approximating sets of states, e.g., those that are reachable after n steps of program execution. An interpolant can be perceived as a proof that no error state (represented by the B sub-formula) is reachable from within the states represented by the interpolant (containing all the states represented by the A sub-formula). Such over-approximation introduces a source of imprecision, which can manifest itself as a non-empty intersection of I and B, representing a spurious error-state reachability. On the other hand, the benefit of employing interpolants lies in a much smaller representation of sets of states compared to the original A sub-formula. Moreover, the spurious errors can be detected and the interpolant *refined*—modified to become more precise over-approximation of A not intersecting with B any more.

An interpolant is usually computed from a proof of unsatisfiability of $A \wedge B$. There are several algorithms for interpolant computation [37, 41, 46] called *interpolation systems*. Interpolants computed by different systems differ in size and in logical strength. The Labeled Interpolation System (LIS) [46] generalizes different approaches and formulates criteria for comparing the strength of different interpolants. It is worth mentioning that for different verification tasks, interpolants of different strength are needed. In addition, interpolants computed by a specific interpolation system have properties that others lack.

Since the motivation for using interpolants in program verification is to reduce the size of set-of-states representation, it is desirable that the interpolants are as compact as possible. Various techniques for achieving this goal are used; they employ reductions of the proof of unsatisfiability [12, 23, 28, 45], from which interpolants are computed, and optimizations of the interpolant construction itself [45]. Smaller interpolants not only save memory, but also the time in the subsequent verification steps in which they are involved.

Contribution. In our work, we focused on faster computation of smaller interpolants by exploiting partial variable assignments. Such an assignment corresponds to ignoring parts of the program as a consequence of added knowledge about, e.g., method parameters. In turn, this results not only in potentially smaller interpolants, but also in more efficient computation of them. Moreover, it does not restrict the application area, since in the case of an empty variable assignment, our technique is equivalent to the standard ones. Our results in this direction are described in Chapters 11 and 12.

CHAPTER 4

Behavior Protocols Verification: Fighting State Explosion

Authors: Martin Mach, František Plášil, and Jan Kofroň

[8] *International Journal of Computer and Information Science, Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, March 2005*

Behavior Protocols Verification: Fighting State Explosion

Martin Mach[1], Frantisek Plasil[1,2], Jan Kofron[1]

Charles University, Prague
Academy of Sciences of the Czech Republic

Abstract

A typical problem formal verification faces is the size of the model of a system being verified. Even for a small system, the state space of the model tends to grow exponentially (state explosion). In this paper, we present a new representation of state spaces suitable for implementing operations upon behavior protocols of software components [1]. The proposed representation is linear in length of the source behavior protocol. By trading space for time, it allows handling behavior protocols of "practical size". As a proof of concept, two versions of a verification tool based on the proposed technique are discussed.

Keywords*:* Formal verification, software components, state explosion, behavior protocols, parse trees.

1. Introduction and motivation

The traditional verification techniques of program correctness are *testing* and *simulation*. However these techniques suffer from two major problems: (i) A working prototype is necessary for the verification, which inherently means belated error discovery within the development cycle. A remedy may require a major change in the program's architecture, which may be very costly in late design stages. (ii) It is usually hardly possible to test all the potential interactions with the program's environment so that some errors may remain undetected during the development, being discovered as late as by an end user.

Formal verification is a well-established method for correctness checking which can be employed during the whole program development cycle. The complete program is described via a mathematical model the properties of which can be verified with the assistance of verification tools. However, as forming of the actual model can be quite complicated, these tools are usually not easy to employ. Another important problem is that the representation of the state space associated with the model tends to exhaust all the memory available for a particular verification tool (the "state explosion" problem).

In this paper, we focus on formal models targeting behavior description of software components. In particular, we address the issue of efficient memory representation of the *behavior protocols* [1], which allows behavior compliance checking of cooperating components.

1.1. Components and behavior

Components are modern foundations of building software applications. Frequently understood as a design entity, a component *provides* some services to its environment and *requires* other services from the environment (other components). A service is usually described as an interface (and the methods in this interface). Therefore, in a typical component model, a component features both *provided* and *required* interfaces, like in Darwin [14] and Fractal [15].

In addition to defining interfaces at the syntax level, some of the component models partially capture also the semantics of components by specifying the desired/allowed sequences of method invocations (behavior of components). Such component models include Wright[5], Darwin[14], and SOFA[3]. In this paper, we focus on the behavior specification via behavior protocols [1] employed in SOFA, an open source component model [3].

1.2. Behavior protocols

A behavior protocol is a regular expression-based expression describing behavior at different levels of granularity (interface, interplay of all interfaces of a component, composition of several components). A behavior is a language over symbols that denote either the start or end of a method invocation (*events*). A behavior protocol features additional operators to enhance expressiveness. These additions do not break regularity of the languages described by behavior protocols. We provide only a basic overview of behavior protocols, for further reference we refer the reader to [1] and [4].

[1]Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1,
Czech Republic
{mach, plasil, kofron}@nenya.ms.mff.cuni.cz,
http://nenya.ms.mff.cuni.cz

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
Pod Vodarenskou vezi 2, 182 07 Prague 8,
Czech Republic
plasil@cs.cas.cz, http://www.cs.cas.cz

Syntax. The symbols denoting *events* are used to describe synchronous and asynchronous method invocations and have the following syntax:

```
(type, interface_name, event_name, flag)
```

where `type` indicates whether `event_name` determines a method invocation accepted on `interface_name` (?), emitted on `interface_name` (!), or it is an internal event taking place within a composed component (τ). Further, `flag` denotes whether the event is a method invocation request (↑) or response (↓). As an example, the acceptance of synchronous call invoking the method b on an interface a is expressed as `?a.b↑ ; !a.b↓`.

Semantics. In addition to the operators defined for regular expressions, i.e. ; (sequencing), + (alternative), * (repetition), several new operators are added to handle restriction, parallelism, and composition. For the purpose of this paper, it is sufficient to mention the operator | (and-parallel) which produces an arbitrary interleaving of traces generated by its operands.

Example. Consider a component representing a file. It provides one interface that contains five methods to manipulate the file: `open`, `read`, `write`, `close`, and `status`. The supported behavior either (i) starts with calling `open`, then an arbitrary interleaving of `read` and `write` follows and finally `close` has to be called; or (ii) allows `status` to be called at anytime (in parallel with (i)). The corresponding behavior protocol takes the form (for simplicity we use shortcut `method_name` for `?method_name↑; !method_name↓`):

```
(open;(read+write)*;close)|status*
```

Compliance. Behavior protocols allow static testing of behavior compliance of tied components. This way questions like "Is it possible to safely replace a component by another one if we know their interfaces and behavior?" or "Is it possible to interconnect these two components if we know the behavior interplay on the provided and required interfaces of each of them?" can be answered. Basically, the components are compliant if they fulfill two conditions based on subset relations. The publication [1] describes the compliance concept thoroughly and also provides an algorithm of compliance verification.

State explosion. Basically, the state space associated with a behavior protocol is the state space of the finite automaton accepting the regular language generated the behavior protocol.

Above, we mentioned that formal verification has typically to cope with the state explosion problem. Also behavior protocols suffer from this problem, because the compliance is tested via the corresponding automata determined by the behavior protocols in question, since any parallel activity causes exponential growth of the state space. For example in the original SOFA verifier [3], the state space corresponding to an expression involving more than 13 parallel operators does not practically fit into the memory available for the verifier even on a decent PC.

1.3. Goals and structure of the paper

To target the problem mentioned above, we designed a novel automata representation, which significantly improves the efficiency of the compliance verifier. In the inherent space versus time tradeoff, it shifts the complexity towards time in such a way that it allows to solve practical problems at least twice as big as the original verifier could handle. The main goal of this paper is to present the basic idea of this novel representation and share with the reader the lessons we learned during experiments with the new verifiers.

The structure of the paper is following. In Section 2, we discuss the flaws of classical automata representations (Section 2.2), while the Sections 2.3 and 2.4 bring the core of the paper by introducing *parse tree automata* and their optimizations. In Section 3, we describe an experimental behavior protocol verifier based on parse tree automata and Section 4 describes an enhanced Java version of the verifier. In Section 5, we evaluate the proposed representation and compare it with other techniques addressing state explosion. Section 6 concludes the paper.

2. Behavior protocol representation

2.1. Representation and efficiency

Different representations of a state space corresponding to a behavior protocol (*expression* for short) have specific benefits and drawbacks. Such a situation makes any reasoning on the representation efficiency a complicated task.

To show the properties of different finite automata representations (*representation* for short), we have identified four criteria proved to be important for a successful choice of a particular representation. The chosen criteria are:

- *Size of representation* is the amount of the memory required to store a (state space) representation. This is determined by all the data structures involved.
- *Building time* is the time required to create the representation from an expression.
- *Space requirement of composed state identifiers* is the amount of memory required to identify the states in a state space.
- *Access time* is the average time needed to determine the list of transitions associated with a state.

2.2. Basic representation techniques

To illustrate how the evaluation criteria help (i) characterize different representation techniques and (ii) show trade-off between time and space complexity, we present an overview of two classical finite automata representation techniques.

Explicit representation is the most simple and straightforward technique to represent an automaton. All necessary information is *explicitly* held in memory – lists of states, transitions, and accepting states (as lists, hash tables, matrices, ...).

As to size of such representation, state explosion is very likely. Also building time is fairly low as the construction of a state space is usually done recursively by composing the state spaces of sub-expressions and as the whole state space has to be traversed during this construction.

On the other hand, explicit representations shine in access time and size of identifiers. Hardly anything can beat the usage of pointers in states identification and retrieving a list of transition from memory.

Size of a representation is the major drawback of explicit representation causing that verification tools avoid using it. As explained in [2], the original SOFA behavior protocol verifier uses this type of representation. States are implemented as Java objects holding lists of labeled references to other states.

Symbolic representation is a group of techniques that use a different approach. The required state space is not generated in advance as in explicit representations but it is rather computed on-the-fly. This approach brings two benefits in terms of fighting state explosion: (i) In most cases, very large numbers of states can be handled, and (ii) the unvisited portions of the space are not generated at all. However access time is slower than in explicit representation because several computations are needed to obtain a list of transitions. Also a state identifier is usually implemented via a composed data structure, hence consuming more memory than a state identifier in the explicit representation technique.

The most recognized member of the symbolic representation technique category is the Ordered Binary Decision Diagram (OBDD) [6] technique. An OBDD is an acyclic directed graph representing a Boolean function $f(x_1,\ldots,x_n) \rightarrow \{0, 1\}$. In this graph, the internal nodes correspond to functional arguments and the two possible terminal nodes correspond to the output of the function. The arguments appear in the same order on the path from the root to leaves (Fig. 1). However the size of an OBDD graph strongly depends on the order of the function arguments.

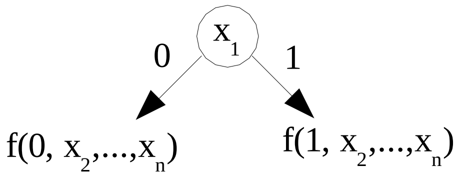


Fig. 1. Root of the decision diagram determining the function $f(x_1,\ldots,x_n)$

There are functions that are described by a graph of linear size for a specific argument ordering and of exponential size for a different ordering. And, unfortunately, deciding on an optimal ordering is an NP-complete problem [6].

To our knowledge, a precise evaluation of using OBDDs for representation of regular expressions has not been provided so far.

2.3. Parse trees and parse tree automata

To tackle the state explosion problem in representation of behavior protocols, we suggest and describe bellow *parse tree automata*, a novel symbolic representation technique.

Parse trees (also *syntax* or *expression trees*) are a common way to represent expressions in memory. They are mainly used to represent mathematic formulas and program source codes in compilers. Obviously, they are also capable to represent behavior protocols (Fig. 2).

A parse tree is a tree structure that describes a given expression unambiguously. When representing behavior protocols, the parse tree features the following important properties:

- Event symbols featuring in an expression appear only in the leaf nodes and operators in inner nodes of the corresponding parse tree.
- The operator nodes representing the repetition and restriction operators are unary; all others are binary.
- Every subtree describes an expression (valid behavior protocol).

The main advantage of parse trees is the size of representation, linearly dependent on the expression length and having no direct relation to the number of states. Also the building time is linear in the length of expression. Evaluation of access time and state identifiers' space requirement will be discussed later after we present parse tree-based representation technique (parse tree automata).

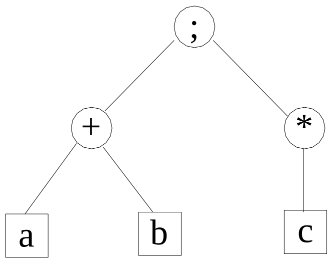


Fig. 2. A parse tree representing (a+b) ; c*

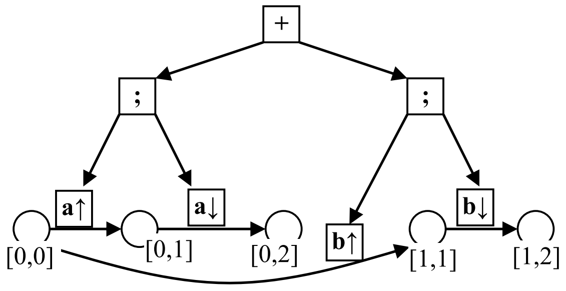
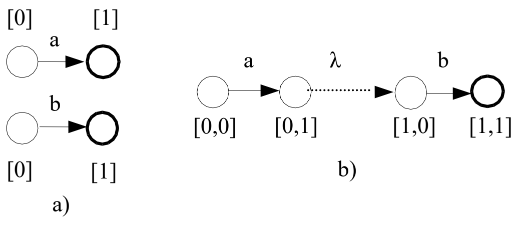


Fig. 3. Generating states and transitions of PTA. Circles represent states. Squares represent nodes of PT; [0,0] denotes the initial state.



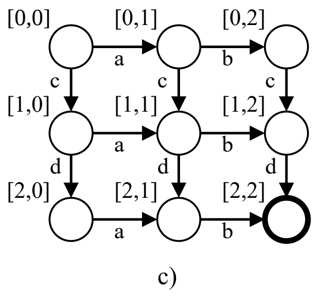


Fig. 4. a) Primitive automata for the "a" and "b" event symbols. b) PTA for "a;b". c) PTA for (a;b) | (c;d). Legend: A dotted arrow represents an implicit transition λ. State identifiers are in brackets (simplified).

Parse tree automata (PTA). Construction of a PTA follows the idea of recursive state space creation in the explicit representation technique. As PTA is a symbolic technique, the actual full state space of PTA is never represented as a single complex data structure. On the contrary, the key idea is to (i) directly represent only the parse tree (PT) of the expression and the *primitive automata* which accept the event symbols in the leaves of the parse tree, (ii) introduce composed state identifiers allowing to denote the current state and avoid unnecessary multiple traversals of PTA states, and (iii) define the transition function of PTA via recursive rules determining the (direct) transitions from a state, given its composed identifier. An example of PTA and its correspondence to a parse tree is illustrated on Fig. 3.

We will demonstrate the idea on three simple examples: (1) representation of a primitive automaton, (2) implementation of automata composition driven by the sequence operator, and (3) implementation of automata composition driven by the parallel operator. Automata compositions driven by the other operators are implemented in a similar manner (a detailed description is in [2]).

A primitive automaton has two states (initial and accepting) and a single transition between them. The transition label is an event symbol (Fig. 4a).

The sequencing operator expresses concatenation of the languages accepted by the left- and right - hand automata PTA_L and PTA_R. To create the respective composed automaton $PTA_;$, it is sufficient to establish implicit transitions (λ) from the accepting states of PTA_L to the initial state of PTA_R (Fig. 4b). The resulting set of accepting states in $PTA_;$ consists of the accepting states of PTA_R . The accepting states of PTA_L are added only if the initial state of PTA_R is accepting. Obviously, modifications of PTA_L and PTA_R are not necessary, since the implicit transitions λ are added in the implementation of the sequencing operator in $PTA_;$.

The parallel operator expresses arbitrary interleaving of all the words of the languages accepted by the left- and right hand automata PTA_L and PTA_R. In order to create the respective product automaton, it is sufficient to establish a state space "grid" and corresponding transitions as illustrated in Fig. 4c.

Composed state identifiers in PTA. To address the idea (ii) above, a state identifier must reflect the structure of the subtree of PT it is associated with and capture the state of the primitive automata within the subtree. For a specific PT, all the top-level identifiers will be of the same size (linear in the size of PT). As a technicality, memory allocation for state identifiers can cause substantial memory overhead. It is recommended to use an allocator that is optimized for allocating small memory chunks of the same size.

Time requirements for generating PTA transitions. The average time required is influenced by the number of PT nodes that have to be visited to calculate the list of transitions associated with a particular state. In each of these nodes some computation is necessary, as the potential transitions are determined on the fly. For each transition, also the state identifier of the target state has to be evaluated for keeping track of the states visited.

The number of visited PT nodes is greatly influenced by the actual operators encountered in PT. For example, for the standard regular expression operators only one subtree has to be visited. On the contrary, encountering a parallel operator means visiting both subtrees.

2.4. PTA optimizations

As discussed in Section 2.3, performance of PTA depends on the number of nodes in PT. If the number of PT nodes were reduced, performance would greatly improve. Therefore we experimented with several optimizations in PTA representation.

Multinodes. The idea of multinodes is to collapse the nodes of PT featuring the same operator into a single node. For example, in Fig. 5 collapsing means representing only a single node for the sequence operator '**;**' (associated with a list of PT subtrees a, b, c, d).

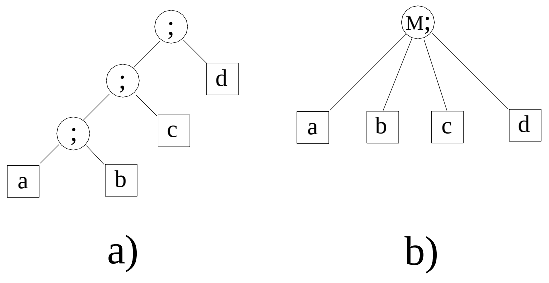


Fig. 5. a) Original parse tree. b) Parse tree with multinodes for the protocol a;b;c;d

This way, access time is greatly improved since less computation is required.

Forward cutting (of primitive automata). Removal of the transitions from the state space, which are discarded by a restriction operator, can be easily achieved by removing the affected event symbols nodes from PT.

Again, such optimization can produce PTs with a smaller number of nodes what results in a smaller state identifiers' space and improved access time.

Explicit subtrees. Since performance of explicit representation is very good for state spaces of "reasonable" size, it can be advantageous to combine both the PTA and explicit representations techniques. It is feasible to select those PT subtrees that imply a small state space (typically not featuring "many" parallel operators) and the states of which are generated more than once (e.g. forced by a parallel operator in a higher level of PT) and represent them via explicit automata embedded in PTA.

We implemented two verifiers based on the PTA representation technique ("Python verifier" and "Java verifier"). These implementations provide a flexible framework that allows simple addition of new parsers, optimizations, and verification backend alternatives as explained below.

3. Python implementation of PTA

Architecture platform. The Python verifier consists of three independent parts (parser, optimizer, backend) orchestrated by a simple application. All the parts of the verifier are implemented in Python [7]. However as the original Python provides only interpreted execution, we use the PSYCO [8] optimizing compiler to improve efficiency.

Parser. The goal of the parser is the creation of a PT representation from an expression. Currently only behavior protocols (Section 1.2) are considered as expressions.

Optimizer supports forward cutting of events and explicit subtrees optimizations. To choose a subtree that should be converted into an explicit automaton, a simple estimate of the number of states described by the subtree is based on assigning weights: the primitive automata get weight 2; for sequencing and alternative operators we sum the weights of the underlying automata, for parallel operators we multiply the weights. All the subtrees, the weight of which does not exceed a specific value, are addressed via explicit representation.

Backend alternatives. To enhance the application area of behavior protocols, we created three backend alternatives: compliance checking, visualization (using Aisee visualization tool [9]), and model checking (using Caesar/Aldebaran model checker [10]). Technically, compliance is checked by evaluating the subset relations of the compliance conditions (defined in [1]) via inspecting the emptiness of intersection of one set and the complement of the other. Visualization of a state space can ease up protocol perception, especially by highlighting counter examples produced by compliance verifier. When the state space gets too large for visualization, checking of specific properties is easier via a model-checking tool such as the Caesar/Aldebaran toolset. The bottom line is that independent tools are used for visualization and model checking; the verifier prepares only source files for them.

Since all backends use exhaustive traversal of the state space, we implemented a general depth-first-search algorithm that provides hooks for the algorithm specific computations during a state space traversal. The algorithm uses state space caching technique [12] to keep the list of visited states.

Implementation details. For particular operators, operator nodes are implemented as classes derived from a single interface that allows the client to obtain the initial state of the state space, list of transitions for a particular state, and list of the accepting states. In addition to the behavior protocol operators, we also implemented operators for language complement and automata product. A state identifier is implemented as a tree of Python 2-tuples.

Benchmarks. We used a slightly modified case study from [1] to assess performance of the Python verifier. The case study features a database server composed of two components and the protocol describing the server's behavior is:

```
!dbAcc.Open;
  (?d.Insert
       {(!dadbAcc.Insert; !dbLog.LogEvent)*}
+
  ?d.Delete
       {(!dadbAcc.Delete; !dbLog.LogEvent)*}
+
  ?d.Query
       {(!dadbAcc.Query)*}) *;
!dbAcc.Close
```

Our enhancements to the case study [1] pertain parallelism for accessing the functionality of the database server (replacing the '+' operator by '|') and the addition of two methods, `insert` and `modify`, to the server interface. The new methods are used in a similar way as their siblings. Using parallelism and the addition of new methods significantly increased the size and complexity of the related state space. These modifications are discussed in [2].

We created four benchmarks (1-4): In (1) we tested the compliance of the protocol described in the case study [1] with the composed protocol of nested components. Both state spaces in (1) were very simple and compliance verification was fast. In the subsequent benchmarks, we (2) replaced the alternative operators by parallel operators and (3) added the `insert` and (4) `modify` methods.

For illustration, the protocol in the (4) variant (most demanding as far as the size of state space generated is considered) was:

```
!dbAcc.open; (
  (?dbSrv.insert↑;!trans.begin;
  (!dbAcc.insert;!lg.logEvent)*;
  (!trans.commit+!trans.abort); !dbSrv.insert↓)
|
  (?dbSrv.delete↑;!trans.begin;
  (!dbAcc.delete;!lg.logEvent)*; (!trans.commit +
  !trans.abort); !dbSrv.delete↓)
|
  (?dbSrv.update↑;!trans.begin;
  (!dbAcc.update;!lg.logEvent)*;
  (!trans.commit+!trans.abort); !dbSrv.update↓)
|
  (?dbSrv.modify↑;!trans.begin; (!dbAcc.modify;
  !lg.logEvent)*; (!trans.commit+!trans.abort);
  !dbSrv.modify↓)
|
  (?dbSrv.query↑;!dbAcc.query;!dbSrv.query↓)
)*;
!dbAcc.close.
```

We benchmarked the consumed memory and required time of the original verifier and of the Python verifier with different optimizer settings. The speed without the forward cutting of primitive automata optimization was very poor, being significantly slower when compared to the original verifier; therefore this optimization was applied in all of the following benchmarks.

		(1)	**(2)**	**(3)**	**(4)**
		Simple protocol from [1]	**Protocol with |**	**Protocol with | and `insert`**	**Protocol with |, `insert` and `modify`**
	Original verifier	12.2MB	16.8MB	70.5MB	*Out of memory limit*
Python verifier	**0 explicit states**	5.9MB	6.2MB	12.3MB	72.4MB
	100 explicit states	5.9MB	6.6MB	10.1MB	46.4MB
	10,000 explicit states	5.7MB	6.7MB	9.9MB	40.2MB
	1,000,000 explicit states	5.7MB	6.4MB	14.0MB	70MB

Table 1. Memory benchmark results of the original verifier and the Python verifier for various sizes of explicit subtrees measured by number of their states.

		(1)	**(2)**	**(3)**	**(4)**
		Simple protocol from [1]	**Protocol with |**	**Protocol with | and `insert`**	**Protocol with |, `insert` and `modify`**
	Original verifier	800.0%	102.9%	197.0%	*Out of memory limit*
Python verifier	**0 explicit states**	100%	100%	100%	100%
	100 explicit states	123.1%	48.9%	45.8%	44.6%
	10,000 explicit states	123.1%	48.9%	34.0%	32.8%
	1,000,000 explicit states	123.1%	97.1%	45.0%	28.2%

Table 2. Relative time requirements: The original verifier vers. Python verifier for various sizes of explicit subtrees measured by number of their states.

The explicit subtrees optimization was applied to a different numbers of states embedded in explicit representation: 0 (no optimization), 100, 10,000, and 1,000,000. The results of the benchmarks were a little bit surprising: The Python verifier (based on PTA) with forward cutting of primitive automata outperformed the original SOFA verifier (based on explicit representation). However, there was a major difference in CPU time dedication: The original verifier spent most of the time by creating the explicit representation, while the actual verification was very quick (about two seconds for (3)). The Python verifier spent some time on optimizations and a significant amount of time on verification. The time spent by the optimizer heavily depended on the size of explicit subtrees. For example, in (4) the creation of explicit subtrees with 1,000,000 states took about 135 seconds. The increase of the overall execution time of (2) and (3) (comparing 10,000 and 1,000,000 states) was caused by the time necessary for creation of explicit subtrees.

4. Java implementation of PTA

To fully incorporate a checking tool into the SOFA environment, we decided to reimplement the Python verifier in the Java language.

The Java verifier uses the approach and techniques employed in the former Python verifier, but it introduces new optimizations and backend features. By these optimizations, both time and space requirements decreased and, therefore, the complexity of the protocols that can be checked was pushed a bit further.

Optimizations. Besides the optimizations included in the Python verifier (explicit subtrees and forward cutting), the multinodes optimization (Fig. 5) was implemented and found very beneficial. This optimization is performed during the construction of a parse tree in a straightforward, efficient way.

Backend alternatives. In the Java verifier we implemented only two backends: compliance checking and visualization, since these two had been identified as the most frequently needed.

For visualization, we decided to use the dot tool of the Graphviz package [16], since it is freely distributed and its features greatly suffice for our purposes. The visualization backend is able to provide both protocol parse tree and graph of the PTA state space. Since the dot tool supports, among other types of output, the Virtual Reality Modeling Language (VRML), this format can be advantageously used for complex protocols both to get the whole picture of the automaton and zoom into its specific parts.

Implementation details. Because of the differences between Python and Java, we had to cope with a lot of specific problems when rewriting the verifier from Python to Java. A main problem was the state identifiers in Java (handled internally by Python): As implied by the argumentation in Section 2.3, we needed state identifiers that could be computed fast and consume as small amount of memory as possible. We could not use Java references, because of the on-the-fly state generation (potentially repeated for a particular state).

Therefore, each state is represented by a *state tree*, where its leaves represent the states of primitive automata, while inner nodes represent the state of the composed automata corresponding to the nodes' subtree. The state identifier of a primitive automaton indicates its active state (0 or 1) (Fig. 4a). The state identifier of a composed automaton is created as concatenation of its children's identifiers. Thus, the resulting state identifier reflects the structure of PT, uniquely denotes a state within the state space, and its length is linear in the size of PT. Obviously, the state identifier of the main automaton is determined at the root of the state tree. The state identifiers are computed in a lazy way (only when actually needed) and are stored in a cache. Traversal of the state space employs frequent comparison of the identifiers (that is quite fast). Even though the computation of state identifiers was optimized for speed, it is still the most time consuming operation in the checking process (since it is performed for each state visit).

Benchmarks. We employed two types of benchmarks: the first type was focused on the benefits of particular optimizations in the Java verifier and the second one on a comparison of performance of the three verifier versions: the original verifier (written in Java), and Python and Java PTA verifiers. Always we used protocols of various complexity; both real-life and "academic" protocols inducing large state spaces were checked.

The real-life protocols included again a set of database server protocols similar to those used in Section 3. The "academic" protocols involved only the parallel operator (such as `a | b`, `a | b | c`, …), which is one of those causing the exponential growth of the state space, so that using it enabled us to generate really large state spaces and easily compute their sizes.

The optimization benchmarks have shown that disabling the forward cutting optimization results in a very poor performance. This is caused by the complement operator expanding the state space to an enormous size. Hence, as well as in Section 3, forward cutting is used in each of the benchmarks below. The benefits of the other types of optimization depend on the concrete structure of the protocols being checked (Table 3). For example, in the case of "academic" protocols using the parallel operator, the most worthwhile optimization are multinodes; the explicit subtrees optimization cannot be used here, because the states of the automaton represented by the only (multi-) node in the parse tree are used only once. While checking the real-life protocols, the explicit subtrees optimization is most beneficial.

Since the most important parameter of the protocol verifier is the state processing speed, in Table 4 we present the comparison of all verifiers based on checking the “academic” protocols (the results of checking the real-life protocols are not so interesting). A comparison of memory requirements is not involved since it is clear from Table 1 that a PTA representation requires a smaller amount of memory than a corresponding explicit representation. In all benchmarks considered below all optimizations were applied.

In the case of “academic” protocols, the Java verifier is faster than the Python verifier even if we turn off the multinodes optimization; the state processing is about two times faster in the Java verifier, which is probably caused by the fact that the Java Virtual Machine outperforms the Python Psyco compiler. On the other hand, the construction of the explicit subtrees is much slower in Java because of the evaluation of the state identifiers; the Python verifier is also able to keep more states (and larger explicit subautomata) in memory, because its state identifiers are shorter. In any case, the PTA approach beats the original explicit state representation.

	Forward cutting only	**All optimization**	**No multinodes**	**No explicit subtree**
Academic (parallel)	100%	76.2%	100.8%	75.7%
Real-life	100%	50.5%	67.7%	81.4%

Table 3. Average relative time the Java verifier spent by checking with various optimizations enabled.

Number of parallel operators used	**Original verifier**	**Python verifier**	**Java verifier**
6	100%	38.3%	22.3%
7	100%	16.5%	7.7%
8	100%	6.9%	2.3%
9	100%	2.7%	0.7%

Table 4. Relative time spent by checking the “academic” (parallel) protocols by all verifiers.

5. Evaluation and related work

Evaluation. The idea of using parse trees for symbolic representation has proven to be useful for the verification of behavior protocols. The newly implemented verifiers outperformed the original SOFA one, both in time and space complexity. The results provide a solid base for the hypothesis that the symbolic PT representation supported by the forward cutting of primitive automata optimization outperforms the explicit representation. Nevertheless, this hypothesis is still to be justified by a more thorough benchmarking.

While experimenting with the proposed technique of PTA, we identified the following implementation issues: (i) Access time was significantly influenced by applying the forward cutting of primitive automata optimization. This implies there might be huge method calls overhead during the list of transition computation. (ii) Another access time improvement may be achieved by an adaptive selection of explicit subtrees, since our benchmarks showed that access time depends on the size of the parse trees as well. (iii) State identifiers may involve allocation of small structures what means a significant memory allocation overhead. Using a customized allocator, the amount of consumed memory might be greatly decreased in both Python and Java cases.

Related work. To our knowledge, there is no other work that would focus on evaluation of an optimal representation for regular expressions. Therefore we can provide bellow only a comparison with representation techniques that face state explosion in other transition systems.

Space explosion handling techniques can be divided into two categories: (i) efficient representation of the state space and (ii) structural simplification of the state space. OBDDs (mentioned in Section 2.2) and their derivatives Multiple-value Decision Diagrams (MDDs) [11] and Multiple-terminal BDDs (MTBDDs) [13] are typical representatives of (i). All these representations suffer from the optimal ordering problem [6]. There are heuristics developed, but they cannot guarantee the optimal results. Structural simplification of the state space is usually achieved by employing several level of abstraction in model description [14]. In fact, this technique was implicitly employed in SOFA component model [3], since a behavior protocol is always defined for a particular level of component nesting (as opposed to [14]) and behavior compliance is evaluated separately at the adjacent levels of component hierarchy.

6. Conclusions and future intentions

In this paper, we presented a new representation of a state space called Parse Tree Automata that addresses the state explosion problem encountered in behavior protocol checking. PTA fights this problem successfully for behavior protocols of “practical size”. Both verifiers based on this representation outperformed the original verifier implemented within the SOFA project not only in memory requirements but in the speed of verification as well.

In the future, we intend to focus on handling the implementation issues described in Section 5. In particular we would like to implement an adaptive version of explicit subtrees optimization and make experiments with various memory allocators.

Acknowledgement

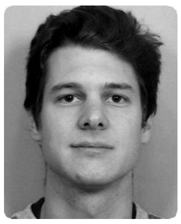
The work was partially supported by the Grant Agency of the Czech Republic (project number 102/03/0672). We are grateful to our colleagues Vladimir Mencl and Jiri Adamek for valuable comments.

References

[1] F.Plasil and S.Visnovsky: Behavior protocols for software components. IEEE Transactions on SW Engineering, 28 (9), Sep 2002.
[2] M.Mach: Formal verification of behavior protocols. Master thesis, Dept. of SW Engineering, Charles University, Prague, 2003.
[3] SOFA project, http://nenya.ms.mf.cuni.cz/sofa/
[4] F.Plasil and J.Adamek: Behavior Protocols Capturing Errors and Updates. Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS, University of Warsaw, 2003.
[5] R.Allen and D.Garlan: A Formal Basis For Architectural Connection. ACM Transactions on Software Engineering and Methodology, Jul 1997.
[6] C.Meinel and T.Theobald: “Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications”. Springer Verlag, 1998.
[7] Python, http://www.python.org
[8] PSYCO compiler, http://psyco.sourceforge.net
[9] Aisee visualization tool, http://www.aisee.com
[10] Caesar/Aldebaran model checker, http://www.inrialpes.fr/vasy/cadp/
[11] A.Srinivasan, T.Kam, S.Malik, and R.Brayton: Algorithms for discrete function manipulation. Int’l Conf. on CAD, 1990.
[12] P.Godefroid, G.Holzmann, and D.Pirottin: State-Space Caching Revisited. Formal Methods in System Design: An International Journal, 1995.
[13] M.Fujita, P.McGeer, and J.Yang.: Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. Formal Methods in System Design: An International Journal, 10, April 1997.
[14] D.Giannakopoulou, J.Kramer, and S.Cheung: Analysing the Behaviour of Distributed Systems using Tracta. Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol. 6(1), Jan 1999.
[15] E.Bruneton, T.Coupaye, and J.Stefani: The Fractal Composition Framework. Proposed Final Draft of Interface Specification version 0.9, The ObjectWeb Consortium, Jun 2002.
[16] Graphviz – open source graph drawing software, http://www.research.att.com/sw/tools/graphviz

Martin Mach is a former Ph.D. student at the Department of Software Engineering, Charles University, Prague. His research was focused on software verification methods, component behavior specification and model checking. He implemented a tool for checking compatibility of components in the SOFA component model (a platform for building component-based application). He is now a software designer and quality tester at Umeå Datakonsulter, Sweden.

Frantisek Plasil is a professor and the vice-chair at the Department of Software Engineering, Charles University, Prague, and also a senior researcher at the Institute of Computer Science of the Academy of Sciences of the Czech Republic, Prague. After receiving his PhD degree from the Czech University of Technology in Prague in 1978, he was with that university until 1994 when he joined Charles University. He has also held visiting positions at the University of Denver, Wayne State University, and the University of New Hampshire, Durham. His research, focused on component-based programming comprising middleware technologies, has been mainly carried out in framework of several projects, including SOFA/DCUP, TOCOOS/ Copernicus/Esprit, PEPiTA and OSMOSE in the ITEA/EUREKA program; the list of industrial partners includes IONA Technologies, France Telecom, Bull Grenoble, Alcatel, and MLC Systeme Ratingen. He is a member of the IEEE Computer Society.



Jan Kofron is a Ph.D. student at the Department of Software Engineering, Charles University, Prague. He is also a member of Distributed Research Group at the department. His research is focused on software verification methods, component behavior specification and model checking. He has implemented and extended a tool for checking compatibility of SOFA components.

CHAPTER 5

Checking Software Component Behavior Using Behavior Protocols and Spin

Authors: Jan Kofroň

[6] *Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, ISBN 1-59593-480-4, pp. 1513-1517, DOI: 10.1145/1244002.1244326, Seoul, Korea, March 2007*

Checking Software Component Behavior Using Behavior Protocols and Spin[*]

Jan Kofron
Institute of Computer Science
Academy of Sciences of the Czech Republic
kofron@cs.cas.cz

ABSTRACT

Using software components is a modern approach for building extensible and reliable applications. To ensure high dependability, a component application should undergo verification, e.g. model checking, to prove it has certain properties. The implementation of an application is usually too complex to be verified at a formal level; therefore, a model being an abstraction of the implementation is to be used. Behavior protocols [11] are a platform for modeling of software component behavior. In this paper, we propose a method for translation behavior protocols to Promela [7], which is consequently used as the input for the Spin model checker [7]. Having the Promela code describing the component behavior, one can efficiently check for the behavior compatibility and LTL (Linear Temporal Logic) properties of cooperating software components.

Keywords

Behavior protocols, Promela, Behavior specification, Verification.

1. INTRODUCTION

Using software components for building reliable (distributed) applications belongs to modern and promising trends of the future of software development. Each software component is potentially a subject for future reuse; therefore, clearly defined interface and semantics are a necessity. To combine components from various vendors, the developer needs a common way for component specification. Unlike description of component interfaces (ADL's, headers, ...), a standard for the specification of component semantics (behavior) has not been established yet.

In this paper, we focus on checking behavior compatibility in hierarchical component models like Fractal [12] and SOFA [6]. In these component models, there are two kinds of components: *primitive* components that have no internal structure (from the architectural point of view) and are directly implemented in a programming language, e.g. Java, and *composite* components consisting of other (either primitive or composite) components. The architecture of an application, i.e., the way the components are connected and nested, is described in an *Architecture Description Language* (ADL) file. Here, the specified component behavior or a link to an external file containing the specification may be present.

[*]This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

1.1 Goals and Structure of the Paper

The goal of our research is to devise a method for verification of behavior compatibility of cooperating components of real-life applications specified using behavior protocols [11] as well as verifying LTL (Linear Temporal Logic) properties of particular components.

Behavior protocols [11] are a method for specification of software component behavior in the SOFA [6] and Fractal [1] component models. Having an entire component application specified by behavior protocols, one can check for behavior compatibility of the application's components.

In this paper, we present a way component behavior compatibility written in behavior protocols can be checked in the Spin model checker [7]; since Spin uses Promela[7] as its input language, we also propose an algorithm for translating behavior protocols into Promela and discuss the benefits of this approach.

The structure of the paper is as follows: In Sect. 2 we describe behavior protocols. In Sect. 3, details of the behavior compatibility check are described, while in Sect. 4 we show how specified behavior can be converted into Promela. Sect. 5 evaluates the contribution. Sect. 6 discusses related work, while Sect. 7 concludes the paper and proposes possible directions of our future work.

2. BEHAVIOR PROTOCOLS

A *behavior protocol* [11] is an expression describing the behavior of a component; the *behavior* means the activity on component interfaces viewed as finite sequences (traces) of accepted and emitted method call events. A behavior protocol is syntactically composed of event denotations (tokens), operators, and parentheses. For a method m on an interface i, there are four event token variants:

Emitting an invocation:	$!i.m\hat{}$
Accepting an invocation:	$?i.m\hat{}$
Emitting a response:	$!i.m\$$
Accepting a response:	$?i.m\$$

Furthermore, several syntactic abbreviations of method calls are defined to improve the readability of the expresions.

The operators in behavior protocols include those used in regular-expressions ('+', ';', and '∗') and a special one (the parallel operator '|' generating an alternative of all possible interleavings of operands' event tokens).

As an example, consider the following behavior protocol:

```
!Callback.AddrInvalidated ∗
|
(
  ?Mgmt.UsePermanentDb^;
  (
    !IpMacPermanentDb.GetAddr ∗
    |
    (!Mgmt.UsePermanentDb$;
     ?Mgmt.StopUsingPermanentDb^)
  );
  !Mgmt.StopUsingPermanentDb$
)∗
```

The protocol describes behavior of a DHCP server, which can behave in two different modes: (1) IP addresses for new clients are automatically generated by the DHCP server (by a preconfigured pattern e.g. valid IP address range) or (2) the IP/MAC address mappings are permanently stored in an external database component and the DHCP server assigns IP addresses to new clients according to their MAC address and the mapping stored in the database.

This behavior is reflected in the behavior protocol as follows: parallel composition of the *!Callback.AddrInvalidated* token with the rest of the protocol means that the component may call this method any time during its execution. Additionally, the component is able to accept a *Mgmt.UsePermanentDb* method request. Before accepting a *Mgmt.StopUsingPermanentDb* method request, it must emit exactly one *Mgmt.UsePermanentDb* response and it may emit any number of *IpMacPermanentDb.GetAddr* calls. Before responding to the *Mgmt.StopUsingPermanentDb*, it accepts a potential response to the *IpMacPermanentDb.GetAddr* to accomplish this method call, if a request event of this method has been emitted and no corresponding response accepted yet. In the first mode, only *!Callback.AddrInvalidated* may be emitted, while in the second mode, which is entered by accepting the *Mgmt.UsePermanentDb ^* event, also methods of the *IpMacPermanentDb* interface are called. The second mode of DHCP server is exited by emitting the *Mgmt.StopUsingPermanentDb$* event.

3. COMPONENT BEHAVIOR COMPATIBILITY

In hierarchical component models, there are two behavior compatibility relations to be taken into account. The first compatibility relation describes the correctness of the communication among components on a particular level of nesting, while the subject of the other relation is to capture the compatibility between each component and its subcomponents. In behavior protocols, the correctness of communication of the former relation is called *horizontal compliance* or absence of composition errors. The latter one is denoted as *vertical compliance.*

For evaluation of both vertical and horizontal compliances a special composition operator *consent* [2] is used. This operator is basically a parallel composition operator synchronizing the protocols on a set of events — let us denote this set $\mathcal{S}$. When synchronizing, two complementary events (differing in their prefixes — '!event' and '?event') from $\mathcal{S}$ form a τ-event. The emitting and accepting events are thus executed in a single atomic transition. Unlike other composition operators, application of the consent operator yields not only the traces corresponding to correct communication among components, but also *error traces* describing the erroneous behavior. The consent operator is able to capture three types of composition errors — *bad activity*, *no activity*, and *divergence.*

The bad-activity error denotes a state when a component (according to its behavior protocol) may emit an event, but there is no other component able to accept such an event[1].

The no-activity error denotes a deadlock, i.e., a state where no component is able to perform any action (neither emit nor accept an event) and at least one component is not in an end state. As the components of an application under consideration are not required to form a closed system, some interfaces may not be bound; in this case, the emit events of such unbound required interfaces are considered to be accepted by "the environment" any time, while "the environment" is considered as being able to emit an event of an unbound provided interface only in the states a component is able to accept it. If this default behavior concerning unbound interfaces is not desirable, a component representing the environment may be constructed adjusting the expected behavior.

The divergence error[2] denotes presence of a cycle within the component behavior when there is no way to reach an accepting state. As mentioned in Sect. 2, behavior protocols describe only finite traces; therefore, each cycle from which no accepting state is reachable is considered as an error.

According to our experience, we believe that undesired infinite program execution appears very rarely when using behavior protocols as the specification platform; therefore we omit the detection of this type of errors in Promela models obtained from behavior protocols.

As an example of how the consent operator works, consider the following behavior protocols P_A and P_B and their consent composition $P_A \nabla_{\mathcal{S}} P_B$:

$$\mathcal{S} = \{\mathit{file.open}\hat{}, \mathit{file.open}\$, \mathit{file.read}\hat{}, \mathit{file.read}\$, \mathit{file.write}\hat{}, \mathit{file.write}\$, \mathit{file.close}\hat{}, \mathit{file.close}\$\}$$

$$P_A : ?\mathit{file.open};\ (?\mathit{file.read})*;\ ?\mathit{file.close}$$

$$P_B : !\mathit{file.open};\ (!\mathit{file.read}+!\mathit{file.write})*;\ !\mathit{file.close}$$

$$P_A \nabla_{\mathcal{S}} P_B : \ (\tau \mathit{file.open};\ (\tau \mathit{file.read})*;\ \tau \mathit{file.close}) + (\tau \mathit{file.open};\ \epsilon \mathit{file.write})$$

The composition of P_A and P_B protocols yields a bad-activity composition error caused by the *file.write* method

[1]Note that in the semantics of behavior protocols requests do not block, as opposed to e.g. Java method calls; therefore, the case when the recipient is not ready to accept a request is considered as an error.

[2]Sometimes referred to as *infinite activity.*

denoted by the ϵ*file.write* token.

4. TRANSLATING BEHAVIOR PROTOCOLS TO PROMELA

Behavior Protocol Checker [9] is a tool for evaluation of horizontal and vertical compliance on models described by behavior protocols. Evaluation of these relations employs exhaustive traversal of the model state space and thus is quite time-consuming. The current version of the Behavior Protocol Checker is limited to state spaces of the size in the order of magnitude of 10^8 states, which is not sufficient in some cases.

The Spin model checker [7] is a state-of-the-art explicit model checker featuring LTL checking abilities, bit-state hashing, and quite user friendly interface. It is able to traverse state spaces of several orders of magnitude higher sizes than Behavior Protocol Checker is. Additionally, there are a number of extensions of Spin extending the Promela language, e.g. dSpin [5], that can be used in the future for translating possible behavior protocols extensions.

These facts motivated us for formulating the "Protocols-to-Promela" translation rules allowing for checking for the composition errors and LTL properties of models described by behavior protocols in Spin.

4.1 Modeling of Communication

The main problem to solve when translating behavior protocols to Promela [7] is modeling of the component communication.

The first idea one probably gets is to use processes to model components and message channels to model communication among them. There are two modes regarding message treatment in Spin — in the first mode, the send-message command gets blocked in the case of the full message buffer, while in the second mode, the message gets lost in such a case. Unfortunately, in behavior protocols, we want each emit event that cannot be accepted immediately to cause a bad activity error. Thus, modeling such behavior using message channels would require incorporation of an algorithmic mechanism (e.g. message counting) to detect bad activity errors; still, bad activity would be probably detected at the end of the verification hardening an error trace construction and finding the error cause.

Another approach for modeling component communication is based on variables. Each component is modeled as a process; moreover, each method of an exported (provided / required) interface is associated with two boolean variables reflecting a wait for a call request (*wrq*) and response (*wrs*). Each time a component starts to wait for a method call, it sets the *wrq* variable to *true*. Later on, when another component decides to emit a call of this method, it first checks the value of *wrq*, and according to the value it either performs the call (it reassigns *false* to *wrq*) or stops the checking process by printing information about the error discovered (similarly with the *wrs* variable). As to the other types of composition errors, the no-activity is detected in a natural way as a Promela deadlock, and, as explained in Sect. 3, the divergence composition error is not detected at all.

4.2 Protocols-to-Promela Translation Rules

Now, we describe the proposed "Protocols-to-Promela" translation algorithm. As aforementioned, for each method, two boolean variables are declared[3]; we denote them *method variables*:

```
bool interface1_method1[2];
bool interface1_method2[2];
...
```

At the beginning, for each behavior protocol a Promela process is created and for all methods the protocol allows to accept in its initial state, the corresponding variable is set to true:

```
interface1_method1[0] = true;
interface1_method3[0] = true;
...
```

Consequently, each process notifies a special process `Main` about finishing its initialization. The `Main` process is used for running the other processes and synchronizing their execution at the beginning and at the end of the verification.

Now, the input protocols are parsed and corresponding Promela code is generated according to each protocol structure. The following table describes the mapping of the behavior protocol operators to Promela.

BP operator	**Promela mapping**
sequence ';'	';'
alternative '+'	`if...fi` with each guard representing the beginning of an alternative branch
repetition '$*$'	`do...od` with break
and-parallel '|'	new process types each representing a parallel branch

Additionally to these basic mapping principles, there are several issues to be addressed concerning particular operators.

If the '+' operator is used to combine various branches starting with an accept event ('?*interface.method*'), after accepting a particular request all the other requests must not be accepted. Therefore, the corresponding method variables have to be reset to false. Mixing of emit and accept events in various branches of an alternative operator is considered as a bad practice and therefore not supported.

If the protocol encapsulated by a repetition operator '*' starts with a request event, the `do...od` statement has to include additional conditional branch '`::skip -> break;`' for nondeterministic termination of the cycle to model any arbitrary number of protocol repetition. Note here that the semantics of this transformation includes also an infinite number of repetitions of the protocol inside the '`do...od`' statement.

If the protocol encapsulated by the '$*$' operator starts with an accept event, it is up to the other components to decide how many times they would call the component associated with this behavior protocol. To model the appropriate behavior and to avoid a deadlock at the end of the cycle, again, the '`do...od`' statement includes an additional conditional branch. It either accept the call following after the repetition part or, if there are no further events, the '`::endofrun -> break;`' statement terminating the cycle at the end of the verification is used. The '`endofrun`' variable is set to

[3]Because of the Promela syntax, the method identifiers used in protocols are modified in the Promela output from '`interface.method`' to '`interface_method`'.

`true` by the `Main` process when no other process is able to perform any action anymore.

As stated in the table above, the '`|`' ('`and-parallel`') behavior protocol operator is modeled using a new process for each parallel branch. The processes are created by the process representing the protocol and started simultaneously (ensured by the '`atomic`' statement). The parent process waits for termination of the child processes before it continues emitting and accepting further events. The notification is performed via a shared variable.

As the semantics of behavior protocols declares that various events may interleave during the execution, but only one event is executed at a time, we use the `atomic` statement to ensure the same semantics in Promela. A protocol waiting for, consequently accepting a method request, and finally emitting a method response looks as follows:

?*interface.method*

The corresponding Promela code fragment takes the form:

```
waitforrequest(interface_method);
acceptrequest(interface_method);
emitresponse(interface_method);
```

The counterpart, i.e., the component emitting the request and consequently accepting a response is described by the following behavior protocol:

!*interface.method*

Here, the corresponding Promela code fragment is:

```
atomic {
  emitrequest(interface_method);
  waitforresponse(interface_method);
}
acceptresponse(interface_method);
```

An example of translating the behavior protocol of DHCP server to Promela can be found in [8].

5. EVALUATION

We have successfully applied the proposed technique on a non-trivial component application [1] consisting of approximately 20 components. The verification time when using the proposed approach and the Spin model checker[4] took less than ten minutes, which we definitely consider as acceptable time.

The Promela code is generated from a behavior protocol in the time linear in the length of the input. Due to the way the semantics of behavior protocols is modeled in Promela, the resulting state space of Promela code is usually about ten times larger than the state space generated by the model specified by behavior protocols. This is because each event has to be explicitly awaited and also emitting an event means several lines of code. Compared with time required for verification of such systems using the proprietary Behavior Protocol Checker[9], Spin is more than an order of magnitude faster. Additionally, an arbitrary LTL property of the model can be verified.

As to alternative approaches, the first idea the reader probably gets after reading the sections above is to specify the component behavior directly in Promela. It is likely that the resulting code would be shorter than the code generated from behavior protocols. On the other hand, it would be significantly longer than a behavior protocols model. Nevertheless, the main problem lies in variables. Since Promela features various data types (integers, booleans, structures, enums), the application designer is in temptation to use them. Our experience shows that description of a slightly more complex system in Promela then usually yields a too large model state space impossible to traverse (even the use of bit-state hashing can't provide a reasonable level of reliability). Therefore, even a bit tricky because of absence of variables, behavior protocols provide a suitable specification platform for testing behavior compatibility of communicating components.

[4]We have used the hash-compact state storing method.

6. RELATED WORK

In [3], the authors use a finite-automata-based description of component behavior and they compose automata for arbitrary components to obtain an automaton modeling behavior of a composite component. They differentiate between control and functional behavior of components. The properties are modeled using the alternation-free μ-calculus, for which an efficient model checking algorithm exists. Using reconfiguration controllers, it is possible to express and check for properties concerning the component application state after and even during an architectural reconfiguration. However, even if the checking process proposed in this paper is performed independently for each composition level, computational requirements of the entire process are substantial even in cases of simple components; complex components are thus beyond the abilities of today's computational systems.

In the Bandera toolset [4], the Java source code is translated to Promela (or another input language of a model checker) to check various properties using Spin. In cases of complex software units, the resulting Promela model yields extremely large state spaces impossible to traverse in a reasonable time with reasonable amount of memory. However, the flexibility and power of the Promela language is demonstrated. Also, if separate checking of particular components is desired, one has to provide a testing environment [10, 13], since the Bandera toolset is able to accept closed code only.

Java PathFinder (JPF) [14] does not translate the Java source code into a modeling language; Instead, it uses a custom Java virtual machine executing the bytecode and checking for three properties: absence of failed assertions, unhanded exceptions, and absence of deadlocks. Moreover, it can be extended to check for a large scale of other properties. Again, however, JPF accepts close code only, so the problem of a suitable environment also has to be solved. As to the scalability of this approach, as Java is a programming language, the state space of even a simple application is quite large.

7. CONCLUSION AND FUTURE WORK

In this paper, we have shown how the compliance relation for communicating components may be evaluated using behavior protocols and the Spin model checker; in particular, we have proposed a procedure for translating behavior specification in behavior protocols to Promela. The mapping of behavior protocol operators and modeling of the components' communication can be done in several ways affecting the size of the resulting model. Using the methods described in this paper, absence of composition errors may be checked even for complex system consisting of tens of components (like in [1]). As the behavior compatibility is checked for each composite component separately, extending of a component-based application by additional components usually increases the checking time requirements in an additive way only. We have also discussed the advantage of using behavior protocols in comparison with Promela as the language for component behavior specification. Nonetheless, during the work on the project [1], we have identified several issues complicating the creating of behavior specification; in particular, it is the absence of procedures/macros and variables.

Our future work will focus on extending the behavior protocol by macros that would allow reusing of protocol fragments in complex protocols. Further, we will focus on stateful components — here variables for storing the component state would be beneficial; therefore, other subject of our future work is extending behavior protocols by variables of some limited domains.

8. REFERENCES

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component reliability extensions for Fractal component model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.

[2] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[3] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)*, August 2006.

[4] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[5] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[6] P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006*, pages 352 – 359. Springer-Verlag, 2006.

[7] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[8] J. Kofron. Software Component Verification: On Translating Behavior Protocols to Promela, technical report no. 2006/11, Dep. of SW Engineering, Charles University in Prague, 2006.

[9] M. Mach, F. Plasil, and J. Kofron. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 6(1), 2005.

[10] P. Parizek and F. Plasil. Specification and generation of environment for model checking of software components. In *Presented at Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006)*, 2006.

[11] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[12] R. Rouvoy and P. Merle. Towards a model-driven approach to build component-based adaptable middleware. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 195–200, New York, NY, USA, 2004. ACM Press.

[13] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proc. of the Eighteenth IEEE Int. Conf. on Automated Software Engineering*, 2003.

[14] W. Visser, P. Mehlitz, J. Penix, D. Giannakopoulou, C. Pasareanu, and M. Mansouri-Samani. Java Pathfinder, `http://javapathfinder.sourceforge.net`, 2006.

CHAPTER 6

Modes in component behavior specification via EBP and their application in product lines

Authors: Jan Kofroň, František Plášil, and Ondřej Šerý

[7] *Information and Software Technology 51/1, pp. 31-41, Elsevier, DOI: 10.1016/j.infsof.2008.09.011, January 2009*

Modes in component behavior specification via EBP and their application in product lines

Jan Kofroň [a,b] František Plášil [a,b] Ondřej Šerý [a]

[a] *Charles University in Prague*
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
Tel: +420 221 914 266, Fax: +420 221 914 323
{jan.kofron, frantisek.plasil, ondrej.sery}@dsrg.mff.cuni.cz

[b] *Academy of Sciences of the Czech Republic, Institute of Computer Science*
Pod Vodárenskou věží 2, 182 07 Praha 8, Czech Republic
Tel: +420 266 053 830, +420 286 585 789
{jan.kofron, frantisek.plasil}@cs.cas.cz

Abstract

The concept of software product lines (SPL) is a modern approach to software development simplifying construction of related variants of a product thus lowering development costs and shortening time-to-market. In SPL, software components play an important role. In this paper, we show how the original idea of component mode can be captured and further developed in behavior specification via the formalism of Extended Behavior Protocols (EBP). Moreover, we demonstrate how the modes in behavior specification can be used for modeling behavior of an entire product line. The main benefits include (i) the existence of a single behavior specification capturing the behavior of all product variants, and (ii) automatic verification of absence of communication errors among the cooperating components taking the variability into account. These benefits are demonstrated on a part of a non-trivial case study.

Key words: Behavior specification, component modes, software product lines
PACS:

1 Introduction

The concept of software components has been around for more than a decade. Component models range from relatively simple, flat component models (e.g., EJB [32]) to more sophisticated hierarchical component models such as Fractal [5] and Koala [30,31]. The latter one is based on Darwin [19] which coined the concept of provided and required interfaces and primitive and composed components allowing component nesting (forming a hierarchy).

Typically, a part of a component-based application involves several operational variants, especially when the part was subject to reuse. This may be reflected both by architecture and behavior variants. However, there is no general consensus on handling this variability. The well-known approaches include modes and product lines.

A mode, as introduced in [13], defines (at design time) a particular alternative of a component's architecture. At runtime, transitions among the modes of the component may take place, however. In principle, a mode is part of a static view on component architecture; it determines the component's internal architecture, the mode of internal components, and is associated with a specific behavior of the component (not necessarily specified in detail). At the same time, the modes of internal components determine a specific mode of the parent.

Software components also play an important role in software product lines (SPL) [30,12]. Work in this field aims at supporting development of software for a set of closely related and likely further evolving products such as consumer electronics. Therefore, capturing variability at different levels of abstraction and stages of software development is a key goal here [7,29,26,9,6,28]. These stages range from modeling software requirements, over design and architecture specification, to code. Consequently, there are many modeling methods targeting variability within such different software artifacts; the key related abstractions include features [23], and, in particular, variation points and their resolution when a specific product is to be instantiated [28,27]. In addition to aspect-oriented programming [24], software components play an important role when considering variability at the level of software architecture. It should be emphasized that a SPL needs to address both variability (configurability) and evolution (modifiability). In a component-based architecture, the variability is reflected in general by some kind of variation points (resolved by configuration parameters) and evolution by the option to replace a component at various levels of component hierarchy.

1.1 Problem statement

Unfortunately, there has been no general consensus on handling architecture variants with respect to component behavior. Specifically, the mode concept is an approach to switching statically determined architecture variants at runtime; however, there is no abstraction to capture how switching among the variants is related to component behavior. If this were determined, automated checks could be employed to verify whether the intended transitions among architecture variants are safely possible in a particular state of the involved components.

Similarly, in SPL, most of the focus is on variability in software architecture, but little attention is paid to variability of a component's behavior in support of its reuse in different architectural variants.

1.2 Goals and structure of the paper

In our group, we have developed a technique of specifying component behavior in a simple process algebra style. In its first variant, Behavior Protocols (BP) [1], the events specified are purely related to issuing request and responses of method calls on the component interfaces, while its more elaborated version, Extended Behavior Protocols (EBP) [16], allows us to capture a component state encoded as a n-tuple of values of enumeration types. The actual expressive power of EBP was tested on a non-trivial component based application developed in the CoCoME component models' contest [22]. There are model checking tools of BP and EBP which can verify correctness of communication among components. The goal of this paper is threefold—to show how

(i) EBP can be used to specify the behavior associated with a specific mode, and also capture the transitions among modes,
(ii) EBP can be employed in product line architecture specification and help derive the behavior and architecture of a particular product variant, and
(iii) the EBP model checking tools can be used for verification of communication correctness of architecture variants.

