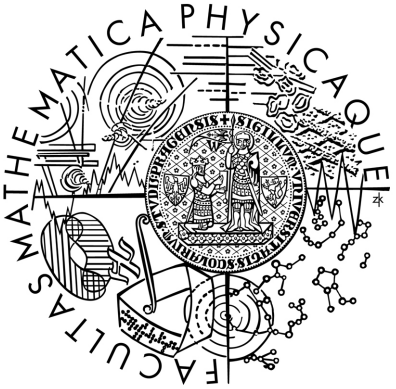
Charles University
Faculty of Mathematics and Physics

HABILITATION THESIS



Jan Kofroň

Verification of Software

Computer Science, Software Systems

Prague, Czech Republic 2018

Contents

1 Introduction **3**

2 Specification of Software Behavior **7**
2.1 Software components and services 7

3 Verification of Source Code **11**
3.1 Explicit model checking 11
3.2 Static analysis 12
3.3 Symbolic verification methods 14

4 Behavior Protocols Verification: Fighting State Explosion **17**

5 Checking Software Component Behavior Using Behavior Protocols and Spin **19**

6 Modes in component behavior specification via EBP and their application in product lines **21**

7 Threaded Behavior Protocols **23**

8 On Partial State Matching **25**

9 Framework for Static Analysis of PHP Applications **27**

10 WeVerca: Web Applications Verification for PHP **29**

11 On Interpolants and Variable Assignments **31**

12 PVAIR: Partial Variable Assignment InterpolatoR **33**

13 Conclusion and future work **35**

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*

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*

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*

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*

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*

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

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*

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*

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*

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/.