2 Background

There are a number of formalisms designed for formal behavior specification of software, including set theoretic (e.g., Z), algebraic (e.g., VDM), and process algebraic (e.g., CCS, CSP, FSP [20]) formalisms. As to component-based software, its structuring of code into components with clearly defined interfaces and bindings encourages modular reasoning, on the one hand. On the other hand, the formalism for behavior specification of software components should reflect the key abstractions commonly used in component models. These abstractions involve methods grouped into interfaces, both provided and required, communication among assembled components via bindings of interfaces, and support of component hierarchies. In addition to BP and EBP, such specialized formalisms include Interface automata [8], Component interaction automata [4], and SEFF [11].

2.1 EBP Example

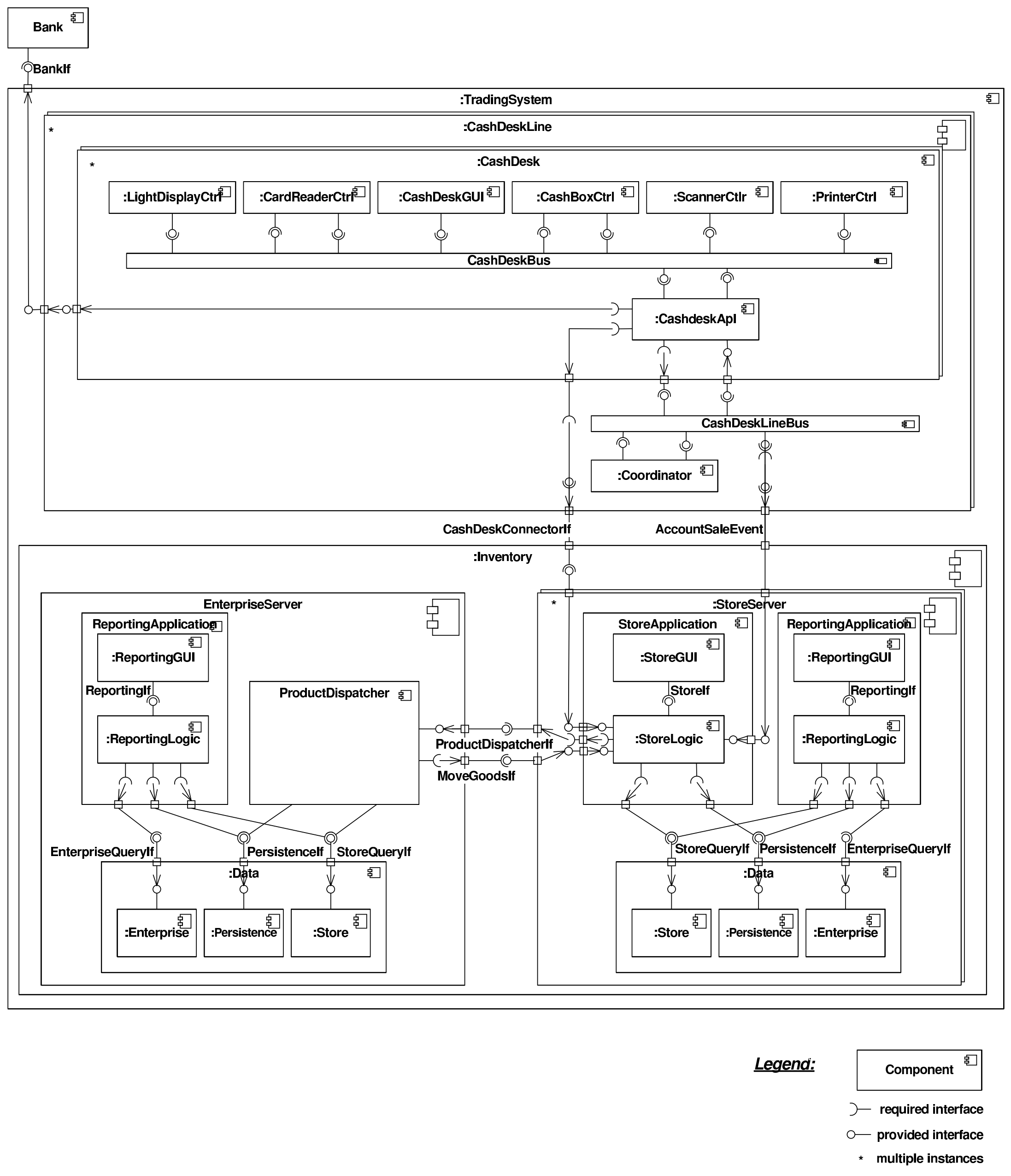


Fig. 1. Architecture of the CoCoME application

First, we shortly describe the CoCoME (Common Component Modeling Example) contest assignment, as the examples in this paper stem from our solution to this assignment [22]. CoCoME was motivated by the need for having a nontrivial canonical example of a component based application that would enable an assessment of strengths and weaknesses of different features and their comparison. Previously, only simple examples (even toy ones) had been used as a proof-of-concept for this purpose.

The assignment of CoCoME is an application for managing a set of stores, each equipped with a cashdesk line. UML component diagram [33] providing an overview of the whole application is in Fig. 1. The assignment consists of a UML specification (component, deployment, sequence, and use-case diagrams), a prototype implementation in Java, and specification of extra-functio-nal properties. A number of teams applied their modeling techniques to the assignment; the results were subsequently evaluated by a jury, pointing out pros and cons of each modeling technique [22].

The example in Fig. 2 provides an intuitive insight and a brief overview of EBP. The example is a slightly simplified version of the EBP specification of the CoCoME CashDeskApplication component. The component contains the actual business logic of a single cash desk in a store; e.g., it maintains information about the progress of a current sale.

An EBP specification of a component consists of three sections: *types*, *vars*, and *behavior*. In the *types* section, the enumeration types of the components' state variables and method parameters are defined. In our case, there is a single enumeration type *states*, which captures the possible states of sale (line 3). The state variables are listed in the *vars* section. In the example, *state* captures the state of a current sale (line 6); it is initialized to SALE_STARTED.

The actual behavior is specified in the *behavior* section. Basic building blocks are: accepting a method call *?interface.method(parameters)* {*reaction*}, issuing a method call *!interface.method(parameters)*, assignment to a state variable *variable* <− *value*, and the *switch* statement, which is used to direct the control flow depending on the value of a state variable. More complex expressions are constructed using the '+' alternative, ';' sequence, '∗' repetition, and '|' parallel (interleaving, no synchronization) operators. They are described in more detail in [21] and [16].

The behavior section of the specification describes which method calls the component can accept and how it reacts on them. In Fig. 2 the reaction depends on the actual value of the *state* variable (note the *switch* statements). In its initial state SALE_STARTED, the CashDesk component can accept bar codes of sale items (ProductBarcodeScannedEvent—line 10) on which it reacts by querying StoreServer about the items and by updating the current total. This can be repeated (line 62). The state is switched to SALE_FINISHED, when the end of BarCode entering is signaled by the GUI component (the corresponding event SALE_FINISHED_EVENT mediated by CashDeskBus is accepted at line 20). After that, the purchase is paid either by cash or a credit card. The former case is reflected by accepting a call from GUI (mediated by CashDeskBus as

```
component CashDeskApplication {
  types {
    states = { SALE_STARTED, SALE_FINISHED, CREDIT_CARD_SCANNED, PAID }
  }
  vars {
    states state = SALE_STARTED
  }
  behavior {
    (
      ?CashDeskAppHandleIf.onEvent(ProductBarcodeScannedEvent) {
        switch (state) {
          SALE_STARTED: {
            !CashDeskConnectorIf.getProductWithStockItem;
            (
              !CashDeskAppDispatchIf.send(ProductBarcodeNotValidEvent) +
              !CashDeskAppDispatchIf.send(RunningTotalChangedEvent)
            )
      } } }
      +
      ?CashDeskAppHandleIf.onEvent(SaleFinishedEvent) {
        switch (state) {
          SALE_STARTED: { state <- SALE_FINISHED }
      } }
      +
      ?CashDeskAppHandleIf.onEvent(CashAmountCompletedEvent) {
        switch (state) {
          SALE_FINISHED: {
            !CashDeskAppDispatchIf.send(ChangeAmountCalculatedEvent);
            state <- PAID
      } } }
      +
      ?CashDeskAppHandleIf.onEvent(CashBoxClosedEvent) {
        switch (state) {
          PAID: {
            !CashDeskAppDispatchIf.send(SaleSuccessEvent);
            !CashDeskDispatchIf.send(AccountSaleEvent);
            state <- SALE_STARTED
      } } }
      +
      ?CashDeskAppHandleIf.onEvent(CreditCardScannedEvent) {
        switch (state) {
          SALE_FINISHED: { state <- CREDIT_CARD_SCANNED }
      } }
      +
      ?CashDeskAppHandleIf.onEvent(PINEnteredEvent) {
        switch (state) {
          CREDIT_CARD_SCANNED: {
            !BankIf.validateCard;
            (
              !CashDeskAppDispatchIf.send(InvalidCreditCardEvent)
              +
              !BankIf.debitCard;
              (
                !CashDeskAppDispatchIf.send(InvalidCreditCardEvent);
                (NULL + state <- SALE_FINISHED)
                +
                !CashDeskAppDispatchIf.send(SaleSuccessEvent);
                !CashDeskDispatchIf.send(AccountSaleEvent);
                state <- SALE_STARTED
            ) )
      } } }
    )*
  }
}
```

Fig. 2. EBP specification of the CashDeskApplication component

CashAmountCompletedEvent—line 25) reporting the cash amount paid by a customer. As a result, GUI is called (again mediated by CashDeskBus—line 28) to report the change to be returned to the customer and the state is switched to SALE_PAID. As soon as the change is returned and the cashbox is closed (CashBoxClosedEvent—line 32), the StoreServer records are updated and the state is set back to SALE_STARTED (lines 35-37). Payment by credit card is initiated by swiping a credit card (CreditCardScannedEvent—line 40), switching the state to CREDIT_CARD_SCANNED, and then either finalized by entering a valid PIN (PINEnteredEvent—line 45), or canceled by switching back to the state SALE_FINISHED.

2.2 Detecting component communication errors via EBP

After an EBP specification of each component has been finished, the tools designed for EBP are applied. They serve to verify correctness of communication among components via checking absence of the following communication errors: *bad activity*, *no activity*, and *unbound requires error*. In general (with a slight simplification), whenever a component calls ('!') a method and the target component is not ready to accept ('?') the call, the bad activity error occurs. No activity represents the situation, when there are components waiting for a method call, while no component is able to emit any. The last error, unbound requires, may be viewed as a special case of the bad activity error, as it represents the situation, when a component issues a call on an unbound required interface. We have developed a tool (Sect. 2.3) which automatically identifies the communication errors and produces a corresponding error trace.

For a composed component, a typical question is whether a *component frame* (the set of externally visible interfaces) is correctly implemented by *component architecture* (composition of subcomponents). The tool can answer the question on the behavioral level, i.e., it reports whether an EBP specification of a composed component is correctly implemented by the behavior of its subcomponents (as described in [1,16,18], compliance of the frame and architecture EBP protocol is verified by a tool).

Considering the CoCoME architecture, the tool can be, e.g., used to show that the EBP specification of CashDesk is correctly implemented by composing CashDeskApplication, CashDeskGUI, LightDisplayCtrl, CardReaderCtrl, CashBoxCtrl, ScannerCtrl, PrinterCtrl, and CashDeskBus (the architecture of CashDesk). Correctness of an entire application can by verified by applying this idea at each level of component nesting.

2.3 EBP in the light of process algebras

In principle, a protocol in EBP is a textual definition of a finite automaton specifying behavior of a component. The essence of the behavior part (as seen in Fig. 2) are expressions forming a simple process algebra close to CSP and partially to CCS. Such an expression generates a set of traces—a trace in EBP is a finite sequence of labels representing the atomic events related to method invocations (the label of a form ?a↑ stands for accepting an invocation of a method with (composed) name a, !a↑ for issuing an invocation, ?a↓ means accepting the response (end) of a method execution, !a↓ means issuing the response). Syntactically, an expression in EBP is composed of labels, operators, and parenthesis '()' and '{}'. The basic operators are: ';' sequencing, '+' alternative, and '|' parallel interleaving with similar semantics as in CSP. However, recursive definitions are not allowed; instead, the repetition operator '*' similar to regular expressions is employed. Therefore only finite traces are considered. Moreover, the parenthesis '{}' serve to easily encode method calls and functionality of methods in the following way: '?a' stands for '?a↑; !a↓', while '?a{P}' stands for '?a↑; P; !a↓', and similarly for '!a' and '!a{P}', where P is again an expression in EBP. For illustration, consider the example in Fig. 3.

```
?CashDeskAppHandleIf.onEvent {            ?CashDeskAppHandleIf.onEvent↑;


  !BankIf.validateCard;                     !BankIf.validateCard↑;
                                            ?BankIf.validateCard↓;
  (                                         (
    !CashDeskAppDispatchIf.send               !CashDeskAppDispatchIf.send↑;
                                              ?CashDeskAppDispatchIf.send↓
    +                                         +
    !BankIf.debitCard;                        !BankIf.debitCard↑;
                                              ?BankIf.debitCard↓;
    (                                         (
      !CashDeskAppDispatchIf.send               !CashDeskAppDispatchIf.send↑;
                                                ?CashDeskAppDispatchIf.send↓

      +                                         +
      !CashDeskAppDispatchIf.send;              !CashDeskAppDispatchIf.send↑;
                                                ?CashDeskAppDispatchIf.send↓;
      !CashDeskDispatchIf.send                  !CashDeskDispatchIf.send↑;
                                                ?CashDeskDispatchIf.send↓

  ) )                                       ) );
}                                         !CashDeskAppHandleIf.onEvent↓
```

Fig. 3. EBP specification of the CashDeskApplication component

In the left column, there is a stripped-off version of the EBP specification in Fig. 2, which was obtained by ignoring the switch constructs and method parameters for simplicity. An equivalent expression where only atomic events are used is in the right column. In principle, this is also a valid CSP specification, in which the events feature composed names (such as !BankIf.validateCard↑)—notice that here a dot means name composition and not CSP prefixing. This simple example illustrates only the use of the '+' and ';' . The operator '|' will be applied later (Fig. 5-10). Obviously, since an expression in CSP determines an LTS [25], an expression in EBP determines also an LTS (more specifically a finite automaton, since recursion in the EBP specification is not employed). Notice that the constructs not employed in Fig. 3 (switches and method parameters) are based on enumeration types, only constants can be assigned to variables of these types, the variables are used only to control switch alternatives, etc., so that the rules for converting them into a finite automaton can be easily articulated.

Composed behavior of two components is determined by parallel composition of their EBP behavior specifications. For this purpose, the binary operator consent (∇_M) is introduced in EBP. In principle, its semantics is similar to the "interface parallel" composition with restriction ($P \; ||_M \; Q$) as known from CSP [25], i.e. in the interleaving of events from P and Q those with names in M synchronize and restriction makes them τ events in the corresponding LTS. However, there are two key semantic differences in case of $P \nabla_M Q$:

(i) The synchronization is based on pair-wise complementarity of the event names (similar to CCS)—the names which differ only in their prefixes '!' and '?' are complementary. For example, events !BankIf.validateCard↑ and ?BankIf.validateCard↑ would synchronize and produce τ.
(ii) Contrary to CSP, if there is no counterpart for an event in M in the other operand of ∇_M, such situation triggers a communication error (an erroneous trace is produced by definition). As mentioned in Sect. 2.2, the following communication errors are defined: *bad activity*, *no activity*, and *unbound required error*.

In our group, for BP we have developed several variants of a model checking tool which identifies communication errors in a ∇_M composition and produces a corresponding error trace [18]. Moreover, for EBP verification, we use a tool transforming EBP specifications into Promela—the input language of the Spin model checker [14]. The reason for choosing Spin is an easier support for future extensions of the EBP language and the efficiency and maturity of Spin—it is a state-of-the-art explicit model checker featuring LTL checking abilities, bit-state hashing, and quite friendly user interface. In addition, there are a number of extensions of Spin, e.g., dSpin extending the Promela language by functions, exceptions, etc. The reason for choosing Spin is an easier support for future extensions of the EBP language.

3 Expressing modes and product lines in EBP

3.1 Behavior modes

To illustrate the way modes are reflected in EBP behavior specification, we present a fragment of the CoCoME component architecture (Fig. 1)—the CashDeskApplication component (Fig. 2). CashDeskApplication works in two operational variants, EXPRESS mode and NORMAL mode. Each of these variants is characterized by a modification of both behavior and architecture of the involved components. In Fig. 4, the architectural variants corresponding to these modes are depicted (we omit other components connected to the CashDeskBus for simplicity).

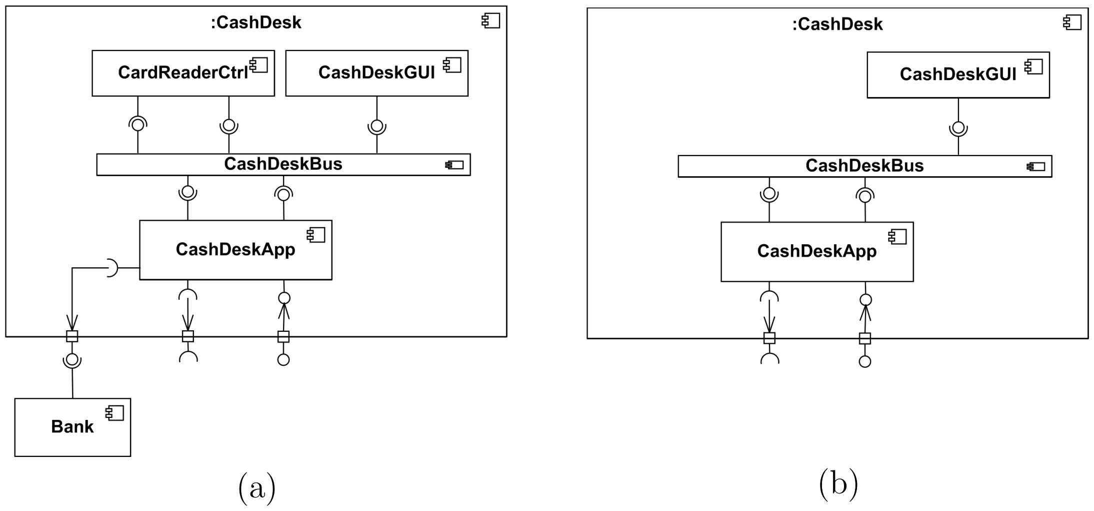


Fig. 4. CashDeskApplication in NORMAL mode (a) and in EXPRESS mode (b)

The EBP snippet in Fig. 5 shows how switching between the modes is reflected in behavior specification of CashDeskApplication. It contains the changes to the CashDeskApplication frame protocol necessary to describe switching to and from the EXPRESS mode. Basically, the original protocol from Section 2.1 is extended by an additional state variable CDAmode (containing the value NORMAL or EXPRESS). The component starts computation in the NORMAL mode (line 8); after receiving an onEvent method call with the parameter value ExpressModeEnabledEvent (line 27), it switches to the EXPRESS mode (line 28).

This example shows that a mode is captured as a specific value of a single state variable (CDAmode in this case). However, in general, a behavior mode of a component can be encoded as an n-tuple of local state variables, each of a specific enumeration type.

The EBP snippet in Fig. 6 describes behavior of the CashDesk component—the parent component of CashDeskApplication. Generally, the behavior modes

```
component CashDeskApplication {
 types {
  states = { SALE_STARTED, SALE_FINISHED, CREDIT_CARD_SCANNED, PAID },
   modes = { NORMAL, EXPRESS}
 }
 vars {
   states state = SALE_STARTED,
   modes CDAmode = NORMAL
 }
 behavior {
   (

     . . .

     +
     ?CashDeskAppHandleIf.onEvent(CreditCardScannedEvent) {
       switch (CDAmode) {
         NORMAL: {
           switch (state) {
             SALE_FINISHED: { state <- CREDIT_CARD_SCANNED }
     } } } }
     +

     . . .

   )* | (
     ?CashDeskAppHandleIf.onEvent(ExpressModeEnabledEvent) {
       CDAmode <- EXPRESS
     }
   )* |  (
     ?CashDeskAppHandleIf.onEvent(ExpressModeDisabledEvent) {
       CDAmode <- NORMAL
     }
   )*
 }
}
```

Fig. 5. EBP specification of the CashDeskApplication component with behavior modes

of a parent and its child components are independent—in each component, there may be state variables capturing the actual behavior mode. In the case of the CashDesk and CashDeskApplication components, however, the modes of both parent and child components are switched simultaneously as a reaction to the ExpressModeEnabled/ExpressModeDisabled events mediated by CashDeskBus.

To provide an example of really independent behavior modes in nested components, let us consider a modified CashDesk component featuring again two behavior modes (CASH and CARD mode) to capture how a customer has decided to pay. Since customers are allowed to pay by cash in both the EXPRESS and NORMAL modes of CashDeskApplication, the behavior mode of CashDesk is not determined by the behavior mode of CashDeskApplication (and vice versa) in this case. Of course, since the CARD mode makes sense only in the NORMAL mode, the modes of CashDeskApplication and CashDesk are not logically independent—the former influences the latter.

```
 1: component CashDesk {
 2:  types {
 3:    modes = { NORMAL, EXPRESS}
 4:  }
 5:  vars {
 6:    modes CashDeskMode = NORMAL
 7:  }
 8:
 9:  behavior {
10:    # Sale related stuff
11:    (
12:      !CashDeskConnectorIf.getProductWithStockItem*;
13:        ( # the bank interface is used only in the EXPRESS mode
14:          switch (CashDeskMode) {
15:            NORMAL: {
16:                       !BankIf.validateCard*;
17:                       !BankIf.validateCard;
18:                       !BankIf.debitCard;
19:            }
20:
21:            EXPRESS: { NULL }
22:          }
23:        )*;
24:      !CashDeskEventDispatcherIf.send(AccountSaleEvent);
25:      !CashDeskEventDispatcherIf.send(SaleRegisteredEvent)
26:    )*
27:    |
28:    # Express mode switch
29:    ?CashDeskAppEventHandlerIf.onEvent(ExpressModeEnabledEvent) {
30:      CashDeskMode <- EXPRESS
31:    }*
32:    |
33:    # Normal mode switch
34:    ?CashDeskAppEventHandlerIf.onEvent(ExpressModeDisabledEvent) {
35:      CashDeskMode <- NORMAL
36:    }*
37:  }
38: }
```

Fig. 6. EBP specification of the CashDesk component with modes

In composition via ∇_M (and the corresponding verification), the behavior modes (state variables) themselves are taken into account indirectly, via the sequences of method calls that are determined by the switch variants which trigger mode switching. In our example, the NORMAL mode is reflected by calling the methods on the BankIf interface (lines 16-18), which is not allowed in the EXPRESS mode (line 21).

3.2 Software product lines

In this section, we show how EBP and behavior modes can be beneficially employed in product line software engineering. To get an idea how EBP can help in SPL architectural design and verification, consider the following example (a slightly generalized part of the CoCoME example). In each store, there is a server maintaining the list of items on and out of stock (Fig. 7). Since the enterprise operates multiple stores, when some goods are running out of stock in a store, they may be brought there from another store (if available) and/or ordered from a supplier. The decision on moving/ordering the goods is realized by a dedicated enterprise server. Depending on whether there is an outlet store at the enterprise premises, the enterprise server either provides also the functionality of the store server (Fig. 9), or it does not (Fig. 8).

These functional alternatives are reflected as architecture variants of a generic server (Fig. 9). The ability to serve as a store server is embodied in the presence of the StoreApplication component, while the enterprise functionality is implemented by the ProductDispatcher component. The other parts, i.e., the ReportingApplication and Data components, are present in all variants.

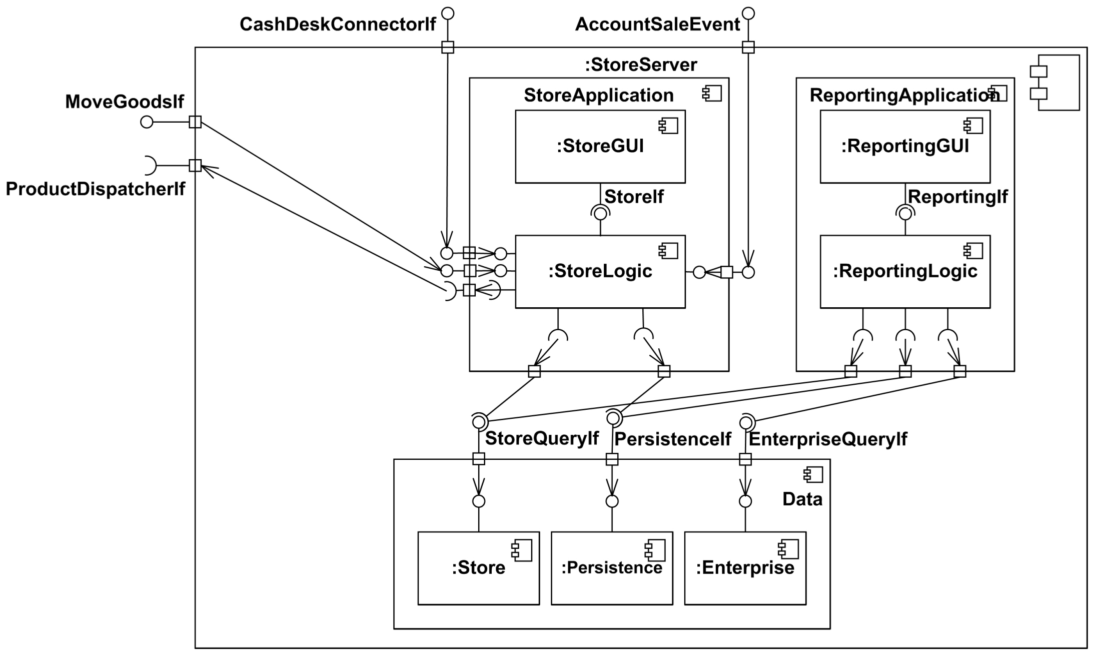


Fig. 7. Store server—present at common stores

In Fig. 10, an EBP specification of the generic server component is provided. Such a behavior specification takes advantage of the fact that some parts of the functionality (behavior) are shared by several variants. The key idea is that the variation points are represented by the switch construct employing the state variables store and enterprise which serve for resolution of these variation points. Both of them can be set to YES or NO; since they are supposed to be set at the design phase by the system architect according to the desired role of the component instance within the system, they are read-only. Particular combinations of the values of store and enterprise reflect the architecture variants described above (except for the combination NO-NO, which results in empty behavior). Such initial assignments to the variables determine the resolution statically.

Behavior modes are suitable for SPL design of unrelated variation points (where state variables are independent). The only difference between a variable determining a product line variant and a variable representing a runtime

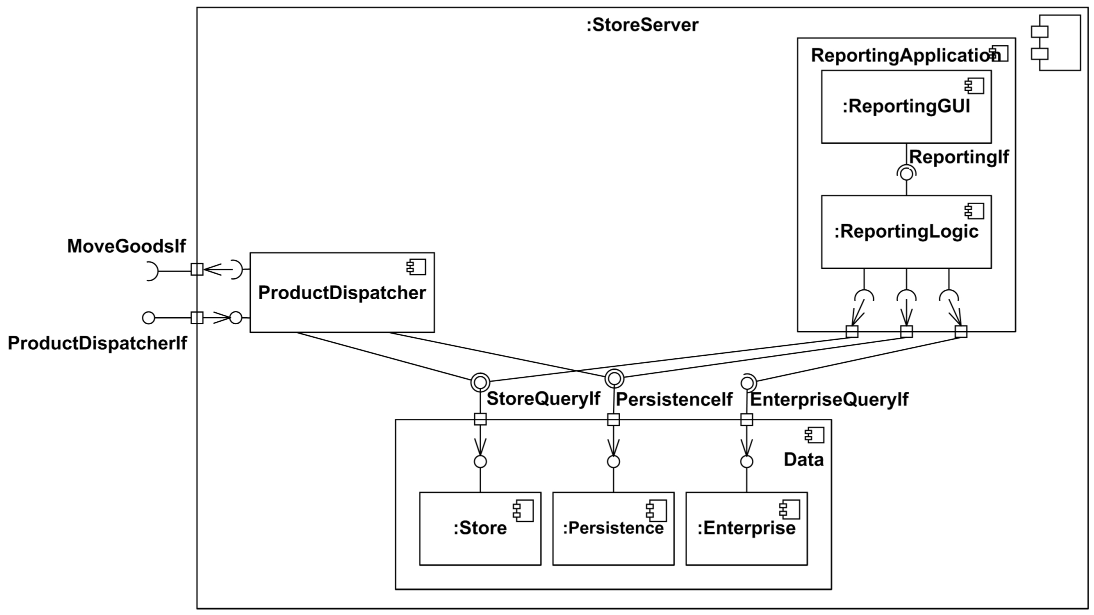


Fig. 8. Enterprise server—present at the enterprise with no store

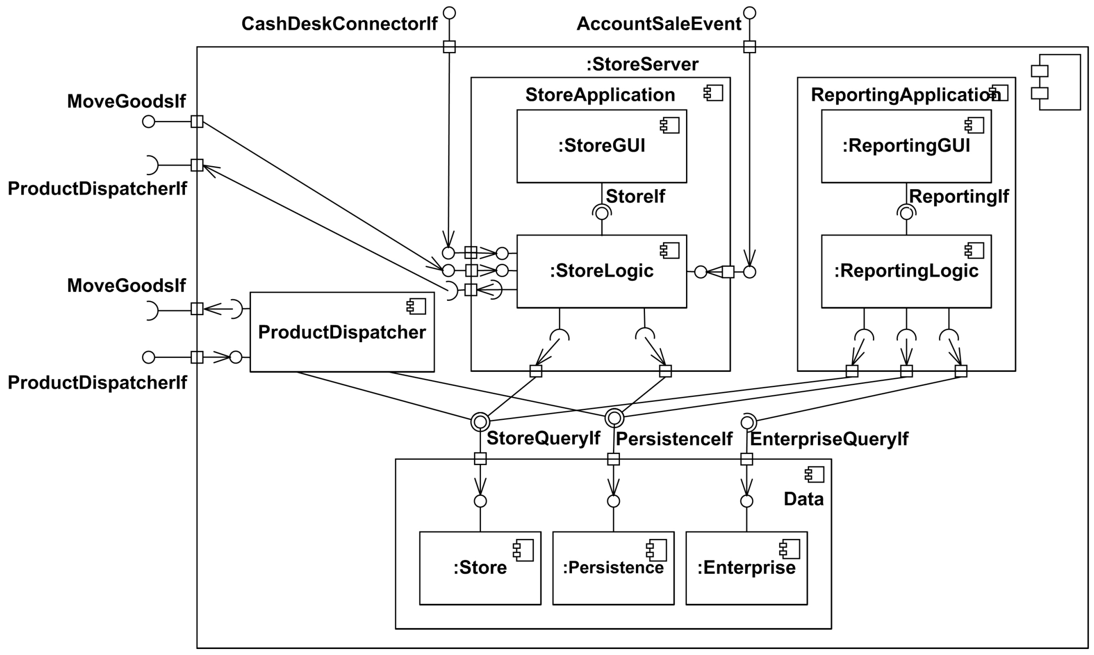


Fig. 9. Generic server

mode is that the value of the former is not modified in the behavior specification. There are no means to enforce this within the EBP language itself; it is up to the designer to obey this principle.

In general, there may be several state variables, each capturing variants of a specific aspect of component behavior and a product line variant. Before the deployment phase, the values of (read-only) variables representing a product line variant are determined to reflect the actual role of each component in the system (resolution of variation points)—it is the case of the store and enterprise variables in Fig. 10. Next, at runtime, the other variables are used to express different behavior modes (and transitions among them) of particular components, as illustrated by the state and CDAmode variables in Fig. 5.

```
component Server {
  types {
    STORE { YES, NO }
    ENTERPRISE { YES, NO }
  }

  vars {
        STORE store = YES
        ENTERPRISE enterprise = NO

  }

  behavior {
    switch (enterprise) {
      YES: {
          (
              ?ProductDispatcherIf.orderProductsAvailableAtOtherStores{
                  !MoveGoodsIf.queryGoodAmount;
                  ((!MoveGoodsIf.acceptFromOtherStore;
                    !MoveGoodsIf.sendToOtherStore)
                    + NULL
                  )
              }
          )*
      }

      NO: { NULL }
    }
    |
    switch (store) {
      YES: {
        (
          ?CashDeskConnectorIf.getProductWithStockItem*;
          (?CashDeskLineEventHandlerIf.onEvent(AccountSaleEvent){
                !ProductDispatcherIf.orderProductsAvailableAtOtherStores +
                NULL
           }
          +
            NULL
          )
        )*
        |
        (
            ?MoveGoodsIf.queryGoodAmount
            +
            ?MoveGoodsIf.sendToOtherStore
        )*
        |
        ?MoveGoodsIf.acceptFromOtherStore*
      }

      NO : { NULL }
    }
  }
}
```

Fig. 10. EBP specification of the Server component with variation points

4 Evaluation and related work

There are several differences between the behavior modes proposed by this paper and the modes introduced in [13]. First, a mode in [13] is a part of the static view on a component's architecture. Moreover, in [13] a mode of a composite component determines both the architecture implementing its frame (i.e., the subcomponents and their bindings) and the modes of subcomponents. Since there is a one-to-one mapping of the mode of a frame and the modes of the subcomponents and their bindings, the architecture and the modes of subcomponents also determine the mode of the encapsulating frame. Regarding the behavior corresponding to different modes, the authors of [13] just assume that every mode of a component is associated with a specific behavior of the component (however, no further details on behavior, or on how modes are switched among, are provided). In other words, modes are a part of the architecture description (technically, they are expressed as (i) labels associated with components and (ii) specification of component instances and their bindings [13]). This way, behavior is determined only at an abstract level. Moreover, employment of the mode idea is distributed over several non integrated tools: Architecture labeling (Darwin), Alloy specification of mode switching constraints [15], and Ponder for specifying runtime policies (such as management and security).

In our approach, on the other hand, mode is a part of behavior specification and it relates to the static view (architecture) only indirectly. Moreover, there is no direct dependency between the behavior mode of a frame and the behavior modes of the subcomponents in its relevant architecture. The only requirement is that the architecture protocol has to be compliant to the frame protocol. It should be emphasized that the compliance evaluation takes into account involvement of mode switching both in the architecture and the frame protocol, i.e., it is verified that these mode switches correspond to each other well. In other words, compliance evaluation helps verify that parent mode switching is correctly reflected in children's behavior. As an aside, an additional benefit of EBP is that the specification scales well due to its textual form (no diagrams) and separation of abstraction layers (frame vs. architecture protocol) which helps keep the state space subject to model-checking in manageable limits.

As to SPL, to our knowledge, there has been no attempt to directly support variation points in behavior specification of a product line. The EBP language allows specifying the behavior of all potential resolutions of variation points in a component frame within a single specification. This is beneficial since significant parts of the specification are typically shared by several variants. An additional benefit of our approach is that from the generic behavior specification of a product line (like the specification of the generic server in Fig. 9) both the architecture and its behavior specification of a variant can be derived; e.g., the architectures in Fig. 7 and Fig. 8 can be inferred in an automatized way.

In addition to [13] discussed above, probably the most related work is [3] emphasizing the need for dynamic software product lines in mobile applications, particularly in the context of self adaptation. We can imagine modifying the variant selection algorithm proposed in [3] in such a way that it would convert its output to trigger an event which could determine a corresponding "dynamic" behavior mode. In our approach, due to the ability to combine both static and non-static variables in the n-tuple determining a behavior mode, we can express behavior of both static (classical) and dynamic product lines.

In [10], dynamic architecture reconfiguration in the context of SPL is modeled as applying the reconfiguration patterns defined at design phase. However, no verification of correctness properties is considered. In [2], behavior of product line variants is modeled in a CSP-based algebra. Nevertheless, all the architecture variants are considered statically and their correctness in terms of deadlock related architectural mismatches is verified separately for each variant. To our knowledge, this is the only process algebra used in SPL.

When compared to general CCS and CSP-like process algebras, EBP is specialized for specification of components. It offers key component abstractions such as methods grouped into interfaces, both provided and required, communication among assembled components via bindings of interfaces, and support of component hierarchies. Another important aspect is representation of a method call by four separate atomic events *!iface.method*$\uparrow$, *?iface.method*$\downarrow$, *?iface.method*$\uparrow$, and *!iface.method*$\downarrow$, which allows for precise specification of interleaving and nesting of method calls.

As an aside, there is no means for expressing behavior modes in the other component behavior modeling formalisms known from literature, such as Interface automata [8], I/O automata [17], and Component interaction automata [4].

5 Conclusion

We have presented a contribution to handling architecture variants with respect to component behavior. The key idea is to use the behavior specification in EBP as a basis for specification of behavior variants. We have shown how EBP can be used to specify the behavior associated with a specific mode of a component and to capture the transitions among the modes at runtime. Moreover, the employment of EBP in the architecture of a product line behavior specification, and the derivation of the architecture and its behavior for a particular product variant were presented. Finally, we have described the use of model checking tools designed for EBP in verifications of component communication correctness in architectural variants.

6 Acknowledgements

This work was partially supported by the Czech Republic project 201/08/0266 and the Czech Academy of Sciences project 1ET400300504.

References

[1] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[2] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.

[3] G. Brataas, S. Hallsteinsen, R. Rouvoy, and F. Eliassen. Scalability of decision models for dynamic product lines. In *Proceedings In International SPLC Workshop on Dynamic Software Product Line (DSPL'07)*, page 10, 2007.

[4] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes*, 31(2):4, 2006.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In *CBSE*, pages 7–22, 2004.

[6] S. Buhne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 41–52, Washington, DC, USA, 2005. IEEE Computer Society.

[7] P. Clements and L. Northrop. *Software product lines: practices and patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, January 2001.

[9] A. Garg, M. Critchlow, P. Chen, C. V. der Westhuizen, and A. van der Hoek. An environment for managing evolving product line architectures. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 358, Washington, DC, USA, 2003. IEEE Computer Society.

[10] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[11] J. Happe, H. Koziolek, and R. H. Reussner. Parametric Performance Contracts for Software Components with Concurrent Behaviour. In F. S. de Boer and V. Mencl, editors, *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS06), Prague, Czech Republic*, Electronical Notes in Computer Science, pages 41–55, September 2006.

[12] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[13] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.

[14] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[15] J. S. Kim and D. Garlan. Analyzing architectural styles with alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM.

[16] J. Kofron. *Behavior Protocols Extensions.* PhD thesis, Charles University in Prague, 2007.

[17] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, November 1988.

[18] M. Mach, F. Plášil, and J. Kofroň. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 2005(1):22–30, 2005.

[19] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[20] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* Wiley, 1999.

[21] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

[22] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS.* Springer, Heidelberg, 2008.

[23] M. Riebisch, D. Streitferdt, and I. Pashov. Modeling variability for object-oriented product lines. In *LNCS 3013*, pages 165–178. Springer, 2004.

[24] D. B. Roberto E. Lopez-Herrejon. Modeling features in aspect-based product lines with use case slices:an exploratory case study., October 2006.

[25] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[26] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, pages 197–213. Springer-Verlag, 2004.

[27] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Managing variability in software product families. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVMG 2004)*, 2004.

[28] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[29] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 130–153, London, UK, 2002. Springer-Verlag.

[30] R. van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM.

[31] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[32] Sun Enterprise Java Beans, `http://java.sun.com/products/ejb`.

[33] Unified Modeling Language, `http://www.uml.org/`.

CHAPTER 7

Threaded Behavior Protocols

Authors: Tomáš Poch, Ondřej Šerý, František Plášil, and Jan Kofroň

[9] *Formal Aspects of Computing, Volume 25, Issue 4 , pp 543-572, ISSN 0934-5043, DOI: 10.1007/s00165-011-0194-3, Springer-Verlag, July 2013*

Threaded Behavior Protocols[1]

Tomáš Poch, Ondřej Šerý, František Plášil, Jan Kofroň

Charles University Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{tomas.poch, ondrej.sery, frantisek.plasil, jan.kofron}@d3s.mff.cuni.cz
http://d3s.mff.cuni.cz

Abstract.

Component-based development is a well-established methodology of software development. Nevertheless, some of the benefits that the component based development offers are often neglected. One of them is modeling and subsequent analysis of component behavior, which can help establish correctness guarantees, such as absence of composition errors and safety of component updates.

We believe that application of component behavior modeling in practice is limited due to huge differences between the behavior modeling languages (e.g., process algebras) and the common implementation languages (e.g., Java). As a result, many concepts of the implementation languages are either very different or completely missing in the behavior modeling languages. As an example, even though behavior modeling languages are practical for modeling and analysis of various message-based protocols, they are not well suited for modeling current component applications, where thread-based parallelism, lock-based synchronization, and nested method calls are the essential building blocks.

With this in mind, we propose a new behavior modeling language for software components, Threaded Behavior Protocols (TBP). At the model level, TBP provides developers with the concepts known from the implementation languages and essential to most component applications. In addition, the theoretical framework of TBP provides a notion of correctness based on absence of communication errors and a refinement relation to verify correctness of hierarchical components. The main asset of TBP formalism is that it links together the notion of threads as used in imperative object oriented languages and the notion of refinement. For instance, this allows reasoning about hierarchical components composed of primitive components implemented in Java without the need of bridging abstractions and simplifications enforced by the modeling languages.

Keywords: Behavior modeling, verification, model checking, refinement, composition, component systems

[1] This work was partially supported by the Grant Agency of the Czech Republic project P202/11/0312 and by the grant SVV-2011-263312.

1. Introduction

The basic idea of component based development (CBD) [LS00] is to compose complex software from well defined artifacts denoted as components. Individual components are developed independently of each other (possibly by different vendors) and communicate through their interfaces. This independence encourages reuse of a single component in different applications requiring similar functionality.

The application architecture plays a key role in the development process. It identifies the individual components of which the application is composed, their functionality, non-functional properties (e.g., performance) and the way the components communicate with each other—interfaces. Each component defines a set of its provided interfaces (set of methods implementing functionality provided to other components), as well as a set of its required interfaces (functionality the component requires from other components). Each interface has its type, and the types of the connected interfaces must correspond to each other (in the meaning common to object oriented languages).

The benefits mentioned so far are closely related to the fact that component internals are hidden to users. This is referred to as the *black box* view. In particular, only the person who implements a primitive component has access to the source code of the component. Other persons are expected to deal with the component via its interfaces and rely on its correctness.

If a component is used in an architecture, it must be ensured that it conforms to its context in the architecture. In practice, the developers rely on syntactic compatibility of interfaces (at the implementation language level) and a precise documentation. This, obviously, does not guarantee a correct result. A developer may overlook subtle details in the documentation which can lead to erroneous behavior. Also, the documentation does not necessarily reflect the actual component implementation. The errors caused by inconsistency of components can be revealed by testing, or in the worst case, manifest themselves after submission to a customer. This weak point of CBD is addressed by many efforts in academia.

Software verification and CBD can benefit from each other [ABC10]. CBD splits a complex program into smaller fragments—correctness of each primitive component can be verified separately. And afterwards, correctness of composition is to be checked. For such compositional verification, each component is equipped with a formal model specifying its behavior, and when composing components it is checked whether their models fit together.

The requirements posed to a formal model to make it applicable in component-based development are formulated as interface theories in [AH01b]. The requirements include support of incremental design and independent implementability. A selective overview of several diverse approaches related to component behavior evaluation, which ranges from LTS based behavior models, through contracts, analysis of implementation, to summaries can be found in [CSS05].

1.1. Problem statement

Component-based development has found its way into industry. However, the component systems used in practice (e.g. EJB [MSD03], DCOM [GG97], CCM [OMG06], Koala [OLKM00]) do not take advantage of behavior modeling and subsequent analysis.

In our experience based on [CoC, ABJ$^+$06], a part of the problem is that there is no suitable specification language for behavioral modeling aiming at component systems in a comprehensive way. Specifically, no specification language allows writing behavior specification in a straight-forward way, while providing features important for analysis of a component-based system such as behavior composition and refinement. Specifically, crafting behavioral models is a time consuming and error prone task. One of the reasons is that the modeling languages of the formal frameworks are too different from the imperative programming languages used by developers on daily basis.

1.2. Goals

The main objective of this paper is to make the application of behavioral modeling in component-based development more suitable for day-to-day practice.

Based on our experience with specification languages of the Behavior Protocols family, in particular Behavior Protocols (BP) [AP04, AP03] and Extended Behavior Protocols (EBP) [Kof07], we propose the specification language Threaded Behavior Protocols (TBP) aiming at fulfilling the following two goals. On the one hand, writing specifications in TBP should resemble programming in an imperative programming language. Since programmers are used to the concepts provided by the imperative languages, this aspect should significantly decrease the effort needed to prepare specifications of individual components. On the other hand, the formal framework should provide means for successful application in CBD. This includes a composition operator, as well as refinement relation, both designed with respect to a precisely-defined notion of correctness and imperative concepts of the language. This way, we want to combine the aim of programming languages at practical usability and existing theoretical frameworks focused on analyses ensuring correctness of behavior.

Even though none of the features of the proposed specification language TBP is utterly novel, the contribution of this paper lies in the way these features are put together within the context of a single language. What makes it unique is that the users can reason on refinement of individual components' models, while the computation is driven by threads as it is in a real implementation. Thus, the user can avoid introducing unnecessary specific concepts, "bridging" different abstractions levels and maintain their mappings as the model evolves.

2. Related Work

During the past decades, several languages and formalisms for behavior modeling of software systems have been proposed. They range from very generic ones (e.g., process algebras) to those specific to components (e.g., Darwin [MDEK95], Wright [All97], Behavior Protocols (BP) [PV02], BIP [BBS06]). In this section, we focus on those based on labeled transition systems (LTS), as they are well studied, meaning that their properties (e.g., decidability of model checking of particular temporal logic formulas) as well as algorithms for formal reasoning are well established.

2.1. Process Algebras

Classical process algebras (e.g., CSP [Hoa85] and CCS [Mil95]) describe a behavior as a set of cooperating processes syntactically defined by a set of recursive equations; each of them associates a process name with an expression determining the process behavior. Formally, the semantics of an expression is given via derivation rules. A number of operators are typically defined to combine elementary actions. The operators include sequencing, alternative, parallel composition typically with the option to create a synchronous product, event hiding, renaming, etc.

When applied in the component context, individual components are represented by separate processes. Synchronization of actions represents issuing a method call on a required interface resp. accepting method call on a provided interface. Names of actions correspond to method names found in the architecture and method parameters and data can be captured in the full calculus by value passing.

A particular example of CSP usage in the context of component systems is the Wright specification language. Wright [All97] is an ADL for defining a component-based architecture enriched by behavioral specification. The key abstractions of the component model include a component and its ports (interfaces), and a connector and its roles (interfaces). An assembly is created by binary bindings of ports to roles. Each of the key abstractions is associated with its behavior specification in the form of a process in a subset of CSP.

The process describing the behavior of a component on its ports is called a computation, while the process specifying behavior of a connector on its roles is called glue. Having the behavior of ports, roles, glues and computations specified in CSP, automated checking of composability (based on refinement and deadlock-free testing) is possible. In [Ros98], the approach taken is to transform Wright specifications into plain CSP and use the FDR tool.

Modeling component behavior in a process algebra in principle assumes that every component is an active entity, which does not correspond to the idea of implementation threads "visiting" other components. This leads to the need of fragmenting threads into cooperating processes, or introducing a specific abstractions for component behavior coordination (connectors in [BBS06])

2.2. Automata-Based Languages

Automata based languages define the LTSes of individual communicating systems (with a straight-forward) graphical representation. Such a definition is typically easy to comprehend, since it does not require deep knowledge of the semantics of the given formalism. On the other hand, drawing complex systems tends to be a tedious and time consuming task. Individual formalisms differ in the supported actions (labels), composition operator and by the studied properties and relations over models.

Interface automata introduced by Alfaro and Henzinger [AH01a] distinguish input actions (?name), output actions (!name) and internal actions (name). The parallel composition is used to form more complex systems from simple ones.

Generally, the composition is defined over pairs of composable (having fitting sets of input and output actions) automata A_1 and A_2. A synchronous product automaton $P_A = A_1 \otimes A_2$ is created; A_1 and A_2 synchronize on complementary actions. The result contains error states if an automaton emits an output action and the counterpart automaton is not able to accept it (there is no complementary input action).

Existence of an error state in the product P_A, however, does not mean that A_1 and A_2 cannot work together. If A_1 and A_2 form an open system (i.e., there are still some input and output actions) there can still be an environment E that avoids the error state. If such E exists, A_1 and A_2 are considered *compatible*.

The refinement relation ($\preceq$) supported by the formalism of interface automata preserves the compatibility. In particular, if P and R are compatible and Q $\preceq$ P then also Q and R are compatible. Such notion of refinement can be directly used in the component context to verify hierarchical architectures.

Apart from the interface automata, there are other formalisms, e.g., I/O automata [LNW07], Component Interaction Automata [ČVZ07] and others.

2.3. UML

Since Unified Modeling Language (UML) [BRJ05] is a de-facto industrial standard used for modeling software systems, it is worth mentioning here as well. Although it addresses the problem of behavior modeling, the semantics of related diagrams (activity, state machine, communication, interaction overview, sequence, and timing diagrams) is not defined precisely enough to allow formal verification. Also a formal notion of composition and refinement is missing. Although UML supports profiles refining the semantics for special cases, none of the profiles has been generally accepted yet. Nevertheless, UML is often used to visualize software designs and even for prototype code generation.

3. Application in Development Cycle

Rigorous application of formal methods in software development always requires substantial effort. This includes running analysis tools and interpretation of their results, preparing models in various languages and formalisms as well as maintaining consistency with concurrent activities in the development process. Therefore, to lower the effort and maximize the benefits of using formal methods, it is necessary to carefully articulate their role in the development process.

In comparison to other development paradigms, component based development has the advantage of an explicitly-defined software architecture and component boundaries. Thus, there is a solid ground upon which the formal methods may be built.

Notion of correctness A key benefit is the guarantee that the created software is in a certain sense correct. The notion of correctness may differ depending on the particular formalism and the available tools. Obviously, the formalism cannot provide guarantees related to the software aspects from which it abstracts. In general, three types of correctness are considered (i) composition correctness, (ii) correctness of implementation, and (iii) user-defined properties.

The behavioral model of a component specifies how the component communicates with its environment. This is sometimes referred to as *contract*. In particular, it specifies what the component provides to the environment, under which assumptions, and what it requires from the environment. As the components are composed in a specific architecture, violations of mutual assumptions can emerge, resulting in a composition error. As an example, consider deadlock or invocation of a method on a component not prepared to accept it (e.g., due to omitted initialization).

Another important property is adherence of the implementation of a primitive component, i.e., a component directly implemented in an imperative language such as Java, to its behavior model. This means that, when used in compliance with the assumptions articulated in its behavior model, the component reacts as specified by its behavior model and without low-level implementation errors (e.g., null pointer dereference).

In addition to these issues, some formalisms provide means to specify and analyze violations of user-defined properties. This allows specifying domain-specific properties reflecting the actual business logic of the application. As an example, consider the requirement that all method calls in a component should be preceded by an initialization and eventually followed by a special method call which frees resources (e.g., open files, database cursors, unfinished transactions). User-specified properties are typically provided in the form of a temporal logic formula (e.g., LTL, CTL).

Top-down approach. In the classical waterfall model, the developer is expected to provide the design in an early stage of the development process. If formal methods are applied, certain aspects of the design are expressed formally. For instance, the behavior model is to be provided right after the application architecture is created. When the architecture is hierarchical, the component behavior models can be created level by level—the behavior models of the top level components can be created prior to the architecture of the lower levels of the hierarchy. At each level, the composition of individual models is checked for correctness and also for fulfilling the behavior specification at the higher level. The relation of the behavior composition to the behavior specification at the higher level is referred to as refinement.

When the architecture is ready, the next step is to provide an implementation of the primitive components. Since the behavior model of the primitive components is already available, it is convenient to generate a skeleton of the implementation from the model. Then, the developers are expected to manually implement the aspects of the behavior not captured by the model (e.g., by means of inheritance). Alternatively to skeleton generation, the developers can provide the whole implementation manually. In such case, it is important to find a way to ensure that the implementation conforms to the model of the primitive components. Otherwise, the whole process is not sound.

The top-down approach allows continuous verification as the behavior model is getting more precise. Thus, since errors at higher levels of hierarchy are detected as soon as possible, they can be fixed with small impact on the lower levels.

Bottom-up approach. The top-down approach is not always applicable. An example is analysis of legacy applications. Also in some of the methodologies, which do not follow the waterfall model, rapid prototyping comes first and later the code is refined and improved. In these cases, implementation precedes creation of the behavior model.

The first step is to provide behavior models for all primitive components. It may be either constructed by a skilled reverse engineering specialist or generated from the implementation by means of static analysis or monitoring. Once the behavior models for all primitive components are available, the architecture can be flattened and checked for composition correctness. This may, however, be an infeasible task for current analysis tools (e.g., due to state explosion). In such case, a hierarchy can help to delegate the task to smaller subtasks by checking each composed component individually. In particular, when a behavior specification of the composite components is provided, refinement ensures that the behavior specification corresponds to the composition of primitive components. In the next step, correctness of composition of components at the higher level is checked and so on.

4. TBP Proposal

Based on the comparison of imperative languages with the formalisms presented so far, we can more closely refine the goals.

Goal 1 TBP will feature a straight-forward syntax and semantics to capture the behavior of a component, as well as the assumptions on the behavior of its environment.

(a) We would like the parts of TBP describing (specifying) the behavior of component to be close to imperative programming languages (we have chosen Java as the representative) in both syntax and semantics.

(b) TBP will support specification of assumptions on an environment.

Goal 2 Theoretical framework of TBP will provide guarantees of correctness in the following sense:

(a) The framework will allow comparing actual behavior of a closed system to the expected behavior. Two kinds of assumption violations are to be detected: *bad activity* and *no activity*[2].
(b) Composition operator will reflect the concept of threads.
(c) There will be a preorder refinement relation considering both bad activity and no activity.
(d) The analyses of correctness should be decidable.

To achieve this goal, the theoretical framework of TBP has to precisely define the semantics of concurrent thread execution in a form suitable for definition of correctness and of the refinement relation.

5. TBP Syntax

This section presents the formalism of Threaded Behavior Protocols (TBP) from the user point of view. In particular, intuitive notion of component execution is used to explain meaning of individual concepts. Moreover, since specification of individual components is a typical usage scenario, this chapter, unless explicitly stated, considers just specification of a single component. Note, however, that the formalism of TBP allows expressing syntactically also the result of a composition.

The basic structure of a TBP specification is formed by five parts—declarations of types, declarations of state variables, reactions on method calls, threads, and provisions. While the reactions and threads specify the behavior exercised by the component itself in the imperative manner, provisions specify the behavior of an environment assumed by the component. The assumptions are stated over sequences of provided method calls.

```
component ComponentName {
  types { ... }
  vars { ... }
  provisions { ... }
  reactions { ... }
  threads { ... }
}
```

5.1. Relation to Component Model

The concept of provided and required methods (often formulated using interfaces—groups of methods) is already present in a typical component model. Since TBP is designed to be an extension of such a model, there is no need to provide explicit syntax construction to determine which methods are provided and which methods are required. Later, in the semantics section (Section 7), we assume Σ_{req}, resp. Σ_{prov}, resp. Σ_{int} to denote the set of component's required, provided, and internal methods, respectively. Their content is taken from the component model.

Moreover, the component models define compositions of components via bindings among the interfaces. Thus, since the information about bindings can be taken from the component model, there is no need to specify syntax for composition of TBP specifications.

5.2. Declarations

The *types* section defines enumeration types. The types may be used for declaration of method parameters, state variables, and local variables. The *vars* section contains definition of state variables important for the behavior. The variables can be accessed from within the component (i.e. reaction or thread) only. Later on, within the threads and reactions sections, the variables are referenced by assignments and conditions.

[2] Due to historical reasons we use the terms "no activity" to denote deadlock and "infinite activity" to denote livelock; "bad activity" denotes a situation similar to the error state of Interface automata [AH01a].

```
types {
  result = {OK, FAILED};
  mode = {INIT, RUNNING, MAINTENANCE, SHUTDOWN}
}
```

```
vars {
  mode actualMode = INIT;
}
```

The types fragment contains declaration of two enumeration types. While the `result` type consists of two enumeration values OK and FAILED, the `mode` type consists of four enumeration values. The following example fragment contains declaration of the `actualMode` state variable. The variable's type is `mode` and the initial value is `INIT`.

There is a special type of variable, *mutex*. A mutex variable serves as a synchronization primitive, upon which the threads can synchronize, e.g., to achieve mutual exclusion.

5.3. Reactions

The *reactions* section contains description of the actual behavior performed by the component in reaction to a method call of either a provided or an internal method (a method that is neither provided nor required, but called by a component's thread). Each reaction specified in the section consists of the method name, declaration of arguments and body. The body begins with declaration of local variables followed by the actual behavior.

```
  reactions {
    interface.methodName(ArgType1 argName1, ArgType2 argName2, ... ):ReturnType {
      LocalVarType localVar = initialValue;
      ...
    }
```

Local Variables and Arguments Each local variable declaration specifies a name, a type, and an initial value. Arguments, on the other hand, consist just of the type and a name. Local variables and arguments may be accessed only from the reaction body. Moreover, the local variables and arguments are not shared by parallel executions of the body—each execution has its own copy.

Elementary Actions The actual behavior consists of elementary actions composed together by control flow operators. There are three kinds of elementary actions. In the following, `val`, `val1`, `val2`, etc. denote values of defined types. The types of parameters in method calls and assignment correspond to expected types. `Var` denotes a variable.

- Method call — `i.a(val1, val2, ... )`
  The execution of the current reaction is postponed and the `a` method on the `i` interface is invoked with parameters `val1`, `val2`, etc. The `a` method is either required or internal. The execution of the current reaction is resumed when the method `a` is finished.
- Return — `return val`
  Explicit termination of reaction. The returned value may be assigned to a variable at the calling site.
- Variable assignment — `var=val`
  The content of the `var` variable is changed to the `val` value. The variable is either local variable or state variable and the value is either constant (enumeration value), another variable or a method call. In the last case, the return value of the invoked method is assigned to the variable.
- empty action — `NULL`

Control Flow Operators Control flow operators correspond to the control flow statements known from imperative languages. The conditions used in the control flow operators are boolean expression consisting of elementary conditions (equality — `var == val`) connected by common boolean operators (`|| && !`). Moreover, there is the non-deterministic condition denoted by `?`.

Apart from the sequence operator (`;`), there are `if`, `switch` and `while` operators used in the same context as in an imperative language. The `sync` keyword denotes a critical section. It cannot be entered by more than one thread simultaneously. Technically, the mutual exclusion is achieved by atomic test-and-set operation over a variable of the built-in type Mutex (`m` in the fragment).

```
if (condition) {thenBranch}
  else  {elseBranch}
```

```
switch(var){
  case constA: aBranch
  case constB: bBranch
  default: defaultBranch
}
```

```
while(condition){
  loopBody
}
```

```
sync(m){
  criticalSect
}
```

When a non-deterministic condition is used, the executed branch is chosen arbitrarily. The loop statement may be executed any finite number of times (i.e. non-deterministic decision whether to continue or not is taken before each iteration).

5.4. Threads

The *threads* section contains description of the autonomous behavior performed by the component's internal threads. Each thread is declared by a name and a body, which consists of local variable declarations followed by description of behavior. The constructs used to describe reaction bodies are used also to describe thread behavior. Each thread may invoke required methods as well as internal methods. It may change a state variable or a local variable. The number of threads is constant, each thread starts its execution immediately at the beginning of the model execution and once a thread reaches its end it does not perform any action.

5.5. Provisions

Previous sections specify the behavior exercised by the component itself in the imperative manner. In contrast, the purpose of the provisions section is to declare the allowed usage of the component—assumptions posed on the environment (i.e., limit the set of environments with which the component is supposed to cooperate). The key distinction from the imperative part is that the assumptions posed on the environment should be weak. While the imperative parts specify the behavior as precisely as possible, the assumptions must not prohibit too many environments, to enable reuse of the component in different contexts.

Each assumption is specified as a set of allowed sequences of component's provided method calls and returns. The set of allowed environments then includes all environments that do not violate the assumption.

Each provision consists of an expression and a set of methods it constraints. The expression defines a language in the same way as a regular expression. An environment fulfills the provision if all traces it generates, when restricted to the methods from the set, belong to the language.

Elementary expression An elementary expression consists of two actions—a provided method call and the corresponding return. Both actions can contain additional data—either the method call parameters or a return value. Each method used in the expression must be a member of Σ_{Prov}.

```
provisions {
  { i.m(val):retVal } for {i.m}
  { i.n() } for {i.n, i.q}
}
```

The fragment contains two provisions. While the first provision guards all invocations of the method `m` on the interface `i`, the second provision guards invocations of the methods `n` and `q`. The first provision states that the method `m` on the interface `i` is expected to be invoked exactly once by the environment. Moreover, the argument value must be equal to `val` and the method returns `retVal`. The second provision states that the environment calls the method `n` exactly once. The arguments used and the return value may be arbitrary, however. Moreover, the method `q` cannot be invoked by the environment at all.

Operators To construct more complex provisions, the following operators are available.

- Regular operators — sequence (`;`), alternative (`+`) and repetition (`*`)

  ```
  { i.m(val);i.n()*+i.q() } for {i.m, i.n, i.q}
  ```

  In the example, the environment must call the method `m` with the argument `val` and later it can either call the method `n` arbitrary many times or the method `q` exactly once. In between, before and after

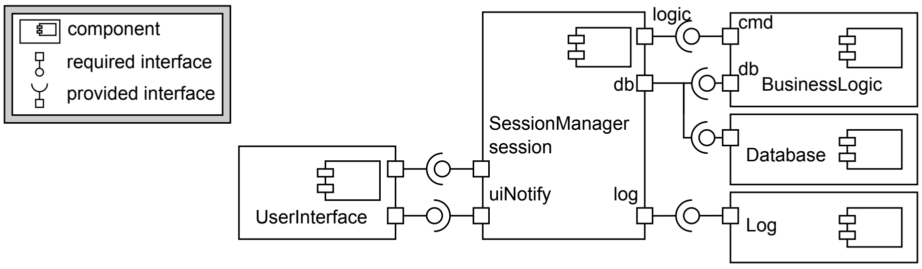


Fig. 1. Architecture employing the SessionManager component

method calls, however, any number of other methods than `i.m`, `i.n`, and `i.q` is allowed, since the other methods are not guarded by the provision.

- Parallel operators — and-parallel(`|`), or-parallel(`||`)
  Parallel operator stands for alternative of all possible interleavings of operands. While for the and-parallel `|` the environment has to follow both operands, for the or-parallel the environment may choose just one operand.
- Reentrancy operator — limited (A `||`n for $n \in \mathbb{N}$) and full (A `||*`)
  A `||`n is equivalent to A `||` A `||` ... `||` A where there are n occurrences of A within the expression. Thus, A may be followed by the environment at most n times in parallel. A `||*` stands for A `||` A `||` A `||` .... Thus, the environment may proceed according to A as many times in parallel as it needs.

```
{ {i.login():ACCESS_DENIED*; i.login():ACCESS_GRANTED; i.n()}|* } for {i.login, i.n}
```

In the example, the environment may attempt to login in parallel. Each attempt, however, must end by a successful login and invocation of the method `n`.

6. Example

6.1. Architecture

Fig. 1 contains an example architecture of a component application. The example contains a fragment of a web-based information system. The application logic is implemented in the BusinesLogic component and the UserInterface component mediates the user input entering the system in form of HTTP requests. The commands from the user, however, do not go directly to the business logic. The communication is intercepted by the SessionManger component that takes care of authentication of commands. SessionManager requires some additional functionality provided by the Log and Database components. The latter one is shared with the BussinessLogic component.

6.2. Behavioral specification for SessionManager

The SessionManager component is expected to work in the environment suggested in Fig 1. In particular, the SessionManager component intercepts the communication between the (web based) user interface and the application logic to provide the authentication feature.

The basic functionality of the component is to associate commands from individual users with sessions. In order to invoke commands, the user interface has to acquire a session id (invoke the `createSession` method provided by SessionManager). Then, if a valid session id is returned, it is used as a parameter for subsequent commands. After SessionManager accepts the command, it checks the session id and passes the command to the business logic. Apart from the main functionality, the component implements a maintenance mode. The mode is automatically turned on when the administrator user is logged in. In the maintenance mode, no new sessions can be created (`createSession` returns `INVALID_SESSION`) and new requests executed in a context of other than the administrator's session causes invalidation of the request session. Moreover, the session timeout is implemented, so that SessionManager may decide on its own to invalidate an arbitrary session.

It is important to emphasize the purpose of the model since it determines the abstractions to be used. In this case, the goal is to describe dependencies of individual method calls in presence of parallelism. In particular, we want to identify deadlocks and wrong ordering of method calls.

For instance, we do not model the particular authentication algorithm since it is not relevant. Just the result influences the sequencing of method calls and the implementation must be prepared for the positive result as well as for the negative one. Moreover, we do not distinguish individual users. There are just several types of sessions (`ADMIN_SESSION`, `USER_SESSION` and `INVALID_SESSION`). In general, the data abstractions are chosen to reflect the conditions in the code that influence the control flow.

In the model, there is just one provision (lines 16-22) prescribing how the user interface can call the methods on the session interface. In particular, user interface is allowed to submit a command (`invokeCmd`), only if the `createSession` method returns a valid session id (i.e. `USER_SESSION` or `ADMIN_SESSION`). The component is able to process requests from several users in parallel (| and |* operators).

Then, there are reactions for the provided methods `createStatement()` and `invokeStatement()` and the reaction for the internal method `terminate Session()` which is invoked from three different placess. When the user explicitly issues the `CMD_LOGOUT` command (line 46) to invalidate an existing normal user session in the maintenance mode (lines 47-48) and finally on timeout (line 69). The timeout is implemented by the only autonomous thread in the specification, Timer.

```
 1 component SessionManager {
 2   types {
 3     DbResult = {DB_GRANTED, DB_REFUSED};
 4     SessionId = {USER_SESSION, INVALID_SESSION, ADMIN_SESSION};
 5     UserId = {ADMIN_ID, USER_ID};
 6     Command = {CMD_LOGOUT, CMD_OTHER};
 7     OperationMode = {NORMAL_MODE, ADMIN_MODE};
 8   }
 9
10   vars {
11     OperationMode opMode = NORMAL_MODE;
12     Mutex m;
13   }
14
15   provisions {
16     {
17       {session.createSession():USER_SESSION;session.invokeCmd(USER_SESSION,?)*}
18      + {session.createSession():ADMIN_SESSION;session.invokeCmd(ADMIN_SESSION,?)*}
19      + session.createSession():INVALID_SESSION
20     }|*
21     for {session.createSession,session.invokeCmd}
22   }
23
24   reactions {
25     session.createSession(UserId userId):SessionId{
26       DbResult queryResult = DB_REFUSED;
27       queryResult = db.query();
28       sync(m) {
29         if (queryResult == DB_GRANTED) {
30           log.log();
31           if (userId==ADMIN_ID){
32             opMode = ADMIN_MODE;
33             return ADMIN_SESSION;
34           }
35           if (opMode==NORMAL_MODE){
36             return USER_SESSION;
37           }
38         } else { log.log(); }
39         return INVALID_SESSION;
40       }
41     }
42
43     session.invokeCmd(SessionId sessionId, Command cmd)
44     {
      if (sessionId == INVALID_SESSION) return
      if (cmd == CMD_LOGOUT || (sessionId == USER_SESSION && opMode == ADMIN_MODE)){
        log.log();
        intr.terminateSession(sessionId);
      } else {
        log.log();
        logic.invokeCmd(cmd);
      }
    }

    intr.terminateSession(SessionId sessionId){
      sync(m){
        if (sessionId==ADMIN_SESSION){
          opMode = NORMAL_MODE;
          log.log();
        }
        uiNotify.sessionTerminated(sessionId);
        log.log();
      }
    }
  }

  threads {
    Timer {
      while(?) {intr.terminateSession(?);}
    }
  }
}
```

7. TBP Semantics

The syntax presented so far accompanied by the information about interfaces from an underlying component model form a TBP specification. Its semantics, as depicted in Fig 2, is defined in two stages.

In model stage, the TBP model is defined. The TBP model is a five-tuple capturing by mathematical means (e.g., Labeled Transition System) the essential information from the TBP specification. Composition is defined at the model stage, so that composition of two TBP models ($\oplus$) yields also a TBP model.

The notion of correctness (i.e. absence of communication errors) and refinement is defined at the LTS stage. The computation of the TBP model is represented by an LTS, either finite or infinite. Communication errors are defined for a closed model (i.e. model which does not exercise any externally observable activity—Σ_{prov} and Σ_{req} are empty) as a property over computation states.

For an open system, refinement is defined as a relation over observation projections. Observation projection is an LTS modeling only externally observable activity (e.g., observation projection of a closed system is a single state with no edges) of the computation. In the special case when the number of external threads expected to use the provided methods of the component is limited to k, the LTS is finite. Finally, notion of refinement is defined as a relation over observation projections.

The motivation behind those stages is to bridge the gap between the TBP specification, which provides relatively rich concepts to the user, and mathematical structures used to clearly define notions of composition, communication errors, and refinement.

7.1. TBP Model

The TBP model precisely defines meaning of the syntax presented so far by mathematical means. As already indicated, provisions differ from the imperative parts of the specification (threads and reactions). Not surprisingly, in the TBP model, those are captured by different means as well. The provisions define a set of important events and a set of allowed traces. To formally capture threads and reactions, a variant of LTS enhanced with variables, guards, and assignments—*Labeled Transition System with Assignments* (LTSA)—is used.

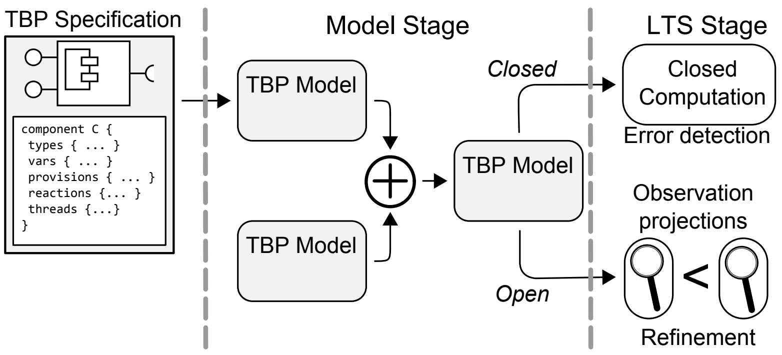


Fig. 2. TBP semantic stages

In the following definitions, let E be a set of enumeration types and V be a set of variables of types from E. Each variable $v \in V$ determines its type and initial value. Dom_e is a set of values of type $e \in E$, Dom_v is a domain of the variable v, $Dom_E = \bigcup_{e\in E} Dom_e$ and $Dom_V = \bigcup_{v\in V} Dom_v$.

A basic element of the transition systems we are going to define is a parameterized transition. It corresponds to a method call, where the method α can contain parameters $v_1, v_2, ..., v_n$:

Definition 1 (Parameterized labels). Let Σ be a set of labels and Par is a set of variables (representing formal parameters). Then, we define the set of parameterized labels $\Sigma_{Par} = \{(\alpha, \langle v_1, v_2, \dots, v_n\rangle) : \alpha \in \Sigma, n \in \mathbb{N} \cup \{0\}, v_i \in Par\}$.

The function $param_i : \Sigma_{Par} \to Par$ returns the i-th parameter of the parameterized label and the function $name : \Sigma_{Par} \to \Sigma$ returns the original label without the parameters.

Labels are later used to model method calls. In particular, parameterized labels allow encoding method parameters as well as return values into labels. Thus, in the following definitions, the set of labels is often parameterized by variables and constants—$\Sigma_{V\cup Dom_E}$.

Definition 2 (Valuation function). The valuation function $\gamma_V : V \cup Dom_E \to Dom_E$ assigns a value to each variable from V. Moreover, it is identity for constants ($\gamma_V(c) = c$ for $c \in Dom_E$). The initial valuation function γ^0_V assigns initial values to all variables. Modification of the valuation function is denoted as $\gamma_V[v \mapsto c]$.

$$\gamma_V[v \mapsto c](a) = \begin{cases} \gamma_V(a) & : a \neq v \\ c & : a = v \wedge v \in V \\ undefined & : a = v \wedge v \notin V \end{cases}$$

This third case in the definition above denotes the situation when v is not a variable but a constant; obviously, the value of a constant cannot be modified.

Definition 3 (Guards). A guard over V, is a finite expression derived using the following rules:

- $true$ is a guard,
- $v == l$, where $v \in Var$, $l \in Dom_v$ is a guard,
- if X and Y are guards, then $X \wedge Y$, $X \vee Y$ and $\neg X$ are also guards.

The actual value of the guard g for valuation γ_V is denoted as $\gamma_V(g)$ and the set of guards over V is denoted as G_V.

Note that a mutex m is considered to be a special case of a variable such that:

$Dom_m = \{LOCKED, UNLOCKED\}, init_m = UNLOCKED$

Definition 4 (Assignment). Assignment over V is a label in the form

- $v = c$ where $v \in V, c \in Dom_v$
  assigning the constant c of the corresponding type to v

- $v = w$ where $v, w \in V, Dom_v = Dom_w$
  assigning the current value of the variable w to v

  Let A_V be a set of assignments over V.

Definition 5 (Labeled Transition System with Assignments). A *Labeled Transition System with Assignments* (LTSA) is a tuple $(S, s_0, F, \delta, \Sigma, V)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, Σ a set of communication labels, V a set of variables and $\delta \subseteq S \times G_V \times (\Sigma \cup A_V) \times S$ is a transition relation.

Definition 6 (LTSA computation state). Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA. We call the tuple (s, γ_V) a computation state of l if $s \in S$ and γ_V is a valuation function for the set V. The tuple (s_0, γ_V^0) is denoted as initial computation state.

Definition 7 (Enabled LTSA transition). Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA and $cs = (s, \gamma_V)$ be its computation state. We call the transition $t = (s, g, \alpha, s')$, $t \in \delta$ *enabled* in cs if $\gamma_V(g)$ is evaluated to true.

In the TBP model, LTSAs are used to capture control flow of imperative parts (i.e., reactions and threads). In this context, the set of labels contains parameterized labels representing issuing of a method call (for all required and internal methods).

Let Σ contain method names ($m \in \Sigma$), $\Sigma^{\uparrow}$ contains events for issuing a method call ($m\uparrow \in \Sigma^{\uparrow}$) and $\Sigma^{\downarrow}$ contains events for accepting a method call result ($m\downarrow \in \Sigma^{\downarrow}$). Moreover, $\Sigma^{\uparrow/\downarrow} = \Sigma^{\uparrow} \cup \Sigma^{\downarrow}$. Then, $\Sigma^{\uparrow}_{V \cup Dom_E}$ contains the events for issuing a method call parameterized by all possible combinations of variables from V and constants from Dom_E.

For purposes of TBP model definition, we define $LTSA^{\Sigma}_{V,E}$ to be a set of all LTSAs using labels from $\Sigma^{\uparrow}_{V \cup Dom_E}$ and variables from V. In such case, each transition in δ is thus guarded by $g \in G_V$ and labeled by $l \in \Sigma^{\uparrow}_{V \cup Dom_E} \cup A_V$. Thus, l represents either issuing of a method call (including parameters), or an assignment involving a variable from V. The return value of a method call is assigned to the special purpose *Ret* variable. Thus, to assign the return value to a user specified variable, it suffices to assign the content of the *Ret* variable.

Informally, the TBP model definition says that a model is defined by an alphabet of methods names, a set of variables, a set of provisions (each captured as a set of allowed traces), a set of reactions (represented as LTSA), and a set of active threads (given also as LTSA).

Definition 8 (TBP model). Let E be a set of enumeration types used in a TBP specification. A *TBP model* is a five-tuple (Σ, P, R, T, G), where:

(a) $\Sigma = (\Sigma_{prov}, \Sigma_{req}, \Sigma_{int})$ denotes disjunct sets of provided, required and internal method names used in the model. In addition, where convenient, we use $\Sigma_{ext} = \Sigma_{prov} \cup \Sigma_{req}$ to denote the set of all externally visible method names, $\Sigma_{all} = \Sigma_{ext} \cup \Sigma_{int}$ for all names and $\Sigma_{imp} = \Sigma_{int} \cup \Sigma_{req}$ for names of methods which can be actively invoked in imperative parts.

(b) G is a set of state variables.

(c) P is a set of provisions $\{P_1, P_2, \ldots, P_n\}$ taking the form $P_i = (filter^{P_i}, traces^{P_i})$, where $filter^{P_i} \subseteq (\Sigma_{prov})$ specifies methods observed by the provision and $traces^{P_i}$ specifies a set of allowed finite sequences of events from $(filter^{P_i})^{\uparrow/\downarrow}_{Dom_E}$.

(d) R is a function: $(\Sigma_{int} \cup \Sigma_{prov}) \rightarrow (L, \mathbb{N} \rightarrow L, LTSA^{\Sigma_{all}}_{G \cup L, E})$ representing a mapping of method names to their local variables (L), a parameter mapping function and reactions in form of LTSA. The L set contains always at least the variable *Ret*.

(e) T is a set of threads $T_1, T_2, ..., T_m$, where $T_i \in (L, LTSA^{\Sigma_{all}}_{G \cup L, E})$ is a tuple specifying a set of local variables and the behavior of the i-th thread in the form of LTSA.

Each provision P_i in (c) represents a set of finite sequences over events representing issuing of a method call and accepting the response. The methods are either provided or internal methods and the events are parameterized by constants from Dom_E. TBP model directly representing a TBP specification states the provisions just over the provided methods. However, after composition, some provided methods become internal methods. The parameters of response events represent return values. This allows reflecting return values in sequences and, in particular, describing an environment which behaves with respect to the value obtained from the method call. Notice, that events are not parameterized by variables as in other cases, but just by constants from Dom_E (actual values of method calls).

The labels in the LTSA for reactions in (d) contains calls of methods from Σ_{imp} and assignments over local variables and state variables. In addition, constants from Dom_E are allowed in method calls. The *Ret* variable is a special purpose variable used to store results from method calls. The parameter mapping function is used to assign parameters from a parameterized method call event α to variables from L.

7.1.1. From TBP Specification to TBP Model

Method sets (Σ), global variables (G), and filters ($filter^{P_i}$) directly correspond to the sets from TBP specifications. Provisions ($traces^{P_i}$) and construction of LTSAs (R,T), however, deserve precise definitions. Formally, we define the function $model(TBPSpec, k)$ taking a TBP specification and the number of threads from the environment allowed to enter the specification in parallel as arguments.

Provisions Every set $traces^{P_i}$ is represented by a finite state machine (FSM). Its construction directly follows the algorithm for construction of FSM from a regular expression. In particular, method calls are translated into a sequence of two events representing issuing a method call and return from a method call. Both events are parameterized—either by parameters or by return values. The parallel operator results in an interleaving of FSMs induced by its operands. Finally, the reentrancy operator is treated as a sequence of or-parallel operators. The number of parallel operators is given by the parameter k of the *model* function.

Imperative Parts The structure of $LTSA^{\Sigma}_{V,E}$ is constructed in a bottom-up fashion. The basic building blocks are method calls and variable assignments. The LTSA representing a method call contains three states sequentially connected by two transitions labeled by a method call and, optionally, a Ret value assignment. A state variable assignment is represented by an LTSA with two states connected by a single transition labeled by the assignment.

The LTSA of a more complex expression is constructed from the LTSAs of its subexpressions. It is also similar to construction of a nondeterministic finite automaton from a regular expression. The sequence operator (';') corresponds to the concatenation of LTSAs, `if` and `switch` correspond to alternative, and the `while` statement is related to repetition. The difference inheres, however, in guards. If the `if` statement contains a deterministic condition (not '?'), the corresponding transitions are equipped with the guards derived from the condition and its negation. Similarly, the edges coming from the final states of the `while` statement may contain a guard.

Finally, a block synchronized by a mutex `sync(m){...}` adds a new initial state to the LTSA connected by a transition to the original initial state. The new transition is labeled by the guard ensuring that the associated mutex is unlocked `m == UNLOCKED` and by the assignment `m = LOCKED`, which locks the mutex `m`. The resulting LTSA has only one (newly added) final state with a transition targeting it from each original final state and labeled by the assignment `m = UNLOCKED`.

7.1.2. Composition

Before defining the composition itself, we first make a simple observation. The names from the sets Σ_{int} and G are not visible to the outer world and thus should not influence the result of the composition. In other words, a protocol defines the same behavior under any arbitrary renaming of Σ_{int} and G. Therefore, without loss of generality, we assume that there are no name clashes in these internal names[3].

Moreover, when two models are being composed, they should not provide a method with the same name as this would yield a binding of a single required interface to multiple provided interfaces, which is not supported. Thus, to capture these requirements, we define notion of composable models.

Definition 9 (Composable models). Let $A = (\Sigma', P', R', T', G')$ and $B = (\Sigma'', P'', R'', T'', G'')$ be TBP models. We say, that A and B are composable iff

- $\Sigma'_{prov} \cap \Sigma''_{prov} = \emptyset$
- $\Sigma'_{int} \cap \Sigma''_{int} = \emptyset$

[3] Formally, this could be also handled by name substitution. However, this would obfuscate the otherwise simple definition.

Composition of two composable TBP models is again a TBP model. The composition makes a union of the corresponding sets of provisions, reactions, threads, and state variables.

Definition 10 (TBP Composition). Let $A = (\Sigma', P', R', T', G')$ and $B = (\Sigma'', P'', R'', T'', G'')$ be composable TBP models. Then:

$A \oplus B = ((\Sigma_{prov}, \Sigma_{req}, \Sigma_{int}), P' \cup P'', R' \cup R'', T' \cup T'', G' \cup G'')$

where
$\Sigma_{prov} = \Sigma'_{prov} \cup \Sigma''_{prov}$,
$\Sigma_{req} = (\Sigma'_{req} \cup \Sigma''_{req}) \backslash (\Sigma'_{prov} \cup \Sigma''_{prov})$ and
$\Sigma_{int} = \Sigma'_{int} \cup \Sigma''_{int}$

Notice that the set of methods provided by composition is a union of methods provided by the original models. This way, a method provided by the input model which is at the same time required by the other input model stays in the set of provided methods in the composition. Thus a provided method can be required (thus, invoked) by several components composed together by sequential application of the composition operator.

Definition 11 (Closed TBP model). Let $M = (\Sigma, P, R, T, G)$ be a TBP model. We call M closed if $\Sigma_{req} = \emptyset$

The essence of a closed model is that it does not communicate with the environment. The closed computation introduced in the following text considers only the threads from T as a source of activity and does not expect the environment to invoke any method.

7.2. Analysis of Closed Models

In this section, we define the computation of a closed TBP model as a finite LTS, called *a closed computation*. Intuitively, a closed computation is created by composition of LTSAs of individual threads and reactions. In particular, the way individual LTSAs are put together is inspired by a typical stack-based execution model of imperative languages.

The number of threads is fixed in the closed TBP model. There is a single stack for each thread. The top of a stack refers to the actual position in the LTSA being currently executed by the thread. Also, the local variables and parameters referenced by LTSA guards and assignments are on the stack. Thus, each *computation state* is represented by a number of stacks and valuation of state variables.

Definition 12 (Computation state). A *Computation state* of the TBP model (Σ, P, R, T, G) is a tuple $(Stacks, \gamma_G)$, where $Stacks$ is a (multi)set of stacks—sequences of tuples (s, γ_L). The size of $Stacks$ corresponds to the number of threads ($|Stacks| = |T|$), γ_G and γ_L are valuation functions and s is a state of an LTSA ℓ. The LTSA ℓ either captures behavior of a reaction or a thread from the model.

A *computation transition* represents an atomic change of the computation state. Such a change is either a modification of a stack or modification of a state variable or both. A change of the stack size corresponds either to issuing a method call or accepting a method call response. Those transitions are labeled by corresponding parameterized labels. The data in those labels are the actual values used in the particular method calls and returns. Thus, if the method names are from Σ, then labels are from $\Sigma^{\uparrow/\downarrow}_{Dom_E}$. Since the labels in LTSA may be also parameterized by variables, *a parameter valuation function* is defined to get the actual values of these variables.

Definition 13 (Parameter valuation function). Let $\gamma_V : V \cup Dom_E \to Dom_E$ be a valuation function. Then, we define the parameter valuation function $\gamma^{\Sigma}_V : \Sigma_{V \cup Dom_E} \to \Sigma_{Dom_E}$ in the following way:
$\gamma^{\Sigma}_V((\alpha, < p_1, p_2, \ldots, p_n >)) = (\alpha, < \gamma_V(p_1), \gamma_V(p_2), \ldots, \gamma_V(p_n) >)$.

In the following definition, a stack is treated as a pair $(top, tail)$ or $null$, where top is the item on the top of the stack, $tail$ is the rest of the sequence and $null$ represents an empty stack.

Definition 14 (Computation transition). Let $l = (S, s_0, F, \delta, \Sigma^{\uparrow}_{V \cup Dom_E}, G \cup L)$ be the LTSA corresponding to the top of the active stack ($s \in S$, L is the set of variables considered by valuation γ_L). Let $(((s, \gamma_L), tail) \cup Stacks, \gamma_G)$ be a computation state of the TBP model (Σ, P, R, T, G), where $((s, \gamma_L), tail)$ is

(a) $$\frac{\delta{:}s \xrightarrow{g,m^{\uparrow}<a_{1..n}>} s', \gamma_{G\cup L}(g)=true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{m^{\uparrow}<\gamma_{G\cup L}(a_{1..n})>} (\{((s',\gamma_L),tail)\}\cup Stacks,\gamma_G)}$$

where $m \in \Sigma_{req}$

(b) $$\frac{\delta{:}s \xrightarrow{g,v=e} s', \gamma_{G\cup L}(g)=true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{\tau} (\{((s',\gamma_L[v\mapsto\gamma_{G\cup L}(e)]),tail)\}\cup Stacks,\gamma_G[v\mapsto\gamma_{G\cup L}(e)])}$$

where $v \in G \cup L$
$e \in G \cup L \cup Dom(v)$

(c) $$\frac{\delta{:}s \xrightarrow{g,m^{\uparrow}<a_{1..n}>} s', \gamma_{G\cup L}(g)=true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{m^{\uparrow}<\gamma_{G\cup L}(a_{1..n})>} (\{((s'_0,\gamma^0_{L'}[p(i)\mapsto\gamma_{G\cup L}(a_i)]),((s',\gamma_L),tail))\}\cup Stacks,\gamma_G)}$$

$\forall i \in \{1..n\}$ and $m \in \Sigma_{prov} \cup \Sigma_{int}$
$R(m) = (L', p, l')$ where L' is a set of local variables, $p : \mathbb{N} \to L'$ is a parameter mapping function, and $l' = (S', s'_0, F', \delta', \Sigma_{G\cup L\cup Dom_E}, G \cup L')$

(d) $$\frac{s\in F}{(\{((s,\gamma_L),((s',\gamma_{L'}),tail))\}\cup Stacks,\gamma_G) \xrightarrow{m^{\downarrow}<\gamma_L(Ret)>} (\{((s',\gamma_{L'}[Ret\mapsto\gamma_L(Ret)]),tail)\}\cup Stacks,\gamma_G)}$$

Fig. 3. Rewriting rules defining a *computation transition*

the stack of the thread producing the transition. Moreover, $m < a_{1..n} >$ is a shorthand for $(m, < a_1, \ldots, a_n >)$ and $m < \gamma_V(a_{1..n}) >$ for $(m, < \gamma_V(a_1), \gamma_V(a_2), \ldots, \gamma_V(a_n) >)$. The transitions among computation states are defined by the rewriting rules in Fig. 3. While the top part of the rule references the LTSA capturing control flow of a method performed by a thread, the bottom part defines a new transition.

In Fig. 3, the rule (a) represents invocation of a required method. The computation transition is labeled by the method identifier parameterized by the actual values of its arguments. The rule (b) represents an assignment. Just the valuation of the given variable is changed and the computation transition is labeled by the silent action. The rule (c) represents invocation of a provided or internal method. In either case, a method reaction is to be executed by the thread. Thus, the target state of the computation transition has a new item at the top of the active stack containing the initial state of the invoked method's reaction. The valuation of local variables contains correct values for the method's arguments. The rule (d) represents the final step of a method reaction. Since the final state of the corresponding LTSA has been reached ($s \in F$), the stack in the target state of the computation transition is popped and the content of the special purpose variable *Ret* (return value) is copied one level higher on the stack. The computation transition is labeled by the return method identifier parameterized by the actual value of the *Ret* variable.

Put together, computation states and transitions form a closed computation:

Definition 15 (Closed computation). Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model. Then the *closed computation* of M is the tuple $C(M) = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$, where:

- Each label from $(\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E}$ is an event representing either issuing or acceptance of method calls parameterized by constants from Dom_E.
- $s_0 = (Stacks_{init}, v^0_G)$ is initial computation state of the model. The set $Stacks_{init}$ contains a stack for each thread $t = (L, LTSA_t)$ from T containing a single item (s^0_t, γ^0_L), where s^0_t is the initial state of $LTSA_t$ and γ^0_L is the initial valuation of L.
- $\delta \subseteq S \times (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\} \times S$ is a transition relation corresponding to the computation transition
- S is a set of computation states reachable by δ from s_0

- $F \subseteq S$ is a set of final states. The state s is final if for each thread t its stack contains just single item (s, L) such that s is a final state of $LTSA_t$.

The closed computation of a closed TBP model is finite as long as the model does not contain (even indirect) recursive calls in reactions. To keep the properties we are interested in decidable, we consider only finite TBP models in the following text. Technically, we prohibit recursion. We believe this is not a huge harm from the practical point of view, since we consider recursive calls among components to be a bad practice. Practically, a tool can detect a possible recursion and refuse to provide any results in such case. Moreover, since there are no required methods, there are no transitions of type (1) (Definition 14)

The definitions presented so far provide us with the precise meaning of a set of TBP specifications composed together such that they form a closed TBP model. The semantics in this case is given in the form of finite LTS, which provide straightforward definition of communication errors in the following sections.

7.2.1. Communication Error

The formalism of TBP distinguishes two kinds of communication errors. There are inherent errors, which appear in consequence of a closed computation, and errors with respect to provisions.

Definition 16. Let $C(M) = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a closed computation of a closed TBP model M.

There is *inherently no activity* in a state $s \in S$ if $s \notin F$ and there are no transitions leading from s. The set of all states with inherently no activity is denoted as $F^{\oslash}$.

There is *infinite activity* in state $s \in S$ if there is no path from s to a final state or to a state where inherently no activity is. The set of all states with infinite activity is denoted as F^{∞}.

There is *internal infinite activity error* in state $s \in S$ if there is infinite activity in s and all paths leaving the state s contain only τ events. The set of all states with internal infinite activity error is denoted as F^{∞}_{int}. Apparently, $F^{\infty}_{int} \subseteq F^{\infty}$.

No activity denotes the situation when a thread gets stuck in a state waiting for a guard which never starts to hold (e.g., to enter a critical section created by `sync` keyword or waiting for a certain value of a variable). In such a case, the thread is waiting for an action to be performed by another thread (leave the critical section, set the variable), which does not occur. Since such a situation is clearly undesirable, such a state is considered to be erroneous.

On the other hand, the infinite activity is desirable in some cases—especially those emphasizing reactive nature of a system while not modeling shutdown at all. The special case of infinite activity—internal infinite activity, however, is undesirable, since there the computation performs only internal actions with no effect observable by other components. For instance, internal infinite activity captures the situation, when a thread is actively waiting (in a loop) for a guard.

We consider as erroneous the states from $F^{\oslash}$ and F^{∞}_{int}, while F^{∞} can be desirable in some models. Additionally, there is another class of errors induced by the provisions.

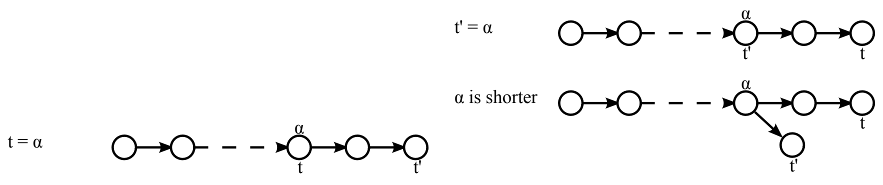
Provisions of individual components express assumptions posed on the environment in the form of traces. As the models are being composed together, the particular environment of the component is being formed. Once the system is closed, it is checked whether all provisions are obeyed.

Since provisions are based on traces, we define traces generated by closed TBP model first.

Definition 17 (Computation trace). Let $C = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a closed computation of a closed TBP model. Then, we call the finite sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$, $\alpha_i \in (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}$ *computation trace*, if there is a sequence of computation states $s_0, s_1, \ldots, s_{n+1}$ such that $\forall i, 0 \leq i < n : \wedge(s_i, \alpha_i, s_{i+1}) \in \delta$.

Moreover, we call the computation trace *terminating* when $s_{n+1} \in F$, *stuck* when $s_{n+1} \in F^{\oslash}$, *diverging* when $s_{n+1} \in F^{\infty}$ and *internally diverging* if $s_{n+1} \in F^{\infty}_{int}$.

The set of terminating computation traces is denoted as $L(C)^{\checkmark}$ and stuck computation traces as $L(C)^{\oslash}$. $L(C)^{\infty}$ contains the set of representatives of diverging computation traces—all diverging traces sharing the same (already diverging) prefix are represented just by the prefix. The set of representatives of internally diverging computation traces is denoted as $L(C)^{\infty}_{int}$. $L(C) = L(C)^{\checkmark} \cup L(C)^{\oslash} \cup L(C)^{\infty}$.



(a) No activity (b) Bad activity

Fig. 4. Examples of provision violation

The set $L(C)$ characterizes a closed computation. It contains all traces leading to successful termination as well as traces leading to no activity and traces leading to diverging states. Having such characterizing set of computation traces, we can define the provision violation.

Definition 18 (Trace restriction). Let $t = \alpha_0, \ldots, \alpha_n$ be a computation trace consisting of parametrized labels $\alpha_i \in (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}$ and $f \subseteq \Sigma_{imp}$ be a set of labels referred to as a filter. Then, the trace restricted by f, $t \upharpoonright f = \alpha'_0, \ldots, \alpha'_m$ is a computation trace consisting of $\alpha'_i \in f^{\uparrow/\downarrow}_{Dom_E}$ such that

$$t \upharpoonright f = \begin{cases} \alpha_0.(t_1 \upharpoonright f) & : name(\alpha_0) \in f \\ t_1 \upharpoonright f & : otherwise \end{cases}$$

where $t_1 = \alpha_1, \ldots, \alpha_n$ is the trace t without the first parametrized label and . is the concatenation operator. Moreover, restriction of the empty trace is the empty trace.

Informally, the trace restriction operator removes from the trace the parametrized labels that are not based on a label belonging to the filter. As an example, consider the following trestriction:
$m_1(v_{11}, v_{12}, ..., v_{1n}); m_2(v_{21}, v_{22}, ..., v_{2n}); m_3(v_{31}, v_{32}, ..., v_{3n}); m_4(v_{41}, v_{42}, ..., v_{4n}) \upharpoonright \{m_1, m_4\} =$
$m_1(v_{11}, v_{12}, ..., v_{1n}); m_4(v_{41}, v_{42}, ..., v_{4n})$.

Definition 19 (Provision violation). Let (Σ, P, R, T, M) be a closed TBP model and C its closed computation. We say, that the provision $P_i \in P$, $P_i = (filter^{P_i}, traces^{P_i})$ is obeyed by a trace $t \in L(C)$ iff $t \upharpoonright filter^{P_i} \in traces^{P_i}$. The provision is not violated in the TBP model if it is obeyed by all the computation traces from $L(C)$. In the other cases, we say that the provision is *violated.*

With violation of provision defined in general, let us discuss specific kinds of violations. Violation of a provision P_i for some i occurs, if there is a trace in L(C) such that its restriction $t \notin traces^{P_i}$. In such case, let $t' \in traces^{P_i}$ be a trace, sharing the longest prefix α with t.

Fig. 4(a) illustrates the case when $t = \alpha$, which is denoted as *no activity.* In such case, the restricted trace t follows the provision up to the certain point (α) and then immediately terminates without following the rest of the trace t'. It corresponds to the situation when the model is expected to perform some action (e.g., close a session), however it does not and just terminates, instead.

Definition 20 (No activity). A TBP model with a closed computation C generating computation traces L(C) and containing a provision (filter, traces) contains *no activity error* if there is $t \in L(C) \upharpoonright filter$, $t \notin traces$ such that it is a prefix of a trace $t' \in traces$.

Fig. 4(b) reflects the remaining two situations. Either $t' = \alpha$ or α is shorter than both t and t'. In the former case, the trace t is a witness of a situation when a source model attempts to invoke a provided method of a target model, which is already in a final state and does not expect any further method calls. The latter case, on the other hand, reflects the situation when the target component does not expect the invoked method, but expects another method call.

Definition 21 (Bad activity). The TBP model with closed computation C generating computation traces L(C) and containing provision (filter, traces) contains *bad activity error* if there is $t \in L(C) \upharpoonright filter$, $t \notin traces$ such that it is not a prefix of any trace $t' \in traces$.

For the purposes of the following text, we define the predicate *ErrFree* over the set of all closed TBP models.

Definition 22 (ErrFree). Let M be a closed TBP model. Then we define

- $ErrFree_{BA}(M) \Leftrightarrow$ there is no bad activity in the model.
- $ErrFree_{\oslash}(M) \Leftrightarrow$ there is neither inherently no activity nor no activity in the model.
- $ErrFree_{\infty_{int}}(M) \Leftrightarrow$ there is no internal infinit activity in the model
- $Errfree(M) \Leftrightarrow ErrFree_{BA}(M) \wedge ErrFree_{\oslash}(M) \wedge ErrFree_{\infty_{int}}(M)$

Assuming C is a closed computation of M, $Errfree(M)$ can be alternatively defined as $(L(C)^{\checkmark} \upharpoonright filter) \subseteq traces \wedge (F^{\oslash} = F^{\infty}_{int} = \emptyset)$

7.3. Analysis of Open Models

So far, a notion of correctness for closed TBP models was presented. In the context of hierarchical component models, however, developers often deal with open systems. Thus, it is necessary to extend the notion to the realm of open TBP models.

Even if there is an error (in the sense of previous paragraphs) in the open system, it often depends on the particular environment whether the error becomes evident or not. The environment may steer the open system away from the error so that the error is never reached in the closed system that results from a composition.

Thus, instead of mere identification of errors in an open system, comparison of open systems with respect to an environment is preferred. In particular, the question is whether a TBP model I behaves correctly in all environments where a TBP model S behaves correctly. Such relation is referred to as refinement in the rest of this paper.

In a typical scenario, I represents a complex model (I stands for the implementation, which is typically a composition of other models) while S is a relatively simple model capturing only the important aspects of behavior with respect to communication with the environment.

Having the refinement relation, the hierarchical system verification is divided into two subtasks. The first task is done when an open system is being created. The developer of a composite component provides the model S and ensures that the actual component behavior corresponds to S—by means of refinement. The second task is done when the open system is put into a particular environment to create either a closed system or an open system on the higher level. In former case, the developer uses the specification S to verify correctness of the closed model by means of the *ErrFree* predicate. In the latter case, the specification S is put together with other components to form the implementation of the composed component. Then, the refinement is checked again on the higher level.

Formally, the refinement is defined as follows:

Definition 23 (Refinement). Let I and S be TBP models and E be the set of all TBP models such that $\forall e \in E : e$ is composable with both I and S, and $e \oplus I$ is a closed TBP model.

- We say that I refines S with respect to bad activity iff
  $\forall e \in E : ErrFree_{BA}(e \oplus S) \Rightarrow ErrFree_{BA}(e \oplus I)$
- We say that I refines S with respect to no activity iff
  $\forall e \in E : ErrFree_{\oslash}(e \oplus S) \Rightarrow ErrFree_{\oslash}(e \oplus I)$
- Finally, we say that I refines S iff
  $\forall e \in E : ErrFree(e \oplus S) \Rightarrow ErrFree(e \oplus I)$

Definition of the refinement is based on the errors considered. Moreover, the implication in these definitions ensures transitivity of all three refinement relations as stated in the following lemma[4].

Lemma 1 (Transitivity). Let A, B and C be TBP models such that A refines B and B refines C. Then, A refines C.

The rest of this section provides a means for deciding whether I refines S. This is done in several steps.

First, an open TBP model is transformed into provision-driven computation. It is LTS similar to the closed computation, however it also contains the transitions labeled by input actions representing the actions the

[4] The proofs can be found in [Poc10].

environment is allowed (or expected) to perform. Those input actions are distinguished from the actions actively performed by the model (output actions).

In the next step, an observation projection is created from the provision-driven computation. The purpose of the observation projection is to resolve the non-determinism in the model. It is a pessimistic approximation of the provision-driven computation representing the behavior of the model as observed by an environment. In particular, the environment is not able to distinguish states of the model reached by the same sequence of observable actions. All these states are represented by a single state (super-state) in the observation projection. Moreover, the super state allows the environment to perform only the actions allowed by a state of provision-driven computation it represents. On the other hand, the observation projection requires the environment to be ready for all actions that may occur in any state the super state represents.

The final step when deciding whether I refines S is parameterized alternation simulation. Basically, the alternation simulation [AH01a] identifies pairs of states which must fulfill a property in order to ensure refinement of specifications. The property P parameterizing the particular variant of refinement is designed to preserve the corresponding error.

Currently, the refinement requires that the number of threads originated in the environment is limited. The limit k goes through all the following definitions and theorems. In this sense, we define a weaker notion of refinement as follows:

Definition 24 (Refinement up to k threads). Let I and S be TBP models and E is a set of all TBP models such that $\forall e \in E : e \oplus I$ is a closed TBP model and e does not invoke more than k provided methods of I in parallel.

- We say that I refines S with respect to bad activity up to k threads iff $\forall e \in E : ErrFree_{BA}(e \oplus S) \Rightarrow ErrFree_{BA}(e \oplus I)$
- We say that I refines S with respect to no activity up to k threads iff $\forall e \in E : ErrFree_{\oslash}(e \oplus S) \Rightarrow ErrFree_{\oslash}(e \oplus I)$
- Finally, we say that I refines S up to k threads iff $\forall e \in E : ErrFree(e \oplus S) \Rightarrow ErrFree(e \oplus I)$

7.3.1. Provision-driven Computation

In contrast to the closed specification, the open specification, besides its internal threads, allows threads from the environment to invoke its provided methods. The way the external threads invoke the provided methods is, however, limited by provisions. Thus, in the provision-driven computation (which is an LTS) also externally triggered activity occurs (i.e. transitions representing invocation of a provided method by an external thread).

Formally, provision-driven computation is a tuple $C_{PD} = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$. Notice that it differs from the computation signature by the set of labels appearing on transitions—while the computation transition uses only Σ_{imp}, there can be also input actions representing provided methods (Σ_{all}). In the remainder of this paper, we use ?m to denote $m \in \Sigma_{prov}$, !m to denote $m \in \Sigma_{req}$ and finally τm to denote $m \in \Sigma_{int}$.

In contrast to closed computation, the number of stacks in the individual states changes to reflect the external threads allowed by provisions to call the provided methods. Moreover, individual states also contain the information about the actual state of provisions.

The first step to create a provision-driven computation is to combine individual provisions of the specification. This is done in two phases. First, each provision is normalized to formally guard all methods from Σ_{prov}. The set of traces allowed by the normalized provision is equal to the set of traces allowed by the original provision. The normalization is done by interleaving all original traces by arbitrary invocations of methods that were not guarded by the original provision. In the second phase, the intersection of normalized provisions is created to achieve combined provision.

Definition 25 (Normalized provision for k threads). Let $P = (filter, traces)$ be a set of provisions of a TBP model M and Σ_{prov} the set of provided method names of M. The normalized provision for k threads is defined as $P^{norm,k} = (\Sigma_{prov}, traces|L((m_1 + m_2 + \ldots + m_n) * ||k))$, where the operator | produces all interleavings of its operands, $m_1, \ldots, m_n \in \Sigma_{prov} \setminus filter$ and $L((m_1 + m_2 + \ldots + m_n) * ||k)$ is a language of traces capturing at most k parallel repetitive invocations of methods $m_1, \ldots, m_n$.

Definition 26 (Combined provisions for k threads). Let $P = \{P_1, P_2, \ldots, P_n\}$ be a set of provisions of a TBP model M where $P_i = (\mathit{filter}^{P_i}, \mathit{traces}^{P_i})$. Then, the combined provisions of M is a provision $\mathit{Prov}_M^k = (\Sigma_{prov}, \bigcap_{i=1,\ldots,n} \mathit{traces}_{P_i}^{norm,k})$ where $\mathit{traces}_{P_i}^{norm,k}$ is the set of traces of the normalized provision $P_i^{norm,k}$.

If the provisions are seen as finite automata, which is possible as each of them defines a regular language, the combined provision is a finite automaton formed as the intersection of automata corresponding to particular normalized provisions.

As long as the environment features less then k threads, the composed provisions allow the environment to perform exactly the same behavior as was allowed by the original set of provisions. The fact is expressed by the following lemma:

Lemma 2. Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model and $N = (\Sigma, \{(\Sigma_{prov}, \mathit{Prov}_M^k)\}, R, T, G)$ is the same model where the set of provisions P was replaced by composed provisions of M for k threads where k is the number of threads in M ($|T|$). Then, $\mathit{ErrFree}(M) \Leftrightarrow \mathit{ErrFree}(N)$.

The Lemma 2 can be proven in the following way. Using the alternative definition of ErrFree(M) ($(L(C)^{\checkmark} \upharpoonright \mathit{filter}) \subseteq \mathit{traces} \wedge (F^{\oslash} = F_{int}^{\infty} = \emptyset)$), one can see that the provisions influence only the $(L(C)^{\checkmark} \upharpoonright \mathit{filter}) \subseteq \mathit{traces}$ part of definition. Moreover, the closed computation $L(C)$ remains the same for both M and N. For a particular provision P_i of M, let us denote $M_{errtr} = (L(C)^{\checkmark} \upharpoonright \mathit{filter}_{P_i}) \setminus \mathit{traces}_{P_i}$ and $N_{errtr} = L(C) \setminus \bigcap_{i=1,\ldots,n} \mathit{traces}_{P_i}^{norm,k}$ and prove $M_{errtr} = \emptyset \Leftrightarrow N_{errtr} = \emptyset$. There is a trace t in M_{errtr} iff $\exists t' \in L(C)$ such that $t = t' \upharpoonright \mathit{filter}_{P_i}$ and at the same time $t \notin \mathit{traces}_{P_i}$. Either t is too short (it is a prefix of a trace from traces_{P_i}) or α is the first symbol of t that differs from the trace from traces_{P_i}. In the former case, t' is just a prefix of a trace from $\mathit{traces}_{P_i}^{norm,k}$ (we were adding interleavings, thus, no trace was shortened). In the latter case, since $\alpha \in \mathit{filter}_{P_i}$, one can find the same (i-th) occurence of α in t' as well and there is no prefix of a trace from $\mathit{traces}_{P_i}^{norm,k}$ that would follow t' up to and inluding α (we were adding just arbitrary interleavings of methods not belonging to filter_{P_i}). Thus, $t' \in L(C) \setminus \bigcap_{j=1,\ldots,n} \mathit{traces}_{P_j}^{norm,k}$.

For the opposite direction, the idea is the same, one just have to use the fact that number of threads is limited by k, thus there is no problem with limited reentrancy in normalized provisions.

Once composed provisions representing the behavior that the model M expects from the environment is available, provision-driven computation can be constructed. The following definitions are based on definitions of closed computation. For instance, the provision-driven computation state is a computation state enriched by a position in the combined provisions.

Definition 27 (Provision-driven state for k threads). Let $M = (\Sigma, P, R, T, G)$ be a TBP model and $\mathit{Prov}_M^k = (S, s_0, \delta, F)$ its composed provisions for k threads. A *state of Provision-driven computation of M for k threads* is a tuple $(\mathit{Stacks}, \gamma_G, s)$, such that the tuple $(\mathit{Stacks}, \gamma_G)$ forms computation state and s identifies a state from Prov_M^k $(s \in S)$.

In addition to the transition of a closed computation, the provision-driven computation transition is changing the state of combined provisions as a method is invoked. Moreover, there are transitions representing invocation of provided methods by threads from the environment.

Definition 28 (Provision-driven transition for k threads). The rewriting rules in Fig. 5 define the transitions among provision-driven states. While the top part of each rule references the computation transition (Def. 14), the bottom part defines a new provision-driven transition. Let $(\mathit{Stacks}, \gamma_G, s_{Prov})$ be a provision-driven computation state of the TBP model $M = (\Sigma, P, R, T, G)$, where s_{Prov} is a state of $\mathit{Prov}_M^k = (S_{Prov}, s_{0Prov}, \delta_{Prov}, F_{Prov})$.

In Fig. 5, the rule (a) produces a transition which does not modify the provision state. The rule (b) and (c) produces a transition representing active invocation of a method by the model. The provision state is modified to reflect the invocation. If the provision does not provide the required transition, the computation state causes bad activity error—$(\mathit{Stacks}, \gamma_G, s_{prov}) \in F^{BA}$. The rule (d) produces a transition representing invocation of a method by environment. The provision state is modified to reflect the invocation.

It is worth to notice that while the transitions defined in (a), (b), and (c) represent an activity performed actively by the model (!m), the rule (d) states that the model is allowed to perform the activity if asked by the environment (?m).

(a) $$\frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G)}{(Stacks,\gamma_G,s_{prov})\xrightarrow{\alpha}_{PD}(Stacks',\gamma'_G,s_{prov})}$$

where $\alpha = \tau$ or α is a required method call

(b)(c) $$\frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G),\delta_{prov}{:}s_{prov}\xrightarrow{\alpha}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\xrightarrow{\tau}_{PG}(Stacks',\gamma'_G,s'_{prov})} \quad \frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G),\delta_{prov}{:}s_{prov}\not\xrightarrow{\alpha}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\in F^{BA}}$$

where α is a provided or internal method call

(d) $$\frac{\delta_{prov}{:}s_{prov}\xrightarrow{m^{\uparrow}<a_{1..n}>}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\xrightarrow{m^{\uparrow}<a_{1..n}>PG}(Stacks\cup\{((s'_0,\gamma'_L),null)\},\gamma_G,s'_{prov})}$$

where $R(m) = (L', p, l')$
s'_0 is the initial state of l'
$\gamma'_L = \gamma^0_{L'}[p(i) \mapsto a_i]$

Fig. 5. Rewriting rules defining *provision-driven transition for k threads*

Definition 29 (Provision-driven computation for k threads). The provision-driven computation of a TBP model $M = (\Sigma, P, R, T, G)$ for k threads is the tuple $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F, !F)$, where:

- A label from $(\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E}$ is either an event representing issuing of a method call or acceptance of a method call parameterized by constants from Dom_E.
- $s_0 = (Stacks_{init}, v^0_G, s^0_{Prov})$ is initial provision-driven state. The set $Stacks_{init}$ contains a stack for each thread $t = (L, LTSA_t)$ from T containing single item (s^0_t, γ^0_L), where s^0_t is the initial state of $LTSA_t$ and γ^0_L is initial valuation of L. s^0_{Prov} is an initial state of $Prov^k_M$.
- $\delta \subseteq S \times (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\} \times S$ is a transition relation corresponding to the provision-driven transition for k threads.
- S is a set of provision-driven computation states reachable by δ from s_0
- $!F \subseteq S$ is a set of imperative final states. The state $s = (Stacks, \gamma_G,\ s_{Prov})$ is a final imperative state if for each model's thread t there is a $stack \in Stacks$ containing just single item (s', L) such that s' is a final state of $LTSA_t$. There are no other stacks in $Stacks$ representing the threads originated in the environment.
- $F \subseteq S$ is a set of final states. The state $s = (Stacks, \gamma_G, s_{Prov})$ is a final state if it is imperative final state and s_{Prov} is a final state of $Prov^k_M$.

In addition to the final states F representing the situation when both internal threads and provisions are in a final state, we also define the set $!F \subseteq S$ to denote the states where all threads are in a final state, but there is no restriction on the provision state ($!F \subseteq F$). Those states reflect the situation when the model is neither obliged to perform any action on its own nor to terminate.

Similarly to the closed computation, using the definition of provision-driven final states, we define the sets $F^{\oslash}$, F^{∞} and F^{∞}_{int}. Moreover, we define F^{BA} to be a set of states causing the bad activity error (Definition 28 (c). In contrast to the closed computation, mere existence of an error state (e.g. $s \in F^{\oslash}$) does not automatically mean that the model is useless. It can still work perfectly in a number of environments which use it in a way such that the error is avoided.

The provision-driven computation is well defined even for closed systems. Such computation features no transitions labeled by input actions (?m). Moreover, it can be used for detection of error states.

Lemma 3. Let M be a closed TBP model containing k threads and $C^k_{PD}(M)$ is its provision driven computation for k threads. The set of error states F^{BA} is empty iff there is no bad activity state in M.

7.3.2. Observation Projection

A key step when deciding whether the I model (implementation) refines the S model (specification) is to compare their ability to work in various enclosing environments—the model I has to work in all environments where S works. Thus, only the model's behavior observable by the environment is important. The environment cannot make any assumptions regarding the internal non-determinism of the model (including internal communication). To include this fact in further reasoning about refinement, we define the *observation projection* of the provision-driven computation. The observation projection is an LTS which abstracts from the non-determinism of the original computation in a pessimistic way—whenever an error can occur due to the non-determinism, it must be reflected in the observation projection.

Each state of the observation projection (super-state) represents a set of states of the original provision-driven computation. The individual states represented by the same super-state cannot be distinguished by any environment. For instance, states originally connected by a τ transition always belong to the same super-state.

Let $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a provision-driven computation of a TBP model M for k threads. Let $A \subseteq S$ be a set of states. We define τ-$closure(A)$ as to be a set of states reachable from $s \in A$ by a set of externally invisible transitions—τ-transitions originated as assignments (cases (a) and (b) in Definition 29).

Definition 30 (Observation projection for k threads). Let $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a provision-driven computation of TBP model M for k threads. Then the *observation projection of M for k threads* is a tuple $C^k_{OP}(M) = ((\Sigma_{ext})^{\uparrow/\downarrow}_{Dom_E}, S_{OP}, s^0_{OP}, \delta_{OP}, F_{OP}, !F_{OP})$ such that

- $S_{OP} \subseteq 2^S$
- $s^0_{OP} = \tau closure(\{s_0\})$
- $\delta_{OP} : S_{OP} \times (\Sigma_{ext})^{\uparrow/\downarrow}_{Dom_E} \to S_{OP}$
  $\delta_{OP}(s_{OP}, ?m) = \tau closure(s'_{OP}) \Leftrightarrow \forall s \in s_{OP} \exists s' \in s'_{OP} : (s, ?m, s') \in \delta$
  $\delta_{OP}(s_{OP}, !m) = \tau closure(s'_{OP}) \Leftrightarrow \exists s \in s_{OP} \exists s' \in s'_{OP} : (s, !m, s') \in \delta$
- $F_{OP} = \{s_{OP} : \forall s \in s_{OP} : s \in F\}$
- $!F_{OP} = \{s_{OP} : \exists s \in s_{OP} : s \in !F\}$

Moreover, we define sets of various error states. In particular

- $E^{BA}_{OP} = \{s_{OP} : \exists s \in s_{OP} : s \in F^{BA}\}$
- $E^{NA}_{OP} = \{s_{OP} : \exists s \in s_{OP} : F^{\oslash} \cup F^{\infty}_{int}\}$
- $E_{OP} = E^{BA}_{OP} \cup E^{NA}_{OP}$

Observation projection simplifies the original provision-driven computation by reducing non-determinism. The internal choices are expected to behave as in the worst case w.r.t. bad activity; if one of the states belonging to a super-state (e.g., several states connected by internal actions) produces an output action, it must be produced also by the super-state. On the other hand, an input action leaving the super-state must be present in all states of the super-state. In other words, output action in the observation projection represents option of the model to emit the action while input action represents obligation to accept it.

Fig. 6 contains a fragment of a provision-driven computation. There are states connected by transitions labeled by internal, input and output actions. The leftmost state is the initial one while the rightmost state is a final state. The rest of the LTS is represented by the dashed line. The corresponding observation projection (Fig. 7) consists of three super-states. The super-state k is the initial state, since one of the states it represents is the initial state of the provision-driven computation. Moreover, there is no transition labeled by an input action leaving k since not all of the states are ready to accept `?p`. On the other hand, there are two output transitions. The one labeled by `!q` leads to the super-state l, while the other one leads to the part of the observation projection which is not depicted. In contrast to k, there is an input transition leaving the super-state l, since all the states in l are ready to accept `?p`. The target states of these transitions are not distinguishable by the environment. Thus, they all belong to the super-state m.

The definitions provided in this section gradually simplify an open TBP model so that, in the end, the observation projection is just a transition system labeled by input and output actions—similarly to interface

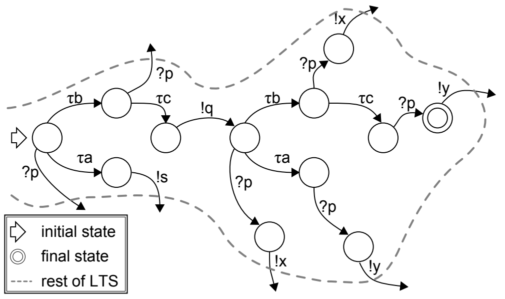


Fig. 6. Fragment of provision-driven computation

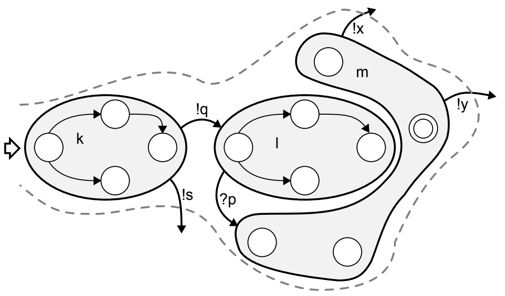


Fig. 7. Fragment of observation projection

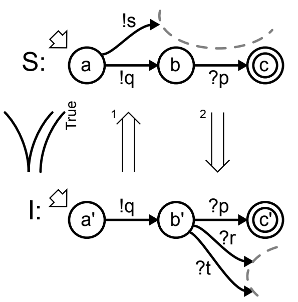


Fig. 8. Alternation simulation example

automata. There are, however, two significant differences. The transition system is deterministic (there are no internal actions and each state contains at most one outgoing transition for each external action). Second, there is also additional termination information associated with super-states.

Determining whether the observation projection $C^k_{OP}(I)$ refines $C^k_{OP}(S)$ is based on the parameterized alternation simulation.

Definition 31 (Parameterized alternation simulation). Let I and S be observation projections of TBP models. Let S_I be the set of states of I, S_S be the set of states of S, δ_I and δ_S be the transition functions of the respective observation projections. Moreover, let $P \subseteq S_I \times S_S$ be a relation. Then, we call the relation $\preceq_P \subseteq S_I \times S_S$ *parameterized alternation simulation* if

- $\forall (s_I, s_S) \in \preceq_P$: $(s_I, s_S) \in P$
- $\forall (s_I, s_S) \in \preceq_P$: $\delta_I(s_I, !m) = s'_I \Rightarrow \exists s'_S : \delta_S(s_S, !m) = s'_S \wedge s'_I \preceq_P s'_S$
- $\forall (s_I, s_S) \in \preceq_P$: $\delta_S(s_S, ?m) = s'_S \Rightarrow \exists s'_I : \delta_I(s_I, ?m) = s'_I \wedge s'_I \preceq_P s'_S$

The relation is extended to observation projections using initial states as follows.

We say that I refines S with respect to the property P ($I \preceq_P S$) iff there is a parameterized alternation simulation $\preceq_P$ such that $s^0_I \preceq_P s^0_S$ where s^0_I is the initial state of I and s^0_S is the initial state of S.

The parameterized alternation simulation stems from the alternation simulation introduced for interface automata in [AH01a]. The main purpose of the alternation simulation is to relate states of the individual observation projections that correspond to each other from the observer's point of view. Let E be an environment enclosing S and the result of composition does not contain any error. Then, when $C^k_{OP}(S)$ exercised by the environment E is in the state s_i, then $C^k_{OP}(I)$ exercised by E must be in the state s_s such that $s_i \preceq_P s_s$.

The additional predicate P allows specifying an additional property that must hold to address a specific error. In particular, $I \preceq_{True} S$ preserves bad activity that occurs in communication among I (resp. S) and its enclosing environment E, however, it considers neither no activity nor bad activity caused by internal communication within I.

Fig. 8 contains an example of two observation projections I and S such that $I \preceq_{True} S$. In particular, $c' \preceq_{True} c$ holds trivially, since there are no leaving transitions and $(c', c) \in True$. The implication 2 and

$c' \preceq_{True} c$ implies $b' \preceq_{True} b$ and the implication 1 and $b' \preceq_{True} b$ implies $a' \preceq_{True} a$. Since the initial states $a' \preceq_{True} a$ then also $I' \preceq_{True} S$

Lemma 4 (Transitivity of $\preceq_P$). Let A, B and C be observation projections of TBP models, and P be a transitive relation over their sets of states S_A,S_B and S_C (i.e. $(s_A P s_B) \wedge (s_B P s_C) \Rightarrow (s_A P s_C)$). Let $A \preceq_P B$ and $B \preceq_P C$. Then $A \preceq_P C$.

7.3.3. Preserving Bad Activity

Theorem 1 (Refinement w.r.t. BA up to k threads). Let I_{OP} resp. S_{OP} be an observation projection of a TBP model I resp. S for k threads. Let E_I^{BA} resp. E_S^{BA} be sets of bad activity error states in the observation projections. Let $BA(a,b)$ be a relation over set of states of observation projections such that

$$BA(a,b) \Leftrightarrow (a \in E_I^{BA} \Rightarrow b \in E_S^{BA}).$$

Then $I_{OP} \preceq_{BA} S_{OP}$ implies that I refines S with respect to bad activity up to k threads.

7.3.4. Preserving No Activity

To define a refinement relation preserving the no activity error additional information is needed in the observation projection. Let A_{OP} be an observation projection. The predicate *running* holds for each state of the observation projection that has to emit an output action. The predicate *running* is defined as follows.

Definition 32 (Running). Let s_{op} be a super-state of the observation projection A_{OP} and S be a set of states in the provision-driven computation represented by s_{op}. Then

$$\begin{aligned} running(s_{op}) = \quad & s_{op} \notin !F \\ & \wedge \\ & \forall s \in S \quad \exists s_\tau, !m, s' : s_\tau \in \tau closure(s) \wedge \delta(s_\tau, !m) = s' \end{aligned}$$

The definition consists of two properties of the state. If the set of states of the provision-driven computation represented by the super-state contains a state representing termination of all active threads ($s_{op} \in !F$), the whole model can terminate on its own while not emitting an output action. The second property states that for each state of the set, there has to be a path consisting of internal actions leading to a state s_τ producing an output action.

Theorem 2 (Refinement w.r.t NA up to k threads). Let I_{OP} resp. S_{OP} be an observation projection of a TBP model I resp. S for k threads. Let E_I^{NA} resp. E_S^{NA} be sets of no activity error states in the observation projections. Let $NA(i,s)$ be a relation over set of states of observation projections such that.

$$\begin{aligned} NA(i,s) = \quad & BA(i,s) \wedge (i \in E_I^{NA} \Rightarrow s \in E_S^{NA}) \\ & \wedge \\ & s \in F \Rightarrow (i \in F \vee running(i)) \\ & \wedge \\ & running(s) \Rightarrow running(i) \end{aligned}$$

Then $I_{OP} \preceq_{NA} S_{OP}$ implies that I refines S with respect to no activity up to k threads.

8. Experiment—TBP model of CoCoME assignment

We tested the capabilities of TBP on a model of a supermarket information system involving row of cashdesks, each featuring a number of devices (keyboard, credit card reader, etc.), and a central database of goods in the store. The model is taken over from the CoCoME contest assignment [RRMP08]. The goal of the CoCoME contest was to compare strengths and weaknesses of different modeling approaches. In particular, each participant provided a model using their own formalism. The approaches of contest participants differ in both goals and means. Some approaches aim at performance modeling and prediction (Paladio, Klapper), while other aim at the functional correctness (rCOS, Java/A, CoIN). The means used by approaches aiming at functional correctness range from finite automata (CoIN), to those employing concepts of preconditions/postcontidion formulas. When compared to goals and means of TBP, the most related approaches used in CoCoME were CoIN, Java/A, and our BP.

By providing a TBP model of the CoCoME example, we can compare it to the models provided by participants [CoC] as well as to the Java implementation, which was created as a part of the assignment. Unfortunately, there is no model of CoCoME in Interface Automata, the formalism which inspired definition of refinement in TBP. On the other hand, in contrast to TBP, we consider Interface Automata to be more a theoretical concept than a specification language intended for applications.

The TBP model of the CoCoME assignment is based on the architecture we already created for the EBP model [BDH+08]. When compared to the other specification languages, it was straight-forward to express multiple bindings in the architecture. In particular, neither reactions, nor provisions of a component providing a method to several other components (e.g., EnterpriseServer shared by many StoreServers) need to specify the particular degree of parallelism. Moreover, the sync keyword approved very useful for modeling mutual exclusion and it significantly simplified the specification of buses. All in all, the means provided by TBP enabled us to create a model that resembles the real implementation more closely than models in CoIN, Java/A and BP.

The TBP model of CoCoME as well as the tool for TBP checking (partially work in progress) is available at [BGR]. Even though the CoCoME assignment is far from being trivial, the initial version of the model was crafted roughly in one day. The analysis of the model revealed a deadlock (in bus access) and also a violation of provisions (the provision of StoreLogic did not allow one to process a sale while the StoreLogic component was processing accepting goods from another store). Specifically the latter would be very hard to discover manually.

9. Evaluation and Discussion

One of the objectives of the TBP design was to provide a formalism that would be simple enough for use by practitioners during day-to-day development. To achieve this goal, both TBP syntax and semantics are designed to be close to common imperative languages. At the same time, another goal was to be able to benefit from formal analyses of models specified upon an LTS background. An important requirement in the scenario was that verification of built-in properties and of refinement is decidable so that both could be analyzed by a tool.

In terms of the Goals, stated in Chapter 4, both of them have been achieved: (1) The syntax of the TBP formalism builds upon the constructs (if, switch, while, sync) and abstractions (threads, types, variables) present in imperative programming languages, thus being easier to comprehend and use by a developer in comparison to the traditional behavior modeling approaches, such as automata and process algebras. (2) The proposed TBP formalism supports reasoning about component composability and refinement, which provides a developer platform supporting both the bottom-up and top-down design. Here, specific kinds of errors, inherent to component applications, are detected (bad activity, no activity). Furthermore, thanks to fulfilling Goal 1, it is possible to reason about conformance of the code (implementation) with the corresponding behavior model (specification)—this, however, is beyond the scope of this paper.

Below we discuss other important TBP properties which make behavior modeling of components to be implemented in imperative object-oriented languages convenient.

Specification of Provisions. Provisions specify how an environment is expected to call the methods of the component. This information is not present in the code itself, at least not explicitly, therefore the developer should state the requirements of the component put on its environment within the TBP specification.

The language of provisions is inspired by BP [AP04]. In contrast to BP, method-call-related events are equipped with parameters and return values of enumeration types. This allows specifying assumptions on the environment with a finer granularity than just imposing particular sequencing of method calls.

Another important aspect is the granularity of assumptions' specification. For a component's model, several provisions guarding different sets of methods are allowed. Some methods can be omitted by the provisions section—these can be invoked arbitrarily by the environment, even in parallel. At the same time, a method can be guarded by several provisions and then the environment has to follow all of them. This approach supports separation of concerns.

	Models		TBP		Stuck	IA Error
	A	B	Bad Act.	No Act.		
a)	$\bigcirc \xrightarrow{!a} \circledcirc$, $\bigcirc \xrightarrow{!b} \circledcirc$	$\bigcirc \xrightarrow{?a} \circledcirc$	Yes	No	No	Yes
b)	$\bigcirc \xrightarrow{!b} \bigcirc$	$\bigcirc \xrightarrow{?a} \bigcirc$	Yes	No	Yes	Yes
c)	$\bigcirc \xrightarrow{?b} \bigcirc$	$\bigcirc \xrightarrow{?a} \bigcirc$	No	Yes	Yes	No
d)	$\circledcirc$	$\circledcirc \xrightarrow{?a} \circledcirc$	No	No	Yes	No

Fig. 9. Communication errors in composition of A and B

Communication. The LTS specified by the model captures parallel interleaving of activities executed by individual threads. Individual threads influence each other only by modifications of state variables (this includes locks).

As long as the parallelism is not explicitly limited in provisions, a method can be invoked as many times in parallel as the environment requires.

In contrast, when process algebras are applied in component behavior modeling, every component is typically modeled by a single process. Communication among components is represented by the synchronization of actions representing method calls. A direct consequence is that method calls are represented as if communication among components was asynchronous—(a) the method call is not accepted if it is not explicitly awaited by the target process and (b) the caller process continues in the computation not waiting for a result. This is far from the semantics of method calls in imperative languages.

While the issue (b) can be resolved by using pairs of actions—an output action at the source process is immediately followed by the corresponding input action waiting for the result (and vice versa), the issue (a) requires further attention—it is most striking when the user wants to model a method which can be invoked in parallel as many times as the environment requires. As an aside, all Java classes exhibit such behavior by default. Process algebras (CSP in particular) support recursion which allows unlimited number of processes starting in parallel, as soon as required by the environment. This however, makes the formalism more complex and the state space of an open component system infinite.

Composition. In contrast to process algebras, composition of specifications in TBP is defined at the syntax level as union of the corresponding sets. The semantics defines how to create an LTS for an open model (observation projection). Since there is no means for composition of two observation projections, two open models are composed at the syntax level into a single model (possibly a closed one) which is transformed into an LTS.

Communication Errors. The definition of communication errors supporting TBP is based on comparison of the behavior actively performed by threads and description of the behavior passively expected by provisions. Since both of them can be expressed as a set of traces, the required properties can be stated as inclusion of these sets.

When comparing the actual behavior to the expected behavior at the trace level, there are hardly any errors other than bad activity and no activity. However, individual formalisms targeting (component) behavior modeling differ significantly in the way traces are constructed, specifically in dealing with non-determinism and performing internal/external choice. An important aspect related to non-determinism is the "degree of optimism" the formalism takes—whether a model containing an error state is considered erroneous: In case of the optimistic approach (Interface Automata), the model is considered correct if there exists an environment making the error state unreachable, while in the pessimistic approach existence of an error state implies the model is erroneous. In TBP, in particular, the error is reported if it can occur, i.e., the pessimistic approach is taken.

Even though it might look simple, there is a caveat to defining communication errors: Fig. 9 illustrates the differences among the TBP errors (bad activity and no activity, introduced in [AP03, AP04]) and the stuck error introduced in [FHRR04a] and discussed in [FHRR04b]. Roughly, stuck error "combines" both bad activity and no activity. Fig. 9 presents pairs of label transition systems demonstrating differences among the errors. In each row, the state at the left-hand side is always an initial state while a double circle represents a final state.

The key differences are characterized by the following four state transitions: The pair (a) demonstrates the difference between the bad activity error and stuck error. While in TBP composition of A and B is considered to be erroneous because of the !b action from A is not accepted by a counterpart in B, the same situation does not cause a stuck error, since it suffices that at least an action is accepted by the counterpart (!a and ?a). Composition of the pair (b) produces a stuck error—no action leaving the initial states is accepted. This composition pair also reflects the difference between a stuck error and no activity. Since the definition of no activity, unlike of stuck error, considers final states and cannot occur at final states, there is a stuck error but not a no activity. The row (c) demonstrates that bad activity distinguishes input and output actions (? and !) while stuck error does not. Finally, the row (d) illustrates the difference between no activity and stuck error. While no activity can occur only in the states that are not final, stuck error does not consider final states at all.

The bad activity error has the same meaning as the error introduced in [AH01a] for Interface Automata; nevertheless, there is no concept of no activity (deadlock) in [AH01a].

Refinement. Fig 9 also demonstrates that the concept of refinement is based on the alternation simulation introduced in [AH01a] for interface automata. For TBP, however, the relation is strengthened to preserve absence of no activity. The definition of refinement for TBP is provided in three steps.

First, the provision-driven computation transforms an open TBP model into an LTS similar to the interface automata—there are input, output, and internal actions. In the next step, observation projection is created in the form of a deterministic LTS. The final step checks the parametrized alternation simulation of two observation projections. This means to compare the options and obligations of the corresponding states from the "implementation" and "specification", i.e., the models being subject to refinement checking. First, the initial states are compared and then the process continues by comparing the pairs identified by the transitions labeled by the same actions.

By defining refinement in three steps, the theoretical framework behind gets more modular. In particular, the second step gathers the information from several states of a single provision driven computation indistinguishable by the environment into a single state of the observation projection. Most importantly, the second step resolves non-determinism; it determines what events important for preserving errors must or may occur. Thus, the third step does not consider these differences, since they are already abstracted away so that it can be defined in a straightforward way as a simple comparison of two states.

In this context, it is worth mentioning the special position of the refinement w.r.t. bad activity. The comparison of the sets of input and output actions in the third step ensures preservation of bad activity occuring on the component boundary on its own. At the same time, the comparison of the sets of actions forms a basis for relating pairs of states. The parameter formula is then applied on these pairs.

Expressiveness compared to Java. By comparing the Java implementation of the CoCoME assignment to its TBP specification, it becomes obvious that TBP specification omits technical details (such as accessing a database via specifically DerbyDB and using ActiveMQ), which make the implementation infeasible for code model checking (e.g., by JPF [HP00]). In principle, to employ such a code model checker for verification we would need a simplified Java model obtained by either (a) omitting details from the implementation, or (b) designing it from scratch. Such a model could be limited to the features available in TBP—limited number of threads, enumeration variables, and assertions in the code for expressing assumptions on the environment. Non-deterministic choices could be modeled by the Random class in Java. By JPF, such a Java model would allow checking deadlock freedom (no activity), as well as checking bad activity as obeying assumptions. To express the desired sequences of method calls, one would have to keep the related state (e.g., position in the sequence, state of the property automaton) manually in a Java variable and check its value in injected assertions or using a custom PropertyListener. In contrast, provisions available in TBP provide much more convenient way to express the desired sequences of method calls. Modeling the TBP reentrancy provision operator in Java would be also very tricky. Finally, even if we limited ourselves to enumeration variables, the size of the state space of the whole composition modeled in Java could become unfeasibly large for non-trivial applications. In TBP, compositional verification using refinement in TBP implies partitioning of the state space, shifting thus feasibility limits further. No similar technique is available in JPF.

10. Conclusion

The proposed specification language, TBP, aims at narrowing the gap between imperative languages used in industrial software development and behavior specification languages.

Key achievements and benefits. While TBP remains a specification language, it tries to get as close as possible to the syntax and semantics of the mainstream imperative languages (Java in particular). Specifically, both the syntax and semantics are inspired by the imperative object-oriented programming languages the industrial developers are used to (Goal 1.a). Thus, the model can be crafted by similar means as the implementation—developers operate with the same concepts and the specification structure can be followed in the implementation as far as to the individual control flow statements. An important benefit is also that synchronization in TBP is inspired by the approach taken by Java. At the same time, TBP specification allows specifying assumptions on the environment, fulfilling thus Goal 1.b.

TBP allows analyzing models of applications in the context of component systems (Goal 2). In particular, TBP supports checking compositional correctness and refinement. The notion of correctness is based on verifying the actual behavior of a closed system against assumptions.

To sum up, TBP is a step towards narrowing the gap between a specification and implementation language. Strictly speaking, most of the TBP features and ideas have already appeared either in the world of behavioral modeling or in an implementation language. TBP, however, puts the ideas together in a novel way—in particular, unique is the option of reasoning on refinement of individual components' models, while the computation is driven by threads as in a real application.

When compared to previous formalisms from the Behavior Protocols family, TBP is a step forward in user friendliness and in the properties of formal framework.

The models related to an implementation are easier to write and understand in TBP than in other specification languages as demonstrated on the session manager example (Sect. 6) and its versions published in [CoC]. Regarding the formal framework, we consider beneficial that the refinement relation preserves both bad activity and no activity errors. Moreover, the refinement relation is modular, which helps enhance it in order to support additional errors. Even though there are other formalisms available supporting similar notion of refinement, to our best knowledge none of them follows the imperative object oriented languages as closely as TBP does. In particular, the behavior based on threads where each thread has its own stack allows the user to follow the object oriented design much closer than in any other formalism and still reason about refinement at the level of application architecture.

Future Work and Open Issues. There are still several features worth modeling, not supported in TBP. Specifically, these include: Dynamic thread creation, support for evolving architectures (software logic often involves dynamically created logical objects important for the high level behavior), and unlimited number of threads considered in the refinement.

We intend to address the last one as follows: To provide a refinement relation supporting unlimited number of threads triggered by environment, we want to add level of parallelism (based on the reentrancy operator) to the observation projection and define a relation that compares implementation and specification in a way that considers the actual level of parallelism. Nevertheless, dynamic thread creation and dynamic object support still remain open issues.

References

[ABC10] Alessandro Aldini, Marco Bernardo, and Flavio Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer, 2010.

[ABJ+06] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component reliability extensions for Fractal component model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.

[AH01a] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[AH01b] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, CMU, 1997.

[AP03] Jiri Adamek and Frantisek Plasil. Behavior protocols capturing errors and updates. In *Proceedings of the 2 nd International Workshop on Unanticipated Software Evolution*, 2003.

[AP04] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BDH$^+$08] T. Bureš, M. Děcký, P. Hnětynka, J. Kofroň, P. Parízek, F. Plášil, T. Poch, O. Šerý, and P. Tůma. CoCoME in SOFA. In *The Common Component Modeling Example: Comparing Software Component Models*, pages 388–417, Berlin, Heidelberg, 2008. Springer-Verlag.

[BGR] Badger—Verification of component behavior specification. `http://d3s.mff.cuni.cz/~sery/badger`.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2005.

[CoC] Modelling Contest: Common Component Modelling Example, `http://agrausch.informatik.uni-kl.de/CoCoME`.

[CSS05] Edmund M. Clarke, Natasha Sharygina, and Nishant Sinha. Program compatibility approaches. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Lecture Notes in Computer Science*, volume 4111, pages 243–258. Springer, Springer, 2005.

[ČVZ07] Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06)*, volume 182 of *ENTCS*, pages 39–55. Elsevier Science Publishers, June 2007.

[FHRR04a] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-Free Conformance. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2004.

[FHRR04b] Cedric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance theory for ccs. Technical report, Microsoft Research, July 2004.

[GG97] Richard Grimes and Dr Richard Grimes. *Professional Dcom Programming.* Wrox Press Ltd., Birmingham, UK, 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International (UK) Ltd., 1985.

[HP00] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[Kof07] Jan Kofron. Checking software component behavior using Behavior Protocols and Spin. In *Proceedings of Applied Computing 2007*, pages 1513–1517, Seoul, Korea, March 2007.

[LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.

[LS00] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of component-based systems.* Cambridge University Press, New York, NY, USA, 2000.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference, ESEC '95 , Barcelona*, 1995.

[Mil95] R. Milner. *Communication and Concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[MSD03] Vlada Matena, Beth Stearns, and Linda Demichiel. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform.* Pearson Education, 2003.

[OLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[OMG06] OMG Group. CORBA Component Model Specification. Technical report, OMG Group, 2006.

[Poc10] T. Poch. *Towards Thread Aware Component Behavior Specifications.* PhD thesis, Charles University, Prague, 2010.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

[Ros98] A. W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science.* Springer, 2008.

CHAPTER 8

On Partial State Matching

Authors: Pavel Jančík and Jan Kofroň

[4] *Formal Aspects of Computing, ISSN: 1433-299X, pp. 1–27, DOI: 10.1007/s00165-016-0413-z, Springer Verlag, January 2017*

On Partial State Matching

Pavel Jančík and Jan Kofroň[1]
Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 25, Praha 1, Czech Republic
pavel.jancik@d3s.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz
http://d3s.mff.cuni.cz

Abstract. During explicit software model checking, the tools spend a lot of time in state matching. This is implied not only by processing a huge number of states, but also by the fact that state representation is usually not small either. In this article, we present two dead variable analyses; applying them during the code-model-checking process results in size reduction of both state representation and explored state space itself. We implemented the analyses inside Java PathFinder and evaluate their impact in terms of memory and time reduction using several non-trivial benchmarks.

Keywords: Explicit model checking, dead variable analysis, optimization, performance.

1. Introduction

In explicit software model checking, the model checkers spend a lot of time in the state matching process. During the state space traversal, state matching identifies equivalent states to avoid multiple exploration of the same parts of the state space. This usually implies computing a state representation for each reached state that is easy to compare and store, and trying to find the state being currently explored in the set of states visited earlier. Many optimizations, such as Partial Order Reduction [Pel93] and thread symmetry [VHB$^+$03], have been introduced to reduce the number of states that need to be explored. At the same time, since the state representation in case of software model checking is usually not of negligible size, its reduction can significantly speed up the state matching process as well. The related optimization techniques in this case focus on fast compression of state representation, such as those in [Huf52, rle], hashing, and identification of unimportant parts of states [SM07, LJ06, BFG99], e.g., the environment variables being the same over the entire program run, to be excluded.

While most of the optimizations are easy and fast to compute and apply if information about entire state space is available (e.g., for partial order reduction, it is clear which traces end up in the same state and what are the potential successors of the states along them), they become challenging if the state space is

Correspondence and offprint requests to: Jan Kofroň, KDSS, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
[1] This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

generated on-the-fly; this is the typical case of today's tools. Then, conservative simplifications have to be made to preserve correctness of the model checking results.

1.1. Problem statement

The existing techniques for state space reduction in the on-the-fly state-space-generation settings focus themselves just either to identify equivalent sets of event sequences resulting in the same program state or in identification of unused parts (w.r.t. future behaviour) [Pel93] or identification of dead variables, but restricting themselves just to local or non-heap variables [BFG99]. However, there is much space for reduction of the state space representation also when data stored in the heap are considered (both global variables and object fields). Involving their consideration in dead variable reduction (DVR) can further improve the state matching performance resulting thus in more scalable and applicable software model checking.

1.2. Goals

In this article, we describe two techniques aiming at identifying dead parts of the states, i.e., data that does not influence the future behaviour of the program, considering also data stored in the heap. We leave the analysis of dead local variables to other techniques, which can be combined with our approach. In particular, we introduce two dead variable analyses (DVA) of data stored in the heap differing in computational demands and precision, both suitable for being applied in an on-the-fly software model checker. Further, we also show the benefits of applying each of them on several benchmarks demonstrating thus the contribution to performance of on-the-fly explicit model checking. This article is an extension of our previous work [JK16]; here, we describe the analyses in more detail as well as provide detailed proofs and more experimental results supporting the significance of our contribution. In addition to that, we also discuss the aspects of DVA on the bytecode level.

2. Background

The purpose of state matching is to detect *behaviourally equivalent states.* For complex representation of program states, it can happen that future behaviours of two or more different states are equivalent. Informally, a state matching algorithm should match states if the future behaviour of the program starting in either of them is the same w.r.t. the property being verified. For reachability properties this means that from both of them, a state violating the property can either be reached or not. Since this view would be difficult to apply in the on-the-fly settings, we adopt a weaker one: Two states are equivalent if the same set of equivalent states can be reached from either one.

We say that a variable is *dead* if it is not read until the program terminates or until its value is (re-)written before being read. Program states which differ in dead variables only are behaviourally equivalent and thus should be matched.

The contribution of DVR is two-fold. First, it reduces the state space, since states differing only in values of the identified dead variables are matched. This means that only a single representative of each set of matched states is explored. Second, state matching (i.e., canonicalization, hashing) can process only the live parts of state representation (ignoring dead parts), thus making the whole state matching process faster. This is also of a particular importance, since explicit-state model checkers spend a large amount of their runtime (approx. 30%) by state matching [NR09]. The former effect applies both for DVAs over local variables as well as for those focusing on the whole program state. On the other hand, the second effect can be observed only if a large enough portion of a program state is identified as dead, which can be expected only for the heap. Instead of ignoring dead variables, some DVR implementations set their value to a predefined constant (e.g., 0 or `null`). In such cases, the former effect is eliminated, since the size of the program state is not reduced at all. Note that the more precise DVA is, the more these effects manifest themselves.

Let us illustrate the aforementioned effects on the Java program in Fig. 1 and the red-black trees in Fig. 2 stored in the `tree` variable. The corresponding class is listed in Fig. 3. The tree is shared among threads and we assume that the program can generate either of them. No assertion is violated irrespective of whether either the left or the right tree has been generated. While the colours in red-black trees are used only in

Thread $T1$

```
1 public void run() {
2   ...
3   assert(tree.contains(5));
4 }
```

Thread $T2$

```
5 public void run() {
6   ...
7   assert(tree.contains(1));
8 }
```

Figure 1. Java program composed of two thread where the `tree` variable contains red-black tree from Fig. 2.

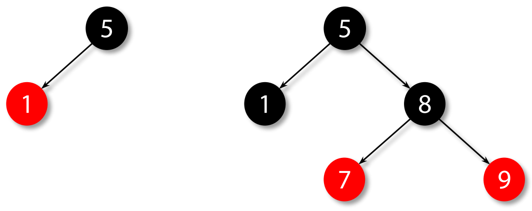


Figure 2. Two different red-black trees, which can be identified as equivalent using our DVA for program in Fig. 1.

modifying operations (insertions and deletions), the `contains` operation does not access them and thus the variables (fields) representing the colours of nodes are dead. The same holds for the right descendant of the root node holding value 5. Since operation `tree.contains(1)` reads only the left descendant of the root node, and operation `tree.contains(5)` accesses only the value of the root node, the whole right sub-tree is dead. It means that the program states where the `tree` variable holds the left resp. right tree of Fig. 2 are equivalent w.r.t. dead variables; the model checker can explore the successors only for the state that is reached first and does not need to re-explore the successors of the (equivalent) state reached later. This way the state space is reduced.

To illustrate the second effect, let us inspect the parts of the tree (in other words the parts of the program states) which are processed by the state matching (in the case of JPF, the data appearing in the state vector). Note that the code in Fig. 1 does not modify the tree below the lines 3 resp. 7. First of all, the `colours` are dead and thus can be ignored by state matching. More importantly, the right descendant of the root node (`tree.right`) is also dead and can thus be omitted from state matching as well. This simple fact causes the whole sub-tree `tree.right` (composed of nodes 7, 8, and 9) not to be accessed and thus it can be omitted from state matching. The latter effect speeds-up verification also if the state space is not reduced (i.e., the former effect does not apply). A prominent example of this effect is representation of environment variables, which form a non-negligible part of the state, however are seldom accessed by the program.

In order to be useful, DVR itself and DVA as its part need to be fast enough to pay off, while still having a modest memory demand. The DVRimplementation should be also compatible with all other state matching optimizations.

Another important question is how to cope with the states having different sets of dead variables; in our example each tree has a different set of them. The tree at the right-hand side has dead variables in nodes 7, 8, and 9, which do not exist in the tree at the left-hand side.

In the following section, we first briefly introduce our approach on the example above, while in Sect. 4 we formalize our approach and show its soundness. In Sect. 5, we describe our two analyses in detail and shows its properties. Sect. 6 compares our approach to related work, while Sect. 7 presents the results of the experiments demonstrating the contribution of our approach Sect. 8 concludes the article.

3. Running example

We have developed two versions of DVA for multi-threaded Java programs; the first one, called *Dynamic DVA* (DDVA), aims at precision. The second one, called *Hybrid DVA*, uses static analysis (for multi-threaded programs), and combines the results of the analysis with the knowledge from the dynamic (runtime) program state. This lightweight analysis is designed to be fast and easy to integrate into state matching.

We illustrate our analyses using the example from Fig. 1 and the `tree` variable holding the right-hand-

```
public class TreeNode {
  public int value;
  public Color color;
  public TreeNode left;
  public TreeNode right;
  ...
}
```

```
  boolean contains(int i) {
    int v = this.value;
    TreeNode child;
    if (v == i) return true;
    if (v < i) child = this.left;
    else child = this.right;
    if (child == null) return false;
    return child.contains(i);
  }
}
```

Figure 3. Red-black tree node representation

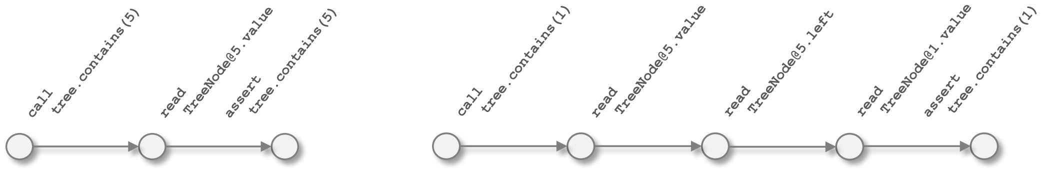


Figure 4. Future transitions of thread $T1$ resp. $T2$.

side tree from Fig. 2. The tree nodes are stored using the data structure from Fig. 3. We add a suffix holding the stored value to distinguish the instances of `TreeNode`s from each other; e.g., the root tree node is denoted as `TreeNode@5`.

Before we show how our analyses work, let us focus on the state space a bit. Thread $T1$ will call the `TreeNode.contains` method, which will read `TreeNode@5.value` only. The $T2$ also calls the `TreeNode.contains` method, which in this case reads `TreeNode@5.value`, `TreeNode@5.left`, and `TreeNode@1.value`. Although both threads are independent of each other and thus it is enough to explore only one of their interleavings, on-the-fly model checkers cannot be aware of this fact (since this requires knowledge about future behaviour of the program). To be safe, they assume, e.g., a possible conflicting write to a given shared field by another thread and create a non-deterministic choice at each of the field reads [PL11]. In other words, the model checker will generate 15 unique states out of which 8 states contain a non-deterministic scheduling choice deciding whether $T1$ or $T2$ will be scheduled for execution (Fig. 5).

3.1. Dynamic analysis

Our dynamic analysis tracks field reads and writes executed by the program and based on them it identifies live parts of the program state (i.e., live addresses). During the depth-first search (DFS), the model checker maintains two data structures related to visited states: (i) a stack of states currently present at the DFS stack, for which full state vectors are stored to enable detection of state-space cycles, and (ii) a set of visited states, for which the pairs $\langle$*live addresses*, *reduced state vector*$\rangle$ are stored; these are used for state matching. The reduced state vectors, which are stored after a state and its successors are fully explored, do not include dead variables.

To illustrate this process on our motivation example from Fig. 1, let us assume that the model checker uses the DFS exploration strategy and that thread $T1$ has a higher priority than $T2$; in other words, if both threads $T1$ and $T2$ can be scheduled at a state, the successor where $T1$ is executed is explored first. The corresponding state space is shown in Fig. 5. The model checker is in state $S1$ just before the `Tree.contains` calls where both threads can be scheduled, thus thread $T1$ is executed first. The model checker executes the call to the `Tree.contains` and continues execution until it encounters the read of `TreeNode@5.value` field. Just before the field read (in state $S2$) it creates a non-deterministic scheduling choice (to be able to read the value possibly modified by other threads) and the execution of thread $T1$ continues. The first executed instruction is the actual read of `TreeNode@5.value`, which caused the scheduling choice; the dynamic analysis stores the fact that at the instance `TreeNode@5`, its field `value` has been read. First, note that our dynamic analysis distinguishes among object instances. Also note that our implementation stores this information during forward exploration (because JPF does not provide another way to obtain this piece of information).

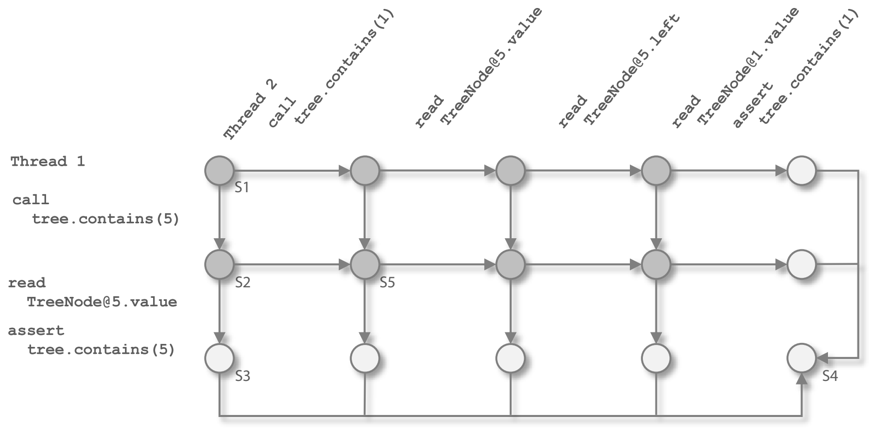


Figure 5. State space of T1 and T2. States with decision choices are in dark grey.

State	`TreeNode@5.value`	`TreeNode@5.left`	`TreeNode@5.right`	`TreeNode@1.value`	`TreeNode@1.left`	`TreeNode@1.right`
$S1$	✓	✓		✓		
$S2$	✓	✓		✓		
$S3$	✓	✓		✓		
$S4$						
$S5$	✓	✓		✓		

Table 1. Live parts of selected states in Fig. 5.

The model checker then continues in forward execution of thread $T1$, it executes the assert statement and thread $T1$ terminates (state $S3$). Since in JPF, a transition has to be executed by a single thread, a new transition is created and, without any non-determinism, the only remaining thread $T2$ is scheduled. The model checker then executes instructions of thread $T2$; among others, the reads of `TreeNode@5.value`, `TreeNode@5.left` and `TreeNode@1.value`. Our analysis stores all these field reads (in the execution order) as in the previous case; the field writes are stored in the same way. All these reads are executed in a single transition, since $T2$ is the only runnable thread; the transition ends in state $S4$, where thread $T2$ terminates.

If the model checker reaches the visited or final state and it starts backtracking, our dynamic analysis uses the stored information to compute live parts of program states. In the example, the model checker has reached a final state and thus dynamic analysis marks all fields in all instances as dead (the program cannot read any of them). The model checker then backtracks from $S4$ to $S3$; during backtracking the dynamic analysis processes the stored information for backtracked transition in the reverse order. Thus due to field reads, the analysis will mark `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value` as live at the beginning of the transition $S3 \rightarrow S4$, while all other data remain dead. This includes, if the right-hand side tree is stored in the `tree` variable, the instances (and their fields) `TreeNode@7`, `TreeNode@8`, and `TreeNode@9` as well as the colors of all `TreeNode` instances. Since there is only a single outgoing transition from state $S3$, the live addresses of state $S3$ are those at the beginning of the transition $S3 \rightarrow S4$. After the part of the entire state space in Fig. 5 is explored, the live addresses of $S1$ are stored. During exploration of another execution (most notably when the other tree from Fig. 2 is created by the program), the state corresponding to the beginning of $T1$ and $T2$ is matched with the one already explored earlier, because the live part of the stored state matches the corresponding part of the state being explored. If no DVR had been performed, the corresponding part of the state space (having the same shape but differing in the `tree.right` field) would have had to be explored.

State matching. Once the live resp. dead addresses are known at a state, dynamic analysis computes a reduced state vector and stores the pair consisting of the set of live addresses and the reduced state vector in the set of visited states. The reduced state vector contains only live values from the static and instance fields, and call-stacks of the threads. Since local variables are not considered by our analysis (i.e., none of them is removed), the complete call-stacks (without any reduction) are stored. In order to speed up state matching, the same optimization as in [SM07] is applied – for each call-stack, we store a list of live addresses used to compute the reduced state vectors corresponding to this call-stack.

The reduced state vectors enable to detect those states which differ in dead parts of the heap only. In our example, if the model checker backtracks over state $S1$, where the example begins, and generates another state with the tree from the right-hand side of Fig. 3, our state matching procedure will match this state with $S1$.

If the model checker reaches a state, it has to check whether the state (or an equivalent one) has been already visited or not, so the state matching is initiated; the model checker uses the call-stack of the current state to find a list of live addresses associated with the call-stack. For each set of live addresses from the list, a reduced state vector of the current state is computed and it is checked whether the pair of live addresses and the reduced state vector is among the visited ones. If so, the state is reported as visited.

3.2. Hybrid analysis

In contrast to our DDVA, where we addressed precision in the case of multi-threaded programs, our hybrid analysis targets scalability and speed, instead.

It is relatively straight-forward to apply DVA on local variables; it can be done e.g., using intra-procedural static analysis. The approach scales well due to a usually small sizes of function (or method) bodies. Moreover, since local variables are thread-local, it naturally works in the case of multi-threaded programs as well. The situation becomes more challenging in the case of global variables (and static fields), since these can be accessed by several threads and their life-time is not limited to a single method only.

Our hybrid analysis consists of two phases. The first phase is executed before the model checker runs. Cheap simple inter-procedural, context-insensitive static analysis is performed, identifying the fields that may be read (i.e., *used* in the terminology of DVA) by a given method before it terminates (up to the end of its body). The analysis gets only a partial view on the liveness; on the other hand, it is cheap to compute. In other words, this phase provides an under-approximation (i.e., a subset) of the live fields at a state. Additional pieces of information are obtained in the second phase, performed at model-checker runtime just before each state matching. Note that the hybrid analysis does not distinguish among object instances; a field is live either at all instances or at none. Therefore, field writes (i.e., *def*s) are not tracked.

In the second phase, the analysis uses information from the call-stacks of the threads; i.e., the locations (instruction pointers) in the methods at the call-stacks. It traverses the call-stacks in the top-down manner and extends the scope for which the liveness is considered. Using the location in the current method (top stack frame), the analysis uses the results from the first phase to obtain fields read before the current method terminates; these fields will be marked as live. Going down the call-stack, the analysis adds the live fields from the caller using the results from the first analysis computed for the location right after the call. This way, we obtain the fields which can be read by a thread before the considered method terminates. Once the whole call-stack is processed, we know which fields the thread may read before it terminates. The above processing of the call-stack is done for each thread and the results are joined together, thus obtaining the final result – the fields which can be read at a given program state. These fields are marked as live and only these are considered in state matching.

Note that even though the second phase is executed many times (before each state matching), it is extremely fast and thus introduces only a minimal overhead.

Using this approach, we obtain more precise results compared to purely static DVA. In particular, the information from call-stack determines the methods that are called; contrary to purely static DVA, which has to consider all possibly called methods and reads in them. Also, the analysis does not have to construct the possibly huge cross-product of thread locations, as it is (typically) done in flow-sensitive analyses of multi-threaded programs.

In state matching, the information is used in a straight-forward way. If the analysis identifies a field as possibly live, the value of the field is included into the state vector, otherwise the value is ignored and not stored. Note that the results of the hybrid analysis intentionally do not distinguish particular instances more precisely (e.g., using allocation sites). While the analysis itself can use a better pointer analysis (e.g., to construct the call graph) to distinguish among instances of the same type, a slower and more complicated state matching algorithm, similar to the one used in dynamic analysis, would be needed to preserve safety in such a case.

4. Formalization

In this section, we first introduce parallel heap-manipulating programs as a parallel composition of guarded-action transition systems and their state space (using the typical small-step-big-step approach). Later, we formally describe our dynamic DVR and show its correctness. In contrast to static DVAs, which identify variables which are dead at a location, our dynamic analysis identifies the addresses on the heap which contain a dead value at a program state.

The memory (*Mem*) is modelled as a function taking an address and returning the value stored at this address. As usual, writing to the memory produces a new function Mem' being equivalent to the original one for all but the written address.

Syntax. A *program* $P = (T_1, \ldots, T_n)$ is a tuple of concurrently executed threads $T_1, \ldots, T_n$ operating over shared memory. For the sake of presentation simplicity, we omit dynamic thread creation in the formalization, since it can be added in a straight-forward way not affecting DVA.

Each *thread* can be seen as a labeled transition system $T_i = (L_i, Tr_i \subseteq L_i \times Guard \times Act \times L_i, l_i^{init} \in L_i)$, where L_i is a set of locations, l_i^{init} is the initial location, and Tr_i a transition relation (can be seen as CFG) with transitions labeled by *Guard*ed *Act*ions. We refer to edges in Tr_i as to *steps*; the steps are executed atomically. Guards and actions can be arbitrary, provided that they satisfy the following conditions (later in this section we show how to extract the information important to DVA): (α) The guards are assumed to be side-effect-free (i.e., they do not modify the memory), and (β) the actions and guards are deterministic (i.e., they act as functions; the same input values result in the same output). Data non-determinism (e.g., user input) is modelled by branching – one branch for each possible user input. The actions encode statements of the source code. They typically represent direct or indirect writes into memory. Without loss of generality, an action can include at most one write into the memory ending up the action; more complex constructs of programming languages are split into several actions. Guards are intended to express conditions and synchronization primitives from the source code. The memory is shared by all the threads; it holds all the thread-local as well as global (shared) data and states of locks (locked/unlocked).

We choose this representation, since we believe that it is best suited for multi-threaded programs. In contrast to *while* language [YG04] in MURPHI, our memory is shared among all threads, thus our approach permits complex interactions involving multiple threads over shared object instances which is common in multi-threaded code. Our formalization does not include primitives for message passing and queues as in the IF model checker [FBG03] (unbounded FIFO queues) and PROMELA [Hol04] (bounded blocking and lossy channels). The motivation for this decision was that queues are usually not an inherent part of programming languages. They are typically included in the language libraries with various semantics – in the form of priority queues, LIFOs, FIFOs, etc.

Semantics. A *program state* $S = (IP, Mem)$ is a pair consisting of function IP (instruction pointers), which for each thread i returns the current thread location $l_i \in L_i$, and function *Mem* representing the content of the memory. In the rest of the chapter, IP_S and Mem_S refer to the corresponding components of state S, respectively. We use $\mathcal{S}$ to denote the set of all possible program states.

For a function f, we write $f[d := val]$ to denote the function which is equivalent to original f for all inputs but d, for which it returns value *val*; we use this syntax to denote execution of a step in a particular thread. We write $Mem[A]$ to denote original memory *Mem*, whose content is modified by the write of action A.

Definition 1 (State space). The state space (i.e., behaviour) of a concurrent program P is the transition system $A_P = (\mathcal{S}, \triangle \subseteq \mathcal{S} \times Guard \times Act \times \mathcal{S}, S^{init} \in \mathcal{S})$ over the program states $\mathcal{S}$, where the initial state S^{init} consists of the initial states of all the threads and the initial state of the memory: $S^{init} \equiv (IP(t) = l_t^{init}\ \forall t \in 1...n, Mem^{init})$.

There is a *transition* $(S, G, A, S') \in \triangle$, where $S \equiv (IP_S, Mem_S)$ and $S' \equiv (IP_{S'}, Mem_{S'})$ iff there exists a thread t and its step $(l_t, G, A, l'_t) \in L_t$ such that:

- $IP_S(t) = l_t$ and $IP_{S'} = IP_S[t := l'_t]$, and

- guard G holds in state S, and
- $Mem_{S'} = Mem_S[A]$ (i.e., action A transforms Mem_S into $Mem_{S'}$).

According to the definition, the guards for all transitions in $\triangle$ hold, hence a transition cannot be blocked by its guard. We include guards into the transitions, because they read the memory, thus are important for DVA. The state space (i.e., $\triangle$) is non-deterministic; two different threads may, at a state, execute the same action guarded by the same guard.

Definition 2 (Transition equivalence). Let $p \equiv S_1 \xrightarrow{G_1,A_1} \cdots$ and $p' \equiv S'_1 \xrightarrow{G'_1,A'_1} \cdots$ be (finite or infinite) traces. We say that p and p' are *transition-equivalent* if they consist of the same sequence of actions and guards: $\forall i : G_i = G'_i, A_i = A'_i$.

Since actions are deterministic, the content of memory in all states along a trace is determined by the initial state of memory and by the actions on the trace. The sets P and P' of traces are equivalent if $\forall p \in P$ there exists a transition-equivalent trace $p' \in P'$ and vice versa. We use $\mathcal{P}$ to denote the set of all traces in the transition system A_P.

Let $read\_addr : \triangle \rightarrow 2^{dom(Mem)}$ be a function, which for a given transition $(S, G, A, S') \in \triangle$ returns the set of memory addresses that are read from the memory by either guard G or action A or both. Similarly, function $write\_addr : \triangle \rightarrow 2^{dom(Mem)}$ for a given transition returns the set of memory addresses to which action A of the transition writes. Recall that guards do not modify the memory thus are not considered by this function. Moreover, from the definition of the action, it follows that the set of written addresses is either empty or it contains a single address; without this restriction, an inter-transition data-flow analysis would be needed if multiple writes occur in a transition to obtain results with the same precision.

In our formalization of the program, we do not exactly specify the form of guards and actions. Instead, we use *read_addr* and *write_addr* to extract their behaviour relevant to DVA (i.e., the read and written addresses). Also note that these functions do not take the guard or action itself, but the whole transition; the purpose is to precisely model indirect memory accesses and arrays, where the address which is read or written depends on the content of the memory (in the initial state of the transition).

Let f be a function and $S \subseteq dom(f)$ be a set. We write $f\lceil_S$ to denote the equivalent partial function with domain restricted to S. We use partial functions to represent program states omitting dead values.

Live addresses. The *read/write_addr* functions can be easily generalized for traces. In a similar way, one can define function *tr_live_addr* which returns the set of addresses that are read before they are written on a given trace, i.e., the set of live variables for the given trace.

Definition 3 (Live trace addresses). Let $p \equiv S_1 \xrightarrow{G_1,A_1} \cdots$ be a trace. Function $tr\_live\_addr : \mathcal{P} \rightarrow 2^{dom(Mem)}$ is defined as:

$$tr\_live\_addr(p) = \{a | \exists i : a \in read\_addr((S_i, G_i, A_i, S_{i+1})) \wedge \forall j < i : a \notin write\_addr((S_j, G_j, A_j, S_{j+1}))\}$$

Note that in contrast to Static DVA, the analysis operates on real program states and state space instead of program locations in CFG. The values stored at live addresses at state S_1 (initial state of the trace) fully determine the behaviour of the trace, i.e., the outcome of the actions on the trace as well as fulfilment of the guards. The *tr_live_addr* function points to the memory addresses whose values are important to follow the trace, and fully determines the computations done (results of the actions) along the trace. If a state S'_1 differs from S_1 in values at dead addresses only (i.e., S_1 and S'_1 equals on $tr\_live\_addr(p)$), then there exists a trace p' from S'_1 being transition-equivalent to p. Moreover, the corresponding states along these equivalent traces differ only in the memory content which is dead w.r.t. the suffices of these traces. Note that in general, two transition-equivalent traces can have different live addresses, however if the initial states of the traces differ at dead addresses only, then $tr\_live\_addr(p) = tr\_live\_addr(p')$.

In the verification, one is typically not interested in a single possible future behaviour starting at a state (i.e., a single trace), but rather in all possible future behaviours (i.e., all traces). To reflect this, we extend *tr_live_addr*. We say that an address is live at a state if there is a trace from the state at which the address is live:

Definition 4. Let a be an address and $S \in \mathcal{S}$ be a program state. Function $live\_addr : \mathcal{S} \rightarrow 2^{dom(Mem)}$ is

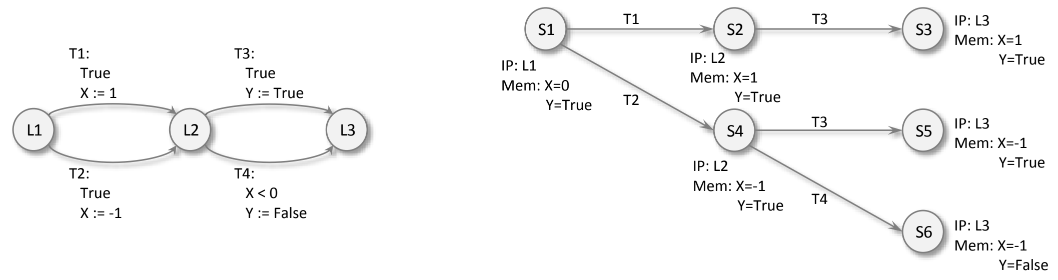


Figure 6. Thread specification (left) and its state space (right)

defined as follows:

$$live\_addr(S) = \{a | \exists \text{ trace } p = S \xrightarrow{*} \cdots \text{ such that } a \in tr\_live\_addr(p)\}$$

The *live_addr* function refers to the memory addresses on whose values the future behaviour of the program depends (once program reaches the given state). Similarly to the trace-based counterpart, if states S and S' equal w.r.t. $live\_addr(S)$ (and of course refer to the same program point IP), then for any trace p from S there is a transition-equivalent trace from S' (i.e., the future behaviours of S is a subset of the future behaviours of S').

In the context of DVA, if two program states equal on their live addresses, their future behaviour is assumed to be the same; informally stated, the sets of states reachable from either one are the same w.r.t. their live addresses. This is generally not true for Guarded-Action LTS, and also for programs, this needs to be proved (e.g., thread synchronization is to be considered).

Consider a single-threaded program $P = (T)$, where thread T is shown in Figure 6; it consists of three locations $L1 - L3$ and four steps $T1 - T4$. All steps except for $T4$ contain guard $True$, step $T4$ contains guard $X < 0$. The actions are assignments to (global) variables X and Y.

The corresponding state space is depicted in Fig. 6 on the right-hand side; it consists of six states $S1 - S6$ and five transitions. Note that for the sake of readability, we used variable names instead of addresses in the example; while this is doable for global variables, it is not suitable for heap instances.

To illustrate that the claim above does not hold for LTSs in general, let us focus on states $S2$ and $S4$. State $S2$ has an empty set of live addresses, since only the assignment $Y := True$ of step $T3$ is executed before the final state is reached; thus only location IP is considered in state matching. State $S4$ has the same location as $S2$, and its set of live addresses equals to that of $S2$. However, the future behaviour of these states is different – from $S4$ there is an additional trace taking step $T4$ into $S6$.

First, note that problem is caused by unsatisfied guards; guard $X < 0$ does not hold in $S2$, thus there is no transition which could make the variable X live. Later in the subsection about *guard restrictions*, we focus on the problem in more detail.

Also note that the claim below the *live_addr* definition holds; from state $S2$ there is a trace $S2 \xrightarrow{T3} S3$. From state $S4$ there is a transition-equivalent trace $S4 \xrightarrow{T3} S5$.

Safety properties. Out of the transition-equivalent traces p and p' one may end up in an error state while the other in a safe state. This is because the values on which the decision whether the state is safe or not depends are ignored. For example, consider Figure 6; transition-equivalent traces $S2 \xrightarrow{T3} S3$ and $S4 \xrightarrow{T3} S5$ and the safety property $(IP = L3 \wedge X > 0)$. While the first trace leads to the safe state $S3$, the second one leads to the error state $S5$.

Let ϕ be (an externally specified) safety property for program P. We use $addr(\phi)$ to denote all the memory addresses which ϕ reads (i.e., the addresses of variables ϕ contains). To be able to distinguish between safe and error states, the addresses $addr(\phi)$ have to be a (live) part of each program state (even in cases the address is not read on any trace). Note that for many common types of program errors, such as assertion violation, the $addr(\phi) = \emptyset$, since the property is encoded in the program itself.

Live parts of state. The aforementioned definitions allow us to introduce a function which omits dead (i.e., irrelevant) parts of program states. For a full state T we define a function $\mathit{reduce\_state}_T$ which reduces full states w.r.t. the future behaviour of the state T.

Definition 5 (Reduced states). Let $S \equiv (IP_S, Mem_S)$ be a full state. The $\mathit{reduce\_state}_T(S)$ function is defined as follows:

$$\mathit{reduce\_state}_T(S) = (IP_S, Mem_S\restriction_{\mathit{live\_addr}(T) \cup \mathit{addr}(\phi)})$$

The $\mathit{reduce\_state}_*$ functions eliminates dead addresses from the domain of memory function Mem_S, while instruction pointers IP_S are not modified.

State matching. An optimal state matching algorithm matches (currently reached) state S with previously (fully) explored state S' if the future behaviours of S are a subset of behaviours from S'. In other words, the states should be matched if (i) they both contain the same value of the instruction pointer and (ii) equal on the data stored at the live addresses of S; formally if $\mathit{reduce\_state}_S(S') = \mathit{reduce\_state}_S(S)$.

This is however hard to implement efficiently, especially if just hashes of the visited states are stored. At the time when state S' was reached, it was not known which states visited later on would be considered in the state matching with S', thus the $\mathit{reduce\_state}_S$ was unknown at that time. And symmetrically at the time when S is reached, from the previously visited state S' only the hash value (of the full state) is known and there is no easy way to get the hash value considering only the live parts of S', i.e., to obtain $\mathit{reduce\_state}_S(S')$.

Alternatively, it is possible to omit the checking for subsets and match states with exactly the same future behaviour; formally expressed $\mathit{reduce\_state}_S(S) = \mathit{reduce\_state}_{S'}(S')$. This check is easier to implement, but the states whose future behaviour is a proper subset of the other are considered different due to different domains of their *Mem* functions. This matching approach should be comparable to the optimal one since for programs, a state does not exhibit a subset of the behaviours (i.e., traces) of another one; either the sets of traces are the same or completely different.

Note that even this alternative way of state matching still needs to precisely know the future behaviour of the currently reached state S. To detect whether state S has been already visited or not, it is not a good idea first to explore the successors of S and then to compute the live addresses from the observed behaviour; such an approach would eliminate one of the benefits of DVA – reduction of the state space. In our hybrid analysis an over-approximation of the future behaviour origins from the static phase; so there is no need to explore the successors. Our dynamic analysis utilizes the observed behaviour to compute live addresses, however this is done only for states which are *known to be new* (i.e., visited for the first time). During state matching, the approach uses the live addresses of previously visited states which are tested against those of S; this is possible due to Theorem 2 below.

Reduced state space. As stated above, for state matching it is possible to consider states w.r.t. their own future behaviour. For a full state $S \equiv (IP_S, Mem_S) \in \mathcal{S}$, we define a *reduced state* R as $R \equiv \mathit{reduce\_state}_S(S)$. We use *reduce_state* without subscript to denote reduction of the state w.r.t. its own future behaviour. The symbol $\mathcal{R}$ is used to denote the set of all reduced states. Upon the reduced states, it is possible to build a state space in a similar way as for full states.

Definition 6 (Reduced state space). Let P be a program with the state space $A_P = (\mathcal{S}, \triangle, S^{init})$. The *reduced state space* R_P *of* P is tuple $R_P = (\mathcal{R}, \triangle_R \subseteq \mathcal{R} \times \mathit{Guard} \times \mathit{Act} \times \mathcal{R}, R^{init})$, where $R^{init} = \mathit{reduce\_state}_{S^{init}}(S^{init})$.

There is a transition $(R, G, A, R') \in \triangle_R$ iff there exists a full state transition $(S, G, A, S') \in \triangle$ such that $\mathit{reduce\_state}_S(S) = R$ and $\mathit{reduce\_state}_{S'}(S') = R'$.

We now justify the correctness of this definition. In the reduced state space, it is easy to see that (1) there are no reads of the undefined values from the memory, and (2) there are no undetermined values in the memory. For a transition $t \equiv ((IP_R, Mem_R), G, A, (IP_{R'}, Mem_{R'})) \in \triangle_R$, (1) can be formally expressed as $\mathit{read\_addr}(t) \in \mathit{dom}(Mem_R)$, while (2) can be expressed as $\mathit{dom}(Mem'_R) \subseteq \mathit{dom}(Mem_R) \cup \mathit{write\_addr}(t)$.

As to (1), for any transition $(R, G, A, R') \in \triangle_R$ if some memory address is read by guard G and/or action A, then from the definition of the *read_addr* function (using the trace containing only this single transition) these addresses are preserved in the reduced state. So both G and A read defined values. Note that the potential memory write of A can be performed to an irrelevant address. In such a case the write is ignored, and the state of the memory is unchanged. This happens if the given memory address is dead w.r.t. the property being verified and if the address is not read later in the program.

As to (2), for any transition $(R, G, A, R') \in \triangle_R$, all values (in the memory) of state R' have to be computed from the values in predecessor state R. For any address $a \in \mathit{dom}(\mathit{Mem}_{R'})$, there are three options: either (i) a is read by the property ϕ (i.e., $a \in \mathit{addr}(\phi)$) and thus the address is in the domain of Mem for all reduced states, in particular Mem_R, or (ii) a is written by action A, thus the value is determined in R', or (iii) a is not written by action A. From the definition of the reduced state space it follows that if there is a reduced transition $(R, G, A, R') \in \triangle_R$, then there is a full transition $(S, G, A, S') \in \triangle$ such that $\mathit{reduce\_state}(S) = R$ and $\mathit{reduce\_state}(S') = R'$. Then from the definition of $\mathit{live\_addr}(S')$, there must be a trace p' starting at S' which (reads a and thus) causes a to be live in R'. This trace p' can be prefixed by the considered transition; and $p = S \xrightarrow{G,A} p'$ can be created. Trace p shows that a is live in S and thus $a \in \mathit{dom}(\mathit{Mem}_R)$.

Below, we show that in our case there is a bisimulation between the full and reduced state spaces (as it is typically done for static DVA). To our best knowledge this is the first proof for Dynamic DVA.

Theorem 1 (Bisimulation). For a program P and a property ϕ, transition system $A_P = (\mathcal{S}, \triangle, S^{init})$ and corresponding reduced transition system $R_P = (\mathcal{R}, \triangle_R, R^{init})$ are bisimilar.

Since reduced states and corresponding full states are equivalent w.r.t. ϕ, it directly follows that the state space of the program is safe w.r.t. ϕ iff the reduced state space is safe (and similarly for error traces). So it is enough to check the safety on the reduced state space. Bisimulation also preserves liveness properties, so these can be safely checked just on reduced state space as well [BK08].

Proof sketch. We first sketch the idea of the proof, while a formal proof is presented later. We will show that $\mathit{reduce\_state}_*$ functions define the relation among the states which is a bisimulation. We assume states $S \in \mathcal{S}$ and $R \in \mathcal{R}$ to be in the relation iff $\mathit{reduce\_state}_S\ (S) = R$.

The direction from full states to reduced states follows directly from the definition. The only non-trivial step is the other direction. For a reduced transition $(R, G, A, R') \in \triangle_R$ executed by thread i and related full state S we will show that there is full transition $(S, G, A, S') \in \triangle$ such that full state S' reduces to R'.

From Def. 6 we obtain a *b*ase full transition $(S_b, G, A, S'_b) \in \triangle$. Using this transition we show that there is also a transition $(S, G, A, S') \in \triangle$ since S and S_b equal on all values that G and A use.

We show that S' and S'_b have the same live addresses: Since S and S_b reduce to the same state R, they have (pair-wise) transition-equivalent sets of outgoing traces. Hence, the sets of traces via S' or S'_b are transition-equivalent as well. It directly follows that S' and S'_b have the same live addresses and, in turn, S'_b and S' reduce to the same reduced state (R').

Before proving Theorem 1, we first introduce a lemma which formally expresses the claim about transition-equivalent traces:

Lemma 1. Let $S_1 \equiv (\mathit{IP}_{S_1}, \mathit{Mem}_{S_1}), S'_1 \equiv (\mathit{IP}_{S'_1}, \mathit{Mem}_{S'_1}) \in \mathcal{S}$ be two states and let $\mathit{reduce\_state}_{S_1}(S_1) = \mathit{reduce\_state}_{S_1}(S'_1)$. For any trace $p = S_1 \xrightarrow{A_1,G_1} S_2 \cdots \cdots S_i \xrightarrow{A_i,G_i} S_{i+1} \cdots$ there exists a transition-equivalent trace $p' = S'_1 \xrightarrow{A_1,G_1} S'_2 \cdots$ such that $\mathit{IP}_{S_i} = \mathit{IP}_{S'_i}$ for all i (i.e., the corresponding states along equivalent traces differ only in the memory content).

Proof of Lemma 1: The proof is constructive and inductive on the trace length. The induction will iteratively append an equivalent transition to the end of trace p' and preserve the following inductive invariant: $\mathit{reduce\_state}_{S_i}(S_i) = \mathit{reduce\_state}_{S_i}(S'_i)$ and $\mathit{IP}_{S_i} = \mathit{IP}_{S'_i}$ and there is a trace p'_i which is transition equivalent to the prefix of p of length i.

The base case (traces with no transitions): In such a case, trace p contains just one state: $p = S_1$ (i.e., the trace of length 0 starting at S_1). A transition-equivalent trace $p' = S'_1$ (a trace staring in S'_1 with no transition) exists trivially. Moreover $\mathit{IP}_{S_1} = \mathit{IP}_{S'_1}$, because $\mathit{reduce\_state}_{S_1}(S_1) = \mathit{reduce\_state}_{S_1}(S'_1)$ and the fact that $\mathit{reduce\_state}$ functions do not modify the instruction pointer (IP).

Inductive step: First, for the initial states S_1 and S'_1 the inductive invariant holds. Let the inductive invariant hold for a trace p of length i. If p is of length i, the inductive invariant is exactly the claim of the lemma. So let us assume trace p to be longer. The inductive invariant gives us transition-equivalent trace $p'_i = S'_1 \longrightarrow \cdots S'_i$ such that $\mathit{reduce\_state}_{S_i}(S_i) = \mathit{reduce\_state}_{S_i}(S'_i)$ and $\mathit{IP}_{S_i} = \mathit{IP}_{S'_i}$. Moreover, there is a transition $p = \cdots S_i \xrightarrow{G_i,A_i} S_{i+1} \cdots$ (i.e., (S_i, G_i, A_i, S_{i+1})); we show that p'_i can be extended with this transition.

Let $S'_{i+1} = (IP_{S_{i+1}}, Mem_{S'_i}[A_i])$. First we show that there is a transition $(S'_i, G_i, A_i, S'_{i+1}) \in \triangle$ that can be appended to p'_i to form p'_{i+1}. The first requirement of Def. 1 is satisfied since $IP_{S_i} = IP_{S'_i}$ (from the induction invariant) and $IP_{S_{i+1}} = IP_{S'_{i+1}}$ (from the definition of S'_{i+1}), thus the step showing the existence of the transition (S_i, G_i, A_i, S_{i+1}) satisfies the condition for transition $(S'_i, G_i, A_i, S'_{i+1})$ as well. The second condition (i.e., guard G_i holds in S'_i) is satisfied, because G_i holds in state S_i and S_i and S'_i equal on live addresses ($\mathit{reduce\_state}_{S_i}(S_i) = \mathit{reduce\_state}_{S_i}(S'_i)$ from the induction invariant). Due to trace $p_{G_i} = S_i \xrightarrow{G_i, A_i} S_{i+1}$ (i.e., a trace with a single transition), all the addresses on which G_i depends are live in S_i. The last condition is satisfied directly by the definition of S'_{i+1}.

Now we show that $\mathit{reduce\_state}_{S_{i+1}}(S_{i+1}) = \mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ (i.e., the states S_{i+1} and S'_{i+1} equal on live addresses). From the definition of S'_{i+1} we know that $IP_{S_{i+1}} = IP_{S'_{i+1}}$. Let us look at the memory. In the following, we use $Mem_{R_{i+1}}$ to denote the memory of state $\mathit{reduce\_state}_{S_{i+1}}(S_{i+1})$ (which is exactly the memory of the reduced state for full state S_{i+1}) and $Mem_{R'_{i+1}}$ to denote the memory of state $\mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ (i.e., the memory of states S'_{i+1} reduced w.r.t. future behaviour of S_{i+1}). We show that the above memory functions are the same (i.e., $Mem_{R_{i+1}} = Mem_{R'_{i+1}}$); in other words, these functions have (i) the same domain and (ii) they evaluate to the same values for any address from their domain. As to (i), because the function $\mathit{reduce\_state}_{S_{i+1}}$ is used to create both memory functions, their domains are the same (i.e., $dom(Mem_{R_{i+1}}) = dom(Mem_{R'_{i+1}})$).

Now we show (ii) i.e., $\forall a \in dom(Mem_{R_{i+1}})$ it holds that $Mem_{R_{i+1}}(a) = Mem_{R'_{i+1}}(a)$. Let us pick an address $a \in dom(Mem_{R_{i+1}})$; this address can be either assigned (written) by action A_i or not. In the former case the values equal (i.e., $Mem_{R_{i+1}}(a) = Mem_{R'_{i+1}}(a)$), because in both cases they are computed by the same action A_i using the same values; the trace $p_{G_i} = S_i \xrightarrow{G_i, A_i} S_{i+1}$ shows that all the addresses read by action A_i are live and thus preserved by $\mathit{reduce\_state}_{S_{i+1}}$ and the induction invariant gives us that Mem_{S_i} and $Mem_{S'_i}$ equal on these addresses.

Let us focus on the latter case, i.e., when address a is not written by action A_i and its value is taken from the predecessor. Formally, it holds that $Mem_{R_{i+1}}(a) = Mem_{S_i}(a)$ and $Mem_{R'_{i+1}}(a) = Mem_{S'_i}(a)$. Address a is preserved by the $\mathit{reduce\_state}_{S_{i+1}}$ function, thus either $a \in addr(\phi)$ or $a \in \mathit{live\_addr}(S_{i+1})$. In both cases address a is preserved by $\mathit{reduce\_state}_{S_i}$; in the first case, the reason comes from the definition of $\mathit{reduce\_state}$ – $addr(\phi)$ addresses are always a part of reduced state. In the second case, the definition of $\mathit{reduce\_state}$ gives us a trace p'_a from state S_{i+1} which demonstrates that address a is live. Let trace p_a be p'_a prefixed by (S_i, G_i, A_i, S_{i+1}) (i.e. $p_a \equiv S_i \xrightarrow{G_i, A_i} p'_a$), because A_i does not write to address a, the trace p_a shows that the address a is live in S_i and thus preserved by $\mathit{reduce\_state}_{S_i}$. We know that $Mem_{R_{i+1}}(a) = Mem_{S_i}(a)$ and $Mem_{R'_{i+1}}(a) = Mem_{S'_i}(a)$, and the inductive invariant gives us that $Mem_{S_i}(a) = Mem_{S'_i}(a)$; thus we have shown that $Mem_{R_{i+1}}(a) = Mem_{R'_{i+1}}(a)$.

Put together, we have proved that $\mathit{reduce\_state}_{S_{i+1}}(S_{i+1}) = \mathit{reduce\_state}_{S_{i+1}}(S'_{i+1})$ – the inductive invariant for $i + 1$. $\square$

Proof of Theorem 1 – Bisimulation: Let $\mathit{reduce\_state}$ be a relation among the full and reduced states, such that full state S and reduced state R are in the relation iff $\mathit{reduce\_state}_S(S) = R$. We show that $\mathit{reduce\_state}$ is a bisimulation, i.e., $\mathit{reduce\_state}$ satisfies the following three conditions:

(1) The initial states S^{init} and R^{init} are in the relation (which comes directly from the definition).

(2) Let there is a full transition $(S, G, A, S') \in \triangle$, and let state R be in the relation with S, i.e., $R = \mathit{reduce\_state}_S(S)$. Then there has to be a reduced transition $(R, G, A, R') \in \triangle_R$ such that $R' = \mathit{reduce\_state}_{S'}(S')$. This comes also trivially from the definition of the reduced state space.

(3) Let there is a reduced transition $(R, G, A, R') \in \triangle_R$ executed by a thread i. Then for any full state S such that $\mathit{reduce\_state}_S(S) = R$ (a pre-image of reduced state R), there exists a full transition $(S, G, A, S') \in \triangle$ such that $\mathit{reduce\_state}_{S'}(S') = R'$.

The only condition remaining to be proved is (3). From the definition of reduced state there exists a *b*ase full transition $(S_b, G, A, S'_b) \in \triangle$ due to which the reduced transition (R, G, A, R') exists. From the definition of $\mathit{reduce\_state}$ it follows that it does not modify the instruction pointer, thus $IP_S = IP_{S_b} = IP_R$ (i.e., the corresponding states S, S_b, and R point to the same program location) and $IP_{S'_b} = IP_{R'}$ (i.e., the base transition is executed by the same thread i as the reduced transition). Moreover the trace $p_b \equiv S_b \xrightarrow{G, A} S'_b$ (i.e., the trace of length 1 consisting exactly of the base transition) guarantees that all the addresses that guard G and action A read are live and thus preserved by $\mathit{reduce\_state}_{S_b}$. States S and S_b are both reduced to state R thus, first, $reduce\_state_{S_b} = reduce\_state_S$ (i.e., the *reduce_state* functions for both states are the same and both states have the same live addresses) and second, the values at the live addresses are the same in both states (i.e., $\forall a \in dom(Mem_R) : Mem_S(a) = Mem_{S_b}(a)$).

Def. 1 gives us that (i) there is step $(l_i, A, G, l'_i) \in L_i$ of thread i such that $IP_{S_b}(i) = l_i$ and $IP_{S'_b} = IP_{S_b}[i := l'_i]$ and (ii) guard G holds in S_b.

Let $S' \equiv (IP_{S'_b}, Mem_S[A])$ be a full state, i.e., it is created from state S by a step of thread i into the same program location as the base (and reduced) transition by executing action A. In order to prove the theorem, we need to show that there is a transition $(S, G, A, S') \in \triangle$ and that S' reduces into R'.

According to Def. 1, there exists transition $(S, G, A, S') \in \triangle$, because $IP_S = IP_{S_b}$, $IP_{S'} = IP_{S'_b}$ so (i) satisfies the first requirement of Def. 1 and guard G holds in S (due to (ii) and because S_b and S equal on all the values that guard G reads). The last requirement of Def. 1 follows directly from the definition of state S'.

We now show that S' reduces to R' (i.e., $reduce\_state_{S'}(S') = R'$); first we show that (iii) states S' and S'_b have the same live addresses (i.e., $reduce\_state_{S'} = reduce\_state_{S'_b}$, in other words the functions are the same) and then that (iv) these states have the same values at the live addresses.

Before moving to (iii), we first show that (v) S' and S'_b have (pair-wise) transition-equivalent sets of traces (using Lemma 1). Let $p_{S'}$ be a trace from state S' (i.e., $p_{S'} = S' \xrightarrow{*} \cdots$). Then there is also a trace p which is formed by $p_{S'}$ prefixed by transition (S, G, A, S') (i.e., $p \equiv S \xrightarrow{G,A} S' \xrightarrow{*} \cdots$). Then $reduce\_state_S(S) = reduce\_state_S(S_b) = R$, because $reduce\_state_S(S) = R$ (from the assumption of the condition (3)), $reduce\_state_S = reduce\_state_{S_b}$, which was shown above, and $reduce\_state_{S_b}(S_b) = R$ (from definition of S_b). Lemma 1 can now be applied on states S and S_b and trace p resulting in a transition-equivalent trace $p' = S_b \xrightarrow{G,A} S_x \xrightarrow{*} \cdots$ such that $IP_x = IP_{S'}$. Now we show that $S_x = S'_b$; the actions are deterministic, so $Mem_x = Mem_{S'_b}$ (more rigorously by the definitions, both equal to $Mem_{S_b}[A]$). Since $IP_x = IP_{S'_b}$ (via $IP_{S'}$), we have shown that the states S_x and S'_b are the same (i.e., $S_x = S_b$). Thus we can remove the first transition of p' to obtain $p_{S'_b}$ from S'_b, which is transition equivalent to $p_{S'}$. Exactly the same reasoning can be used to show that for any trace from S'_b there is a transition-equivalent trace from S'.

As to (iii), it remains to show that functions $reduce\_state_{S'}$ and $reduce\_state_{S'_b}$ are the same. From the definition of the *reduce_state* functions it follows that they modify only the *Mem* part of the program state by restricting its domain. Assume that $reduce\_state_{S'}$ preserves an address $a \in dom(Mem_{S'})$; then it holds either $a \in addr(\phi)$ or $a \in live\_addr(S')$. In the former case, the address a is also preserved by $reduce\_state_{S'_b}$ from the same reason (the safety property ϕ is the same for the whole program). In the latter case, we use the definition of *live_addr*, which gives us a trace $p_{S'}$ – a witness why a is a live address (i.e., $a \in tr\_live\_addr(p_{S'})$). Above we have shown that there is a trace $p_{S'_b}$ transition equivalent to $p_{S'}$ (see (v)). Trace $p_{S'_b}$ (having the same live addresses as $p_{S'}$, since the traces equal on *tr_live_addr* in their initial states) is a witness that address a is live in S'_b (i.e., $a \in LA(S'_b)$). Thus a is also preserved by $reduce\_state_{S'_b}$. The same reasoning can be applied to show that all addresses preserved by $reduce\_state_{S'_b}$ are also preserved by $reduce\_state_{S'}$, which together gives us that $reduce\_state_{S'} = reduce\_state_{S'_b}$.

The only remaining part is (iv). $IP_{S'}$ and $IP_{R'}$ equal, because we know that $IP_{R'} = IP_{S'_b}$ and $IP_{S'} = IP_{S'_b}$ from the definition. Let us focus on the memory parts; we show that for any address a preserved by $reduce\_state_{S'}$, the states S' and S'_b have the same value (i.e., $Mem_{S'}(a) = Mem_{S'_b}(a)$).

Address a can be either written by action A (taken from the transition (S, G, A, S')) or not. In the former case, the value written by A is the same in $Mem_{S'}$ and $Mem_{S'_b}$, because the outcome of the actions is deterministic and the values used by action A to compute its result are the same (S and S_b both reduce into R thus equal on live addresses and due to the single transition trace $\xrightarrow{G,A}$, the addresses read by A are live in these states). Let us focus on the latter case in which address a is not modified by A and thus the values (in states S' and S'_b) are the same as in their predecessor (in states S and S_b respectively); formally $Mem_{S'}(a) = Mem_S(a)$ and $Mem_{S'_b}(a) = Mem_{S_b}(a)$. The predecessors have the same value at this address (i.e., $Mem_S(a) = Mem_{S_b}(a)$); because S and S_b both reduce into R and thus equal on live addresses and a is live in both S and S_b. Address a is live in S, because it is live address in S'; so there exists a trace p' – witness that the address is live. The trace $p = S \xrightarrow{G,A} p'$ (i.e., prefixed by transition (S, G, A, S')) shows that the address is also live in S as we need.

We have shown (iii) and thus $reduce\_state(S') = R$. It means we have created state S' such that there is a transition (S, G, A, S') and *reduce_state*$(S') = R$ so we have shown (3). This proves the bisimulation theorem. $\square$

Guard restriction. While the bisimulation theorem guarantees soundness of the approach, the definition of the *reduce_state* function is not suitable for efficient state matching. On-the-fly explicit state model checkers can get a precise set of live addresses for a state, however they can be computed only *after* the state is fully explored, which is too late to directly reduce the state space. Below, we articulate Lemma 2, which permits us to use *live_addr* (i.e., the *reduce_state*$_*$ function) of one state to safely reduce another (still unexplored) one. The theorem, however, cannot be applied on general guarded-action LTS. Hence, we first introduce a syntax restriction for guards.

Definition 7 (Guard-restriction). Program $P = (T_1, ...T_n)$ satisfies *guard-restriction* if for each thread $T_i = (L_i, Tr_i, l^{init})$ and each step $(l_i, G, A, l_j) \in Tr_i$ such that guard G is non-trivial (i.e., different from $True$) there exists also a step $(l_i, \neg G, A', l_k) \in Tr_i$, (in the same thread t from the same location l_i) with the complementary (negated) guard $\neg G$.

Both guards read the same data and in any case exactly one of the guards is satisfied. The guard-restriction is inspired by the way source code is converted into transition systems. The guards are used to encode conditions (`if-then-else`) and synchronization primitives. The conditions are encoded this way naturally. Synchronization primitives (such as synchronized blocks, thread joins, and wait-notify) can also be easily modelled this way by adding an (active-waiting) self-loop for the blocked state. This means that source code (e.g., Java) yields guard-restricted transition systems.

Let us now focus on the purpose of the guard restriction. Without the guard restriction, if two states C and V equal w.r.t. relevant addresses of V, we know that the future behaviours of C includes all the future behaviours of V. They are in the subset relation, because there can exist a trace from C which cannot be followed from V; it means if the trace is followed from V, the corresponding transition to follow the trace can be missing due to an unsatisfied guard. Note that unsatisfied guards do not contribute to relevant addresses. It means that C and V do not behave equally w.r.t. (unsatisfied) guards. Exactly this case is illustrated in examples from Figure 6, which do not satisfy the guard restrictions. Guard restrictions, on the other hand, eliminate this issue. Consequently, if two states C and V equal w.r.t. relevant addresses of V, they have (pair-wise) transition-equivalent sets of traces and also the same set of relevant addresses. This is formally expressed in the following lemma:

Lemma 2. Let P be a program that satisfies guard restriction, and $A_P = (\mathcal{S}, \triangle, S^{init})$ be its state space. Let $C, V \in \mathcal{S}$ be two states such that *reduce_state*$_V(C) =$ *reduce_state*$_V(V)$. Then *reduce_state*$_C =$ *reduce_state*$_V$.

Proof sketch. We show that for each trace from C a transition-equivalent trace with the same set of *tr_live_addr* from V exists and vice-versa; thus the states have the same set of live addresses and hence the *reduce_state* functions. The direction from V to transition-equivalent trace from C is informally stated just below the definition of (trace) relevant addresses, being exactly the claim of the Lemma 1. As to this part, the guard restriction is not applied here.

The other direction, i.e., for any trace from C, a transition-equivalent trace from V exists, is shown by contradiction. Assume that p is the shortest trace from C, such that there is no transition-equivalent trace from state V and that states C and V satisfy the requirements of the lemma. Since $p \equiv C \xrightarrow{G,A} C'$ is the shortest such a trace, it follows that the first transition cannot be followed from V. In the opposite case we could have moved along the common prefix of traces p and p' (thus finding shorter traces from states C_i and V_i).

Now we exploit the guard restriction to show that trace p cannot exist. Guard G cannot be trivial (in such a case transition $\xrightarrow{G,A}$ would exist from V). Moreover, due to the guard restriction, there is a transition $V \xrightarrow{\neg G,A'} V'$ thus all the addresses that $\neg G$ (as well as G) reads are live in V. Because C and V equal w.r.t. the relevant addresses of V (formally expressed as *reduce_state*$_V(C) =$ *reduce_state*$_V(V)$), and all the addresses that guard G reads are relevant, it follows that in C guard G is not satisfied (in fact its negation holds). Thus there is no transition $C \xrightarrow{G,A} C'$ and trace p does not exist. In turn, in the whole state space, there is no contradicting p (i.e., a trace without a transition-equivalent counterpart).

5. Implementation

We have developed two independent DVAs to identify live fields of the heap instances and implemented them in Java PathFinder (JPF) [VHB+03]. The analyses differ in precision, computation complexity as well as in the way live addresses are obtained. To make our approach clear, we present the algorithms in pseudo-code below. To emphasize our extensions to classical model checking, we present the standard algorithm we build upon in Alg. 1.

Algorithm 1 Common DFS model checking algorithm

```
1:  procedure Main
2:      ModelCheck(P, $s\_init$, [$s\_init$])

3:  procedure ModelCheck(program $P$, state $s$, trace $t$)
4:      if IsErrorState($s$) then throw Unsafe($s, t$)
5:      if Visited($s$) then return VISITED
6:      SetVisited($s$)
7:      for transition $alfa \in$ EnabledTransitionsIn($s$) do
8:          $s\_succ \leftarrow alfa(s)$
9:          ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
10:     return SAFE
```

Our Dynamic DVA (DDVA) tracks field reads and writes executed by the program. The live addresses for a given program state are computed once all the future behaviours of the program state are fully explored. Thus our DDVA can be used only with the default Depth-First Search (DFS) with no test-like heuristics (those traversing only a part of the state space). The analysis is exact for loop-less state space, it marks particular field as live if and only if there exists a real trace from that state on which the value of the field is read (before it is overwritten). On the other hand, the analysis requires additional memory and has a bigger computational costs compared to our Hybrid DVA. DDVA has to store the observed live addresses (i.e., *reduce_state* function) together with the program state (i.e., its state vector or its hash).

Our Hybrid DVA combines static analysis and dynamic information from program states; hence its name. Static analysis provides an over-approximation of future program behaviour (in terms of field reads), thus the information about live addresses is computed once the state is reached (before its successors will be explored). So the analysis can be safely used together with various test-like heuristics and state space exploration strategies. It is less precise than our DDVA, however, it offers very low computational and memory overhead; only small amount of memory is required to store the results of static analysis.

In our work, we aim at identifying irrelevant content stored in the heap. Note that we disregard irrelevant data stored in the local variables, which is a subject of other analyses that can be combined with ours.

Heap of Java programs. Heap is typically formally modelled as an array of values where references (pointers) are indexes into the array. While this straightforward approach is generic and can express any usage of the heap, it is too low-level for object-oriented languages. JPF (as well as other explicit-state model checkers) typically represent heap in a more complex way. This helps to better express the semantics, and it makes various optimizations (e.g., heap canonicalization) easier. It leads us not to represent the DVA related information as addresses, but to follow a higher-level abstraction used in model checkers. It also provides us with an easier integration into state matching. Below, we describe how the elements of heap are represented; such a representation is directly used in our DDVA. Our Hybrid DVA uses a simpler representation, which is described later.

In object-oriented languages, the objects are allocated on the heap. In Java, the objects are of two types – arrays and instances of classes. Instead of addresses for referring to objects, we use pairs consisting of a unique object ID (i.e., the representation of an address) and a field name (its index since the name may not be unique) for class instances.

The class instances do not contain only fields, but also a type. The type of each instance is stored in its headers. The type can be accessed via introspection (reflection), by the `instanceof` Java operator, and indirectly via virtual method calls. The instance cannot be marked as dead (i.e., completely ignored by state matching), even if no field is read from the instance, if a virtual method is called on the instance; its type is live, since if another instance with a different type had been used instead of the original one, the program would behave differently; it will execute a different (virtual) method. Thus, for each instance (i.e., unique instance ID), the analysis stores whether its type is live or not.

In Java, monitors are other properties of instances; each instance has associated a monitor (i.e., a lock), upon which the threads can synchronize. The state of monitors (locked, unlocked, notified, etc.) is of course an important part of the program state and thus has to be considered by state matching. The monitors realize the `synchronized` blocks and methods; each monitor needs to be accessed at the beginning and at the end of the corresponding synchronized section, and so each monitor is live. That is why our analysis assumes all monitors to be live.

Arrays are other objects stored on the heap. They are handled in a similar way as class instances (since, in Java, arrays are instances as well, but with no fields and methods defined by user), however, instead of field names, we store which particular indices in the array are live – we store pairs consisting of a unique object ID and an index. The arrays are more difficult to handle than normal instances, since they can vary in length. Instances of a single class have a fixed set of fields and thus a fixed size. For arrays, we also store whether their length has been accessed, in other words whether the length attribute is live or dead. In rare cases, when an array is not accessed at all, but still the instance is important (e.g., it is compared to another reference), we even omit the size of the array in state matching. Of course, if any index of the array is live, then the array length is live as well; if the length of corresponding arrays differs, an attempt to access an index may result in the `IndexOutOfBoundsException` only in one state with the shorter array. It means that we also have to store the type of each array as for a normal class instance (the length of the array is a part of the array type).

Our analysis also handles static fields. In JPF, static data are stored in a (special) heap, where each class has its own instance. Because of that, we handle the static fields in the same way as object instances; for each such a field, we store a pair consisting of a unique class identifier and a field name for identification. Note that a particular class can be loaded into JVM several times (each time with a different defining classloader) and each such loading results in a different and incompatible type, featuring its own static fields (sharing the names). Even though these classes have the same name, we support this in our approach, since JPF assigns them different identifiers.

Analysis of JVM bytecode. Above, we focused on specific properties of the heap. We need to track not only addresses, but also, e.g., types and array lengths.

In the following paragraphs, we move from LTS to JVM bytecode. Our analyses, as described, operate over LTS, however, the program is not executed as guarded-action LTS. Java programs are represented by JVM bytecode, which is directly executed by the JPF model checker and observed for purposes of our analyses. The mapping from LTS to bytecode is quite straightforward; however, guards need special attention. Moreover, our DDVA requires a guard-restricted transition system; so below (1) we focus on guards and show that JVM bytecode can express only guard-restricted transition systems. Later, we show (2) how to obtain *read_addr* and *write_addr* resp. theirs counterparts for programs with heap.

In LTS, evaluation of guards influences which transitions the program may take; if LTS encodes a source code, then guards are used to express conditions and synchronization primitives. In JVM, there are no guards. Instead, bytecode (assemblers) have *compare instructions* to evaluate conditions, and *conditional jump* instructions, to take a particular branch (i.e., transition). In particular, JVM bytecode contains the compare `[D/F]CMP[G/F]` instructions, and conditional jumps specialized for a specific condition (e.g., the `IF_ICMPLE` instruction which compares two top integer values on the stack and jumps if the one-but-top value is less than or equal to the top one).

Let us focus on the guard restriction. The restriction has been introduced to ensure that the reduced states behave in the same way w.r.t. both satisfied and unsatisfied guards; in other words, the restriction makes the reads of unsatisfied guards visible. Informally, in order to enable JVM to decide whether a given transition can be taken, it first has to evaluate the condition (i.e., the guard); thus, the reads of (satisfied as well as unsatisfied) guards are visible. Therefore, for JVM bytecode, we can assume that it satisfies the guard restriction.

Let us now focus on the conditions and synchronization primitives in more detail. In case of conditions (e.g., `if-then-else` and loops), first, the condition (i.e., the guard) is evaluated (so its reads are visible) and based on the result, either the `then` or the `else` branch (i.e., the transition) is taken. In case of synchronization

a bytecode instruction	b Object instances Fields	c Type	d Index	e Arrays Length	f Type	g Static Fields
`getfield`	✓					
`putfield`	✓					
`getstatic`						✓
`putstatic`						✓
`checkcast`		✓			✓	
`instanceof`		✓			✓	
`[a/b/c/d/f/i/l/s]aload`			✓	✓		
`[a/b/c/d/f/i/l/s]astore`			✓	✓		
`arraylength`				✓		
`invokeinterface`		✓				
`invokevirtual`		✓				
`athrow`		✓				
`monitor[enter/exit]`						

Table 2. Heap manipulating JVM bytecode instructions

primitives, in particular the `MONITORENTER` and `MONITOREXIT` bytecode instructions, which correspond to the beginning and end of `synchronized` sections, respectively, we always consider the corresponding monitor to be live, as argued above. To focus more on the bytecode level, independently of the fact whether the `MONITORENTER` would block the thread or not, the execution of the instruction is started and thus the read of the monitor can be observed in either case – (blocking the thread or acquiring of the monitor) as needed for the guard restriction.

Below, we focus on extraction of relevant information for JVM bytecode. First, we identify the bytecode instructions that need to be considered by the analysis, later we focus on more specific JVM properties.

In Tab. 2, there is a list of bytecode instructions which manipulate references and accesses the heap; for each instruction, we mark whether it depends on or modifies values in the interesting parts of the heap mentioned above. Not all instructions which manipulates with references are interesting for DVA; e.g., **`aconst_null`**, **`if[not]null`**, **`areturn`**, **`aload`** do not directly access the heap, so they can be safely ignored by DVA.

Field accessing (`[PUT/GET][field/static]`) instructions do not make the type of the instance live. The particular field that is read or written is determined during compilation; there is no dynamic-dispatch as in case of virtual methods. Field accesses are thus independent of the actual runtime type of the corresponding instance. Similarly, the **`*aload`** resp. **`*astore`** instructions read resp. write data from/to arrays; the proper instruction is determined at compile-time using static types; its behaviour is independent of the actual runtime type (in case of `aaload`), thus these instructions do not mark type information as live.

As to the invoke instructions, it is obvious that their behaviour does not depend on the values of the fields. However, as mentioned above, since the runtime type of the instance on which the method is invoked influences which particular method is executed, the dynamic dispatch is applied (**`invokeinterface`**, **`invokevirtual`**). Note that **`invokestatic`** and **`invokespecial`** do not involve dynamic dispatch, the called method is determined statically during compilation, thus runtime type information is not used by them.

The **`athrow`** instruction throws an exception; its behaviour is independent of the type of the instruction, however, the exception handling, i.e., the decision which exception handler matches the exception depends on the type of the exception instance.

The **`invokedynamic`** instruction is the most complex; its purpose is to support dynamic languages using

JVM. Our implementation of the DVA analyses does not consider this instruction, because the particular JPF we used does not support this instruction.

The behaviour of the locking instructions (as well as calls to synchronized methods) has been described above. JPF stores a state of all locked monitors into the state vector; our DVR's do not modify the way monitors are serialized. Moreover, in order to lock/unlock the monitor, it needs to be accessed. This guarantees that the monitor instance is live and thus not omitted from program state.

The DVA does not have to consider the JVM instructions which create new objects (i.e., `new` and `ldc`). The created objects do not exist in predecessor states; so obviously these objects cannot be live in them.

Not only bytecode can manipulate with the heap content; also native functions (in JPF realized by modelled functions) can access it. The most obvious example is the introspection, which allows JPF to read or modify fields and call methods. Another example is the output to the console (i.e., `System.out`); the method `print(String)` is modelled in JPF and its native body reads the whole content of the provided string.

Furthermore, in addition to the types of non-determinism present during normal Java programs execution, JPF features methods for efficient modelling of the user or environment behaviour. These include, e.g., `Verify.getBoolean()`, `Verify.getInt()`. JPF process these methods by creating a temporal variable storing the selected value in them and creating a choice for each possible one (i.e., branches in the state space). Even when using these methods, the program satisfies the guard-restriction property.

5.1. Dynamic DVA

Our DDVA monitors interesting instructions described above and based on their appearances, it computes live addresses for each state. The algorithm of DDVA is listed in Alg. 2.

Algorithm 2 Dynamic DVR

```
11: procedure Main
12:     ModelCheck($P$, $s\_init$, [$s\_init$])

13: procedure ModelCheck(program $P$, state $s$, trace $t$)
14:     if is_error_state($s$) then  throw  Unsafe($s$, $t$)
15:     ($visited$, $live\_addrs$) ← IsVisited($s$)
16:     if $visited$ then return (VISITED, $live\_addrs$)
17:     if $s \in trace$ then return (LOOP, $\{all\_addrs\}$)
18:     $live\_addr\_s \leftarrow \{\}$
19:     for transition $alfa \in$ EnabledTransitionsIn($s$) do
20:         ($s\_succ$, $heap\_accesses$) ← $alfa(s)$
21:         ($\_$, $live\_addrs\_s\_succ$) ← ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
22:         $live\_addrs\_s\_alfa$ ← update_live_addrs ($live\_addrs\_s\_succ$, $heap\_accesses$)
23:         $live\_addrs\_s \leftarrow live\_addrs\_s \cup live\_addrs\_s\_alfa$
24:     SetVisited(ReduceState(($s$, $live\_addrs\_s$), $live\_addr$))
25:     $map\_stacks2live\_addrs[s.threads.callStacks]$ += $live\_addrs\_s$
26:     return (SAFE, $live\_addrs\_s$)

27: procedure IsVisited(state $s$)
28:     for $live\_addrs\_v \in map\_stacks2live\_addrs[s.threads.callStacks]$ do
29:         $r\_s\_candidate$ ← ReduceState($s$, $live\_addrs\_v$)
30:         if Visited($r\_s\_candidate$, $live\_addrs\_v$) then return (TRUE, $live\_addrs\_v$)
31:     return (FALSE, {})
```

Live addresses. While JPF executes the program forwards, for each transition (i.e., each forward step) (line 19–23), the analysis records all relevant instructions (line 20–21); to be more specific, the addresses (a part of the heap) to mark resp. unmark as live. JPF stops advancing forward once it reaches either the end of the program or a visited state (line 16). In the former case, there are no live addresses, so no address is marked as live (this corresponds to the empty set in our algorithm) (line 18). In the latter case, the live addresses stored for the matched (previously visited) state are used (line 15). When JPF backtracks a transition (i.e., goes backwards along a program trace), the analysis computes a union of live addresses for the target state of the transition and the reads and writes recorded for the transition itself (line 23); this way, live addresses for the source of the transition are computed. To obtain live addresses for state s, the live addresses from the beginning of all the outgoing transitions from s are merged; an address is live at state s if and only if it is live at the beginning of (at least one) transition starting in s.

Due to the DFS exploration strategy, all outgoing transitions are explored before the state is backtracked from. Thus, the live addresses for state s (i.e., the $\mathit{reduce\_state}_s$ function) as well as its reduced state r_s can be computed and stored for the state matching purposes after JPF backtracks over s. The only exception are cycles in the state space (not just in CFG). If a cycle is detected after reaching (an already visited) state s, the live addresses for it (being in the stack of unprocessed states) are not known. In order to preserve correctness, we have to mark all addresses in s as live (line 17). Note that it does not necessarily mean that the program has to loop forever; after a few iteration of the cycle, a non-deterministic choice can be taken, which will exit the loop. Alternatively, there is a more elaborated approach, which does not lose precision. It is possible to postpone the computation of live addresses for the cycle states. First, all the traces leading out of the cycle (remaining non-deterministic choices in the states forming the cycle) have to be explored, and later, after all of them are explored and their live addresses are known, the sets of live addresses for the cycle states can be computed – their live addresses are propagated along the states forming the cycle to simulate any number of loop iterations. This way, the sets of live addresses can be obtained even for state spaces with loops in a precise way. Nonetheless, since the practical outcome of this approach is not clearly visible, we postpone the experiments with it as future work.

State matching. After JPF reaches a state (after a forward step), state matching is initiated to decide whether the state has already been visited or it is a new one (line 27–31). Java programs satisfy the guard-restriction property, so Lemma 2 can be applied here; the state matching process attempts to find (line 28) a state V and a corresponding function $\mathit{reduce\_state}_V$ (i.e., *live_addrs_v*) satisfying the requirements of the lemma (line 30).

For the purpose of state matching, a helper map is maintained (*map_stacks2live_addrs*, line 28). For each call-stack, the map holds live addresses (in other words *reduce_state* functions) used to reduce the states with the particular call-stack. This map is used as an optimization reducing the number of live addresses to be examined.

State matching proceeds as follows: First, the set of possible live addresses (i.e., *reduce_state* functions) is obtained using the call-stack of the current state s (line 28). These live addresses are examined one-by-one whether their live values are the same as in s (line 29–30). In other words, a candidate reduced state is created using the current state s and its live addresses are examined (formally $\mathit{reduce\_state\_v}(s)$ is computed). Then, using the standard state matching function, the algorithm determines if a previously visited state v exists such that $\mathit{reduce\_state\_v}(s) = \mathit{reduce\_state\_v}(v)$ (line 30). If so, due to Lemma 2, a state equivalent to s has been already visited (line 30); if not, s is a new state (line 31).

5.2. Hybrid DVA

The other analysis of ours – Hybrid DVA – focuses on scalability instead of precision. It is designed to be fast in presence of threads and to allow for efficient state matching. The algorithm of our hybrid analysis is listed in Alg. 3.

For each program state, it identifies the fields which can be read (on any object of the given type) before the program terminates by any trace from that state. In its nature, the analysis is similar to the analyses proposed in [PL11], but we adapted it for the purpose of state matching. It consists of two phases: (1) static data-flow analysis, whose results are combined with (2) the (verification time) information from the currently reached state. The first phase is done once before a JPF run, while the second one is executed on demand during state-space exploration just before state matching.

Static phase. Backward flow-sensitive context-insensitive data-flow analysis over full inter-procedural control flow graph (ICFG) is used to obtain information about future behaviour of the program (line 33). For

Algorithm 3 Hybrid DVR

```
32: procedure MAIN
33:     $partial\_liveness \leftarrow$ HdvrComputePartialLiveFields($P$)
34:     ModelCheck($P$, $s\_init$, [$s\_init$])

35: procedure MODELCHECK(program $P$, state $s$, trace $t$)
36:     if IsErrorState($s$) then throw Unsafe($s$, $t$)
37:     $live\_fields \leftarrow$ HdvrComputeLiveFields($s.callstacks$)
38:     $reduced\_s \leftarrow$ ReduceState($s$, $live\_fields$)
39:     if IsVisited($reduced\_s$) then return $VISITED$
40:     SetVisited($reduced\_s$)
41:     for transition $alfa \in$ EnabledTransitionsIn($s$) do
42:         $s\_succ \leftarrow alfa(s)$
43:         ModelCheck($P$, $s\_succ$, $t$ + ($alfa$, $s\_succ$))
44:     return SAFE

45: procedure HDVRCOMPUTELIVEFIELDS(state $s$)
46:     $live\_fields \leftarrow \{\}$
47:     for Thread $t$ : $s.threads$ do
48:         for StackFrame $f$ : $t.callstack$ do
49:             $live\_fields \leftarrow live\_fields \cup partial\_liveness[f.instruction\_pointer]$
50:     return $live\_fields$
```

Instruction	Transfer function
(branching point)	$\text{after}[\ell] = \bigcup_{\ell' \in \text{succ}(\ell)} \text{before}[\ell']$
ℓ: v = o.f	$\text{before}[\ell] = \text{after}[\ell] \cup \{\text{ClassName(o).f}\}$
ℓ: return	$\text{before}[\ell] = \emptyset$
ℓ: call M	$\text{before}[\ell] = \text{before}[\text{M.entry}] \cup \text{after}[\ell]$
ℓ: other instr.	$\text{before}[\ell] = \text{after}[\ell]$

Figure 7. Transfer functions for the static phase of Hybrid DVA.

a given location (i.e., bytecode instruction) l, the analysis computes an over-approximation of the set of all fields which may be read before *returning from the method* containing l (including potential reads from the nested method calls and spawned threads). The data flow facts are pairs `ClassName.FieldName` which unambiguously identify all program fields. In contrast to [PL11], the facts do not include allocation sites of objects; even though potentially less precise, this allows for fast state matching, used together with heap canonicalization.

The transfer functions are defined in Fig. 7. The result of the static phase is the least fixpoint over the equations determined by these functions. If static analysis encounters the field-read instruction, it adds the field into the resulting set (see the second rule). Because the analysis summarizes all instances into a single data-flow fact, it does not treat field writes in any special way (the last rule applies).

The field reads are not propagated via exit-return edges (see the third rule). This blocks propagation along infeasible paths in ICFG, in particular those where a related call-entry edge starts and exit-return edge leads to a different method. For a program state, the complete picture about future reads is computed at the dynamic phase. This technique gets precision of a context-sensitive analysis at the computational cost of a fast context-insensitive analysis.

Dynamic phase. In the dynamic phase, the results from the static phase are utilized to get complete information about future field reads, first for each thread, then for a program state. The call-stack (of each thread) is processed in the top-down manner (line 48). The static analysis result for the current instruction on the top stack frame contains the possible field reads, before the current method is exited. Going down the

Benchmark	LOC	Threads
AlarmClock	250	4
CLIF-BladeInsertAdapter	2780	3
Cache4J	600	3
CoCoME	3500	4
Deos	2160	3
Elevator	400	3
FTDemo	1300	3
LinkedList	291	3
Producer-Consumer	180	3
RepWorkers	630	3
Simple JBB	2300	3

Table 3. Sizes and numbers of threads for each benchmark.

call-stack, for each stack frame, the analysis joins (i.e., adds) the field reads computed in the static phase for the instruction just after the call instruction (current instruction pointer in the stack frame) (line 49). This way, future behaviour being considered is extended to the end of the given stack frame. Once the entire call-stack is processed, the result contains all the fields the thread can read before it terminates. At the end, future reads of all threads are joined together to obtain the future field reads for the program state (line 50).

State matching. Hybrid DVA provides us with an over-approximation of live addresses. The dynamic phase depends only on information from the call-stacks, thus for the same call-stacks the analysis yields the same live addresses (irrespective of the content of the heap). So there is no need for complex state matching as in the case of aforementioned Dynamic DVR; put simply, a reduced state (in a representation suitable for state matching) is created, which contains only the live fields of object instances. For the state matching purposes, only the reduce state is used. In contract to Dynamic DVR, the live fields (i.e., a Hybrid-DVR equivalent of live addresses) does not need to be stored in addition to reduced states; each reduced state already contains all the information required to reconstruct the live fields.

We found that an efficient implementation is crucial in order to speed-up the verification. We heavily employ bit-vectors and block-operations in our hybrid DVR implementation.

6. Evaluation

Implementation of both DVRs together with an experimental setup, and all related data are accessible at `http://d3s.mff.cuni.cz/software/jpf-psm/`.

6.1. Benchmarks

We evaluated our approaches on 11 benchmarks – CoCoME [B$^+$07], FTDemo [A$^+$06], CLIF [Dil09], the Cache4j and the Elevator (both from PJBench suite [PJB]), Simple JBB, and a set of small benchmarks taken from the CTC repository [CTC] (AlarmClock, LinkedList, Deos, Producer-Consumer, ReplicatedWorkers). The information on LOC and number of threads created during the programs' runs are listed in Tab. 3. All the benchmarks are multi-threaded, use heap, are error-free, and their state space (not the CFG) is acyclic, thus the reported data are related to their complete state spaces. The reason for not including the benchmarks with cyclic state spaces is that our current implementation does not support them. However, there is no principle obstacle – we just have not implemented the support for them so far. The benchmarks were taken from [PL15]; we have chosen them for several reasons. We already knew they worked with JPF,

Benchmark	original JPF		JPF with Hybrid DVR				JPF with Dynamic DVR			
	states	time	states		JPF time + SA time		states		time	
AlarmClock	573 362	2:15	573 362	100%	1:58 + 0:03	89%	573 362	100%	2:08	95%
CLIF-BladeInsertAdapter	50 627	0:21	43 512	86%	0:14 + 0:04	86%	31 491	62%	0:47	224%
Cache4J	5 106 128	17:53	5 105 482	100%	17:03 + 0:04	96%	2 880 839	56%	14:18	80%
CoCoME	2 213 005	23:19	2 103 729	95%	19:44 + 0:05	85%	1 319 894	60%	13:36	58%
DeOS	625	0:01	370	59%	0:01 + 0:04	500%	211	34%	0:01	100%
Elevator	7 304 096	29:34	7 256 407	99%	27:12 + 0:03	92%	6 947 527	95%	33:44	114%
FTDemo	59 354	0:40	56 308	95%	0:31 + 0:06	92%	53 646	90%	0:35	88%
LinkedList	2 038 840	5:51	2 038 840	100%	5:10 + 0:04	89%	1 974 486	97%	6:15	107%
ProdConsumer	6 074 085	19:35	6 074 055	100%	16:46 + 0:04	86%	1 237 457	20%	3:46	19%
RepWorkers	15 363 223	56:42	15 362 801	100%	48:26 + 0:05	86%	12 052 301	78%	50:11	88%
SimpleJBB	109 861	2:30	85 655	78%	1:37 + 0:05	68%	60 222	55%	1:10	47%
Overall	38 893 206	2:38:42	38 700 521	100%	2:18:42 + 0:47	88%	27 131 436	70%	2:06:31	80%

Table 4. Experimental results – runtime and state space size. "JPF time" represents the time spent by JPF at runtime. The "SA time" is the time spent in static analysis of HDVR. "Time" is the overall running time, if there is no static-analysis phase.

Benchmark	original JPF		JPF with Hybrid DVR				JPF with Dynamic DVR					
	memory	vector size	memory		vector size		memory		DVA memory	vec. size		DVA size
AlarmClock	2 802.82 MB	5 117.84	557.12 MB	20%	1 010.88	20%	593.42 MB	21%	246.81 MB	625.88	12%	451.37
CLIF-BladeInsertAdapter	907.05 MB	18 778.60	198.48 MB	22%	4 775.00	25%	21.28 MB	2%	7.80 MB	440.76	2%	259.72
Cache4J	30 351.17 MB	6 224.80	12 300.51 MB	41%	2 518.32	40%	3 727.82 MB	12%	517.32 MB	1 160.56	19%	188.30
CoCoME	32 290.57 MB	15 292.08	16 097.64 MB	50%	8 015.64	52%	4 157.39 MB	13%	2 299.96 MB	1 467.60	10%	1 827.18
DeOS	3.62 MB	6 064.60	0.81 MB	22%	2 287.80	38%	0.41 MB	11%	0.25 MB	799.88	13%	1 242.39
Elevator	44 217.47 MB	6 339.88	19 410.64 MB	44%	2 796.92	44%	7 661.87 MB	17%	1 894.32 MB	862.48	14%	285.91
FTDemo	1 437.22 MB	25 382.60	455.65 MB	32%	8 477.16	33%	178.55 MB	12%	92.27 MB	1 678.36	7%	1 803.53
LinkedList	10 347.93 MB	5 313.96	1 887.42 MB	18%	962.68	18%	1 057.93 MB	10%	7.09 MB	550.08	10%	3.77
ProdConsumer	30 995.96 MB	5 342.88	8 675.65 MB	28%	1 489.68	28%	1 084.43 MB	3%	368.92 MB	598.32	11%	312.61
RepWorkers	87 001.07 MB	5 930.04	34 958.10 MB	40%	2 378.04	40%	9 302.25 MB	11%	695.21 MB	740.84	12%	60.48
SimpleJBB	5 452.43 MB	52 033.12	2 112.35 MB	39%	25 851.12	50%	366.94 MB	7%	205.17 MB	2 808.68	5%	3 572.39

Table 5. Experimental results – memory consumption. The vector sizes are averages over all vectors in a particular analysis. The "DVA memory" is the memory used solely by the DVA analysis. The "DVA size" is the average amount of bytes needed to store the DVA information for each state vector.

they generate a relatively small yet non-trivial state space, and since we worked with them before, we could compare the results with other techniques.

As to the default JPF configuration, we disabled *heap canonicalization*, because it is not compatible with our analyses. The reason is that in our implementation, we rely on one-to-one mapping of unique ids to objects, which does not exist if heap canonicalization is used. Theoretically, it is possible to overcome this restriction of our analyses, but it would imply an additional overhead for maintaining a one-to-one mapping by ourselves. On the other hand, the following default optimizations were left turned on: *final fields filtering*, *live threads*, and *dynamic lock analysis*.

Cache4j is a simple framework for in-memory caching of Java objects. We use the configuration with LRU eviction algorithm and a blocking cache. Usage of the cache is modelled by two parallel threads accessing it.

CLIF is an open-source stress-testing platform, which is able to generate various kinds of traffics and measure resource usage for the system under test. The core of CLIF is based on the Fractal component model [B$^+$04]. Our benchmark consists of one of its internal component, which is responsible for adding measured blade servers, and a generated environment, which simulates its usage. The environment was generated from a behaviour specification using [JPK12].

The *CoCoME* benchmark is a prototype of a cash-desk system for supermarkets. It consists of an inventory management sub-system (storing a product database) and a cash-desk line formed by a set of cash desks. The application consists of a test driver, which simulates two clients served in parallel.

The *Elevator* benchmark is a simulator of elevators in a building. We use the configuration with two elevators and four actions executed by a simulated person.

FTDemo is a high-level component-based prototype of a software system providing Wi-Fi internet access at airports. The demo consists of around twenty software components, handling, e.g., user authentication, payment for network access, and IP address allocation. We ran the system with two simulated users in parallel.

Simple JBB is a simplified version of the SPEC JBB 2005 benchmark, which is a model of an enterprise information system for concurrent processing of clients' requests. It models several databases (e.g., orders and stock) and transactions that operate upon these databases. Some simplifications were necessary to make the benchmark run inside JPF and to reduce the size of the state space to a reasonable level.

6.2. Experimental objectives

The main question to answer by our experiments was whether and to which extent the DVRs improve the performance of software model checking in the case of Java PathFinder, both in terms of the runtime and memory consumption. In particular, this involves both contribution to reduction of the state vectors and the reduction of the number of explored states. We were also interested in the overhead of our analyses, that is the memory and time consumption of Dynamic DVR and the static part of Hybrid DVR to see whether the analyses actually pay off. In other words, the aim was to determine whether employing dead analyses of heap data can broaden the set of Java programs that can be verified by JPF.

6.3. Results

All the experiments were run on a Linux machine with an Intel(R) Xeon(R) X5687 (3.60GHz) CPU and 192GB memory with the JPF filtering serializer. Tab. 4 and Tab. 5 summarize the results of the benchmarks.

In Tab. 4, the number of states and the running time (in the hour:min:sec format) for each type of DVR is reported for each benchmark. The percentages are related to the values obtained by original JPF. We ran each benchmark ten times and report the average run times. The standard deviation of JPF run times was negligible with the average of 0.67% and maximum of 2.95%, which practically means few seconds in the case of the most complex benchmarks. The information about memory (state space, vector sizes, etc.) were constant across the runs. On average, Hybrid DVR yields a reduction of 12% in the verification time, while using Dynamic DVR results in a 20% speed-up and a reduction of 30% in the number of states. Note that the DeOS benchmark took in the case of Hybrid DVR five times longer to complete than in the original JPF settings. This is due to static analysis performed at the beginning of the run, which, even though it reduced the state space to 59%, did not pay off. Since the original state space consists of 625 states only, we do not consider this case an issue.

Tab. 5 lists the information on memory consumption of the state matching using full state vectors. We report the memory consumed solely by state matching [2] – we summed up the sizes of state vectors used (they are all arrays of `int`). The default state matching in JPF stores only hashes of state vectors, thus its memory requirements are negligible. In the table, both the absolute amount of memory and the percentage reduction in the case of our analyses are presented. We also report reduction of the state vector size; the reduction is smaller compared to reduction of the memory requirements, however, it is still significant. It is caused by fewer program states being visited. The reduction of the vector size is larger for Dynamic DVR than Hybrid DVR, because dynamic DVR is more precise.

We consider the vector shrinking particularly interesting. In the case of Hybrid DVR, memory consumption is reduced to one third on average, both in terms of the total consumption and vector sizes, while the memory consumption for Dynamic DVR is an order of magnitude lower.

An interesting example is the Alarm clock benchmark, where neither Hybrid nor Dynamic DVR reduce the number of states, but the verification is faster (cf. Tab. 4). This shows that the overhead of our DVAs is

[2] At the beginning, we used the *-Xmx* Java option to limit the maximal memory consumption of the Java process to estimate the approximate amount of memory needed for each benchmark.

low; the analyses identify dead parts of program states faster than the state matching processes these dead parts (e.g., compute state hashes). In this particular case, the state vectors reduce to 20 and 12 per cent of the original size for Hybrid and Dynamic DVR, respectively (cf. Tab 5). The main source of dead variables is the `System.properties` object.

Hybrid DVR. In the case of Hybrid DVR, we present both the JPF runtime and the static phase duration. Typically, static analysis of live field takes less than five seconds (with a maximum of six seconds). If the Java standard library uses introspection to create class instances of a dynamic class (i.e., it calls `Class.forName(*).newInstance()`), it has a major impact on the static-analysis runtime. This is due to processing a large number of classes from standard libraries. Since static analysis considers also the used standard Java libraries, the whose size of processed Java bytecode is considerably larger than the size of the verified benchmarks, this phase does not represent a bottleneck of our approach.

Dynamic DVR. Our dynamic analysis needs an additional amount of memory to store live addresses of program states. This additional amount of memory is always required, independently of whether full state vectors or only their hashes are used in state matching. In Tab. 5, we report the memory used to store live addresses in the DVA memory column. On average, a single set of live addresses takes 910 bytes; the consumed memory ranges from 256KB for DeOS, to 2.3GB for CoCoME. Since multiple states often have the same set of live addresses (and differ only in the values at these addresses), the amount of memory required to store them is one to two orders of magnitude less than the memory needed to store and match full program states (instead of their hashes – JPF default). In Tab. 5, the column DVA size reports how much additional memory per program state is needed to store DVA information; due to sharing of the live addresses, the size is smaller than the average of 910 bytes.

6.4. Summary

The results of the benchmarks demonstrate that the contribution to performance in both the Hybrid and Dynamic DVR cases is significant. Moreover, according to our experience with software model checking, the most frequent reason why model checking fails is an insufficient amount of memory available, whereas the runtime is usually acceptable (even though it can take up to several hours). Regarding this, we consider the reduction in memory consumption when using our DVRs especially valuable.

Let us have a look at the aspects influencing the decision which analysis to choose for model checking of a particular input. In general, Hybrid DVR usually pays off if the benchmark runtime takes just several minutes. In this case, the speed up of the entire process is higher than the time required for the static analysis phase. The only situations when HDVR would not be suitable (from the practical point of view) is when many new (not implemented inside JPF) native methods are used. This requires creating models for them in JPF first (this is not DVR specific) and then create models for static analysis itself (HDVR specific), which represents a significant additional amount of work. The usefulness of dynamic DVR is hard to predict in general, but we foresee some heuristics based on random state space search and sampling the stack traces, which would provide an insight as to the DDVR benefits for a specific Java program (without model checking it). In particular, the size the set of live addresses at particular stack-trace influences the overall runtime in terms of number of comparisons when searching for an equivalent state.

It would be also useful to see the results if our techniques are combined with DVRs for local variables. We have not implemented a combined analysis mainly for the following reasons: The corresponding techniques are already known, so there would not be a scientific contribution in that. Next, the local DVRs only reduce the state space, not the state vectors. Finally, it would not be clear what is the effect of the local DVRs and what is the effect of the global (heap) ones' reduction (turning on and off each of them would not give a clear insight), but, on the other hand, we agree that it would show the practical contribution.

The selected benchmarks share some properties that, in general, might slightly bias the final observation about their results. First of all, all of them are multi-threaded. Even though this seems to be more general than including also single-threaded benchmarks, the (data) non-determinism appearing in single-threaded programs may cause the reductions to work slightly worse or better. Another aspect of the benchmarks is that their state spaces do not contain cycles. Unlike in the previous case, we do not expect programs with cyclic state space to have significant impact on reduction, regardless of the way it is handled, i.e., simplified or precise (see Sect. 5.1). Last aspect we would like to mention here is that we restricted ourselves, due to implementation platform, to explicit model checking. In principle, there is no obstacle in applying our reductions also in the context of symbolic model checking, but it is worth mentioning that a number of technical details have to be addressed there.

7. Related work

In this section, we compare the presented DVRs to related approaches both in general and showing the differences on our running example from Sect. 2.

Dynamic DVA. Let us continue with the example after exploring the $S3$ state. After state $S3$ is fully processed, the model checker will backtrack the transition $S2 \rightarrow S3$. The dynamic analysis uses the records stored for this transition and it adds `TreeNode@5.value` among live variables; because this field is live in state $S3$ (i.e., in the target state of the transition) adding it has no effect and at the beginning of transition $S2 \rightarrow S3$, the live variables are the same as in $S3$, i.e., `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`. Because this is not the only outgoing transition from $S2$, the model checker has to explore also the transitions where thread $T2$ is scheduled before thread $T1$ – transition $S2 \rightarrow S5$. Using the same approach as before it discovers that only `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value` are live at the beginning of transition $S2 \rightarrow S5$. To compute live variables for $S2$, our analysis merges the live variables at the beginning of all outgoing transitions; the live variables again include `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`. Since the model checker has already explored all outgoing transitions from $S2$, it backtracks transition $S2 \rightarrow S1$ to state $S1$ and our analysis computes that at the beginning of the transition, the live variables are again `TreeNode@1.value`, `TreeNode@5.left`, and `TreeNode@5.value`.

In [LJ06], Lewis et al. introduce DVA, which uses dynamic information obtained at runtime to improve the precision of static analysis. Their analysis supports heap and interrupts; however, it lacks support for multi-threaded programs. Once a state is reached, Lewis's DVA spawns an additional forward simulation to get a depth-limited knowledge about future behaviour of the state. The results from the simulation are used in subsequent static analysis. In particular, the information of the simulation is used for elimination of the edges in CFG that cannot be taken. Our approach differs from that of Lewis in three aspects: our DDVA (i) has full knowledge about future behaviour, (ii) computes this information cheaply during state space exploration, and (iii) is thus more precise. Consider Lewis's DVA with a bound of two transitions. In this case, the simulation also finds the trace $S1 \rightarrow S2 \rightarrow S5$ where both threads stop at the read of `TreeNode@5.value`. Hence, using this bound, the analysis is not able to block the path in CFG that accesses the `TreeNode.right` fields. In turn, Lewis's analysis will imprecisely mark `TreeNode.right` fields in $S1$ as live, in contrast to our dynamic analysis which correctly identifies these fields as dead and ignores their values.

The DDVA introduced by Self and Mercer [SM07] is similar to our approach. Live addresses are computed from observed reads and writes in the same way. The approaches differ in the way non-determinism is handled; our DDVA explores all possible future executions from a state (i.e., all outgoing traces) and then merges their live variables, while DDVA of [SM07] operates only on a single trace. It lacks knowledge about reads and writes on other traces, hence, at non-deterministic (branching) states, it marks all the variables as live. In our running example, both analyses get the same result for $S3$, however in $S2$, where a non-deterministic choice is present, DDVA of [SM07] marks all fields as live in contrast to our analysis. The dynamic DVA in [SM07] seems to be more reasonable for sequential programs, where it provides (some) dead variables even for states which are not fully processed. However, our approach is better suited for multi-threaded programs, where transitions are quite short (often only a few instructions), and where DDVA of [SM07] becomes considerably less precise.

In the case of programs without non-determinism both methods equal and yield the *DVA maximal reduction* [SM07], while in the other case, our approach provides more precise results (as shown above). Moreover, for our Dynamic DVR, we also proved bisimulation between the original transition system and the transition system over reduced states (without dead variables). However, in [SM07], their definition permits additional transitions in DVA abstract state space which do not have pre-image transitions in the full state space. From this point of view, it is obvious that the original state space and the corresponding reduced one cannot be bisimilar.

Hybrid DVA. Similar DVA focusing just on local variables is used in, e.g., the SPIN model checker [Hol04], MURPHI [mur, YG04], and BANDERA [C+00]. To obtain complete results for global variables and static fields, the effects of other threads need to be considered; in [BFG99], a *control-graph* – cross-product of all locations of all threads is created and static DVA is done over this structure; for programs, this can be a bottleneck. Each thread (its ICFG) is composed of a large number of locations, thus their cross-product can be potentially of the same size as the state space, e.g., if values of variables are determined by the location as in the classical Dining Philosophers problem. Moreover, the number of parallel threads can be unbounded, and in such cases the approach of [BFG99] cannot be used at all.

To our best knowledge, the analysis presented in [BFG99] is the only (Static) DVA, which supports concurrency and global variables. However, it does not support heap (instances), since it was designed for verification of specification languages. In contrast to our Hybrid DVA, which uses cheap ICFG analysis, the analysis of [BFG99] creates so called *control graph* – a cross-product of all possible locations in all threads, whose size can be comparable to the one of the corresponding (full) state space. The live addresses are computed by static analysis over the control graph, i.e., as the least fixpoint using reads and writes of the actions. This implies that their analysis is expensive for multi-threaded programs, which have thousands of locations (i.e., instructions) per thread.

8. Conclusion and future work

In this article, we presented two novel dead variable analyses, one purely dynamic, the other, hybrid, combining static and dynamic analyses. Based on them, we designed dead variable reductions of the state space for Java programs. We presented a proof of bisimulation between the original and reduced state spaces, which allows us for establishing properties of the original program by means of analysis (model checking) of the reduced state space. We have implemented our technique in Java PathFinder, an explicit-state software model checker for Java programs. Using a large set of both real and academic programs, we demonstrated the benefits of the proposed reduction in terms of both verification time and memory consumption.

As aforementioned, local variables are a target of other analyses and thus we do not focus on them. As future work we envision a combination of dead variable reduction on the local and global level, i.e., involving both local and heap variables. Also, as already mentioned, we plan to investigate the impact of a precise dead variable analysis for loops, which, on the one hand, require both more time and memory to compute, while on the other hand provides precise information about dead variables.

References

[A+06] Jiří Adámek et al. Component Reliability Extensions for Fractal Component Model. `http://d3s.mff.cuni.cz/software/ft/`, 2006.

[B+04] Eric Bruneton et al. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *LNCS*, pages 7–22. Springer, 2004.

[B+07] Lubomír Bulej et al. CoCoME in Fractal. In *LNCS 5153*, pages 357–387, 2007.

[BFG99] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State space reduction based on live variables analysis. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1999.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* The MIT Press, Cambridge, Mass, 2008.

[C+00] James C. Corbett et al. Bandera: extracting finite-state models from java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 439–448. ACM, 2000.

[CTC] Concurrency Tool Comparison repository. `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`.

[Dil09] Bruno Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annals of telecommunications - annales des télécommunications*, 64(1):101–120, 2009.

[FBG03] Jean-Claude Fernandez, Marius Bozga, and Lucian Ghirvu. State space reduction based on live variables analysis. *Sci. Comput. Program.*, 47(2-3):203–220, 2003.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 9(40):1098–1101, September 1952.

[JK16] Pavel Jančík and Jan Kofroň. Dead variable analysis for multi-threaded heap manipulating programs. In *Proceedings of 31st ACM Symposium on Applied Computing.* ACM, 2016.

[JPK12] Pavel Jančík, Pavel Parízek, and Jan Kofroň. BeJC: Checking Compliance Between Java Implementation and Behavior Specification. In *Proceedings of the 17th International Doctoral Symposium on Components and Architecture*, WCOP '12, pages 31–36, New York, NY, USA, 2012. ACM.

[LJ06] Micah Lewis and Michael Jones. A dead variable analysis for explicit model checking. In John Hatcliff and Frank Tip, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 48–57. ACM, 2006.

[mur] MURPHI Model Checker. `http://formalverification.cs.utah.edu/Murphi/`.

[NR09] Viet Yen Nguyen and Theo C. Ruys. *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, chapter Memoised Garbage Collection for Software Model Checking, pages 201–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Pel93] Doron Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 409–423, London, UK, UK, 1993. Springer-Verlag.

[PJB] Parallel Java Benchmarks. `https://bitbucket.org/pag-lab/pjbench`.

[PL11] Pavel Parízek and Ondřej Lhoták. Identifying future field accesses in exhaustive state space traversal. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 93–102. IEEE Computer Society, 2011.

[PL15] Pavel Parízek and Ondřej Lhoták. Model checking of concurrent programs with static analysis of field accesses. *Science of Computer Programming*, 98, Part 4:735 – 763, 2015.

[rle] Run-length encoding. `https://en.wikipedia.org/wiki/Run-length_encoding`.

[SM07] Joel P. Self and Eric G. Mercer. On-the-fly dynamic dead variable analysis. In Dragan Bosnacki and Stefan Edelkamp, editors, *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, volume 4595 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2007.

[VHB$^+$03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[YG04] Karen Yorav and Orna Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.

CHAPTER 9

Framework for Static Analysis of PHP Applications

Authors: David Hauzar and Jan Kofroň

[2] *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015),
DOI: 10.4230/LIPIcs.ECOOP.2015.68910.4230/LIPIcs.ECOOP.2015.689, Prague, Czech Republic, July 2015*

Framework for Static Analysis of PHP Applications*

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract

Dynamic languages, such as PHP and JavaScript, are widespread and heavily used. They provide dynamic features such as dynamic type system, virtual and dynamic method calls, dynamic includes, and built-in dynamic data structures. This makes it hard to create static analyses, e.g., for automatic error discovery. Yet exploiting errors in such programs, especially in web applications, can have significant impacts. In this paper, we present static analysis framework for PHP, automatically resolving features common to dynamic languages and thus reducing the complexity of defining new static analyses. In particular, the framework enables defining value and heap analyses for dynamic languages independently and composing them automatically and soundly. We used the framework to implement static taint analysis for finding security vulnerabilities. The analysis has revealed previously unknown security problems in real application. Comparing to existing state-of-the-art analysis tools for PHP, it has found more real problems with a lower false-positive rate.

1998 ACM Subject Classification F.3.1

Keywords and phrases Static analysis, abstract interpretation, dynamic languages, PHP, security

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

To analyze programs precisely and soundly, static analysis needs to resolve method calls, include statements, and accesses to data structures. Since in dynamic languages, targets of method calls and include statements can depend on information about values (and types) of expressions, value analysis tracking values of all primitive data types present in the language needs to be performed. Moreover, due to frequent use of dynamic data structures such as associative arrays and objects, value analysis needs to be combined with heap analysis. These depend on each other also the other way round—since array indices and object properties can be accessed using arbitrary expressions, heap analysis needs value analysis to evaluate these expressions. This makes any end-user static analysis that takes this into account overly complex.

In this paper we present a static analysis framework for languages with dynamic features [10] based on abstract interpretation [1]. The framework automatically resolves dynamic features and makes it possible to define static analyses without taking these features explicitly into account.

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization.

Lattice	$(L, \sqsubseteq, \sqcup)$	$(Bool, \implies, \vee)$	
Initial value	$init(v)$	$true$	if $v \in$ `$_POST` $\cup$ `$_GET` $\cup \ldots$
		$false$	otherwise
Transfer function	$[\![LHS = RHS]\!]$	$var = \bigvee_{r \in RHS} r$	if $var \in LHS$
		$var = var$	otherwise
	$[\![st]\!]$	$var = var$	if st is not assignment

Table 1 Propagation of tainted data.

In particular, our contributions include:

- The architecture of the static analysis framework for dynamic languages and the way dynamic features are automatically resolved.
- Description of all necessary analyses that are needed to automatically resolve dynamic features. We define value analysis that tracks values of all primitive data types of PHP. We articulate our assumptions on heap analysis to take dynamic index and property accesses into account—indices and properties are created when they are accessed for the first time and accesses can be performed using arbitrary value expressions, yielding statically unknown values.
- Composition of all necessary analyses allowing to define these analyses independently. Here the main challenge is defining the interplay of value analysis and heap analysis taking dynamic features into account. The composition is sound; if the analyses being composed are sound, the resulting analysis is sound as well.

2 Motivation

As a motivation example, consider static taint analysis, which is often used for security analysis of web applications. It can be used for detection of security problems, e.g., vulnerability of an application to SQL injection and cross-site scripting attacks. Static taint analysis can be described as follows. The program point that reads user-input, session ids, cookies, or any other data that can be manipulated by a potential attacker is called *source*, while a program point that prints out data to a browser, queries a database, etc. is referred to as *sink*. Data at a given program point are *tainted* if they can pass from a source to this program point. Tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potentially malicious parts. Program is *vulnerable* if it contains a sink that uses data that are tainted and not sanitized.

Static taint analysis can be performed by computing the propagation of tainted data and then checking whether tainted data can reach a sink. The specification of forward data-flow analysis computing the propagation of tainted data is shown in Tab. 1[1]. The analysis is specified by the lattice of data-flow facts and lattice operators, the initial values of variables, and the transfer function.

Consider now the code in Fig. 1. The code contains two vulnerabilities to XSS attack [7]. The first vulnerability corresponds to the call at line (25), the second vulnerability corresponds to the call at line (26). In both cases the method `show` of `Templ1` can be called (line (5)) with the parameter `$msg` being tainted and going to the sink. Taint analysis defined using our framework uses just the information in Tab. 1 and can still detect both vulnerabilities. This

[1] For simplicity we omit the specification of sanitization.

```
 1  class Templ {
 2    function log($msg) {...}
 3  }
 4  class Templ1 : Templ {
 5    function show($msg) { sink($msg); }
 6  }
 7  class Templ2 : Templ {
 8    function show($msg) { not_sink($msg); }
 9  }
10  function initialize(&$users) {
11    $users['admin']['addr'] = get_admin_addr_from_db();
12  }
13  switch (DEBUG) {
14    case true: $mode = "log"; break;
15    default: $mode = "show";
16  }
17  switch ($_GET['skin']) {
18    case 'skin1': $t = new Templ1(); break;
19    default: $t = new Templ2();
20  }
21  initialize($users);
22  $id = $_GET['userId'];
23  $users[$id]['name'] = $_GET['name'];
24  $users[$id]['addr'] = $_GET['addr'];
25  $t−>$mode($users[$id]['name']);
26  $t−>$mode($users['admin']['addr']);
```

Figure 1 Running example

is possible only because the framework automatically resolves control flow and accesses to built-in data structures. That is, the framework computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$mode` can contain values `"show"` and `"log"`. Based on this information, it automatically resolves calls at lines (25) and (26). As the framework automatically reads the data from and updates the data to associative arrays and objects, tainted data written at line (23) are read at line (25). Moreover, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads this tainted data.

3 Static Analysis Framework

The architecture of the framework is shown in Fig. 2. The analysis is split into two phases. In the first phase, the framework computes control flow of the analyzed program together with the shape of the heap and information about values of variables, array indices and object properties and evaluates expressions used for accessing data. The control flow is captured in the intermediate representation (IR), while the other information can be accessed using the data representation. In the second phase, end-user analyses of the constructed IR are performed.

Data representation allows accessing analysis states. In particular, it allows reading and writing values, and modifying shape of data structures. Next, it performs join and widening of analysis states and defines their partial order. Importantly, data representation defines the interplay of heap, value, and declaration analyses allowing each analysis to define these operations independently on other analyses.

The implementation of heap analysis tracks the shape of the heap and must provide

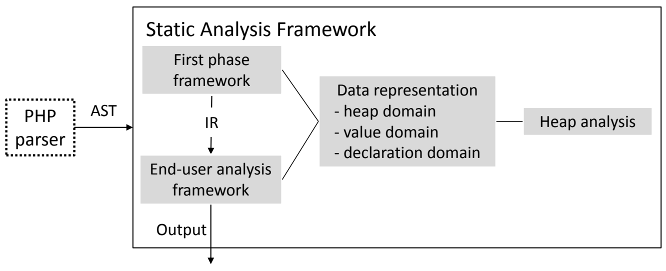


Figure 2 Architecture of the framework.

information that the value analysis needs to read values from data structures, update values to data structures, and to join values stored in data structures.

The implementation of the first phase must provide information necessary for computing control flow of the program and the information that the heap analysis needs to access data. That is, it must define value analysis that tracks values of PHP primitive types, evaluates value expressions modeling native operators, native functions, and implicit conversions. Next, the implementation must define declaration analysis handling declarations of functions, classes, and constants. Finally, it must compute targets of throw statements, include statements, and function and method calls.

The implementations of end-user analyses define additional value analyses. In contrast to value analysis for the first phase, which must track values of PHP primitive types, end-user value analyses can use an arbitrary value domain. This is possible because (1) control flow is already computed, (2) the shape of the heap is computed and dynamic data accesses are resolved (i.e., value expressions specifying data accesses are evaluated). That is, all information that the data representation needs to determine accessed variables, array indices, and object properties is available. (3) Data representation combines heap, value, and declaration analyses automatically.

3.1 Intermediate Representation

The intermediate representation (IR) of our analysis is a graph, in which each node contains an instruction. There are two types of nodes in the graph—*value nodes* and *non-value nodes*. Value nodes compute and store representation of values while non-value nodes perform other actions. The graph has two types of edges. *Flow edges* represent potential control flow between instructions of the program—they define ordering in which program instructions can be executed. *Value edges* connect nodes that use values (e.g., operators) with nodes that represent these values (e.g., operands).

Each node has associated an analysis state stored in data representation. The state is modified by transfer function defined for the node and the resulting state is propagated to successor nodes connected with flow edges. If a node has more predecessors the states of predecessors are joined.

Note that transfer functions for most of the value nodes are defined as identity—they do not modify the analysis state. That is, most of the value nodes just compute values (e.g., evaluate expressions) or compute information that specify data access to values (e.g., compute possible names of variables that they represent). This information is stored in data representation, but it is not a part of the analysis state and thus it is not propagated to successor nodes. Instead, nodes that use these values (e.g. operator nodes) are connected with value nodes (e.g. operands) using value edges. If an operand value is needed when evaluating the operator, the value edge is used to get the value from the operand.

Example 1: As an example, consider the intermediate representation corresponding to the statement `$$a=b($c)`. The statement assigns the value computed by function `b` to a variable with the name given by the value of variable `$a`. The resulting intermediate representation is depicted in Fig. 3. Note that the node corresponding to the assignment instruction is connected using a value edge with the source of the assignment (the node containing the value computed by the function `b`) and with the target of the assignment (the node representing the assigned variable—`$`). Next, the latter node is connected using a value edge with the node representing possible names of the assigned variable (node `$a`). △

The nodes can be of different types. In the following, we denote value nodes by adding superscript V; for each node, its parameters are the value nodes that are connected with the node using value edges:

variable$^V[n^V]$: represents a variable—stores the information necessary for accessing the variable in data representation. The parameter n^V is the value node that represents the name of the variable. Note that reading n^V yields an arbitrary value from the abstract string domain and can thus represent more concrete string values—names. Consequently, the variable node can represent more concrete variables.

property-use$^V[o^V, f^V]$, index-use$^V[a^V, i^V]$: property-use^V stores the information for accessing a property of the given object. Parameter o^V is the value node storing the representation of the object and f^V is the value node storing the name of the property. Again, reading o^V and f^V yields abstract values and the property-use^V node can get representation of more properties. The index-use^V is similar and it is used for accessing arrays.

assign$^V[l^V, r^V]$: represents the assignment of the right operand r^V to the left operand l^V and stores the information for accessing this value. While the parameter l^V is a value node whose type can be variable, property-use, and item-use, the parameter r^V is an arbitrary value node.

alias$^V[l^V, r^V]$: represents the alias statement. The alias statement is similar to the assignment statement. However, besides performing the assignment, the alias statement creates explicit alias between its parameters and both parameters of the alias statement must be variables, object properties, or array indices.

expression$^V[e, o_1^V, ..., o_n^V]$: represents the expression e with operands $o_1^V, ..., o_n^V$. It stores the representation of the result.

assume$[c]$: represents assumption implied, e.g., by *if* and *while* statements. It indicates whether the condition c is satisfiable. If the condition is unsatisfiable, the flow is not propagated to descendant nodes. If the condition is satisfiable, the analysis state is refined according to the condition and then propagated to descendant nodes.

constant-declaration$[d]$: represents declaration of a constant.

function-declaration$[d]$: represents declaration of a function.

class-declaration$[d]$: represents declaration of a class.

call$^V[n^V, o^V, a]$, construct$^V[n^V, a]$: represents a call of a function whose name is specified using the value node n^V on an object specified using the value node o^V with arguments specified using a list of value nodes a. The construct^V nodes are similar to call^V nodes and are used for `new` expressions. Note that reading n^V, o^V, and elements of a yields abstract values that can represent more concrete values.

return$[e^V]$: represents a return from a function. e^V represents the value of a return expression.

include$[p^V]$: represents the inclusion of the script given by the path specified by the value node p^V. Again, a path can represent more concrete values.

eval$[c^V]$: evaluates the code fragment specified by value node c^V.

native-method$^a[]$: represents execution of a native method or a native function with arguments specified using a list of value nodes a.

extension$[f, a]$: is used to dynamically extend the control from IR node f. This is necessary when the information needed to determine the control flow from the node is computed by the analysis. This is the case of calls to functions, methods and constructors, and the include and eval statements. During analysis, for each dynamically discovered control flow from the node, a single extension node is added. Parameter f is the node that is extended. Parameter a is used in the cases that the control flow is extended because of a function, method, and constructor call and it is a list of value nodes representing parameters of the call.

extension-sink$[n]$: represents a join point of all the extensions of node n.

try-scope-start$[c]$ and try-scope-end$[c]$ represent the start and the end of a `try` block. Parameter c represents catch blocks associated with the `try` block.

throw$[v^V]$: represents the `throw` statement. Parameter v^V is the node representing the value to be thrown.

catch$[v^V]$: represents a catch block. It contains a node representing the first node of the catch block as a flow child. Parameter v^V is the node representing the value to be thrown.

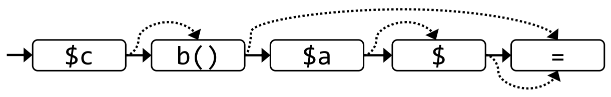


Figure 3 Intermediate representation of the statement `$$a=b($c)`. Solid edges are flow edges, dashed edges are value edges.

3.2 Building IR

To determine control flow of the analyzed application, the information from value analysis is needed. Thus, the IR is built gradually during the analysis.

Initially, IR for the entry script of the application (typically `index.php`) is built. This IR contains caller nodes—the nodes corresponding to function, method, and constructor calls, script inclusions, and eval statements. Since at this point, the information needed to compute control flow from these nodes is not yet available, the control flow is initially directed to the nodes that follow the calls.

The control flow is then extended during static analysis. When processing a caller node, the analysis framework provides the first phase implementation with all information already computed by the analysis that is relevant to determine the control flow. Using this information, the first phase implementation finds appropriate function or method definitions or scripts to be included, and it computes IRs representing their control flow. The first phase implementation can build new IRs or use existing IRs, which are then shared among multiple caller nodes. This way, the first phase implementation can control context sensitivity. Finally, the control flow of the caller nodes is extended with computed IRs.

IRs are not connected to caller nodes directly—extension node is inserted between each caller node and the entry node of the connected IR and extension-sink node is inserted between each final node of the IR and the node following the call. While an extension node binds actual parameters to formal parameters for function, method, and constructor calls, an extension-sink joins states of final nodes of all the IRs that extend the corresponding caller node.

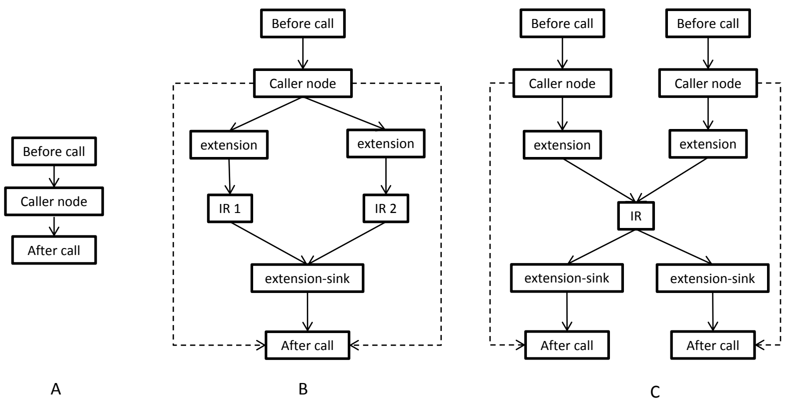


Figure 4 Building IR. Initial IR—the control flow of the caller node has not yet been extended (A). IR after processing the caller node during static analysis. The control flow of the caller node is extended with two IRs—IR 1 and IR 2 (B). Single IR shared between multiple caller nodes (C).

Example 2: Fig. 4-A shows IR after it is initially built. Fig. 4-B shows IR after extending the caller node during the analysis. In this case, the caller is extended with more IRs—this can happen, e.g., if a method is called on an object that can be of more types. Fig. 4-C shows the case when a single IR is shared by multiple callers. △

3.3 Analysis Domain

The states of our abstract domain have the form of $\overline{\text{State}} = \overline{H} \times \overline{V} \times \overline{F}$ where $\overline{H}$ is a state of the heap analysis, $\overline{V}$ is a state of the value analysis, and $\overline{F}$ is a state of the declaration analysis. The heap analysis tracks the shape of the heap and approximates concrete heap locations with heap identifiers (HId), while the value analysis tracks values on heap identifiers. While the heap analysis and value analysis need to interplay, the declaration analysis is independent from both.

Declaration Analysis

Declaration analysis is necessary, because in PHP and other dynamic languages, the names of functions, classes, and constants are bound to concrete definitions during runtime. The analysis thus needs to track these definitions. A state of a declaration analysis $\overline{F}$ is a set of class, function, and constant declarations and lattice operators of the analysis are $\langle \overline{F}, \subseteq, \cup, \cap \rangle$.

Example 3:
Consider the following PHP code:

```
1  if ($_GET[1]) {
2    class A {
3      public $a = 2;
4      function f($p) { return $p + 1;}
5    }
6  } else {
7    class A {
8      public $a = −1;
9      function f($p) { return $p − 1;}
10   }
11 }
12 $x = new A();
13 $y = $x−>f($x−>a); // $y can be −2, 0, 1, 3
```

Since the condition at line (1) is statically unknown, the declaration analysis computes that both declarations of class `A` can be used at line (12). Consequently, the call at line (13) can be done with two possible arguments and has two possible callees resulting in four possible results. △

Heap Analysis

In PHP and other dynamic languages, variables as well as array indices and object properties need not be declared and can be accessed with arbitrary expressions, which can yield statically unknown values. If a specified variable, index, or object property exists, it is overwritten; if not, it is created.

To be able to capture this semantics, the heap analysis approximates arrays, objects, array indices, object fields, and even variables[2] with heap identifiers and the heap analysis can create new heap identifiers both during assignment and join operation. Whenever a new heap identifier is created, it is initialized with an existing heap identifier that stores values from statically unknown assignments to the new identifier that could happen before the creation.

Creation of new heap identifiers corresponds to *materialization* in shape analysis [20]. The *summary* heap identifiers summarize all the heap elements that could be updated by statically unknown assignments and have not been materialized yet[3]. When there is a need to distinguish a heap element from other heap elements summarized by the same summary heap identifier, a new heap identifier is materialized from the summary identifier. This happens, e.g., when an array index is assigned for the first time with a statically known target. In this case, the array index is approximated by the summary heap identifier representing all indices that could be updated only by statically unknown assignments in the pre-state and by the newly materialized heap identifier in the post-state. Materializations are defined as a set of pairs of heap identifiers $\mathrm{Mat} = \mathcal{P}(\mathrm{HId} \times \mathrm{HId})$. The meaning of a single materialization is that the first heap identifier from the pair is materialized from the second, summary heap identifier.

Note that materialization makes the naming scheme of the heap analysis flow-dependent—depending on the program location, a concrete heap element can be approximated by different heap identifiers. This makes an interplay of the heap analysis and the value analysis more challenging. Since the value analysis tracks values on heap identifiers, materializations, which change the naming scheme, need to be applied also to value analysis. Later, we define this

[2] Variables are treated as indices of a special associative array representing the symbol table.

[3] Heap elements that have not been assigned by any assignment are summarized by a special heap identifier `uninit`.

application using the standard abstract interpretation interface of the value analysis. This makes it possible to update the state of the value analysis automatically, without modifying the value analysis ad-hoc. That is, any value analysis that complies with the standard abstract interpretation interface can be used.

Value

Heap id	Value
id	{AnyString}
?	{UndefString}
Users-admin-addr	{admin}
Users-admin-?	{UndefString}
Users-?-name	{UndefString, AnyString}
Users-?-?	{UndefString}
uninit	{UndefString}

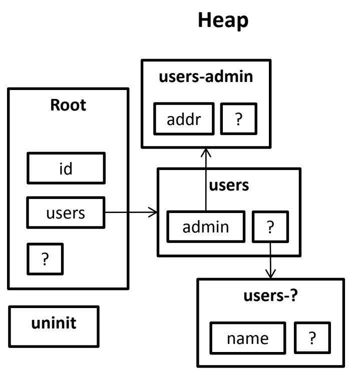


Figure 5 The heap and string part of the value component of the state after the update at line 23 in Fig. 1.

Example 4: Fig. 5 shows the heap and value component of the analysis state after the update at line 23 in Fig. 1. In the following, we use the heap domain developed in [11] and the set domain as a value domain. Note that the value domain tracks values just over these heap identifiers that can contain values. Other heap identifiers are present only in the heap domain.

The heap component of the state contains heap identifier `Root` representing the array corresponding to the symbol table and heap identifier `uninit` representing the uninitialized heap elements. The symbol table array contains three heap identifiers (`id`, `users`, and `?`), which represent program variables (`$id` and `$users`) and statically unknown variables. For heap identifier `id`, value analysis tracks the value `AnyString`, while heap identifier `users` is present only in the heap domain and points to another array. Heap identifier `users-admin` represents index `$users[admin]`, while heap identifier `users-?` represents statically unknown indices of array `$users`. Both heap identifiers represent arrays corresponding to the next dimensions of array `$users`. Finally, heap identifiers `users-admin-addr`, `users-admin-name`, `users-admin-?`, `users-?-name`, and `users-?-?` represent indices of these arrays. Since these heap identifiers store values, they are tracked by the value analysis. △

We assume that heap analysis is provided with lattice operators $\langle \overline{H}, \sqsubseteq_{\overline{H}}, \sqcup_{\overline{H}}, \sqcap_{\overline{H}} \rangle$. Operator $\sqsubseteq_{\overline{H}}$ specifies a partial order, $\sqcup_{\overline{H}}$ is the join operator, and $\sqcap_{\overline{H}}$ is the meet operator. The semantics of heap analysis is given by transfer function $[\![\cdot]\!]_{\overline{H}} : \overline{H} \mapsto \overline{H}$.

Moreover, we assume that the heap analysis provides function read : $AE \mapsto \mathcal{P}(\text{HId})$ for reading data from the heap. The function returns a set of heap identifiers identified by given *access expression*. Access expression is obtained from IR nodes of type variable^V, property-use^V, and index-use^V. In the case of variable^V, access expression is just the set of values, in the case of property-use^V, and index-use^V, it is a sequence of sets of values. Each set from the sequence contains values that can be used to access the corresponding dimension of an array or the corresponding object in the object reference chain. That is, access expressions can represent multi-dimensional updates. This is necessary in order to model semantics of non-decomposable multi-dimensional updates [11].

Example 5: Consider reading the values stored at index `$users[10]['name']` from the state depicted in Fig. 5. The access expression for the index is $\{users\}\ \{10\}\ \{'name'\}$. Calling the read function provided by the heap component with this access expression as argument returns the heap identifier `users-?-name`. This heap identifier is then used to get the resulting values from the value domain. The value domain returns values `UndefString` and `AnyString` meaning that the index `$users[10]['name']` can be uninitialized and can contain statically unknown string value.

Similarly, when reading the index `$users[$_GET[1]]['name']`, the access expression is $\{users\}\{*\}\{'name'\}$, the read function returns heap identifiers `users-?-name` and `users-admin-name`, and the subsequent call to the value domain returns values `UndefString`, `AnyString`, and `'addr'`. △

We additionally assume that the heap analysis provides function joinToValue : $\overline{H} \times \overline{H} \mapsto \mathrm{Mat} \times \mathrm{Mat}$. This function takes the heap parts of analysis states to be joined as arguments and for each joined analysis state, it returns materializations of the heap identifiers created when performing the join operation.

Finally, we assume that heap analysis provides function assignToValue : $\overline{H} \times \mathrm{AE} \mapsto \mathrm{Mat} \times \mathcal{P}(\mathrm{HId}) \times \mathcal{P}(\mathrm{HId})$. This function takes an analysis state before the assignment and the access expression identifying the target of the expression as arguments and returns a triple: (i) materializations of heap identifiers created when performing the assignment, (ii) the heap identifiers representing heap elements that certainly must be updated, and (iii) the heap identifiers representing heap elements that only may be updated.

Value Analysis

The states of the value analysis have a form of $\overline{V} = \overline{V_1} \times \overline{V_2}$ where $\overline{V_1}$ is a state of the value analysis in the first phase and $\overline{V_2}$ is a state of the value analysis in the second phase (end-user analysis). The first phase of the analysis modifies the first component of the state V_1, the second phase of the analysis modifies the second component of the state V_2.

The user of the framework can define both the value analysis in the first phase and the value analysis in the second phase. However, since values that are used to compute control-flow and targets of data accesses are computed in the first phase, the user is more constrained in the first phase.

Second phase

The domain for the second phase tracks information over heap identifiers and it is provided with lattice operators $\langle \overline{V_2}, \sqsubseteq_{\overline{V_2}}, \sqcup_{\overline{V_2}}, \sqcap_{\overline{V_2}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{V_2}} : \overline{V_2} \mapsto \overline{V_2}$, and widening operator $\triangledown_{\overline{V_2}}$.

First phase

In the first phase, the value analysis tracks values of PHP primitive types over heap identifiers:

$$\overline{V_1} = \mathrm{HId} \mapsto \overline{\mathrm{Value}_1}$$
$$\overline{\mathrm{Value}_1} = \overline{\mathrm{Undef}} \times \overline{\mathrm{Null}} \times \overline{\mathrm{Bool}} \times \overline{\mathrm{Num}} \times \overline{\mathrm{String}}$$

Since PHP has dynamic type system—variables, array indices, and object properties do not have declared types, and they can store values of different types depending on context—, $\overline{\mathrm{Value}_1}$ can store values of all primitive types.

To define the value analysis of the first phase, the user of the framework must for each component $\overline{C}$ of $\overline{\text{Value}_1}$ provide the framework with lattice operators $\langle \overline{C}, \sqsubseteq_{\overline{C}}, \sqcup_{\overline{C}}, \sqcap_{\overline{C}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{C}} : \overline{C} \mapsto \overline{C}$, and widening operator $\nabla_{\overline{C}}$. The lattice operators for $\overline{\text{Value}_1}$ are defined component-wise. Moreover, for each pair $(\overline{C_1}, \overline{C_2})$ of components of $\overline{\text{Value}_1}$, functions $\text{Conv}_{\overline{C_1 C_2}} : \overline{C_1} \mapsto \overline{C_2}$ and $\text{Conv}_{\overline{C_2 C_1}} : \overline{C_2} \mapsto \overline{C_1}$ must be provided. These functions are used to model type conversions, which are ubiquitous in dynamic languages.

It should be noted that even though value analysis in the first phase is defined independently of heap analysis, which simplifies its definition, intricate value semantics of PHP makes the definition inherently complex. The framework thus provides default implementations of all components of $\overline{\text{Value}_1}$ including transition functions for PHP native operators and library functions. For the default implementation of the numeric component, we used the interval domain. For the default implementation of the string component, we used the domain based on sets of strings. Its lattice structure is $\langle \mathcal{P}(\text{String}), \subseteq \rangle$ where String are concrete strings. To make the height of the lattice finite and thus guarantee termination, the size of sets is limited by a constant; value AnyString represents the sets of larger sizes.

Example 6: This example illustrates the states of $\overline{\text{Value}_1}$ with the default implementation of its components.

The abstract value $(\bot, \bot, \text{AnyBool}, \bot, \bot)$ represents concrete values `true` and `false` of type Boolean, the abstract value (undef, $\bot, \bot$, true, {"foo", "bar"}) represents the following concrete values: uninitialized value, the Boolean `true`, the string `"foo"`, and the string `"bar"`. △

3.4 Lattice Order and Meet

The lattice order $\sqsubseteq_{\overline{\text{State}}}$ and meet operator $\sqcap_{\overline{\text{State}}}$ for analysis states are defined component-wise:

$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqsubseteq_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) \iff h_1 \sqsubseteq_{\overline{H}} \overline{h_2} \wedge \overline{v_1} \sqsubseteq_{\overline{V}} \overline{v_2} \wedge \overline{f_1} \subseteq \overline{f_2}$$
$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcap_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcap_{\overline{H}} \overline{h_2}, \overline{v_1} \sqcap_{\overline{V}} \overline{v_2}, \overline{f_1} \cap \overline{f_2})$$

3.5 Applying Materializations to Value Analysis

Materializations allow the heap analysis to create new heap identifiers during the assignment and join operations. As discussed in Sect. 3.3, materializations change the naming scheme of the heap analysis; since value analysis tracks values of heap identifiers, these changes must be applied also to the value domain. This is carried out by function applyMaterializations : $(\overline{V} \times \text{Mat}) \mapsto \overline{V}$ that applies materializations to a state of the value analysis:

$$\text{applyMaterializations}(\overline{v}, M) = \overline{v_n} \textit{ where } M = \{(t_1, s_1), (t_2, s_2), ..., (t_n, s_n)\},$$
$$\overline{v_0} = \overline{v}, \forall j \in [1..n] : \overline{v_j} = [\![t_j = s_j]\!]_{\overline{V}}(\overline{v_{j-1}})$$

3.6 Join and Widening

The join of two facts is defined as the set of all facts that are implied independently by any. The join and widening of two states (h_1, v_1, f_1) and (h_2, v_2, f_2) are defined as follows:

$$
\begin{aligned}
&(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcup_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \sqcup_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})\\
&(\overline{h_1}, \overline{v_1}) \triangledown_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \triangledown_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})\\
&(\overline{m_1}, \overline{m_2}) = \text{joinToValue}(\overline{h_1}, \overline{h_2})\\
&\overline{v_1'} = \text{applyMaterializations}(\overline{v_1}, \overline{m_1})\\
&\overline{v_2'} = \text{applyMaterializations}(\overline{v_2}, \overline{m_2})
\end{aligned}
$$

The declaration and heap parts of input states are joined independently on other parts. To perform the join of value parts, heap analysis provides value analysis with materializations of heap identifiers in each joined state. Then, the materializations are applied to the value components of joined states and finally, the updated value parts are joined. Note that the latter two operations are done just by means of standard abstract interpretation interface provided by value analysis.

Example 7: Fig. 6 shows joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. For brevity we omit declaration components.

First, the heap components of analysis states are joined. For the first state to be joined, heap analysis materializes heap identifiers `arr-1-3`, `arr-1-?`, and `arr-?-?` from the heap identifier `uninit` representing undefined heap identifier. That is, there were no statically-unknown assignments that could update the materialized identifiers. Application of materializations to the value component of the first analysis state to be joined thus just adds the materialized identifiers and initializes them with value `UndefString`.

For the second state, heap analysis materializes heap identifiers `arr-1-?`, `arr-1-2`, and `arr-1-3`. Since there was statically-unknown assignment that could update the latter identifier, this identifier is materialized from the identifier `arr-?-3` representing the target of this statically-unknown assignment. Application of materializations to the value component of the second analysis state to be joined thus initializes the identifier `arr-1-3` with values `UndefString` and `'second'`.

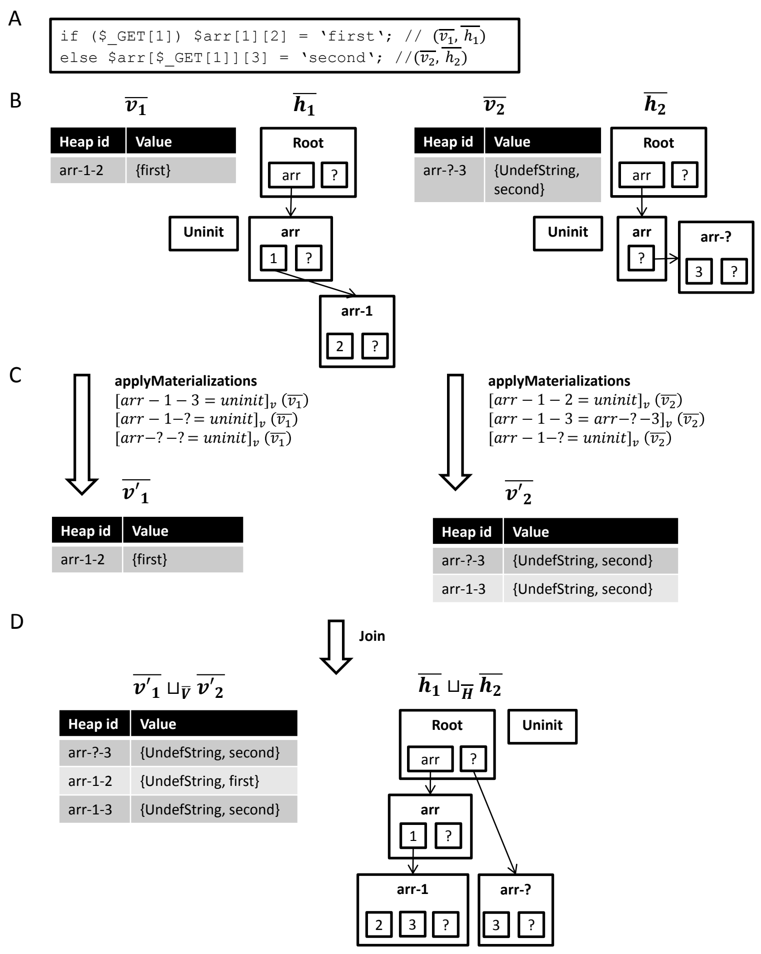
Finally, the value components of analysis states after applying materializations $\overline{v_1'}$ and $\overline{v_2'}$ have the same set of heap identifiers and can be joined independently on the heap components. △

3.7 Transfer Functions

For each kind of node in the intermediate representation, a transfer function maps an abstract state before the node to an abstract state after the node.

We describe the transfer function for the node assign$^V[l^V, r^V]$, where both parameters l^V and r^V are nodes of type variable^V, property-use^V, or index-use^V. Each of these nodes allows getting an access expression, which provides heap analysis information necessary for accessing heap identifiers representing heap locations stored in the node. The access expression for the left-hand side of the assignment l^V is l^V.AE, the access expression for the right-hand side of the assignment r^V is r^V.AE.

To define the transfer function, we first define function strongUpdate : $\overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$. The first parameter is a state of value analysis that is being updated, the



```
if ($_GET[1]) $arr[1][2] = 'first'; // (v1, h1)
else $arr[$_GET[1]][3] = 'second'; //(v2, h2)
```

$\overline{v_1}$

Heap id	Value
arr-1-2	{first}

$\overline{v_2}$

Heap id	Value
arr-?-3	{UndefString, second}

applyMaterializations
$[arr-1-3=uninit]_v\ (\overline{v_1})$
$[arr-1-?=uninit]_v\ (\overline{v_1})$
$[arr-?-?=uninit]_v\ (\overline{v_1})$

applyMaterializations
$[arr-1-2=uninit]_v\ (\overline{v_2})$
$[arr-1-3=arr-?-3]_v\ (\overline{v_2})$
$[arr-1-?=uninit]_v\ (\overline{v_2})$

$\overline{v'_1}$

Heap id	Value
arr-1-2	{first}

$\overline{v'_2}$

Heap id	Value
arr-?-3	{UndefString, second}
arr-1-3	{UndefString, second}

$\overline{v'_1} \sqcup_{\overline{V}} \overline{v'_2}$

Heap id	Value
arr-?-3	{UndefString, second}
arr-1-2	{UndefString, first}
arr-1-3	{UndefString, second}

Figure 6 Joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. The corresponding code (A), value and heap components of joined states (B), applying materializations to value components of states to be joined (C), result of the join (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

second parameter is the set of heap identifiers that are updated, and the last parameter is the set of heap identifiers representing new values:

$$\text{strongUpdate}(\overline{v}, T, S) = \overline{v_n} \ \textit{where}\ T = \{t_1, t_2, ..., t_n\}, \overline{v_0} = \overline{v}$$

$$\forall j \in [1..n] : \overline{v_j} = \bigsqcup_{s \in S} [\![t_j = s]\!]_{\overline{V}}(\overline{v_{j-1}})$$

Next, we define function weakUpdate : $\overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$:

$$\text{weakUpdate}(\overline{v}, T, S) = \bigsqcup_{t \in T, s \in S} \overline{v} \sqcup_{\overline{V}} [\![t = s]\!]_{\overline{V}}(\overline{v})$$

While after strong update, heap identifiers can have just new values, after weak update, they either can have the original values or the new ones [18]. This effect is approximated by joining the analysis state before the update with the analysis state after the update.

The transfer function for updating the state $(\overline{h}, \overline{v})$ with $\text{assign}^V[l^V, r^V]$ is defined as:

$$[\![\text{assign}^V[l^V, r^V]]\!]_{\overline{\text{state}}}(\overline{h}, \overline{v}, \overline{f}) = ([\![l^V.\text{AE} = r^V.\text{AE}]\!]_{\overline{H}}(\overline{h}), \overline{v'''}, \overline{f})$$
$$(m, u_{must}, u_{may}) = \text{assignToValue}(\overline{h}, l^V.\text{AE})$$
$$\overline{v'} = \text{applyMaterializations}(\overline{v}, m)$$
$$\overline{v''} = \text{strongUpdate}(\overline{v'}, u_{must}, \text{read}(\overline{h}, r^V.\text{AE}))$$
$$\overline{v'''} = \text{weakUpdate}(\overline{v''}, u_{may}, \text{read}(\overline{h}, r^V.AE))$$

The transfer function for the heap part of the state is defined by the heap domain itself, and it is not influenced by the value domain. To define the transfer function for the value part of the state, the heap domain provides the value domain with necessary information via function assignToValue. This information consists of: (1) m—information necessary to materialize the heap identifiers that were defined by the assignment (2) heap identifiers representing the heap elements that are certainly targets of the assignment, and (3) heap identifiers representing the heap elements that may be targets of the assignment.

Then, the materializations are applied to the value domain. Finally, the heap identifiers that are certainly targets of the assignment are strongly updated with values of the heap identifiers read from the right-hand side of the assignment, and the heap identifiers that only may be targets of the assignment are weakly updated. The same way as in the case of the join operation, all these updates are performed just by means of the transfer function for the assignment provided by value analysis.

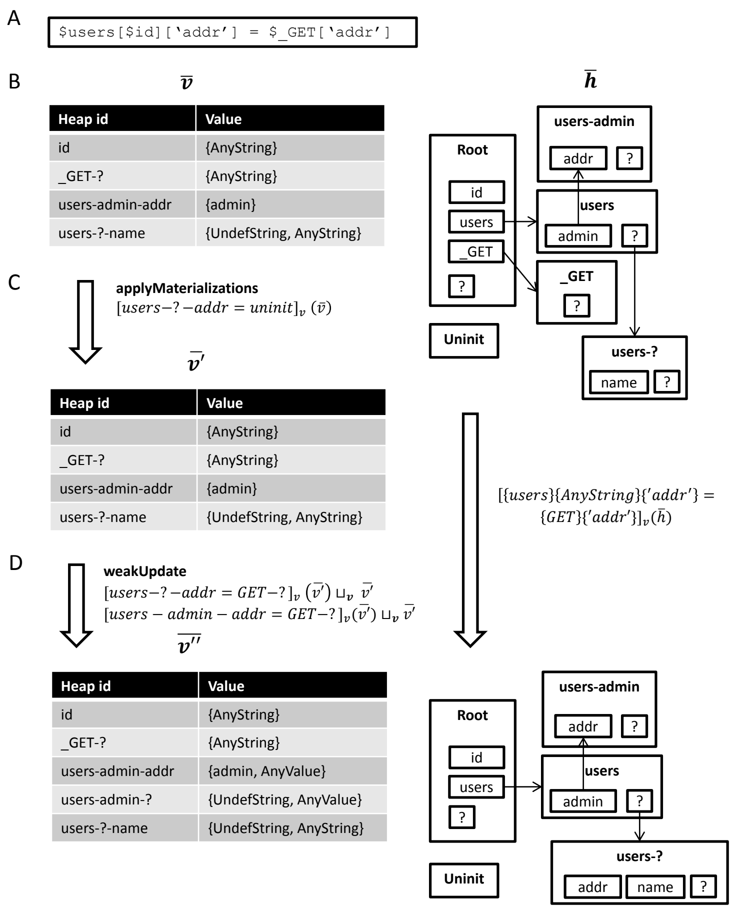
Example 8: Fig. 7 illustrates the transition function for the assignment at line 24 in Fig. 1 (`$users[$id]['addr'] = $_GET['addr']`). First, the access expressions for the source and the target of the assignment are obtained from the corresponding IR nodes. For the source of the assignment, the access expression is $\{\_GET\}\{addr\}$, for the target of the assignment, the access expression is $\{users\}\{AnyString\}\{addr\}$. Note that in the latter case, the value for the second dimension of the access is specified by the variable `$id`.

Second, the access expressions are used to specify the update. During the update, the heap component materializes the heap identifier `users-?-addr` and this change is propagated to the value component via the function applyMaterializations. Note that since there have not been any statically unknown assignments that could update this heap identifier, it is materialized from the identifier `uninit` representing undefined values. That is, the identifier `users-?-addr` is added to the value component and initialized with `UndefString`.

Finally, the heap component specifies that identifiers `users-?-addr` and `users-admin-addr` are weakly updated and the update is propagated to the value component. Since the target of the assignment is not statically known, no heap identifiers are strongly updated. △

3.8 Summary Heap Identifiers

Value analyses are designed to track information on local variables, while we use them to track information on heap identifiers, which can represent many concrete heap locations—summary identifiers [9]. For an example of summary identifiers, consider heap identifiers representing targets of statically unknown assignments and heap identifiers representing a single allocation-site in that many concrete heap locations can be allocated. While value analysis can treat heap identifiers that represent a single heap location exactly the same way as local variables, for summary identifiers, it must take into account that they represent more heap locations.


A
$users[$id]['addr'] = $_GET['addr']
B
$\overline{v}$
$\overline{h}$
Root
id
users
_GET
?
Uninit
users-admin
addr
?
users
admin
?
_GET
?
users-?
name
?
C
applyMaterializations
$[users-?-addr = uninit]_v\ (\overline{v})$
$\overline{v'}$
$[\{users\}\{AnyString\}\{'addr'\} = \{GET\}\{'addr'\}]_v(\overline{h})$
D
weakUpdate
$[users-?-addr = GET-?]_v\ (\overline{v'}) \sqcup_v \overline{v'}$
$[users - admin - addr = GET-?]_v(\overline{v'}) \sqcup_v \overline{v'}$
$\overline{v''}$
Root
id
users
?
Uninit
users-admin
addr
?
users
admin
?
users-?
addr
name
?

Heap id	Value
id	{AnyString}
_GET-?	{AnyString}
users-admin-addr	{admin}
users-?-name	{UndefString, AnyString}

Heap id	Value
id	{AnyString}
_GET-?	{AnyString}
users-admin-addr	{admin}
users-?-name	{UndefString, AnyString}

Heap id	Value
id	{AnyString}
_GET-?	{AnyString}
users-admin-addr	{admin, AnyValue}
users-admin-?	{UndefString, AnyValue}
users-?-name	{UndefString, AnyString}

Figure 7 Transfer function for the assignment. The code of the assignment (A). The value and the heap component $(\overline{v}, \overline{h})$ of the state before the assignment (B). Applying materialization of the identifier `users-?-addr` to the value component of the state (C). The value component $\overline{v''}$ and the heap component of the state after the assignment (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

First, summary heap identifiers must be always weakly updated. In our framework, heap analysis has to take this into account in function assignToValue, which defines identifiers that are weakly and strongly updated by the assignment. This is enough for non-relational value domains—these value domains can otherwise treat summary heap identifiers the same way as local variables.

However, in the case of relational value domains, it is additionally necessary to treat differently assignments from summary heap identifiers. Consider the code:

```
1  $a = $users[$_GET[1]];
2  $b = $users[$_GET[2]];
3  if ($a != $b) {...}
```

Our heap analysis represents both `$users[$_GET[1]]` and `$users[$_GET [2]]` by the same summary heap identifier `users-?`. Our technique would thus abstract the semantics of assignment at line (1) as $[\![a = users-?]\!]_{\overline{V}}$ and the semantics of assignment at line (2) as $[\![b = users-?]\!]_{\overline{V}}$. If v werte a relational domain, the analysis would relate both identifiers `a` and `b` with the summary identifier `users-?` and incorrectly infer that the `if-then` branch can never be reached. This problem was studied by Gopan [9] et. al., who showed that it is wrong to correlate summarized identifiers with non-summarized ones and they proposed a way to extend existing relational domains to deal with this problem.

In our framework, the value domain in the first phase is non-relational and all value domains for end-user analyses that we have implemented so far are also non-relational. To use relational value analyses, these analyses need to be extended to summary dimensions as described by Gopan [9] and the heap analysis has to additionally provide the framework with the information which heap identifiers are summaries.

3.9 Soundness

Our analysis framework allows for defining sound analyses. If the semantics of heap analysis, the semantics of value analysis, and the semantics of declaration analysis plugged into our framework are sound, the semantics of the resulting composed analysis is sound as well. In the following, we will state the fundamental assumptions on value and heap analyses[4]. The traditional soundness argument in abstract interpretation is:

▶ **Definition 1** (Soundness of analysis semantics). Given a set of abstract states $\overline{S}$, abstract semantics $[\![\cdot]\!]_{\overline{S}}$, a set of concrete states S, concrete semantics $[\![\cdot]\!]_S$, and concretization function $\gamma : \overline{S} \mapsto \mathcal{P}(S)$, the abstract semantics $[\![\cdot]\!]_{\overline{S}}$ is sound with respect to the concrete semantics $[\![\cdot]\!]_S$ iff for each statement st and analysis state $\overline{s} \in \overline{S}$ it holds:

$$([\![st]\!]_{\overline{S}}(\overline{s}) = \overline{s}' \wedge [\![st]\!]_S(\gamma(\overline{s})) = s') \implies s' \subseteq \gamma(\overline{s}')$$

Hence, to prove the soundness of the analysis semantics, it is necessary to define the structure of concrete states, their semantics, concretization function, and then prove that it satisfies proposition of Def. 1.

The soundness argument is based on the assumption that the heap semantics and the value semantics are sound. While for the value semantics, the soundness can be specified just using Def. 1 and we can thus use any sound value analysis in our framework, for the heap semantics, we must pose further assumptions.

First, we assume that function read provided by heap analysis complies with the semantics of concrete dereferencing. That is, for each abstract state and each access expression, the heap identifiers returned by function read must represent all concrete locations given by dereferencing using the access expression in all the concretizations of the abstract state.

Next, we assume that the updates given by semantics function assignToValue are sound with respect to the semantics of concrete dereferencing. That is, for the left hand site of assignment, the heap identifiers representing updates given by function assignToValue and an access expression in an abstract state must represent all concrete heap locations R given by dereferencing using the access expression in all the concretizations of the abstract state. Moreover, all the heap identifiers that are in the strong-update set (u_{must}) must exactly represent all the heap locations in set R and all the other heap identifiers that represent the sets of heap locations with non-empty intersection with R must be in the may-update set (u_{may}).

Finally, we assume that the materializations produced by heap analysis are coherent with respect to the modifications of heap analysis. That is, (1) in the post-state, the heap

[4] We will omit the declaration analysis. It needs not interplay with the other analyses and can be treated completely separately.

identifiers that are not sources of materializations must represent the same concrete heap locations as in the pre-state, (2) for each heap identifier that is materialized and its source it must hold that in the post-state each of them represents the subset of the heap locations represented by the source of the materialization in the pre-state, and (3) in the post-state both heap identifiers together must represent all the heap locations represented by the source of the materialization in the pre-state.

Note that we do not require different heap identifiers to represent non-overlapping portions of concrete heap. That is, there can exist two different heap identifiers with *overlapping concretizations*, i.e., there exists a concrete heap location approximated by both heap identifiers. This allows using heap analyses modeling the semantics of assignment by reference more precisely [11]. Consider, e.g., the statement `$a = &$b`. In the concrete semantics, the statement makes `$a` and `$b` pointing to the same heap location. We allow heap analysis to model this concrete location by more heap identifiers with overlapping concretizations—e.g., heap identifier i_1 for variable `$a` and heap identifier i_2 for variable `$b`. To be sound, heap analysis must update these heap identifiers coherently—e.g., if heap identifier i_1 is updated, heap identifier i_2 is updated as well. This is guaranteed by the soundness of updates stated above. Heap identifiers with overlapping concretizations can enable more strong updates. Consider the following example:

```
if ($_GET['INPUT']) $a = &$b;
else $a = &$c;
$a = 1;
```

There are two concrete heap locations in this example. If heap analysis uses less than three heap identifiers to represent these concrete locations, it must perform weak update at line 3. In case of three heap identifiers, their concretizations must overlap; it allows heap analysis to perform strong update on the heap identifier for `$a` and weak updates of those for `$b` and `$c`.

4 Evaluation

To evaluate the precision and scalability of our framework, we used it to implement static taint analysis and we applied it to the NOCC webmail client[5] and a benchmark application comprising of a fragment of the myBloggie weblog system[6], with a total of over 16 kLoC. The benchmark application contains 13 security problems; the number of problems contained in the webmail client is not known.

We compared the results of our analysis with PIXY [15] and PHANTM [17], the state-of-the-art tools for security analysis and error discovery in PHP applications. Both these tools compute control-flow of analyzed applications, model PHP data structures, and perform value analysis. Both these tools detect accesses to uninitialized elements. In addition, PHANTM detects type mismatch errors and PIXY detects taint errors, i.e., flows of sensitive data to critical commands. Our analysis detects both type of errors.

Tab. 2 shows the summary of results. Out of 13 errors in the benchmark application, 8 errors were accesses to uninitialized elements and 5 errors were taint errors. Since PIXY is not designed to detect taint errors we did not use taint errors to assess the error coverage for PIXY. The table shows that the analysis defined using our framework outperforms the other tools both in error coverage and number of false positives when analyzing the

[5] http://nocc.sourceforge.net/
[6] http://mybloggie.mywebland.com/

	myBloggie				**NOCC 1.9.4**			
Lines	648				15,605			
	W	**C**	**F**	**T**	**W**	**C**	**F**	**T**
Our framework	16	100	19	0.9	34	NA	62	84
Pixy	16	69	44	0.6	NA			
Phantm	43	38	93	2.5	426	NA	NA	90

Table 2 Comparison of tools for static analysis of PHP. W/C/F/T: **W**arnings / error **C**overage (in %) / **F**alse-positives rate (in %) / analysis **T**ime (in s).

benchmark application. As to the analysis of NOCC, while Pixy was even not able to analyze the application, Phantm reported a huge number of alarms, which together with a high false-positive rate made its output almost useless[7].

Our analysis discovered all 13 problems in myBloggie. One of the false alarms reported by our analysis is caused by imprecise modeling of the built-in function `date`. Our analysis only models this function by types and deduced that any string value can be returned by this function. However, while the first argument of the function is `"F"`, the function returns only strings corresponding to English names of months. When the value returned by this function is used to access the index of an array, our analysis incorrectly reports that an undefined index of the array can be accessed. Two remaining false alarms are caused by path-insensitivity of the analysis. The sanitization and sink commands are guarded by the same condition, however, there is a joint point between these conditions, which discards the effect of sanitization from the perspective of path-insensitive analysis.

Our analysis found three previously unknown vulnerabilities in the NOCC email client and ten other problems (e.g., calling a function with an argument that is not declared and superfluous implicit conversions). False-positive alarms were caused by imprecise modeling of PHP built-in functions, path-insensitivity of the analysis, and using non-relational value domains.

5 Related Work

The present work builds on a large body of work on static program analysis of dynamic languages. The pioneering works [12, 30, 26, 27] omit modeling some of the important parts of the analyzed languages. The unmodeled parts include references, dynamic accesses to associative arrays, and object-oriented features.

Pixy [16] performs security taint analysis of PHP programs and provides information about the flow of tainted data. Pixy performs a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with literal and alias analysis to achieve precise results. Its main limitations include an incomplete support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, omitting type inference, and a limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

Andromeda static taint analyzer [24] fights the problem of scalability of taint analysis by computing data-flow propagations on demand. It uses forward data-analysis to propagate

[7] Because of a huge number of alarms reported by Phantm, we assessed its false-positives rate just for myBloggie, not for NOCC.

tainted data and ignores propagation of other data. If tainted data are propagated to the heap, it uses backward analysis to compute all targets to which the data should be propagated. Andromeda analyzes Java, .NET, and JavaScript applications. The drawback of the approach is that it propagates only taint information. Especially for dynamic languages, the control-flow of the application can depend on other kind of information which is then not available. To reduce this problem, Andromeda uses F4F [21], which reduces the amount of information that is not known statically.

Phantm [17] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims at detection of type errors. To obtain precise results, Phantm is flow-sensitive, i.e., it is able to handle situations when a single variable can be of different types depending on program location. However, it omits updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

TAJS [14] is a JavaScript static program analysis infrastructure that infers type information. To gain precise results, the analysis is context-sensitive and precisely models intricate semantics of JavaScript, including prototype objects and associative arrays, dynamic accesses to these data structures, and implicit conversions. It tackles the problem that dynamic features of JavaScript make it impossible to construct control-flow before static analysis by constructing control-flow on-the-fly during the analysis. Since TAJS models JavaScript semantics precisely, it has been successfully used to enable additional analyses. In [4, 5], the TAJS program analysis infrastructure is used to build a tool for refactoring JavaScript programs and in [13] TAJS is used to enable technique of statically resolving `eval` constructs. However, TAJS combines heap and value (type) analysis ad-hoc, which results in intricate lattice structure and transfer functions. Next, TAJS assumes that updates to multi-dimensional arrays and objects can be decomposed to updates of length one. While this is true for JavaScript, this assumption leads to loss of precision in the case of some other dynamic languages such as PHP and Perl.

Since the excess of information that are only available at runtime pose a major problem to static analysis, several techniques have been developed that try to enable static analysis of dynamic languages by making this information statically available prior to static analysis. F4F [21] focuses on static taint analysis of web applications that use frameworks. They use a semi-automatically generated specification of framework-related behaviors to reduce the amount of statically-unknown information, which arises, e.g., from reflective calls. Phantm [17] reduces the number of information that static analysis must compute and possibly overapproximate by first executing the application, collecting this information and then invoking static analysis from a particular runtime state. Wei et. al. [28] reduce the number of statically-unknown information in JavaScript by using a technique of blended static analysis [2]. They first execute a test suite and for each test they record its execution trace. Then, for each execution trace, they extract its call graph, types of created objects, and dynamically generated code and perform static analysis of the application with using this information. Finally, they combine solutions from different execution traces into a single solution for the application.

Recently, there has flourished a rich body of work on precise and sound points-to analysis for dynamic languages. Sridharan et. al. [22] present static flow-insensitive points-to analysis for JavaScript modeling objects in JavaScript using associative arrays that can be accessed by arbitrary expressions. To enhance the precision and scalability of the analysis, they identify correlations between dynamic property read and write accesses. If the updated location and stored value can be accessed by the same first class entity (variable), it is extracted to a function parametrized by this entity; this function is then analyzed context-sensitively with the context being the variable. Thus, the correlation between the update and store is preserved. Wei et al. [29] present points-to analysis for JavaScript. Their analysis is partially flow-sensitive—it stores points-to information for every CFG segment with a single state-update statement. Next, their analysis is context-sensitive—to reflect the fact that properties can be added to objects at runtime, it uses a receiver object, its chain of prototype objects, its local properties and their object values as a context. This makes it possible to differentiate between two calls received by the object with the same creation site but different properties. Finally, to perform more strong updates in case of property-writes they use access path edges in their points-to representation. For a property-write (e.g., `$x->p = $y`) where the dereferenced variable (`$x`) points to more objects, weak-updates of properties in these objects (`p`) are performed. However, the access path edge (`<x, p>`) is strongly updated. If the property is read and there exists a corresponding access path edge, it is used instead of points-to edges (for `$z = $x->p` the access path edge `<x, p>` is used and just objects pointed-to by variable `$x` are read). In our previous work [11], we presented points-to analysis for PHP modeling associative arrays that could be accessed using arbitrary expressions. Additionally, our analysis precisely models the semantics of PHP explicit aliases and the semantics of multi-dimensional updates—in PHP or Perl, updates create indices if they do not exist and initialize them with empty arrays if needed; on contrary, read accesses do not.

While heap and value static analyses have been studied mainly as orthogonal problems, to support verification of real programs, they usually need to be combined together [6, 25]. Since in dynamic languages, data structures can be dynamically accessed with arbitrary expressions, this problem of combining heap and static value analysis is particularly relevant in this domain.

Clousot [3] preprocesses the program applying heap analysis, and uses a value numbering algorithm to compute under-approximation of must-alias to replace heap accesses with heap identifiers. Value analysis then tracks values of variables and also of the heap identifiers. While the approach allows for using arbitrary value analysis, it only allows for using specific heap analysis, which cannot use the information provided by value analysis, and the technique is not sound.

Miné et. al. [19] combine type based pointer analysis and numeric value analyses in a generic way. The pointer analysis models pointer arithmetic, union types and records of stack variables in C programs. The general limitation of this technique is that it relies on type based heap analysis, which is too coarse for many applications. In particular, their technique does not support summary nodes and dynamic allocation.

Fu [8] combines numeric value analysis and points-to analysis. His method uses points-to analysis to partition possibly infinite set of heap references into a finite set of abstract locations (heap identifiers) and use value analysis to track values of variables and also of heap identifiers. The method is both generic—it allows for reusing existing analyses as black-boxes—and automatic—it does not require any annotations specific to a particular heap and value analysis to be provided. The fundamental limitation of the technique is that it relies on flow-independent naming scheme for points-to analysis. That is, a concrete reference is always mapped to the same abstract location independently of program location. On one hand, this assumption allows the technique to assume that change of the heap component of the analysis state has no effect on the value component of the state and that two states can be joined component-wise. On the other hand, this assumption poses a substantial limitation to modeling of adding new object fields and array indices using statically-unknown updates.

To illustrate the limitation, consider that a statically-unknown index of an empty array `$a` is updated (`$a[rand()]=..`). At this point, points-to analysis must represent all concrete indices of the array with a single abstract location h. Next, if a concrete index of the array, e.g., `$a[1]`, is updated (`$a[1]=..`), the analysis must still represent the index `$a[1]` with h and thus cannot distinguish this index from other indices in `$a`. That is, a statically unknown update makes the updated array (object) index-insensitive (field-insensitive) for all indices (fields) added after the update. As both modeling statically-unknown updates and field-sensitivity of heap analysis is crucial for static analysis of dynamic languages [23, 29], the assumption of flow-independent naming scheme is too limiting in this context.

Ferrara [6] introduced the concept of substitutions overcoming the limitation of flow-indepenent naming scheme when combining value and heap analysis. Substitutions allow heap analysis to materialize and summarize abstract locations, i.e., to replace a single abstract location in the pre-state with more abstract locations in the post-state and to replace more abstract locations in the pre-state with a single abstract location in the post-state. Ferrara defined how the analyses are composed when the substitutions are given and showed the assumptions on the heap and the value analyses in order to make their composition sound. However, his work cannot be directly applied in the context of dynamic languages. First, it does not model dynamically added fields and indices to objects and arrays, which is essential for dynamic languages. Then, Ferrara allows only heap analyses with non-overlapping heap identifiers. As we explain in Section 3.9, some heap analyses developed for dynamic languages use overlapping heap identifiers to perform more strong updates. Moreover, his work does not allow heap analyses to explicitly specify which updates are strong and which updates are weak thus reducing the precision of the composed analysis. In our work we focused specifically on heap analyses for dynamic languages overcoming these limitations.

6 Conclusion

In this paper, we presented a framework for static analysis of dynamic languages, in particular PHP applications.

The framework employs a two-phase analysis architecture—in the first phase, the dynamic constructs present in the analyzed code are resolved, while the analysis in the second phase can proceed in a way similar to a one for a language without dynamic features. This way, the framework provides a developer with high-level API for implementing various kind of analyses upon the code without the need to cope with dynamic features of the language. To allow resolving dynamic features, the framework combines heap, value, and declaration analyses. We described the necessary requirements on these analyses and the way these analyses are composed together generically and soundly. That is, our framework allows for combining various heap and value analyses while guaranteeing that if the analyses being composed are sound, the composed analysis is sound as well.

The framework is provided with default implementations of heap analyses and first phase analyses. To demonstrate usefulness of our framework, we implemented taint analysis of PHP application and applied it on real PHP application. We have shown that the tool is able to reveal real (previously unknown) security holes, while producing less false-positive alarms comparing to other state-of-the-art tools.

As for future work, we aim at improving the performance and precision of the analyzes provided by the framework especially in terms of scaling to large applications. In particular, this includes the scalability improvements of the heap analysis, implementation of more choices of context-sensitivity, and devising precise modeling of more library functions. Next, we plan to enhance our implementation of security analysis and use the framework for implementing additional end-user analyses.

References

1 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

2 Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07*, pages 118–128. ACM, 2007.

3 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS '10*, LNCS, pages 10–30. Springer-Verlag, 2011.

4 Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *OOPSLA '11*, pages 119–138. ACM, 2011.

5 Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *OOPSLA '13*, pages 323–338. ACM, 2013.

6 Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI'05*, LNCS, pages 302–321. Springer-Verlag, 2014.

7 Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, May 2007.

8 Zhoulai Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In *VMCAI '05*, LNCS, pages 282–301. Springer-Verlag, 2014.

9 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS '04*, LNCS, pages 512–529. Springer-Verlag, 2004.

10 David Hauzar and Jan Kofroň. Weverca. `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`, 2014.

11 David Hauzar, Jan Kofroň, and Pavel Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS '14*, EPTCS, pages 56–70. Open Publishing Association, 2014.

12 Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04*, pages 40–52. ACM, 2004.

13 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA 2012*, pages 34–44. ACM, 2012.

14 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS'09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

15 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

16 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06*, pages 258–263. IEEE Computer Society, 2006.

17 Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *RV'10*, LNCS, pages 300–314. Springer-Verlag, 2010.

18 Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16, New York, NY, USA, 2011. ACM.

19 Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES '06*, pages 54–63. ACM, 2006.

20 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

21 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA '11*, pages 1053–1068. ACM, 2011.

22 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.

23 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12*, LNCS, pages 435–458. Springer-Verlag, 2012.

24 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE'13*, LNCS, pages 210–225. Springer-Verlag, 2013.

25 Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE 2005*, LNCS, pages 227–236. Springer-Verlag, 2005.

26 Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07*, pages 32–41. ACM, 2007.

27 Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08*, pages 171–180. ACM, 2008.

28 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*, pages 336–346. ACM, 2013.

29 Shiyi Wei and BarbaraG. Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *ECOOP 2014*, volume 8586 of *LNCS*, pages 1–26. Springer Berlin Heidelberg, 2014.

30 Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*. USENIX Association, 2006.

CHAPTER 10

WeVerca: Web Applications Verification for PHP

Authors: David Hauzar and Jan Kofroň

[1] *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM'14), LNCS, DOI: 10.1007/978-3-319-10431-7_24, Grenoble, France, September 2014*

WeVerca: Web Applications Verification for PHP (Tool Paper)[⋆]

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract. Static analysis of web applications developed in dynamic languages is a challenging yet very important task. In this paper, we present WeVerca, a framework that allows one to define static analyses of PHP applications. It supports dynamic type system, dynamic method calls, dynamic data structures, etc. These common features of dynamic languages cause implementation of static analyses to be either imprecise or overly complex. Our framework addresses this problem by defining end-user static analyses independently of value and heap analyses necessary just to resolve these features. As our results show, taint analysis defined using the framework found more real problems and reduced the number of false positives comparing to existing state-of-the-art analysis tools for PHP.

1 Introduction

PHP is the most common programming language used at the server side of web applications. It is notably used, e.g., by Wikipedia and Facebook. PHP as well as other dynamic languages contains dynamic features, such as dynamic type system, dynamic method calls (names of called methods are computed at run-time), and dynamic data structures (names of object fields are computed at run-time and object fields can be added at run-time). These features provide flexibility accelerating the development. However, they make applications more error-prone and less efficient. Consequently, they shift more work to tools for error detection, code refactoring, and code optimization.

For most of these tools, static program analysis is a necessary prerequisite. Unfortunately, dynamic features pose major challenges here. To precisely resolve these features, the end-user analysis (e.g., taint analysis) needs to be combined with value and heap analyses. Importantly, these analyses must interplay. To resolve dynamic accesses to data structures, the heap analysis needs to evaluate value expressions and the value analysis must track values over heap elements—array indices and object fields.

In this paper we present WeVerca[1], an open-source static analysis framework for PHP. WeVerca allows to define end-user static analyses independently

⋆ This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by Charles University institutional funding SVV-2014-260100.

[1] `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`

of dynamic features. This is possible because: (1) WeVerca defines an interplay of value and heap analyses allowing to define these analyses independently of each other. (2) WeVerca comes with default implementations of context-sensitive heap and value analyses that model associative arrays and prototype objects, track values of PHP primitive types, and model library functions, native operators, and type conversions. (3) WeVerca defines how information from heap and value analyses are used to resolve dynamic features (i.e., to compute control-flow and resolve dynamic data accesses). As a proof of the concept, we implemented static taint analysis for detection of security problems.

2 Example

As an example, consider static taint analysis, which is commonly used for web applications. It can be used for detection of security problems, e.g., SQL injection and cross-site scripting attacks. The program point that reads user-input, session ids, cookies, or any other data that can be manipulated by a potential attacker is called *source*, while a program point that prints out data, queries a database, etc. is referred to as *sink*. Data at a given program point are *tainted* if they can pass from a source to this program point. A tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potential malicious parts of it. Program is *vulnerable* if it contains a sink that uses data that are tainted and not sanitized.

Static taint analysis can be performed by computing the propagation of tainted data and then checking whether tainted data can reach a sink. The propagation of tainted data computed by forward data-flow analysis is shown in Tab. 1[2]. The analysis is specified by giving the lattice of data-flow facts, the initial values of variables, the transfer function, and the join operator.

Lattice	L	$true$	
Top	$\top$	Bool	
Initial value	$init(v)$	$true$	if $v \in$ `$_SESSION` $\cup$ ..
		$false$	otherwise
Transfer function	$TF(LHS = RHS)$	$var = \bigvee_{r \in RHS} r$	if $var \in LHS$
		$var = var$	otherwise
	$TF(n)$	$var = var$	if n is not assignment
Join operator	$\sqcup(x, y)$	$x \vee y$	

Table 1. Propagation of tainted data.

Consider now the code in Fig. 1. At lines (1)–(9) classes for processing the output are defined. They can either log the output or show the output to the user. While the `Templ1` class uses a *sink* command to show the output, `Templ2` uses a *non-sink* command (e.g., does not send the output to the browser directly, but sanitizes it first). At lines (13)–(16) the application mode is set based on the value of `DEBUG` either to `log`—the application will log the output—or to `show`—the application will show the output to the user. At lines (17)–(20) the skin is set based on user input. At line (21), the array `$users` is initialized with

[2] For simplicity we omit the specification of sanitization.

the address of administrator. This value is not taken from any source and can be directly shown to the user. Note the update at line (11) is correct even if the variable `$users` is uninitialized. In PHP, if a non existing index is updated, it is automatically created and if the update involves next dimension, the index is initialized with an empty array. At lines (23)–(24) information about the user name and user address is assigned to the array `$users`. Note that this information is tainted. Finally, at lines (25)–(26) data are processed to the output.

```
1   class Templ {
2     function log($msg) {...}
3   }
4   class Templ1 : Templ {
5     function show($msg) { sink($msg); }
6   }
7   class Templ2 : Templ {
8     function show($msg) { not_sink($msg); }
9   }
10  function initialize(&$users) {
11    $users['admin']['addr'] =
              get_admin_addr_from_db();
12  }
13  switch (DEBUG) {
14    case true: $mode = "log"; break;
15    default: $mode = "show";
16  }
17  switch ($_GET['skin']) {
18    case 'skin1': $t = new Templ1(); break;
19    default: $t = new Templ2();
20  }
21  initialize($users);
22  $id = $_GET['userId'];
23  $users[$id]['name'] = $_GET['name'];
24  $users[$id]['addr'] = $_GET['addr'];
25  $t−>$mode($users[$id]['name']);
26  $t−>$mode($users['admin']['addr']);
```

Fig. 1. Running example

The code contains two vulnerabilities. At lines (25) and (26) the method `show` of `Templ1` can be called, its parameter `$msg` can be tainted and the parameter goes to the sink. Taint analysis defined using WeVerca detects both vulnerabilities. Note that the definition of taint propagation uses just the information in Tab 1. This is possible only because WeVerca automatically resolves control-flow and accesses to built-in data structures. That is, WeVerca computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$mode` can contain values `show` and `log`. Based on this information, it resolves calls at lines (25) and (26). Moreover, as WeVerca automatically reads the data from and updates the data to associative arrays and objects, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads tainted data.

3 Tool description

The architecture of WeVerca is shown in Fig. 2. For parsing PHP sources and providing abstract syntax tree (AST) WeVerca uses Phalanger[3]. The analysis is split into two phases. In the first phase, the framework computes control-flow of the analyzed program together with the shape of the heap and information about values of variables, array indices and object fields. Then it also evaluates expressions used for accessing data. The control-flow is captured in the intermediate representation (IR), while the other information is stored in the data representation. IR defines the order of instructions' execution and has function calls, method calls, includes, and exceptions already resolved. In the

[3] `http://www.php-compiler.net/`

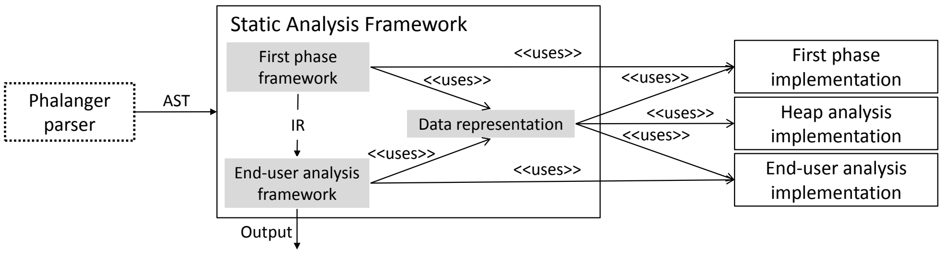

Static Analysis Framework
First phase framework
IR
End-user analysis framework
Data representation
<<uses>>
Phalanger parser
AST
Output
First phase implementation
Heap analysis implementation
End-user analysis implementation

Fig. 2. The architecture of WeVerca

second phase, end-user analyses of the constructed IR are performed. The tool includes the following parts:

- **Data Representation** stores analysis states and allows to access them—it allows to read values from data structures, write values to data structures, and modify the shape of data structures. Next, it performs join and widening of the states and defines their partial order. Importantly, data representation defines the interplay of heap and value analyses allowing each analysis to define these operations independently. WeVerca contains implementation of heap analysis described in [1]. It supports associative arrays and objects of an arbitrary depth (in PHP, updates create indices and properties if they do not exist and initialize them with empty arrays and empty objects if needed; on contrary, read accesses do not, so updates of such structures cannot be decomposed). Accesses to these structures can be made using an arbitrary expression yielding even statically unknown values.
- **First-phase implementation** must define value analysis that tracks values of PHP primitive types and evaluates value expressions. Next, it must handle declaration of functions, classes, and constants. Finally, it must compute targets of include statements and function and method calls, and it must define context sensitivity. WeVerca contains a default implementation of the first phase providing fully context-sensitive value analysis precisely modeling native operators, native functions, and implicit conversions.
- **End-user analyses** can be specified using an arbitrary value domain. This is possible because (1) control-flow is already computed, (2) the shape of the heap is computed and dynamic data accesses are resolved—all information that data representation needs to discover accessed variables, indices, and fields are available. (3) Data representation combines heap and value analyses automatically, i.e., to perform operations with analysis states, it uses standard operations of combined analyses. The framework contains an implementation of static taint analysis as a proof-of-the-concept.

4 Results

To evaluate the precision and scalability of the framework, we used the framework to implement static taint analysis and we applied it to a NOCC webmail client[4] and a benchmark application comprising of a fragment of the myBlog-

[4] http://nocc.sourceforge.net/

gie weblog system[5], with a total of over 16,000 lines of PHP code. While the benchmark application contains 13 security problems, in the case of the webmail client, the number of problems is not known.

Tab. 2 shows the summary of results together with the results of PIXY [3] and PHANTM [4], the state-of-the-art tools for security analysis and error discovery in PHP applications. The table shows that the analysis defined using WEVERCA outperforms the other tools both in error coverage and number of false positives when analyzing the benchmark application. While it took WEVERCA more than 5 minutes to analyze the webmail client and 52 alarms were reported, PIXY was even not able to analyze this application. PHANTM analyzed the application in two minutes, however, the false-positive rate of 93% makes its output almost useless.

Out of 13 problems in the benchmark application, WEVERCA discovered all of them. One of the false alarms reported by WEVERCA is caused by imprecise modeling of the built-in function `date`. WEVERCA only models this function by types and deduced that any string value can be returned by this function. However, while the first argument of the function is `"F"`, the function returns only strings corresponding to English names of months. When the value returned by this function is used to access the index of an array, WEVERCA incorrectly reports that an undefined index of the array can be accessed. Two remaining false alarms are caused by path-insensitivity of the analysis. The sanitization and sink commands are guarded by the same condition, however, there is a joint point between these conditions, which discards the effect of sanitization from the perspective of path-insensitive analysis. While the first false-alarm can be easily resolved by modeling the built-in function more precisely, the remaining false alarms would require more work. One can either implement an appropriate relational abstract domain or devise a method of path-sensitive validation of alarms.

	Lines	**WeVerca** W/C/F/T	**Pixy** W/C/F/T	**Phantm** W/C/F/T
myBloggie	648	16/**100**/**19**/2.2	16/69/44/**0.6**	43/23/93/2.5
NOCC 1.9.4	15605	52/NA/NA/332	NA	426/NA/NA/**130**

Table 2. Comparison of tools for static analysis of PHP. W/C/F/T: **W**arnings / error **C**overage (in %) / **F**alse-positives rate (in %) / analysis **T**ime (in s). The best results are in bold.

5 Related work

The existing work on static analysis of PHP and other dynamic languages is primarily focused on specific security vulnerabilities and type analysis.

Pixy [3] performs taint analysis of PHP programs and it provides information about the flow of tainted data using dependence graphs. It involves a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with

[5] http://mybloggie.mywebland.com/

literal and alias analysis to achieve precise results. The main limitations of Pixy include limited support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, omitting type inference, and limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

Phantm [4] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims at detection of type errors. To obtain precise results, Phantm is flow-sensitive, i.e., it is able to handle situations when a single variable can be of different types depending on program location. However, they omit updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

TAJS [2] is a JavaScript static program analysis infrastructure. To gain precise results, it models prototype objects and associative arrays, dynamic accesses to these data structures, and implicit conversions. However, TAJS combines combines heap and value analysis ad-hoc, which results in intricate lattice structure and transfer functions.

6 Conclusion and future work

In this paper, we presented WeVerca, a framework for static analysis of PHP applications. WeVerca makes it possible to define static analyses independently of dynamic features, such as dynamic includes, dynamic method calls, and dynamic data accesses to associative arrays and objects. These features are automatically resolved using information from heap and value analyses, which are automatically combined.

Our prototype implementation of static taint analysis outperforms state-of-the-art tools for analysis of PHP applications both in error coverage and the false-positive rate. We believe that WeVerca can accelerate both the development of end-user static analysis tools and the research of static analysis of PHP and dynamic languages in general.

For future work, we plan to improve the scalability and precision of analyses provided by the framework. In particular, this includes the scalability improvements of data representation, implementation of more choices of context-sensitivity, more precise widening operators, and devising precise modeling of more library functions.

References

1. D. Hauzar, J. Kofroň, and P. Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS'14*, EPTCS, 2014.
2. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS'09*. Springer-Verlag, 2009.
3. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *S&P'06*. IEEE, 2006.
4. E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP Analyzer for Type Mismatch. In *FSE'10*. ACM, 2010.

CHAPTER 11

On Interpolants and Variable Assignments

Authors: Pavel Jančík, Jan Kofroň, Simone Fulvio Rollini, and Natasha Sharygina

[5] *Proceedings of Formal Methods in Computer-Aided Design 2014, DOI: 10.1109/FMCAD.2014.6987604, Lausanne, Switzerland, October 2014*

On Interpolants and Variable Assignments

Pavel Jancik, Jan Kofroň
Faculty of Mathematics and Physics,
Charles University, Prague, Czech Republic
Email: name.surname@d3s.mff.cuni.cz

Simone Fulvio Rollini, Natasha Sharygina
Faculty of Informatics,
University of Lugano, Switzerland,
Email: name.surname@usi.ch

***Abstract*—Craig interpolants are widely used in program verification as a means of abstraction. In this paper, we (i) introduce *Partial Variable Assignment Interpolants (PVAIs)* as a generalization of Craig interpolants. A variable assignment focuses computed interpolants by restricting the set of clauses taken into account during interpolation. PVAIs can be for example employed in the context of DAG interpolation, in order to prevent unwanted out-of-scope variables to appear in interpolants. Furthermore, we (ii) present a way to compute PVAIs for propositional logic based on an extension of the Labeled Interpolation Systems, and (iii) analyze the strength of computed interpolants and prove the conditions under which they have the path interpolation property.**

I. Introduction

In software model checking Craig interpolants play an important role. They are typically used to refine an abstraction of a program. Many techniques have been introduced to compute interpolants for various theories such as propositional logic, conjunctive fragments of linear arithmetic, and octagon domain. For propositional logic, McMillan's [9] and Pudlák's [11] interpolation systems are well established; they are generalized by the Labeled Interpolation Systems [6] (LISs), which permit to systematically compute interpolants of different logical strength from the same refutation.

Given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig interpolant is a formula I such that A implies I, I is inconsistent with B and I is defined over the common variables of A and B. In other words, I is an over-approximation of A (which can be used to abstract the behavior of a system, represented by A) disjoint from B (which often represents unacceptable behaviors).

In this paper, we introduce *Partial Variable Assignment Interpolants (PVAIs)* – a generalization of Craig interpolants – which, in addition to the standard subdivision of an unsatisfiable formula (the *interpolation problem*) into A and B, is parametric in a *partial variable assignment (PVA)*. A PVA defines a *sub-problem* on which a PVAI is focused. A sub-problem is obtained from the interpolation problem by removing the clauses (constraints) satisfied by the assignment. Due to the specialization, (1) it is possible to restrict the variables occurring in an interpolant to those relevant to the sub-problem, i.e. those shared between the A and B parts of the sub-problem. Moreover, since the irrelevant constraints (those not occurring in the sub-problem) need not be considered by interpolation, (2) the interpolants for the sub-problem can be of smaller size, compared to Craig interpolants computed from the interpolation problem.

This work is partially supported by: ICT COST Action IC0901, the Grant Agency of the Czech Republic project 14-11384S, and Charles University Foundation grant 203-10/253297.

In the motivating example in Sec. II we show how PVAIs apply to program verification. For instance, in the context of abstract reachability graphs (ARG) (and DAG interpolation [2]), an interpolation problem is the encoding of a whole ARG (representing all paths in the ARG), while for a given ARG node i the related sub-problem represents the set of paths that pass through that node. An over-approximation of the states reachable at i via these paths (a *node interpolant*) can be computed by means of a PVAI. Properties of PVAIs guarantee that the interpolant contains only in-scope program variables.

An alternative approach could be to solve each sub-problem separately, which involves calling a SAT/SMT solver for each sub-problem and applying standard Craig interpolation. The method we propose allows one to perform just a single call to a solver for an interpolation problem which encompasses all the sub-problems, thus (i) processing the parts common to multiple sub-problems only once. A single solver call results in a single proof from which all the interpolants for the sub-problems are computed. The presence of a single proof, in turn, enables (ii) generating collections of interpolants which satisfy properties relevant to verification, such as path interpolation [7], [13]. Such collections are hard to obtain if multiple proofs are involved. In the case of PVAIs, a collection may consist of the interpolants associated with different sub-problems.

We also propose the new framework of *Labeled Partial Assignment Interpolation Systems* (LPAISs) – a generalization of LISs, which computes PVAIs for propositional logic. We define the notion of logical strength for LPAISs and show how introducing a partial order over LPAISs allows to systematically compare the strength of the computed interpolants (a feature intuitively relevant to verification since it affects the coarseness of the over-approximations realized by interpolants [12]). We also show how LPAISs can be used to generate collections of interpolants which enjoy the path interpolation (inductive step) property. These results can be applied in the context of ARGs, where the path interpolation property of computed node interpolants (labels) guarantees well-labeledness [10] of the ARG.

```
1: int max(int i, int j) {
2:   if (i > j)
3:     return i;
   else
4:     return j;
5: }
   // The main function
6: assert(max(random(), 0) >= 0);
```

Figure 1. Motivating example

$$\begin{tikzcd}
& 1 \arrow[d, "\tau_{12} \equiv j = 0"] & \\
& 2 \arrow[dl, "\tau_{23} \equiv i > j"'] \arrow[dr, "\tau_{24} \equiv \neg(i > j)"] & \\
3 \arrow[dr, "\tau_{35} \equiv result = i"'] & & 4 \arrow[dl, "\tau_{45} \equiv result = j"] \\
& 5 \arrow[d, "\tau_{56} \equiv \neg(result >= 0)"] & \\
& 6 &
\end{tikzcd}$$

Figure 2. Abstract reachablity graph

$$\begin{aligned}
\mu_1 &\equiv (n_1 \Rightarrow n_2) && \wedge ((n_1 \wedge n_2) \Rightarrow \tau_{12})\\
\mu_2 &\equiv (n_2 \Rightarrow (n_3 \vee n_4)) && \wedge ((n_2 \wedge n_3) \Rightarrow \tau_{23}) \wedge\\
& && \wedge ((n_2 \wedge n_4) \Rightarrow \tau_{24})\\
\mu_3 &\equiv (n_3 \Rightarrow n_5) && \wedge ((n_3 \wedge n_5) \Rightarrow \tau_{35})\\
\mu_4 &\equiv (n_4 \Rightarrow n_5) && \wedge ((n_4 \wedge n_5) \Rightarrow \tau_{45})\\
\mu_5 &\equiv (n_5 \Rightarrow n_6) && \wedge ((n_5 \wedge n_6) \Rightarrow \tau_{56})
\end{aligned}$$

$$\mathsf{Cond} \equiv n_1 \wedge \mu_1 \wedge \mu_2 \wedge \mu_3 \wedge \mu_4 \wedge \mu_5$$

Figure 3. The Cond formula

II. Motivation

In the following, we illustrate a possible application of PVAIs, which originally motivated this work; nonetheless, the proposed PVAIs are not limited to this context. As an example, consider the source code on the left-hand side of Fig. 1 and the corresponding ARG in Fig. 2. Node i is associated with location i in the program. Node 1 is the initial node, while node 6 is the node representing an error location. The *edge constraints* τ_{ij} encode the semantics of the corresponding program statements. Note that τ_{12} originates from the call to the `max` function in `main`, on line 6. Further, in node 3, the parameter i is the only in-scope variable; similarly in node 4 the parameter j is the only in-scope variable. A variable is in-scope at a given node, if there is a path through the node where the variable is used before as well as after the node.

In the context of software verification, an important question is whether an error location is actually reachable from the initial location of a program – this is known as the *reachability problem*. The question can be answered by computing, for each node i, the set of states reachable at i via paths in the program ARG [4], [10]. Typically, it is enough to compute an over-approximation of these states, i.e. a *node interpolant*. To this end, the ARG is converted into a Cond formula[1], which represents all execution paths in the ARG. An auxiliary *structure-encoding* Boolean variable n_i is introduced for each node i in the ARG; for each i (except for the error node), a *node formula* μ_i is created, which encodes the labels on the outgoing edges (Fig. 3).

For illustration, we describe the meaning of μ_2. The first conjunct $n_2 \Rightarrow (n_3 \vee n_4)$ expresses that after reaching node 2, a path has to proceed to a successor node (3 or 4). The second conjunct $(n_2 \wedge n_3) \Rightarrow \tau_{23}$ guarantees that if a path goes via the edge $2 \rightarrow 3$, the semantics of the edge is preserved (i.e., the constraint τ_{23} is satisfied). Similarly, the third conjunct enforces the semantics of the edge $2 \rightarrow 4$.

The Cond formula is satisfiable if and only if a feasible path exists that leads from node 1 to node 6 in the ARG. Suppose now that Cond is unsatisfiable; then a node interpolant for each node i can be computed. First the ARG needs to be partitioned into A and B – so that A corresponds to the antecedents of i, B to all the other nodes in the ARG – and then a Craig interpolant I is generated as an over-approximation of the states reachable at i. For instance, in the case of node 3, A would be set to $n_1 \wedge \mu_1 \wedge \mu_2$ and B to $\mu_3 \wedge \mu_4 \wedge \mu_5$. However, employing standard Craig interpolation in this manner to compute a node interpolant I is not sufficient; out-of-scope variables might in fact belong to both A and B, they could therefore appear in I, and should be consequently eliminated. Variable j, for example, could appear in the interpolant for node 3. Even though out-of-scope variables can be eliminated by resorting to quantification, followed by a quantifier-elimination phase, this approach is a well-known bottleneck in verification.

[1]Cond has the same meaning as ArgCond in [3].

Computing node interpolants using PVAIs effectively solves the problem of out-of-scope program variables. Suppose that a node interpolant is to be computed for a node k; the created PVA assigns False to all structure-encoding variables corresponding to nodes not lying on the paths through k. By setting a variable n_j to False, in fact, the paths via node j are blocked; moreover, the whole node formula μ_j is satisfied and thus μ_j is not a part of the sub-problem for node k. On the other hand, the PVA assigns n_k to True to express that each considered path has to pass through k (the node for which the interpolant is computed). In particular, to compute an interpolant for node 3, we assign n_3 to True and n_4 to False to block the path through node 4; the rest of variables remain unassigned. This assignment satisfies (and thus removes) $n_2 \Rightarrow (n_3 \vee n_4)$, $(n_2 \wedge n_4) \Rightarrow \tau_{24}$ and μ_4 from the sub-problem (see Fig. 4). In the A part, the sub-problem for node 3 contains the edge labels (and consequently the program state variables) related to the path from node 1 to node 3, and in the B part information related to the path from node 3 to node 6. The program state variables shared by the A and B parts of the sub-problem are the in-scope variables, which are exactly those that may appear in PVA interpolants.

III. Preliminaries

A *clause* is a finite disjunction of literals. We use angle brackets $\langle\Theta\rangle$ to denote the clause built over the literals in Θ. Let $\langle\Theta, p\rangle$ and $\langle\Theta', \overline{p}\rangle$ be clauses. Using variable p as the *pivot*, their resolution yields the clause $\langle\Theta, \Theta'\rangle$. In the following, we consider propositional formulas in *conjunctive normal form*,

$$\begin{aligned}
\pi_3 &\equiv n_3 \wedge \overline{n_4}\\
A_3 &\equiv n_1 \wedge\\
&\quad (n_1 \Rightarrow n_2) \wedge ((n_1 \wedge n_2) \Rightarrow j = 0) \wedge\\
&\quad \wedge ((n_2 \wedge n_3) \Rightarrow i > j)\\
B_3 &\equiv (n_3 \Rightarrow n_5) \wedge ((n_3 \wedge n_5) \Rightarrow result = i) \wedge\\
&\quad (n_5 \Rightarrow n_6) \wedge ((n_5 \wedge n_6) \Rightarrow \neg(result >= 0))
\end{aligned}$$

Figure 4. The A and B parts of the sub-problem for node 3

i.e., as conjunctions (or equivalently sets) of clauses. We use $\mathsf{Var}(l)$ to denote the variable of literal l and $\mathsf{Var}(A)$ for the variables occurring in the set of clauses A.

We adopt the definition of resolution proof from [6]: a resolution proof is a tuple (V, E, cl, piv, s), where V is a set of vertices in the proof, E is a set of edges. Each inner vertex v represents resolution of its antecedent vertex-clauses (specified by cl) using the pivot $piv(v)$. A refutation proof derives an empty clause in the sink vertex s.

Since the resolution proofs take the set of clauses as input, the input formula is first converted into a conjunction of clauses. Thus in the following we use the terms formula and set of clauses interchangeably.

A *Craig interpolant* [5] for the pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable is a formula I such that (1) $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$.

An *interpolant sequence* for the unsatisfiable formula $A_1 \wedge A_2 \wedge ... \wedge A_n$ is a tuple of formulas $(I_0, I_1, ...I_n)$, where I_i is an interpolant for $(A_1 \wedge ... \wedge A_i, A_{i+1} \wedge ... \wedge A_n)$. If for all i, $I_i \wedge A_i \Rightarrow I_{i+1}$, then $(I_0, I_1, ...I_n)$ is said to satisfy the *path interpolation* (PI) property. In [7], it was proved that the path interpolation property holds for any LISs, including the well-known McMillan's and Pudlák's systems, whenever the interpolant sequence is computed from the same proof.

Let A be a set of clauses. A *variable assignment* assigns either True ($\top$) or False ($\bot$) to each variable in the $\mathsf{Var}(A)$ set. The variable assignment can be seen as a conjunction of literals. A *partial variable assignment* (PVA) π assigns values only to a subset of variables in $\mathsf{Var}(A)$. A PVA π can be used as an assumption w.r.t. A (i.e., $\pi \models A$) to restrict the set of models of A to those compatible with π.

Definition 1 (Clauses under assignment): Let A be a set of clauses and π be a PVA over $\mathsf{Var}(A)$. We define the sets of *satisfied* clauses $A_\pi = \{\langle\Theta\rangle | \langle\Theta\rangle \in A \text{ and } \pi \models \langle\Theta\rangle\}$ and *unsatisfied* clauses $A_{\overline{\pi}} = \{\langle\Theta\rangle | \langle\Theta\rangle \in A \text{ and } \pi \not\models \langle\Theta\rangle\}$.

Satisfied clauses contain at least one literal evaluated to $\top$ under π, while, for unsatisfied clauses, every literal is either unassigned or falsified. The unsatisfied clauses $A_{\overline{\pi}}$ determine the sub-problem. We use $\pi \models l$ to express that a literal l evaluates to $\top$ in a given PVA π.

IV. Partial Variable Assignment Interpolants

In this section, we formally define *partial variable assignment interpolation*, which, in addition to the subdivision of an unsatisfiable formula into A and a B parts, requires specification of a PVA.

Definition 2: Let R be an (A, B)-refutation and π a partial variable assignment over $\mathsf{Var}(A \wedge B)$. A *partial variable assignment interpolant* (PVAI) is a formula I such that:

(D2.1) $\pi \models A \Rightarrow I$
(D2.2) $\pi \models B \wedge I \Rightarrow \bot$
(D2.3) $\mathsf{Var}(I) \subseteq \mathsf{Var}(A_{\overline{\pi}}) \cap \mathsf{Var}(B_{\overline{\pi}})$
(D2.4) $\mathsf{Var}(I) \cap \mathsf{Var}(\pi) = \emptyset$

In the following we use (A, B, π) to denote that a PVAI is computed from an (A, B)-refutation using the partial assignment π.

Since $\pi \models (A \Leftrightarrow A_{\overline{\pi}})$, D2.1 and D2.2 can be equivalently rewritten as $\pi \models A_{\overline{\pi}} \Rightarrow I$ and $\pi \models B_{\overline{\pi}} \wedge I \Rightarrow \bot$; in other words, I is an interpolant for the sub-problem $(A_{\overline{\pi}} \wedge B_{\overline{\pi}})$. Note that even after removing (the satisfied) clauses, the sub-problem remains unsatisfiable (assuming π).

On the other hand, a PVAI cannot be obtained from standard interpolants by application of a partial assignment ($I[\pi]$). The reason is that, in addition to assigned variables (disallowed by D2.4), rule D2.3 excludes from the PVAI also all unassigned (out-of-scope) variables that occur in satisfied clauses only, which can still appear in $I[\pi]$.

Calling a solver multiple times can be quite resource-consuming. An (A, B)-refutation proof is independent of a PVA; this important fact allows to call the solver only once on the overall problem $A \wedge B$, and later to introduce various PVAs (representing relevant sub-problems) for which the PVAI can be efficiently computed.

Although Craig interpolation has many applications in program verification, verification tools often require interpolation sequences with specific properties [7]. The PVAI for all the sub-problems are computed from the same proof, thus they are related to each other. The existence of a single proof permits the application of a standard proving technique in the area of interpolation – structural induction over a refutation proof – to show various properties of PVA interpolant sequences. All the techniques where interpolants for different sub-problems are computed using different proofs (e.g., applying a solver directly on each sub-problem, or incremental solving with assumptions) do not, per se, guarantee any properties of their sequences.

V. Labeled Partial Assignment Interpolation System

To show that PVAIs are not just a theoretical concept, we present the framework of *Labeled Partial Assignment Interpolation Systems*, a generalization of LISs [6], which computes PVAIs for propositional logic, and prove its soundness. Next, in order to prove the path interpolation property, we introduce the concept of logical strength on LPAISs, which allows one to systematically compare the strength of the generated interpolants.

In order to define LPAISs, first we have to extend the definitions of labeling functions and locality from LISs to take variable assignments into account. Note that if no variable is assigned, LPAISs are equivalent to LISs.

A labeling function assigns labels to literals in a refutation; the labeling drives the computation of an interpolant from the proof and determines its strength.

Definition 3 (Labeling function): Let $L = (S, \sqsubseteq, \sqcap, \sqcup)$ be the lattice of Fig. 6, where $S = \{\bot, a, b, ab, d^+\}$ and $\bot$ is the least element, and let $R = (V, E, cl, piv, s)$ be a resolution proof over a set of literals Lit. A function $\mathsf{Lab}_{R,L} : V \times \mathsf{Lit} \rightarrow S$

Leaf v:	$\langle\Theta\rangle, [I]$	
$I = \langle\Theta\rangle[\pi]\rrbracket_{b,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in A_{\overline{\pi}}$	Hyp-$A_{\overline{\pi}}$
$I = \neg\langle\Theta\rangle[\pi]\rrbracket_{a,v,\mathsf{Lab}}$	if $\langle\Theta\rangle \in B_{\overline{\pi}}$	Hyp-$B_{\overline{\pi}}$
$I = \top$	if $\langle\Theta\rangle \in A_\pi \cup B_\pi$	Hyp-A_π, Hyp-B_π

Inner vertex v:	$\dfrac{v_1 : \langle p, \Theta_1\rangle, [I_1] \qquad v_2 : \langle \overline{p}, \Theta_2\rangle, [I_2]}{\langle\Theta_1, \Theta_2\rangle, [I]}$	
$I = I_1 \vee I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$	Res-a
$I = I_1 \wedge I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$	Res-b
$I = (I_1 \vee p) \wedge (I_2 \vee \overline{p})$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$	Res-ab
$I = I_2$	if $\mathsf{Lab}(v_1, p) = d^+$	Res-d^+
$I = I_1$	if $\mathsf{Lab}(v_2, \overline{p}) = d^+$	Res-d^+

Figure 5. Labeled Partial Assignment Interpolation System

$$\begin{tikzcd}
 & d^+ \arrow[d, dash] & \\
 & ab \arrow[dl, dash] \arrow[dr, dash] & \\
a \arrow[dr, dash] & & b \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Figure 6. Lattice of labels (according to $\sqsubseteq$)

is called *labeling function* for a refutation R iff $\forall v \in V$ and $\forall l \in \mathsf{Lit}, \mathsf{Lab}_{R,L}$ satisfies the following conditions:

(D3.1) $\mathsf{Lab}_{R,L}(v, l) = \bot$ if and only if $l \notin cl(v)$, and

(D3.2) $\mathsf{Lab}_{R,L}(v, l) = \mathsf{Lab}_{R,L}(v_1, l) \sqcup \mathsf{Lab}_{R,L}(v_2, l)$, where v_1, v_2 are the predecessor vertices.

From condition D3.2 it follows that the labeling function is fully determined once the labels in the leaves have been specified. We omit subscripts R and L if clear from the context.

Naming conventions: Let us assume a pair of sets of clauses (A, B) and a PVA π. The clause sets are split into four groups, the unsatisfied clauses $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$ which specify the sub-problem and are taken into account during interpolation, and the satisfied clauses A_π and B_π, which are disregarded.

We distinguish among the following kinds of variables, depending on the standard notions of locality and sharedness, as well as on where the variables appear in the four groups of clauses. We say that a variable k is *unassigned* if $k \notin \mathsf{Var}(\pi)$. An unassigned variable k is:

$A_{\overline{\pi}}$-local	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$
$B_{\overline{\pi}}$-local	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$-shared	if $k \in \mathsf{Var}(A_{\overline{\pi}})$ and $k \in \mathsf{Var}(B_{\overline{\pi}})$
$A_{\overline{\pi}}B_{\overline{\pi}}$-clean	if $k \notin \mathsf{Var}(A_{\overline{\pi}})$ and $k \notin \mathsf{Var}(B_{\overline{\pi}})$

The properties above are independent of the occurrence of k in $\mathsf{Var}(A_\pi)$ and $\mathsf{Var}(B_\pi)$. The "clean" variables occur only in the satisfied clauses, thus are out-of-scope and cannot appear in a PVA interpolant.

We say that a variable k is *McMillan-labeled* if, whenever k is $A_{\overline{\pi}}B_{\overline{\pi}}$-shared or $A_{\overline{\pi}}B_{\overline{\pi}}$-clean, k is labeled b (the labels of the remaining variables are not limited to b). If all variables are McMillan-labeled, a LIS reduces to McMillan's interpolation system [6], which yields the strongest interpolant that LISs (and LPAISs) can produce from a given refutation proof.

A variable k is labeled *consistently* if all occurrences of k in a refutation have the same label.

Not all labeling functions can be used to generate interpolants; in LPAIS, interpolants are computed if a locality preserving labeling is used.

Definition 4: A labeling function Lab for an (A, B, π)-refutation R is *locality preserving* iff $\forall v \in V, \forall l \in cl(v)$:

(D4.1) $\mathsf{Lab}(v, l) = d^+ \Leftrightarrow \pi \models l$

(D4.2) $\mathsf{Var}(l)$ is unassigned and $A_{\overline{\pi}}$-local $\Rightarrow \mathsf{Lab}(v, l) = a$

(D4.3) $\mathsf{Var}(l)$ is unassigned and $B_{\overline{\pi}}$-local $\Rightarrow \mathsf{Lab}(v, l) = b$

(D4.4) $\mathsf{Var}(l)$ is unassigned and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean $\Rightarrow$ it is consistently labeled a or b.

Locality constraints provide freedom in labeling $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables; the choice of labels directly affects the strength of the computed interpolants. The label of $A_{\overline{\pi}}B_{\overline{\pi}}$-shared variables can be set freely to a, b, or ab. The same holds for falsified literals; their labels are irrelevant since they are removed by the assignment filter (defined below).

The D4.2 and D4.3 rules are equivalent to the locality requirements of LIS, where A-local and B-local variables must be labeled a and b, respectively. D4.1 concerns the satisfied literals. The label d^+ is used in the interpolation process to identify resolutions with an assigned pivot and parts of the proof which are not relevant to the sub-problem. The D4.4 requirement is specific to PVAI and deals with variables which occur in the satisfied clauses only. The requirement guarantees that such variables do not occur in the interpolant, because ab-resolution cannot be applied. Further, note that for the empty assignment the locality constraints reduce to those of LISs, since D4.1 and D4.4 do not apply to any literal.

Filters: For a clause $\langle\Theta\rangle$, a labeling function Lab, a resolution-proof vertex $v \in V$, and a label c, we define the *match filter* $|$ as $\langle\Theta\rangle|_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c = \mathsf{Lab}(v, l)\}$; it preserves only the literals with the specified label. Similarly, we define the *upward filter* $\upharpoonleft$ as $\langle\Theta\rangle\upharpoonleft_{c,v,\mathsf{Lab}} = \{l \in \langle\Theta\rangle \mid c \sqsubseteq \mathsf{Lab}(v, l)\}$; it preserves the literals with labels above c in Fig. 6. The subscripts Lab, v are omitted if clear from the context. Given a partial assignment π and a clause $\langle\Theta\rangle$, we also define the *assignment filter* $\langle\Theta\rangle[\pi] = \{l \in \langle\Theta\rangle \mid \mathsf{Var}(l) \notin \mathsf{Var}(\pi))\}$, which removes all the assigned literals (satisfied and falsified ones).

Moreover, we assume that filters have a higher precedence than negation. E.g., $\neg\langle\Theta\rangle[\pi]\upharpoonleft_a$ can be equivalently rewritten as $\neg((\langle\Theta\rangle[\pi])\upharpoonleft_a)$.

An interpolation system is a procedure for computing an interpolant from a refutation. It assigns a partial *vertex-*

interpolant to each vertex of the refutation, yielding the final interpolant at the sink vertex.

Definition 5: For a locality preserving labeling function Lab and an $(A,\ B,\ \pi)$-refutation R, Fig. 5 defines the *Labeled Partial Assignment Interpolation System* $\mathsf{Lpaitp}(\mathsf{Lab}, R)$.

An LPAIS produces interpolants in the following way: first the vertex-interpolants for leaves of the refutation proof are computed using the rules in the upper part of Fig. 5 (hypothesis rules). Depending on the occurrence of the vertex-clause $\langle\Theta\rangle$ in A or B sets, the corresponding rule describes the transformation of the vertex-clause into a vertex-interpolant. Later, going down through the proof from leaves to the sink, the vertex-interpolants for inner vertices are computed using rules in the lower part of Fig. 5. The labels assigned to the pivots determine how vertex-interpolants of both predecessors are combined. This process ends at the sink vertex where the PVAI is derived. The interpolants are computed in time linear to the size of the proof.

The main difference compared to LISs are the additional d^+ rules. For instance, consider the last rule, where $\mathsf{Lab}(v_2, \overline{p}) = d^+$. In contrast to the standard rules, the partial interpolant is simpler, because it does not contain I_2, omitted due to the variable assignment. Generally, these rules *cut out* the satisfied sub-tree of the proof. Usually, the later in the refutation the assigned variable is resolved, the larger sub-tree is pruned and the smaller the resulting interpolant is.

The differences between LPAISs and LISs are motivated by the way variable assignments work. The new d^+ rules can be seen as a specialization of the ab resolution rule if a PVA π is assumed. A similar relationship holds for the hypothesis rules in the leaves of a refutation. These rules are equivalent to LIS hypothesis rules if applied on a clause under the assumed assignment. The changes we introduce w.r.t. LISs are of two kinds: those in LPAISs rules force specialization of the interpolant on a sub-problem, while the changes in the locality constraints remove unassigned out-of-scope variables from the interpolant.

Theorem 1 (Correctness): $\mathsf{Lpaitp}(\mathsf{Lab}, R)$, for an $(A,\ B,\ \pi)$-refutation R and a locality preserving labeling function Lab, generates a partial variable assignment interpolant.

Proof sketch: By structural induction over R we show that, for each vertex v of a resolution proof, the following invariants hold:

$$\pi \models A \wedge \neg\langle\Theta\rangle{\upharpoonright_{a,v,\mathsf{Lab}}} \Rightarrow I_v$$
$$\pi \models B \wedge \neg\langle\Theta\rangle{\upharpoonright_{b,v,\mathsf{Lab}}} \Rightarrow \neg I_v$$

I_v is the partial vertex-interpolant and $\langle\Theta\rangle$ is a vertex-clause of v. These invariants yield the PVAI constraints (D2.1, D2.2) at the sink vertex, where $\neg\langle\Theta\rangle = \top$. The full proof can be found in [8]. □

The attentive reader may notice that the locality constraints, as well as the way LPAISs compute interpolants, are *symmetric* for the A_π and B_π sets of satisfied clauses. It reflects the fact that these clauses are not a part of the sub-problem under consideration, thus irrelevant for PVAI interpolants. Given a fixed π, the satisfied clauses can be moved freely between the A and B sets; both computed interpolants and locality of the labeling functions are not affected if satisfied clauses are moved. This fact allows us to articulate the *strength* theorem in an elegant way.

A. Strength

Interpolation systems based on labeling provide some freedom in the choice of labels (e.g., for shared variables); this choice affects the resulting interpolants, in particular their strength. In the following we investigate this relationship in more detail.

$$\begin{tikzcd}
b \arrow[d, dash] \\
ab = d^+ \arrow[d, dash] \\
a \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 7. Strength ordering ($\preceq$)

Definition 6 (Strength order): Let $\preceq$ be a pre-order relation defined on the set of labels $S = \{\bot, a, b, ab,\ d^+\}$ as: $b \preceq ab = d^+ \preceq a \preceq \bot$ (see Fig. 7). Let Lab and Lab' be labeling functions for a refutation R. We say Lab *is stronger than* Lab', denoted as $\mathsf{Lab} \preceq \mathsf{Lab}'$, if for all vertices $v \in V$ and for all literals $l \in cl(v)$ it holds that $\mathsf{Lab}(v,l) \preceq \mathsf{Lab}'(v,l)$.

Note that labels ab and d^+ are of the same strength and can be exchanged if the locality requirements permit; b is the strongest label, while a is the weakest one a literal can get.

The following theorem states that the introduced strength order on labeling functions also orders the produced interpolants by logical strength.

Theorem 2 (Interpolant strength): Let Lab be a locality preserving labeling function for an $(A,\ B,\ \pi)$-refutation R, and Lab' be a locality preserving labeling function for $(A,\ B,\ \pi')$-R. Let I be a partial variable assignment interpolant for $\mathsf{Lpaitp}(\mathsf{Lab}, R)$ and I' be a PVAI for $\mathsf{Lpaitp}(\mathsf{Lab}', R)$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi, \pi' \models I \Rightarrow I'$.

Note that when π and π' are *empty* assignments, we obtain exactly the theorem on interpolant strength from [6]. Also note that the theorem permits different variable assignments for the interpolants. Thus it relates the interpolants generated for different sub-problems (e.g., interpolants considering different sets of paths through a given ARG node). Since both π and π' are assumptions of the formula $I \Rightarrow I'$, the theorem applies to cases common to both sub-problems (i.e., to the shared paths). Both interpolants (I and I') have to be computed using the same A and B parts, thus interpolants for different ARG nodes cannot be compared using this theorem; a generalization in this direction is shown in the following sub-section.

In the following proof, we need a new type of filter. Let Lab and Lab' be labeling functions to be compared by strength and v be a vertex of the refutation proof. The new *weakened-labels filter* $\Vert_v^{\mathsf{Lab},\mathsf{Lab}'}$ preserves the literals whose label is weaker in Lab' than in Lab. E.g., the filter preserves a literal l if the strongest labels b ($\mathsf{Lab}(v,l) = b$) is weakened into label a or

ab in $\mathsf{Lab}'(v,l)$, while it filters-out a literal if both functions assign label a to it. The vertex and the labeling functions are omitted if clear from the context.

Proof sketch (Theorem 2): By structural induction over R, we show that for each vertex of the resolution proof the following invariant holds:

$$\pi, \pi' \models I_v \wedge \neg\langle\Theta\rangle|\downarrow_v \Rightarrow I'_v$$

$\langle\Theta\rangle$ is the vertex-clause, I_v and I'_v are the partial vertex-interpolants for the vertex v as generated by our interpolation system using the labeling functions Lab and Lab', respectively. The full proof in [8] shows that the invariant holds for all combinations of rules that can be used to define the vertex-interpolants I_v and I'_v. □

Similarly to LISs, for a fixed variable assignment there is a lattice of LPAISs ordered according to the strength of labeling functions. The top element of the lattice involves the strongest labeling function, which assigns label b to $A_{\overline{\pi}}B_{\overline{\pi}}$-shared and $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables, while the labeling function of the bottom element assigns label a to them. Theorem 2 claims that LPAISs produce interpolants ordered by strength according to the lattice.

B. Path interpolation property

Several verification approaches such as [3], [10], [14] depend on the *path interpolation* property (PI). In [13] the authors show that LISs can be employed to generate path interpolants by providing a sequence of labeling functions that are decreasing in terms of strength. In this subsection we study conditions for labeling functions that have to be satisfied in order to guarantee the PI property of interpolant sequences generated by LPAISs.

First, we show that the PI property holds if the same partial assignment along a sequence is used to compute the interpolants (i.e., considering the same set of paths at different ARG nodes). Later on, we generalize the result to permit different partial assignments for particular interpolants (i.e., relating node interpolants).

Fixed PVA: To show the PI property, it is enough to prove that, for any consecutive interpolants in the sequence, it holds: $I \wedge S \Rightarrow I'$, where I is an interpolant for $(A, S \cup B, \pi)$, I' is an interpolant for $(A \cup S, B, \pi)$, and S is a set of clauses.

For LISs, [13] defines a set of *labeling constraints* on the labeling functions used to compute the interpolants I and I'; if the labeling constraints are satisfied, the interpolants have the PI property. However, we prove the PI property in another way, more suitable for LPAISs. Given a labeling function to compute the interpolant I, we define the strongest labeling function which can be used to compute the successor interpolant I'.

Definition 7: Let Lab be a labeling function for an $(A, S \cup B, \pi)$-refutation R. The *strongest successor labeling* function Lab^S (for the set S) is defined in Fig. 8.

It is easy to see that Lab^S is a valid labeling function and that if Lab is locality preserving, then Lab^S is locality preserving for $(A \cup S, B, \pi)$. Hence, Lab^S can be used to compute an interpolant for $(A \cup S, B, \pi)$.

The first alternative (D7.1) forces label a for all literals which become $(A_{\overline{\pi}} \cup S_{\overline{\pi}})$-local due to the shift of the clauses in S from the B to the A part. Any locality preserving function Lab' has to also assign the label a to these literals. So, it is easy to see that if $\mathsf{Lab} \preceq \mathsf{Lab}'$ then also $\mathsf{Lab}^S \preceq \mathsf{Lab}'$. This expresses the meaning of *strongest*. Moreover, $\mathsf{Lab} \preceq \mathsf{Lab}^S$, because either the labels are equal or the weakest label a is used in the labeling Lab^S.

The following lemma states the PI property for the strongest successor labeling.

Lemma 1: Let Lab be a locality preserving labeling function for an $(A,\ S \cup B,\ \pi)$-refutation R and let $\mathsf{Lpaltp}(\mathsf{Lab}, R) = I$. Let Lab^S be the strongest successor labeling for Lab and S, and $\mathsf{Lpaltp}(\mathsf{Lab}^S,\ (A \cup S, B, \pi)) = I'$.

Then $\pi \models I \wedge S \Rightarrow I'$.

Proof sketch: By structural induction over R, we show that for each vertex v of the resolution proof the following invariant holds:

$$\pi \models I_v \wedge S \wedge \neg\langle\Theta\rangle|\downarrow_v \Rightarrow I'_v$$

$\langle\Theta\rangle$ is the vertex-clause, I_v and I'_v are the partial vertex-interpolants for the vertex v as generated by our interpolation system using the labeling functions Lab and Lab^S, respectively. The full proof can be found in [8]. □

Lemma 1 guarantees the PI property only if the sequence of the strongest successors labeling functions is used. Below we generalize this result in such a way that the strength of the labeling function can decrease along the sequence; Theorem 3 states the main result for a fixed partial assignment – the path interpolation property.

Theorem 3: Let Lab and Lab' be locality preserving labeling functions for an $(A,\ S \cup B,\ \pi)$-refutation R and $(A \cup S,\ B,\ \pi)$-R, respectively. Let $\mathsf{Lpaltp}(\mathsf{Lab}, R) = I$ and $\mathsf{Lpaltp}(\mathsf{Lab}', R) = I'$.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi \models I \wedge S \Rightarrow I'$.

Proof: Let I^S be the partial variable interpolant for the strongest successor labeling function Lab^S. From Lemma 1 it holds that $\pi \models I \wedge S \Rightarrow I^S$. As shown above $\mathsf{Lab}^S \preceq \mathsf{Lab}'$; so Theorem 2 can be applied and $\pi \models I^S \Rightarrow I'$. □

The result in this case is the same as for LISs. In the following we focus on the case when different PVAs are used, and the situation becomes more challenging.

Different PVAs: The goal to prove when different partial assignments π and π' are used to compute interpolants I and I' (respectively) is:

$$\pi, \pi' \models I \wedge S \Rightarrow I'$$

Looking back at the motivating example, for each node in the ARG a different partial variable assignment is typically used; thus, the generalization done in this section is needed to relate the interpolants of adjacent ARG nodes. Assume node

$$\mathsf{Lab}^S(v,l) = \begin{cases} a & \text{if } \mathsf{Var}(l) \in \mathsf{Var}(S_{\overline{\pi}}) \ \wedge\ \mathsf{Var}(l) \notin \mathsf{Var}(B_{\overline{\pi}}) \ \wedge\ \mathsf{Var}(l) \notin \mathsf{Var}(\pi) & (D7.1) \\ \mathsf{Lab}(v,l) & \text{otherwise} & (D7.2) \end{cases}$$

Figure 8. Strongest successor labeling function

interpolants I_2 for node 2 and I_3 for node 3. The desired property is then $I_2 \wedge \tau_{23} \Rightarrow I_3$ (well-labeledness in the context of ARGs [3], [10]), which follows from the aforementioned goal. In Theorem 4, we work out the conditions the labeling functions (for I_2 and I_3) have to satisfy so that the interpolants have the desired property.

Assignments: Having two different PVAs π and π', the expression (π, π') represents the PVA formed by the union of π and π'. We say that a PVA σ is an *extension* of a PVA π, if $\sigma \Rightarrow \pi$ (viewing the PVAs as conjunctions of literals). In other words, σ can be created from π by assigning additional variables. In case of conflicting π and π' (assigning one $\top$ and the other $\bot$ to a particular variable), the goal above holds trivially and therefore we omit the case from now on.

Definition 8: We say that the variable is *assignable* if it is McMillan-labeled and not $A_{\overline{\pi}}$-local.

Each assignable variable must have label b, therefore, after assigning it, its label becomes weaker. The following theorem states the main result for different PVAs.

Theorem 4: Let Lab be a locality preserving labeling function for an $(A, S \cup B, \pi)$-refutation R and let $I = \mathsf{Lpaltp}(\mathsf{Lab}, (A, S \cup B, \pi))$. Let Lab' be a locality preserving labeling function for $(A \cup S, B, \pi')$-R and let $I' = \mathsf{Lpaltp}(\mathsf{Lab}', (A \cup S, B, \pi'))$.

Suppose that (i) $A_{\overline{\pi}} \subseteq A_{\overline{\pi'}}$, (ii) $B_{\overline{\pi'}} \subseteq B_{\overline{\pi}}$, (iii) the variables assigned by π' and not by π are assignable in Lab, and (iv) the variables assigned by π and not by π' are not $B_{\overline{\pi'}}$-local.

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then it holds $\pi, \pi' \models I \wedge S \Rightarrow I'$.

Intuitively, the constraints (i) and (ii) prevent from comparing interpolants of unrelated sub-problems. The only way to violate the constraint (i) $A_{\overline{\pi}} \subseteq A_{\overline{\pi'}}$ is to assign a new variable by π'. In terms of ARGs, it means that π' blocks some paths in addition to those blocked by π. The interpolant I over-approximates the states reachable in the corresponding node via non-blocked paths in the A part. If the assignment π' blocks some paths related to I' in addition to those blocked by π, then I' may not cover (over-approximate) the states coming from the blocked paths, thus it may be not implied by I. A similar reasoning can be used for (ii).

Proof sketch: The overall idea of the proof is shown in Fig. 9. The proof consists of four simpler steps. In the first step (① $\rightarrow$ ②) new variables get assigned by π', in the second step (② $\rightarrow$ ③) the clauses of S are moved. In the third step (③ $\rightarrow$ ④) the assignment π is removed, in the last step (④ $\rightarrow$ ⑤) the labeling function is weakened. In the second line of Fig. 9, it is expressed how the interpolation problem is divided into A and B parts and which PVA is used. In all but the second step the division into A and B parts does not change, thus Theorem 2 can be used to relate particular interpolants with each other via implications; in the second step the partial variable assignment does not change, so Theorem 3 is utilized.

To be able to apply this scheme (Theorems 2 and 3), locality preserving labeling functions of decreasing strength are needed. The third line of Fig. 9 specifies a labeling function for each step. The idea of the approach is similar to the one used for fixed variable assignments. In each step, we create the strongest possible labeling function; in particular for the first step (① $\rightarrow$ ②) we create an *extended-assignment labeling* function ($\mathsf{Lab}^{+}_{\pi \rightarrow (\pi,\pi')}$) – the strongest locality-preserving labeling function if new variables get assigned. For the second step (② $\rightarrow$ ③) we use the strongest successor labeling function as defined in Def. 7. For the third step (③ $\rightarrow$ ④) we create a *restricted-assignment labeling* function ($\mathsf{Lab}^{-}_{(\pi,\pi') \rightarrow \pi'}$) – the strongest locality-preserving labeling function if variables get unassigned. For the sake of space, we skip the definitions of the aforementioned labeling functions and proofs of the required properties; they can be found in [8].

Via the above construction we create the strongest locality-preserving labeling function ($\mathsf{Lab}^{-}_{(\pi,\pi') \rightarrow \pi'}$) for $(A \cup S, B, \pi')$ which satisfies $\mathsf{Lab} \preceq \mathsf{Lab}^{-}_{(\pi,\pi') \rightarrow \pi'}$. In the last step (④ $\rightarrow$ ⑤) we decrease the strength into Lab', in the same way as it is done for Lab^S in Theorem 3.

The last line of Fig. 9 shows how the interpolants in each step are related to each other and how the overall claim of this theorem follows from the particular steps. □

C. Application to ARGs

While the locality constraints are simple to satisfy for a single interpolant, the situation becomes more complicated if several interpolants need to be related by the path interpolation property. In such a case, the labels of the literals have to be chosen in an appropriate way. In the following, we briefly discuss how to set labels for ARG nodes (using the same encoding as in our motivating example) to apply Theorem 4 and, thus, to obtain well-labeled node interpolants.

Recall that in ARGs there are two kinds of variables – (1) *structure encoding* (n_i), which can be assigned, and (2) program variables, which are not assigned. The first rule is that the structure encoding variables have to be McMillan-labeled (obtaining the strongest possible labels). This rule and the properties of ARG encoding are enough to satisfy the (i)–(iv) requirements of Theorem 4.

Only the last requirement – $\mathsf{Lab}_i \preceq \mathsf{Lab}_j$ – restricts also the labels for program variables. It is easily satisfied in ARGs by a quite simple general rule: once an $A_{\overline{\pi}}B_{\overline{\pi}}$-shared or an $A_{\overline{\pi}}B_{\overline{\pi}}$-clean literal gets a label weaker than the strongest label b at a node, the same or a weaker label has to be assigned at all its successor nodes, until it becomes $A_{\overline{\pi}}$-local.

$$\begin{array}{ccccccccccc}
 & \textcircled{1} & \rightarrow & \textcircled{2} & \rightarrow & \textcircled{3} & \rightarrow & \textcircled{4} & \rightarrow & \textcircled{5} \\
 & (A, S \cup B, \pi) & & (A, S \cup B, (\pi, \pi')) & & (A \cup S, B, (\pi, \pi')) & & (A \cup S, B, \pi') & & (A \cup S, B, \pi') \\
 & \mathsf{Lab} & \preceq & \mathsf{Lab}^{+}_{\pi \rightarrow (\pi,\pi')} & \preceq & \mathsf{Lab}^{S}_{(\pi,\pi')} & \preceq & \mathsf{Lab}^{-}_{(\pi,\pi') \rightarrow \pi'} & \preceq & \mathsf{Lab}' \\
\pi, \pi' \models & I \wedge S & \overset{T2}{\Longrightarrow} & I^{+} \wedge S & \overset{T3}{\Longrightarrow} & I^{S} & \overset{T2}{\Longrightarrow} & I^{-} & \overset{T2}{\Longrightarrow} & I'
\end{array}$$

Figure 9. Idea of Theorem 4

Apparently, if for all nodes in an ARG the strongest possible labeling functions are used (i.e., all variables are McMillan-labeled), the aforementioned rules on labeling functions are satisfied, and well-labeled node interpolants are obtained.

A well-known inherent property of node interpolants is that for a path p in ARG the resulting node interpolants do not form path interpolants. A node interpolant summarizes information about all paths via the node. To be able to express this "summary", the variables shared (between A and B) on any path via the node need to be employed; we call these in-scope variables. However these variables are not necessarily AB-shared in the selected path p.

Still, path interpolants for a single path can be computed from the overall problem by means of PVAIs. Using a PVA that blocks all paths except for the one of interest, LPAISs yield path interpolants focused only on that path and over the variables shared on that path.

VI. Related work

To the best of our knowledge, the only strongly related works in this area are [1], [3].

The approach of [3], implemented in the UFO tool, can handle linear integer arithmetic. The main idea of the technique is to linearize a DAG into a single path; after that, standard path interpolants are computed and, if out-out-scope variables are present in the interpolants, quantification is used to remove these variables. So, in general the approach leads to quantified interpolants, while LPAISs yield quantifier-free interpolants.

In [1], the authors present a different solution to the problem of out-of-scope variables. Instead of quantification, the following operations are proposed to remove them: (a) assigning constants to variables in the interpolant ($\top$ or $\bot$ in case of propositional logic) or (b) modifying the structure of the DAG encoding. Comparing to (a), our approach is more general. We naturally handle any provided assignments, thus it is possible to assign additional variables to obtain the same interpolant as suggested by [1]. Moreover, we provide more flexibility, e.g., in the case of $A_{\overline{\pi}}B_{\overline{\pi}}$-clean variables one may choose either label b to obtain a stronger interpolant, or label a to get a weaker one. In our work we also show the constraints under which a property relevant to verification – the path interpolation property – holds, which is not guaranteed in [1].

An aspect common to the above approaches is that they are applied as post-processing techniques, after an interpolant has been computed and only if it contains out-of-scope variables. On the contrary, our method is integrated into the computation of the interpolant, and simplifies the proof on the fly according to the corresponding variable assignment, yielding a possibly smaller interpolant.

VII. Conclusion

In this paper, we introduced the new concept of Partial Variable Assignment Interpolants, which, unlike Craig interpolants, permits specialization to sub-problems specified in the form of variable assignments. We showed how PVAIs find application in the context of Abstract Reachability Graphs and DAG interpolation. We also developed the new framework of Labeled Partial Assignment Interpolation Systems, which can be used to compute PVAIs for propositional logic, and showed its properties.

As future work, we plan to extend the framework of LPAISs and to introduce a PVA interpolation system for linear integer arithmetic – a theory particularly relevant to program verification.

Acknowledgment.: Special thanks go to Ondřej Šerý for his valuable contribution.

References

[1] Albarghouthi, A., Gurfinkel, A.: DAG-Interpolation for Software Model Checking (2013), http://cav2013.forsyte.at/files/aws_albarghouthi.pdf
[2] Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: SAS '12. LNCS, vol. 7460, pp. 300–316 (2012)
[3] Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: TACAS '12. LNCS, vol. 7214, pp. 157–172 (2012)
[4] Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In: CAV '12. LNCS, vol. 7358, pp. 672–678 (2012)
[5] Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic pp. 269–285 (1957)
[6] D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI'10. LNCS, vol. 5944, pp. 129–145 (2010)
[7] Gurfinkel, A., Rollini, S.F., Sharygina, N.: Interpolation Properties and SAT-Based Model Checking. In: ATVA '13. LNCS, vol. 8172, pp. 255–271 (2013)
[8] Jančík, P., Kofroň, J.: On Partial Variable Assignment Interpolants. Tech. Rep. 2013/5, Dept. of Distributed and Dependable Systems, Charles University in Prague (2013), http://d3s.mff.cuni.cz/publications/download/D3S-TR-2013-05-PVAI.pdf
[9] McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV '03. LNCS, vol. 2725, pp. 1–13 (2003)
[10] McMillan, K.L.: Lazy Abstraction with Interpolants. In: CAV '06. LNCS, vol. 4144, pp. 123–136 (2006)
[11] Pudlák, P.: Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
[12] Rollini, S., Alt, L., Fedyukovich, G., Hyvärinen, A., Sharygina, N.: PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification. In: LPAR (2013)
[13] Rollini, S.F., Sery, O., Sharygina, N.: Leveraging Interpolant Strength in Model Checking. In: CAV '12. LNCS, vol. 7358, pp. 193–209 (2012)
[14] Vizel, Y., Grumberg, O.: Interpolation-sequence based Model Checking. In: FMCAD '09. pp. 1–8. IEEE (2009)

CHAPTER 12

PVAIR: Partial Variable Assignment InterpolatoR

Authors: Pavel Jančík, Leonardo Alt, Grigory Fedyukovich, Antti E.J. Hyvärinen, Jan Kofroň, and Natasha Sharygina

[3] *Proceedings of FASE'16, DOI: 10.1007/978-3-662-49665-7_25, Eindhoven, Netherlands, April 2016*

PVAIR: Partial Variable Assignment InterpolatoR*

Pavel Jančík[2], Leonardo Alt[1], Grigory Fedyukovich[1], Antti E. J. Hyvärinen[1], Jan Kofroň[2], and Natasha Sharygina[1]

[1] University of Lugano, Switzerland, `{name.surname}@usi.ch`
[2] Charles University in Prague, Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems, Czech Republic
`{name.surname}@d3s.mff.cuni.cz`

Abstract Despite its recent popularity, program verification has to face practical limitations hindering its everyday use. One of these issues is scalability, both in terms of time and memory consumption. In this paper, we present Partial Variable Assignment InterpolatoR (PVAIR) – an interpolation tool exploiting partial variable assignments to significantly improve performance when computing several specialized Craig interpolants from a single proof. Subsequent interpolant processing during the verification process can thus be more efficient, improving scalability of the verification as such. We show with a wide range of experiments how our methods improve the interpolant computation in terms of their size. In particular, (i) we used benchmarks from the SAT competition and (ii) performed experiments in the domain of software upgrade checking.

1 Introduction

Symbolic model-checking algorithms rely on expressing a verification problem as a logical formula and determining whether the formula satisfies a given property. Many sub-tasks of model-checking, such as computing safe inductive invariants for programs and summarizing functionality with respect to properties critical to program correctness, rely on over-approximating parts of the formula. To keep the formal verification manageable and the run time low it is critical that the over-approximations are suitable for the model-checking task at hand. *Craig interpolation* [7] is a process for computing over-approximations of first-order formulas that has proven useful in both program verification and automatic approximation refinement [15]. The idea in applying Craig interpolation in model checking is to reduce the over-approximation process into finding a compact interpolant I such that I is satisfied by all models of the part being over-approximated but still entails the properties of interest with respect to the rest of the formula. The *Labeled Interpolation System* (LIS) [8] is a widely used framework for computing Craig interpolants in propositional logic from a resolution refutation. The flexibility of LIS allows it to be used in a variety of verification tasks that place additional requirements for the interpolants [18].

In some tasks, (e.g., when proving safety of certain types of program updates or speeding up model-checking with parallel computing) it is useful to compute over-approximations of the formula under assumptions which are specific to the particular

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by the SNF projects number 200020_163001 and 200021_153402.

application problem. However, the LIS framework in its original form does not allow for computing interpolants under assumptions. There are several reasons why such *focused interpolants* would be beneficial in particular in the LIS framework. Firstly, the focused interpolants are in general smaller and therefore more manageable for the model checker. Secondly, the properties of interpolants provided by the LIS framework, such as the path interpolation property [13], can be preserved in the focused interpolants. Thirdly, several focused interpolants can be computed from a single resolution refutation, while constructing a resolution refutation is computationally expensive. In [12], we introduced an interpolation system exploiting partial variable assignments to improve efficiency of interpolant computation. We proved that following a set of requirements put on labeling during interpolation results in interpolants with the path interpolation property, which is required by some verification tools, e.g. [1], to work.

This paper presents the *Partial Variable Assignment InterpolatoR* (PVAIR), the first implementation that is able to construct such focused interpolants. The implementation is based on the *Labeled Partial Assignment Interpolation System* (LPAIS) [12], an extension of LIS which supports focusing the interpolant in the manner required by the verification applications. The PVAIR solution is generic and can be used in various model checking-based scenarios. In this paper, in addition to providing the description of the tool architecture, we also report an initial experimental study on how the interpolants constructed with PVAIR behave in different example tasks. The results show a significant improvement in both interpolant size and the overall model checking time, suggesting that the approach is viable for constructing focused interpolants.

The general intuition behind the applications of PVAIR is that sometimes a symbolic model checker can provide a partial truth assignment for the formula being verified, coming from the knowledge of the program structure and meaning of the variables. As a result, some constraints of the formula can get satisfied; the LPAIS framework allows for removing such clauses during the interpolant computation. This improves the interpolation in two ways: the interpolation becomes faster, and the resulting interpolant can be significantly smaller. Because of the latter the interpolants can be handled in a more efficient way during the subsequent computation. PVAIR is built on top of the open-source tool PeRIPLO [18], which provides resolution proofs and is able to optimize the proofs for interpolation through transformations. PeRIPLO has been used in various verification projects, including function summarization in eVolCheck [10] and FunFrog [22], both as an interpolation engine and as a SAT solver.

We experimentally studied the performance of PVAIR on a set of its potential applications. We compared it to PeRIPLO during computation of a summary for a particular function using eVolCheck. In this experiment, PVAIR was used to rule out the program paths that do not call the function. We also applied PVAIR in more generic settings, when constructing interpolation problems from a subset of the SAT Competition benchmarks. This experiment resembles closely the scenario of computing focused interpolants for a divide-and-conquer approach for parallel model checking. In both types of benchmarks, we report a substantial reduction in interpolant sizes. As shown in the eVolCheck use case, smaller interpolants also result in considerably faster upgrade-checking steps.

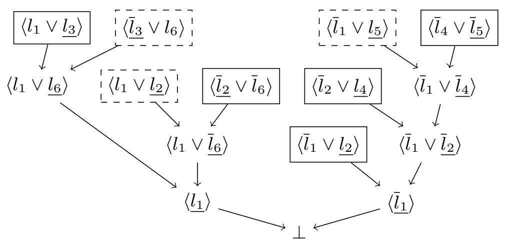


Figure 1: Refutation resolution proof; the clauses from A-part and B-part are in dashed and full boxes, respectively.

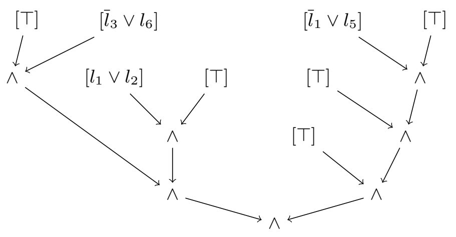


Figure 2: McMillan's interpolant.

2 Preliminaries and Background Theory

A *literal* is a Boolean variable l or its negation $\bar{l}$. A *clause* is a disjunction over a set of literals. We use angle brackets $\langle\Theta\rangle$ to denote the clause built from the literals in set Θ. A propositional formula in Conjunctive Normal Form (CNF) is a conjunction (or equivalently set) of clauses. A *resolution proof* for a set of clauses Φ is a rooted DAG with each node having either no antecedents (*leaf node*) or exactly two antecedents (*inner node*). Each node in the resolution proof is associated with node clause; from now on we use proof node and corresponding node clause equivalently. A leaf node corresponds to an input clause from Φ. Each inner node with two antecedents $\langle\Theta_1, p\rangle$ and $\langle\Theta_2, \bar{p}\rangle$ has node clause $\langle\Theta_1, \Theta_2\rangle$, thus representing a resolution where p is the *pivot* variable.

Given an unsatisfiable CNF formula Φ and its (A,B)-partitioning into $A \wedge B$ parts, a Craig interpolant [7] is a formula I such that I is implied by A ($\models A \Rightarrow I$), unsatisfiable with B ($\models B \wedge I \Rightarrow \bot$), and defined over common symbols (variables) of A and B. An interpolant can be seen as an over-approximation of A still being strong enough to be unsatisfiable with B.

Example 1: Fig. 1 shows a resolution refutation proof for CNF formula $\Phi = \langle l_1 \vee l_2\rangle \wedge \langle \bar{l}_3 \vee l_6\rangle \wedge \langle \bar{l}_1 \vee l_5\rangle \wedge \langle l_1 \vee l_3\rangle \wedge \langle \bar{l}_2 \vee \bar{l}_6\rangle \wedge \langle \bar{l}_4 \vee \bar{l}_5\rangle \wedge \langle \bar{l}_2 \vee l_4\rangle \wedge \langle \bar{l}_1 \vee l_2\rangle$. Assume a (A,B)-partitioning with A consisting of the conjunction of the first three clauses and B of the remaining five clauses. There might not be just a single interpolant for an unsatisfiable formula; many different ones of various strengths can exist. Formula $I_1 \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge (\bar{l}_6 \vee l_2)]) \wedge (\bar{l}_1 \vee l_5)$ is one of the possible interpolants which can be computed from the proof in Fig. 1 using LIS. Fig. 2 shows how McMillan's interpolant $I_2 \equiv (l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ can be derived (after constant propagation) from the proof in Fig. 1, e.g., by LIS or LPAIS with an empty assignment. Note that for convenience we write the partial interpolant associated to a particular node of the proof into brackets.

As an over-approximation, Craig interpolants express properties for all models of the formula. However, this might be unnecessarily strong for some applications. For example, while constructing a function summary through interpolation, it is possible to consider only the models corresponding to the paths going via the summarized function. Based on the encoding of the function body, a variable assignment blocking all the other paths can be derived. This applies also for the case of Abstract Reachability Graphs (ARGs). The label of a particular ARG node is an over-approximation of reachable states at that node. Since the paths in ARG which do not go via the node cannot influence the reachable states at that node, for each node it is possible to compute variable assignment blocking these paths; in other words, the assignment permits only the models corresponding to paths via the node. The node labels are computed by interpolation, however it is actually enough to compute a formula that is an interpolant for the models consistent with the assignment.

Focused interpolants. A *Partial Variable Assignment* (PVA) π assigns value $True$ resp. $False$ to some variables from formula Φ; alternatively, PVA can be seen as a conjunction of literals. Given a partial variable assignment π, a set of clauses A can be partitioned into A_π – a subset of clauses from A satisfied by the assignment, and the remaining clauses $A_{\overline{\pi}}$ which are not satisfied by π. For a given unsatisfiable formula Φ, its partitioning into $A \wedge B$ and a partial variable assignment π, a *Partial Variable Assignment Interpolant* [12], shortly *focused interpolant*, is a formula I such that $\pi \models A \Rightarrow I$ and $\pi \models B \wedge I \Rightarrow \bot$ and I is defined over unassigned shared variables between $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$, i.e., the symbols common to the π-unsatisfied parts of A and B. In other words, it is an interpolant, but only for models which agree on the values of variables assigned by π. Due to the weakened requirements, the focused interpolants can be of a smaller size compared to the Craig interpolants. The focused interpolants can be alternatively seen as Craig interpolants for the unsatisfied parts of the input – *sub-problem*, i.e., for $A_{\overline{\pi}} \wedge B_{\overline{\pi}}$ where literals falsified by the assignment are removed.

Example 1 (cont.): Let us assume assignment $\pi \equiv \bar{l}_2$ (i.e., assigning $False$ to variable l_2) and the set of clauses from our previous example. Given the assignment, B can be split into $B_\pi \equiv \langle \bar{l}_2 \vee \bar{l}_6 \rangle \wedge \langle \bar{l}_2 \vee l_4 \rangle$ and $B_{\overline{\pi}} \equiv \langle \bar{l}_4 \vee \bar{l}_5 \rangle \wedge \langle \bar{l}_1 \vee l_2 \rangle$. A_π is empty thus $A_\pi \equiv \top$ and $A_{\overline{\pi}} \equiv A$.

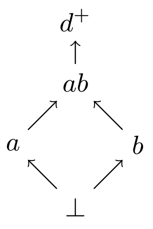
Craig and focused interpolants differ in the variables which could occur in the interpolant. The shared variables between A and B (i.e., those that can appear in a Craig interpolant) are l_1, l_2, l_5 and l_6. Since focused interpolants consider for the shared variables only unsatisfied parts of A resp. B (i.e., $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$), fewer variables are shared; in our example only l_1 and l_5 could appear in a focused interpolant, which are those which can appear in a Craig interpolant for the sub-problem.

Given an assignment and a Craig interpolant, an alternative way to reduce the interpolant size is to assign the values inside the interpolant formula and propagate the Boolean constants. In this case the interpolants from the above example result in $I_1[\pi] \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge \bar{l}_6]) \wedge (\bar{l}_1 \vee l_5)$ and $I_2[\pi] \equiv l_1 \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$. None of them is a valid focused interpolant since both contain variable l_6. Note that $I_2[\pi]$ can be equivalently rewritten as $l_1 \wedge l_5 \wedge (\bar{l}_3 \vee l_6)$x; in general, such a transformation requires a complex analysis and not all interpolants can be transformed into focused interpolants as I_1 shows. This means that the aforementioned techniques can be used to reduce the size of the formula, however not to compute focused interpolants. Below we introduce a method to compute focused interpolants for propositional logic which produces interpolants smaller than the approach above.

Leaf v:	$\langle\Theta\rangle, [I]$		
$I =$	$-\langle\Theta\rangle\vert_{b,\pi}$	if $\langle\Theta\rangle \in A_{\overline{\pi}}$	Hyp-$A_{\overline{\pi}}$
	$\neg\langle\Theta\rangle\vert_{a,\pi}$	if $\langle\Theta\rangle \in B_{\overline{\pi}}$	Hyp-$B_{\overline{\pi}}$
	$\top$	if $\langle\Theta\rangle \in A_{\pi} \cup B_{\pi}$	Hyp-A_{π}, Hyp-B_{π}

Inner vertex v:	$\dfrac{v_1 : \langle p, \Theta_1\rangle, [I_1] \qquad v_2 : \langle \overline{p}, \Theta_2\rangle, [I_2]}{\langle\Theta_1, \Theta_2\rangle, [I]}$		
$I =$	$I_1 \vee I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$	Res-a
	$I_1 \wedge I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$	Res-b
	$(I_1 \vee p) \wedge (I_2 \vee \overline{p})$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$	Res-ab
	I_2	if $\mathsf{Lab}(v_1, p) = d^+$	Res-d^+
	I_1	if $\mathsf{Lab}(v_2, \overline{p}) = d^+$	Res-d^+

Table 1: Labeled Partial Assignment Interpolation System



$$\begin{tikzcd}
 & d^+ & \\
 & ab \arrow[u] & \\
a \arrow[ur] & & b \arrow[ul] \\
 & \perp \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 3: Lattice of labels ($\sqcup$).

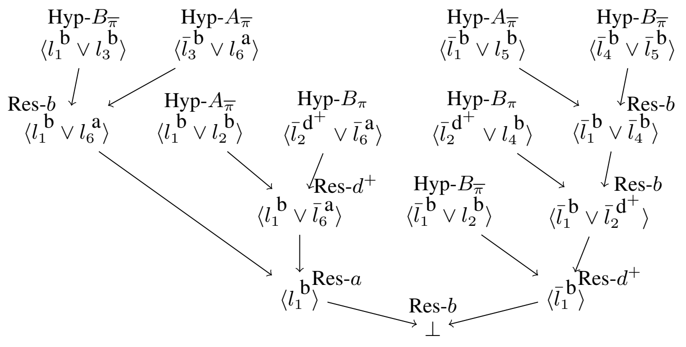

Hyp-$B_{\overline{\pi}}$
$\langle l_1^{b} \vee l_3^{b}\rangle$
Hyp-$A_{\overline{\pi}}$
$\langle \bar{l}_3^{b} \vee l_6^{a}\rangle$
Hyp-$A_{\overline{\pi}}$
$\langle \bar{l}_1^{b} \vee l_5^{b}\rangle$
Hyp-$B_{\overline{\pi}}$
$\langle \bar{l}_4^{b} \vee \bar{l}_5^{b}\rangle$
Res-b
$\langle l_1^{b} \vee l_6^{a}\rangle$
Hyp-$A_{\overline{\pi}}$
$\langle l_1^{b} \vee l_2^{b}\rangle$
Hyp-B_{π}
$\langle \bar{l}_2^{d^+} \vee \bar{l}_6^{a}\rangle$
Hyp-B_{π}
$\langle \bar{l}_2^{d^+} \vee l_4^{b}\rangle$
Res-b
$\langle \bar{l}_1^{b} \vee \bar{l}_4^{b}\rangle$
Res-d^+
$\langle l_1^{b} \vee \bar{l}_6^{a}\rangle$
Hyp-$B_{\overline{\pi}}$
$\langle \bar{l}_1^{b} \vee l_2^{b}\rangle$
Res-b
$\langle \bar{l}_1^{b} \vee \bar{l}_2^{d^+}\rangle$
Res-a
$\langle l_1^{b}\rangle$
Res-d^+
$\langle \bar{l}_1^{b}\rangle$
Res-b
$\perp$

Figure 4: Labeled proof and rules to be applied at proof nodes.

Labeled Partial Assignment Interpolation System (LPAIS) — an extension of the Labeled Interpolation System [8] — yields focused interpolants from the resolution refutation of $A \wedge B$.

In LPAIS, each literal in the clauses of the resolution proof is assigned a label a, b, ab, or d^+. Labels a, b, and ab have the same meaning as in LIS, while the label d^+ is used for the literals from the assignment π. The lattice of labels is defined by the Hasse diagram in Fig. 3. The labels are specified via a *labeling function* Lab; e.g., $\mathsf{Lab}(v_2, \overline{p})$ is the label of literal $\overline{p}$ at node v_2 of the proof. The label of a literal in an inner node v is computed using join operator $\sqcup$ (defined by Fig. 3) from the labels of the literal in the antecedent nodes ($\mathsf{Lab}(v, l) = \mathsf{Lab}(v_1, l) \sqcup \mathsf{Lab}(v_2, l)$, where v_1 and v_2 are the antecedent nodes of v). Formal definition of labeling function as well as the requirements that labels must satisfy are described in [12].

Example 1 (cont.): Fig. 4 shows how LPAIS assigns labels to literals; the label of a literal is shown as superscript. When choosing the strongest possible labeling, LPAIS yields, for empty assignments, McMillan's interpolants; in particular, only variables occurring in $A_{\overline{\pi}}$ but not in $B_{\overline{\pi}}$ are labeled a (i.e., l_6), all the others (except for the literals from the assignment) re-labeled b.

The labeled partial assignment interpolation system assigns a *partial interpolant* $[I]$

to each proof node according to the rules described in Tab. 1. The partial interpolants of the leaf nodes are directly constructed from the node clauses (it means those forming $A \wedge B$) using the rules in the upper part of Tab.1. The applied Hyp-$*$ rule is determined by the set inclusion check in the middle column; in particular by occurrence of the node clause in $A_{\overline{\pi}}$, A_π, B_π and $B_{\overline{\pi}}$. A partial interpolant for the Hyp-$A_{\overline{\pi}}$ rule, defined as $\langle \Theta \rangle|_{b,\pi}$, represents a clause which is created from the node clause $\langle \Theta \rangle$ by omitting the literals over the π-assigned variables and those whose label differs from b. In particular node clause $\langle \bar{l}_3^{\,\mathrm{b}} \vee l_6^{\mathrm{a}} \rangle$ yields partial interpolant $\langle \bar{l}_3^{\,\mathrm{b}} \vee l_6^{\mathrm{a}} \rangle|_{b,\pi} \equiv [l_3]$. The leaf nodes with clauses satisfied by π have the partial interpolant $\top$.

For inner nodes, the rule from Tab. 1 is chosen based on the labels of the pivot in the antecedents (denoted by v_1 and v_2). Note the Res-d^+ rules, which correspond to the case where the pivot is satisfied by the assignment in one of the antecedents. In these cases, the partial interpolant is the same as the partial interpolant in the antecedent not being satisfied by the assignment; due to such nodes the size of the LPAIS interpolant is smaller compared to the LIS interpolant.

Example 1 (cont.): Fig. 5 shows how focused interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$ for our example can be derived. Note the dotted arrows at nodes corresponding to Res-d^+ resolutions; they highlight the antecedents whose partial interpolants are ignored and their sub-trees do not contribute to final focused interpolant. Also note that the focused interpolant I_π is smaller compared to both $I_1[\pi]$ and $I_2[\pi]$ from the examples above.

An assignment applied onto (interpolant) formula (i.e., if $I[\pi]$ is computed) can reduce the size of the formula only if the assigned variable appears in the (interpolant) formula (i.e., the variable has to be shared). However, LPAIS reduce the size of the interpolants even if the assigned variable does not appear in the interpolant, since the reduction is done as a part of interpolant computation and not as a post-processing step.

PVAIR implements the LPAIS framework. The tool can generate the McMillan's [16], Pudlák's [17], and McMillan's$'$ [8] interpolants and their equivalents in presence of assignments. Additionally, PVAIR supports constructing different interpolants by providing different labelings for the literals in the leaves. The relative logical strength of interpolants constructed with LPAIS from the same resolution refutation is determined

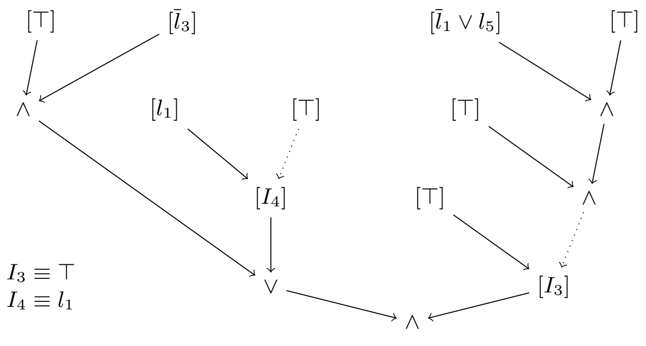


Figure 5: Focused interpolant I_π, using labeling of Fig. 4.

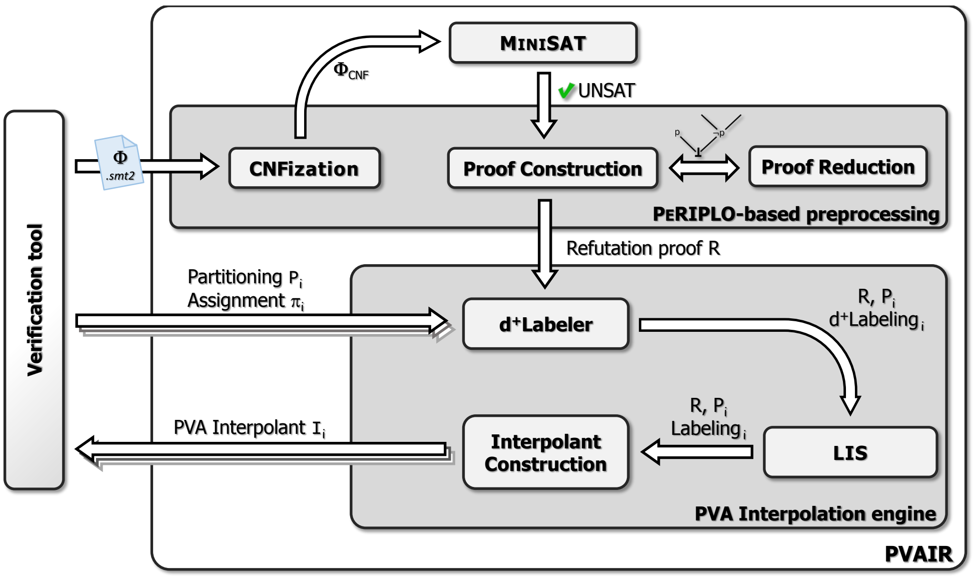


Figure 6: PVAIR architecture.

by the labeling function used. For instance, the McMillan's focused interpolants are sufficiently strong to have the path-interpolation property.

3 The Tool Architecture

The PVAIR architecture is shown in Fig. 6. It takes a propositional formula, its (A, B)-partitioning, and a partial variable assignment as input and produces focused interpolants if the input formula is unsatisfiable. The input can be provided either in a file in the SMT-LIB 2.0 format or via a C++ API.

When a verification tool decides to compute interpolants (e.g., to obtain either function summaries in the case of upgrade-checking and over-approximations of reachable states for covering checks) it constructs an input formula Φ which encodes the program being verified. Further, based on the way the input formula is constructed, the verification tool decides how to partition it (e.g., to obtain a summary of a given function) and which partial variable assignment to use (e.g., depending on the changes detected in the new version of the program). These inputs are then passed to the PVAIR tool.

The workflow of the PVAIR tool is as follows. First, the input formula is passed to the PERIPLO-based preprocessing module. Since the formula can be in an arbitrary form, it is transformed into CNF (the top box in Fig. 6) using an efficient, structure-sharing version of the Tseitin encoding [25]. Its satisfiability is then determined using the MINISAT 2.2.0 solver [9].

In the case of an unsatisfiable input, an initial refutation is extracted from the solver in the compact MINISAT internal proof format. The format is then transformed into a resolution DAG to allow more efficient handling of the proof (Proof Construction). In particular, using the resolution DAG form, the proof can be compressed using well-known proof reduction techniques such as structural hashing or pivot recycling [19,20]

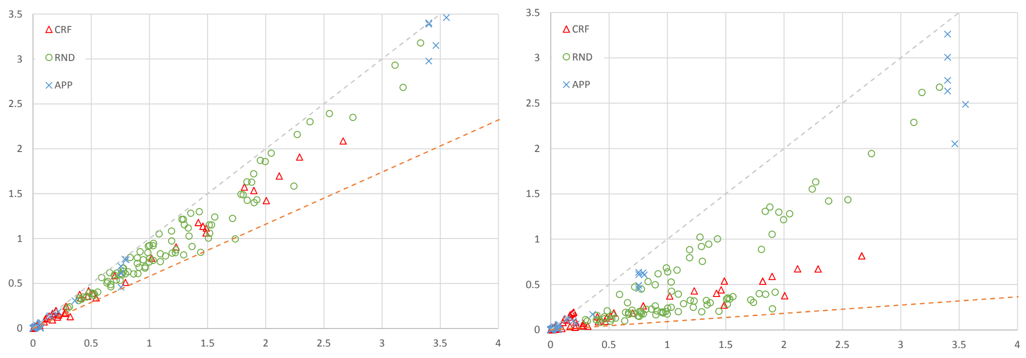


Figure 7: Comparison of interpolant sizes computed without variable assignment [x] and with one variable assigned [y] (left) and five variables assigned (right).

available in PERIPLO (Proof Reduction). The proof reduction techniques can be enabled/disabled via a configuration file or API.

Once the resolution proof R is computed, it is passed together with the partitionings and variable assignments to the interpolation engine (the bottom box in Fig. 6). From this point on, any number of partial variable assignments π_i and partitionings P_i (into $A_i \wedge B_i$) can be given as input to the tool and used to construct the corresponding interpolants I_i. Note that in any case only one SAT-solver call will be made during the entire execution. The first step inside the PVA interpolation engine is labeling all the literals in $A \wedge B$. The d^+Labeler will distribute d^+ labels among the literals according to the assigned variables, whereas the LIS will label the remaining literals according to the partitioning and the selected LIS-based interpolation algorithm (which can be chosen in the configuration file or via API). When the labeling is complete, it is used together with the partitioning and resolution proof R to compute interpolants (Interpolant Construction).

The construction starts by computing partial interpolants (according to the upper part of Tab. 1) for the leaf nodes of the refutation. The computation then proceeds from the leaves to the root node. In each inner node, depending on the label of the pivot, a partial interpolant of the node is computed by combining the partial interpolants from the antecedent nodes (the bottom part of Tab. 1). During the interpolant construction partial interpolants are optimized using Boolean constant propagation and structural sharing (hashing). The final interpolant is computed in the root node.

For the details on PVAIR usage, we refer the reader to the Tutorial section of the tool web page available at `http://verify.inf.usi.ch/pvair`.

4 Experiments

We ran PVAIR on two types of experiments: (1) SAT Competition benchmarks and (2) computational problems generated by the EVOLCHECK tool during verification procedure. To demonstrate the tool performance, we measured the size of produced interpolants and its effect on the total verification time.

Focused Itp.	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	92.8 %	83.0 %	78.1 %	83.7 %
5 vars	76.2 %	45.2 %	31.5 %	47.6 %
20 vars	48.3 %	10.1 %	4.8 %	15.0 %

Itp. from sub-prob.	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	69.5 %	55.0 %	65.5 %	58.8 %
5 vars	24.4 %	5.7 %	9.7 %	9.1 %
20 vars	0.12 %	0.01%	0.39%	0.09%

Table 2: Average interpolant sizes by category and number of assigned variables.

4.1 SAT Competition

From the way focused interpolants are computed by PVAIR it is obvious that they are smaller compared to the Craig interpolants. In this part we illustrate the actual difference. Compared to experiments on functions summaries in the latter part, SAT Competition provides us with a larger set of more heterogeneous kinds of benchmarks. This helps one to see how the reduction of the size varies among inputs from different domains.

For experiments, we chose 47 unsatisfiable benchmarks from the SAT Competition[3] from all categories – 12 from the *Application* (APP), 11 from the *Crafted* (CRF), and 24 from the *Random* (RND) sets. Since the benchmarks are not partitioned, we generated six partitionings for each benchmark; we simulated the typical way the path interpolants are computed, i.e., we randomly chose n, first n clauses of the benchmark belonged to the A-part, the remaining clauses to the B-part. No assignment is given by authors of the benchmarks, thus for each partitioning we generated five random variable assignments consisting of a single, five, resp. twenty assigned variables. Assignments of various sizes indicate how the reduction scales w.r.t. the number of assigned variables.

Since focused interpolants can be seen as Craig interpolants for a sub-problem, for each pair of partitioning and assignment, we created the sub-problem instance and used PVAIR to computed the Craig interpolant. Sub-problems are simpler compared to the benchmark from which they were generated, so interpolants for sub-problems are typically smaller compared to Craig interpolants of the benchmark. However, the interpolant for each sub-problem is computed from a different refutation proof; in contrast to focused interpolants which, for a particular benchmark, are all computed from the same proof. The path interpolation property [13], which is often exploited during program model checking, might be missing in this case.

As to the interpretation of the results: *No assignment* reflects the state-of-the-art approaches, where Craig interpolants are used directly. Focused interpolants show how the size of the interpolants can be reduced if the model checker (i.e., a tool generation the input) provides a reasonable assignment together with a partitioning. The interpolants for a sub-problem can be seen as an alternative to focused interpolants because of their similar meaning, however these interpolants lack the properties of the focused ones.

For comparison, we use McMillan's interpolants – a widely used approach. The proof reduction techniques were disabled; we used the default PERIPLO settings. All benchmarks were run on a Linux blade server with Xeon X5687 CPU using the timeout of 60 minutes and the memory limit of 20GB using the Parallel environment [24].

Fig. 7 compares the sizes of the computed interpolants. Each point in the graph corresponds to a single partitioning of a benchmark; the x-axis represents the interpolant

[3] http://www.satcompetition.org/

size if no assignment is provided (Craig interpolant) while the y-axis represents the size of the focused interpolants with a single (resp. five) assigned variable(s). For presentation clarity, the y-axis is the average size of all five random assignments generated for a given partitioning. The values on axes represent millions of nodes if an interpolant is represented as DAG (counting literals and Boolean operators). The orange dashed line shows the average size of Craig interpolants for sub-problems. This illustrates what price is paid by focused interpolants for the path interpolation property and a single SAT solver call. Both graphs show interesting reduction in size for focused interpolants as well as substantially larger reduction in case of five assigned variables. In both graphs the same partition of the same benchmark share the same x-value, thus it is possible, especially for the larger ones, to compare their reductions.

Tab. 2 summarizes the results shown in the graphs above, reporting precise numbers. The table on the left-hand side compares the sizes of focused interpolants to Craig interpolants (in the *No assignment* row). The *No assignment* row shows the average size of Craig interpolants for a given benchmark type. The remaining rows show the relative sizes of focused interpolants w.r.t. the *No assignment* row. The application benchmarks exhibit a smaller reduction compared to the other types, and even for twenty assigned variables, the interpolants are half in the size of the Craig interpolants. The table on the right-hand side compares the sizes of Craig interpolants for the benchmark with the Craig interpolants for sub-problems (corresponding to the assignments used in the left table). The table shows that these interpolants are on average smaller compared to the focused ones. The more variables are assigned, the bigger the difference is. While the sizes are comparable for a few assigned variables, the price paid for the path interpolation property of focused interpolants is high for larger assignments (e.g., twenty variables) .

Time and memory demands are crucial properties of each interpolation tool. The reduction in overall running time and required memory roughly correspond to the reduction of interpolant sizes; e.g., PVAIR is 11% faster and requires 9% less memory on average if a single variable is assigned. The time and memory savings occur as well during the interpolant computation phase due to smaller interpolants being handled.

4.2 Applying PVAIR for Checking Software Upgrades

The usefulness of PVAIR is motivated by the tremendous role of interpolation in model checking. One of the possible applications of PVAIR is checking software upgrades by means of function summarization [23] implemented in the tool eVOLCHECK. Given a program S and an assertion a, eVOLCHECK verifies S with respect to a (i.e., proves that $S \wedge \neg a$ is unsatisfiable) and, for each function call in S, it constructs the interpolant and treats it as a *function summary*. In [21] we show that even if the constructed function summary is an over-approximation of the function behavior of S, it preserves the safety of the assertion a in S.

eVOLCHECK *validates* the computed function summaries to over-approximate the behavior of the corresponding functions of a program upgrade, U. In that context, programs S and U must have a non-empty set of common function calls. eVOLCHECK traverses this set starting from the deepest level of the (unwound during preprocessing) function call-tree and checks whether each original function summary still over-approximates the new behavior of the corresponding function. If there is a function call, the original summary of which does not over-approximate the new behavior, eVOLCHECK propagates the check to the caller function. If there is no function to propagate then U is unsafe. If at some depth of the unwound call-tree all the function summaries are proven to be valid, then U is safe, and eVOLCHECK reconstructs the summaries for the modified function calls.

Applying PVAIR *to* eVOLCHECK. Consider the case when U is obtained from S by *removing some functionality*. Then by construction, the original summaries of S are still valid over-approximation of the new function behavior in U. But at the same time, they might be unnecessarily general and consume excessive memory. While the use of the original summaries does not break soundness of the further upgrade checking, it is practical to refresh (and possibly shrink) the summaries to become more accurate with respect to U.

The *refreshed* summaries may be used to verify a further updated program W that additionally may *introduce new functionality* with respect to U. On the other hand, the summaries may be also used to speed up verification of a new assertion b, implanted in the code of U [21]. To enable both scenarios, the constructed summaries need to be externally stored and further migrated across the verification runs. Thus, the size of the summary also becomes important.

While eVOLCHECK does not provide a way to refresh summaries except of complete re-verification of U from scratch, PVAIR becomes particularly useful. Let $\Delta_{S,U}$ denote the behavioral difference of S and U, i.e., the set of behaviors of S not present in U. If the set $\Delta_{S,U}$ is non-empty, it could be exploited by PVAIR to generate the partial interpolants that represent new summaries for each function in U. These updated summaries are still guaranteed to preserve safety of the assertion a in U.

Experiments. We experimented with PVAIR on a set of 10 pairs of different benchmarks written in C. Notably, all benchmarks used non-linear arithmetic operations. After the required propositional encoding (i.e., bit-blasting), the resulting large-size formulae have been a bottleneck for solving and interpolation using the original eVOLCHECK approach.

In our experiments, for each pair of programs, S and U, we obtained U from the corresponding S by assigning guards in some conditional expressions. In particular, we replaced `if P do A else do B` by `assume(P); A`. This is equivalent to assigning `P = true`, and $\Delta_{S,U}$ consists of the behaviors specified by `assume(¬ P); B`. For simplicity, in our experiments, we assumed that $\Delta_{S,U}$ affected only a single function f.

The results of our experiments are shown in Tab. 3. For each S and U, we identified $\Delta_{S,U}$ and obtained the set of conditional expressions to be assigned in S (column *#var. assigned*). Then we performed two steps: (1) *constructed* the summary of f without/with $\Delta_{S,U}$; and (2) *validated* the corresponding summaries of f with respect to the new code in U. This experiment illustrates to what extent:

- the use of PVAIR yields smaller summaries compared to the ones by PERIPLO,
- the use of smaller summaries improves the overall performance of eVOLCHECK.

We collected the size of the resulting interpolants and total verification time needed to perform steps (1) and (2). We used the Pudlák interpolation algorithm [17] to construct the “orig” interpolants (the ones constructed without $\Delta_{S,U}$).

C program		Interpolant (function summary) size				Verification time (sec)			
name	# var. assigned	# var. orig.	# var. PVAI	# cl. orig	# cl. PVAI	boot. orig.	boot. PVAI	upgr. orig.	upgr. PVAI
Test 0	3 vars	15227	62.61 %	45192	62.21 %	18.93	99.17 %	4.025	65.96 %
Test 1	1 var	23273	78.46 %	69330	78.31 %	10.36	99.24 %	4.034	77.79 %
Test 2	2 vars	31278	59.19 %	93345	58.98 %	8.71	100.32 %	3.878	57.61 %
Test 3	1 var	12236	63.80 %	36219	63.31 %	7.34	100.12 %	1.256	71.50 %
Test 4	2 vars	20447	74.57 %	60852	74.37 %	12.40	101.94 %	2.982	81.35 %
Test 5	3 vars	24716	32.50 %	73659	32.05 %	12.20	102.94 %	3.855	39.46 %
Test 6	3 vars	33076	37.89 %	98739	37.58 %	12.63	102.16 %	7.951	40.05 %
Test 7	1 var	12478	57.47 %	36945	56.91 %	8.88	100.29 %	2.350	57.96 %
Test 8	1 var	21201	50.42 %	63114	50.04 %	14.46	97.55 %	3.706	50.94 %
Test 9	2 vars	20314	39.71 %	60453	39.22 %	21.42	101.26 %	4.581	40.30 %

Table 3: eVolCheck verification statistics.

As can be seen from the table, the use of PVAIR helped eVolCheck to make the function summaries up to 60% smaller compared to the ones produced by PeRIPLO (columns *#var. orig* vs. *#var. PVAI*, and *#cl. orig* vs. *#cl. PVAI*), while taking almost no additional time (columns *boot. orig.* vs. *boot. PVAI*). Furthermore, eVolCheck spent up to 60% less effort in the validating step (columns *upgr. orig.* vs. *upgr. PVAI*), in which the model checker finally confirmed that the new code is safe. In other words, in the considered verification scenario and driven by PVAIR, eVolCheck improved both, the size of the summaries and the overall verification time, without sacrificing soundness of the entire model checking procedure.

5 Related work

This section compares the PVAIR approach with various techniques for reducing the size of an interpolant based on variable assignments, proof compression, and interpolant post-processing.

Variable assignments. Given a variable assignment, the most straightforward way to reduce the interpolant size is to apply the assignment directly onto the interpolant formula and propagate Boolean constants. This idea is used in the UFO [1] tool. Due to the tight integration into the interpolation process, LPAIS yields smaller interpolants compared to this simple approach. Since the assignment is considered by LPAIS already during the interpolant construction, this results in larger parts of the interpolant being cut away.

Proof compression. Interpolants are often derived from a resolution proof and therefore their size is roughly proportional to the size of the proof. Several methods for compressing a resolution proof exist [2,11,4,2,19,6]. Different variants of these techniques are applied in PdTRAV [5] verification framework, the PeRIPLO tool, and the Skeptik [3] proof transformer, just to name a few examples. In this work, the reduction of the interpolant size is based on the fact that only a proof of the unsatisfied part of the input formula is needed. Since the omitted (i.e., satisfied) parts can be important w.r.t. other assignments, the proof compression techniques cannot remove these parts from the proof. As a result, these techniques are orthogonal and PVAIR can benefit from proof compression if applied.

Interpolant post-processing. Once an interpolant is computed, various techniques can be used to reduce its size. Such techniques include constant propagation, structural sharing, and various equivalence and subsumption checks. PdTRAV, for example, internally uses BDD-based sweeping to detect the equivalences and balancing/rewriting over And-Inverter Graphs [14] representation to further reduce the size of an interpolant. Any such post-processing technique producing smaller equivalent formulae can be applied to the interpolants produced by the PVAIR tool.

6 Conclusions

In this paper we presented the PVAIR interpolation tool, which exploits partial variable assignments obtained from an application-specific source to compute focused interpolants. The tool uses the extension of the labeled interpolation system, LPAIS, to construct the interpolants from a resolution refutation. We presented a potential application for the focused interpolants, in particular in software upgrade checking where we require the path interpolation property. We performed an initial study using a wide range of experiments varying the size of the partial variable assignment. The results show a good improvement compared to the baseline and suggest that the approach taken for computing focused interpolants has significant potential in reducing the interpolant size and model checking time. In the future we plan to integrate the PVAIR tool into a concrete implementation of a parallel model checker as well as to study other applications of model checking where partial assignments arise naturally.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In *TACAS*, pages 157–172, 2012.
2. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-Time Reductions of Resolution Proofs. In *HVC*, pages 114–128, 2008.
3. J. Boudou, A. Fellner, and B. W. Paleo. Skeptik: A Proof Compression System. In *IJCAR*, pages 374–380, 2014.
4. J. Boudou and B. W. Paleo. Compression of propositional resolution proofs by lowering subproofs. In *TABLEAUX*, pages 59–73, 2013.
5. G. Cabodi, C. Loiacono, and D. Vendraminetto. Optimization Techniques for Craig Interpolant Compaction in Unbounded Model Checking. In *DATE*, pages 1417–1422, 2013.
6. S. Cotton. Two Techniques for Minimizing Resolution Proofs. In *SAT*, pages 306–312, 2010.
7. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Symbolic Logic*, pages 269–285, 1957.
8. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant Strength. In *VMCAI*, pages 129–145, 2010.
9. N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, pages 61–75, 2005.
10. G. Fedyukovich, O. Sery, and N. Sharygina. eVolCheck: Incremental upgrade checker for C. In *TACAS*, pages 292–307, 2013.
11. P. Fontaine, S. Merz, and B. W. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In *CADE-23*, pages 237–251, 2011.
12. P. Jancik, J. Kofroň, S. F. Rollini, and N. Sharygina. On Interpolants and Variable Assignments. In *FMCAD*, pages 123–130, 2014.

13. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In H. Hermanns and J. Palsberg, editors, *TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.
14. A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *DAC*, pages 232–237, 2001.
15. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.
16. K. L. McMillan. An Interpolating Theorem Prover. In *TACAS*, pages 16–30, 2004.
17. P. Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Symbolic Logic*, pages 981–998, 1997.
18. S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. PeRIPLO: A Framework for Producing Effective Interpolant-based Software Verification. In *LPAR*, pages 683–693, 2013.
19. S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, pages 1–41, 2014.
20. S. F. Rollini, O. Sery, and N. Sharygina. Leveraging Interpolant Strength in Model Checking. In *CAV*, pages 193–209, 2012.
21. O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based Function Summaries in Bounded Model Checking. In *HVC*, pages 160–175, 2011.
22. O. Sery, G. Fedyukovich, and N. Sharygina. FunFrog: Bounded model checking with interpolation-based function summarization. In *ATVA*, pages 203–207, 2012.
23. O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD*, pages 114–121, 2012.
24. O. Tange. GNU Parallel – The Command-Line Power Tool. *The USENIX Magazine*, pages 42–47, 2011.
25. G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, 1969.

CHAPTER 13

Conclusion and future work

This thesis provides an overview of my contribution to the field of software verification. It ranges from creating semantic models for behavior of software components to techniques improving the practical complexity of the verification tools. I emphasized that while advances in the direction of new algorithms' development are of a great importance, new optimizations of verification tools and their performance are a necessity.

Building reliable and error-free software is a very important goal nowadays. Absence of errors in software can be achieved by different means, at various stages of the development process. On one hand, a formal specification of desired properties at the design phase helps to create software that is maintainable, scalable, and satisfies high-level requirements. On the other hand, verification of properties at the code and bytecode level can assure absence of low-level errors at runtime. Thus, we address the issues of software correctness during the whole development process, at all levels of design.

Verification (especially by means of model checking) of software properties is an algorithmically undecidable problem in general. Nonetheless, successful attempts to develop methods and tools deciding validity of certain properties in particular cases have been made; despite being often either unsound or incomplete, such tools are very useful in practice. Another challenge in this area is capturing (and verifying) high-level design properties, such as security and privacy aspects of user data, at the code level; while finding a possible assertion violation is definitely very useful in the debugging phase, high-level properties are usually not provable at the code level. In addition, maintaining correspondence of a high-level design with the code in important aspects is rarely addressed in research. Hence in my view, the next step to be taken in this area is to develop methods for linking the high-level (design) properties with abstractions at the code level allowing for maintaining and verifying consistency between these two levels. This can be achieved, e.g., by inserting assert-like statements and special annotations into the code or creating a particular structure of method (or function) bodies. Even though some tools providing such functionality have already been made, e.g., for UML, little attention has been paid to preserving important properties so far (*traceability*) during the development process.

To achieve this goal, extending an existing programming language by new abstractions or design a new one, supporting this kind of connections, is needed. To avoid changes breaking desired properties, support at the side of an integrated development environment (IDE), preferably also providing the verification functionality, becomes a necessity.

While the paragraphs above describe our vision at a high level, below, we pinpoint particular steps to be taken helping in achieving the goal of a practically usable verification platform.

In the area of static analysis of dynamic languages, we plan to improve the efficiency of the memory representation to keep precision of our analysis and improve the performance in terms of memory consumption, which is currently the main limiting factor of the framework. While the precision is satisfactory—an acceptable rate of false negatives is produced, unlike in the case of other tools, memory demands for analysis of more complex PHP programs is beyond what a usual desktop PC can offer, making the framework hard to be used on daily basis by software developers. This means either proposing a better memory representation or extending the analysis algorithm to differentiate among particular situations (memory patterns) making the representation more compact.

In the area of symbolic software-verification methods, improving performance is one of the main factors motivating further research. Even though by our improvements, we manage to decrease both memory demands and verification time significantly, our method still suffers from low practical usability in terms of scaling to large programs. A very promising direction here is to extend the partial variable assignment interpolation system currently encoding the input program into propositional formulas to a first-order logic. Using a logic such as Linear Integer Arithmetic (LIA) allows one to drop the expensive step of encoding the program variables into Boolean ones, substantially decreasing the size of the verification condition. Instead of an SAT solver, an SMT solver is to be used, then. Even though an SMT call is usually more expensive than an SAT call, the size of the formula can be significantly smaller for a higher-order logic; recent research results show viability of such an approach.

Bibliography

Included Publications

[1] D. Hauzar and J. Kofroň. WeVerca: Web Applications Verification for PHP. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 296–301, Cham, 2014. Springer International Publishing.

[2] D. Hauzar and J. Kofroň. Framework for Static Analysis of PHP Applications. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] P. Jančík, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, J. Kofroň, and N. Sharygina. PVAIR: Partial Variable Assignment InterpolatoR. In P. Stevens and A. Wasowski, editors, *Fundamental Approaches to Software Engineering: 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, pages 419–434, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[4] P. Jančík and J. Kofroň. On partial state matching. *Formal Aspects of Computing*, pages 1–27, 2017.

[5] P. Jancik, J. Kofroň, S. F. Rollini, and N. Sharygina. On Interpolants and Variable Assignments. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 22:123–22:130, Austin, TX, 2014. FMCAD Inc.

[6] J. Kofroň. Checking Software Component Behavior Using Behavior Protocols and Spin. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1513–1517, New York, NY, USA, 2007. ACM.

[7] J. Kofroň, F. Plášil, and O. Šerý. Modes in Component Behavior Specification via EBP and Their Application in Product Lines. *Inf. Softw. Technol.*, 51(1):31–41, Jan. 2009.

[8] M. Mach, F. Plášil, and J. Kofroň. Behavior Protocol Verification: Fighting State Explosion. *International Journal of Computer and Information Science*, 6(1):22–30, 2005.

[9] T. Poch, O. Šerý, F. Plášil, and J. Kofroň. Threaded behavior protocols. *Formal Aspects of Computing*, 25(4), July 2013.

Referenced Publications

[10] The consolidated Ada Reference Manual, consisting of the International Standard (ISO/IEC 8652:2012): Information Technology – Programming Languages – Ada, 2012.

[11] Autosar: AUTomotive Open System ARchitecture. http://www.autosar.org/.

[12] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-Time Reductions of Resolution Proofs. In *HVC*, pages 114–128, 2008.

[13] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 54–65. ACM, 2007.

[14] E. Borde and J. Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*. ACM, June 2011.

[15] J. Boulanger. *Static Analysis of Software: The Abstract Interpretation.* Wiley, 2011.

[16] M. Bozga, J. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, pages 164–178, 1999.

[17] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, Sept. 2006.

[18] T. Bures, P. Hnetynka, and F. Plasil. Runtime concepts of hierarchical software components. *International Journal of Computer & Information Science*, 8(S):454–463, sep 2007.

[19] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.

[22] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] S. Cotton. Two Techniques for Minimizing Resolution Proofs. In *SAT*, pages 306–312, 2010.

[24] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[25] E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, 85/1980:169–181, 1980.

[26] M. Fahndrich. Static verification for code contracts. In *SAS'10 Proceedings of the 17th international conference on Static analysis*. Springer Verlag, September 2010.

[27] G. Fedyukovich, A. C. D'Iddio, A. E. J. Hyvärinen, and N. Sharygina. Symbolic detection of assertion dependencies for bounded model checking. In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, pages 186–201, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[28] P. Fontaine, S. Merz, and B. W. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In *CADE-23*, pages 237–251, 2011.

[29] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3), 2012.

[30] K. Havelund. Java pathfinder user guide. *NASA Ames Research*, 1999.

[31] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[32] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[33] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[34] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, pages 1087–1106, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[35] M. Lewis and M. Jones. A dead variable analysis for explicit model checking. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 48–57, 2006.

[36] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[37] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. CAV'03*, pages 1–13, 2003.

[38] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[39] P. Parízek, F. Plášil, and J. Kofroň. Model checking of software components: Combining java pathfinder and behavior protocol model checker. *2012 35th Annual IEEE Software Engineering Workshop*, 00:133–141, 2006.

[40] PHP: Hypertext Preprocessor. http://www.php.net/.

[41] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

[42] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[43] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[44] R. Reussner, I. Poernomo, and H. W. Schmidt. Reasoning about software architectures with contractually specified components. In *Component-Based Software Quality*, pages 287–325, 2003.

[45] S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, pages 1–41, 2014.

[46] S. F. Rollini, O. Sery, and N. Sharygina. Leveraging interpolant strength in model checking. In *Proc. CAV'12*, volume 7358 of *LNCS*, pages 193–209. Springer, 2012.

[47] J. P. Self and E. G. Mercer. On-the-fly dynamic dead variable analysis. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, pages 113–130, 2007.

[48] M. Shema. *Hacking Web Apps: Detecting and Preventing Web Application Security Problems.* Syngress Media. Syngress, 2012.

[49] V. Štill, P. Ročkai, and J. Barnat. Divine: Explicit-state ltl model checker. In M. Chechik and J.-F. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 920–922, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[50] Tesla car crash. https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk, June 2016.

[51] N. Tillmann and P. de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, pages 134–153, Prato, Italy, April 2008. Springer Verlag.

[52] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, Mar. 2000.

[53] Microsoft Visual Studio. https://www.visualstudio.com/.