Appendix A – Research Program

StaRVOOrS: Unified Static and Runtime Verification of Object-Oriented Software

1 Purpose and Aims

Nowadays, we witness a significant quest from software industry for *lightweight formal methods*. By that, we mean methods which achieve a high degree of confidence in desired (sub-)system properties, while satisfying high demands on usability and automation. The reasons for the increased interest, in the software area, for formal methods in general are the following, parallel developments:

- *Model driven development.* There is an ever more dominant role of models in the software development process. Well defined graphical or textual notations are used to achieve unified documentation, semi-automated code generation, simulation, and to some extent analysis. As compared to traditional development methods, the gap to artifacts used for formal specification and verification is much closer here.
- *Automated software engineering.* Related to the above, there is a trend to (partly) automate ever more steps in the development cycle. As a side effect, it becomes more natural for developers to invest into meta-level artifacts.
- *Exploding complexity of embedded software.* Both the number and size of software units embedded into mechanical or electr(on)ical environments is increasing dramatically. The demands on the safety of such units is typically extremely high.
- *Software standards and certification.* In specific domains (e.g., avionics, car, medical), standards for architecture, interfaces, and processes are becoming very important. Their aim is to improve interoperability and quality assurance. Safety critical certification regimes, such as for the avionics domain ED-12B/DO-178B (for Europe/USA), have contributed to unparalleled safety in their respective domains. As late as Dec. 2011, the new versions ED-12C/DO-178C were approved, now containing a dedicated 'Formal Methods' supplement.[1]
- *Application focus of program verification.* Fundamental concepts of program verification have been around for decades, but only lately have arisen many techniques that are tailored to widely used languages and platforms.
- *Increased efficiency of program verification.* Verification technology has become a lot more efficient, and automation has increased significantly.

In spite of the above, academic and industrial R&D in software verification is in danger of not fulfilling the—now high—expectations they are facing from industry. Even if *static verification* of software has become more relevant, effective, and efficient as compared to earlier days, the community has a hard time overcoming certain inherent limitations. Certain static verification techniques have high precision, in which case powerful judgements are still too hard to achieve automatically. Other static verification techniques use abstractions to enable increased automation, in which case important, or even critical, aspects of the real, concrete system are easily missed, not to speak of the fundamental difficulty of crafting the right abstraction.

In reaction to this, there is a recent trend towards more *lightweight* formal methods, which are easier to exploit but give limited guarantees. One such lightweight method is

[1] Another supplement of ED-12C/DO-178C is 'Object-Oriented Technology', which demonstrates the growing role of object-orientation in safety critical domains.

runtime verification. As compared to static verification, it has complementary strengths and weaknesses. Runtime verification combines full precision of the execution model (even including the real deployment environment) with full automation. On the other hand, it only ever judges the observed runs, and cannot judge alternative and future runs. Another drawback of runtime verification is the computational overhead of monitoring the running system which, although typically not very high, can still be prohibitive in certain settings.

The **overall purpose** of the StaRVOOrS project ('Unified Static and Runtime Verification of Object-Oriented Software') is to *provide a unified, lightweight to use but powerful in result, method for specifying and verifying, with a variety of confidence levels, properties of parallel object-oriented software systems.*

The **specific goals** of StaRVOOrS are:

1. To combine the strengths of static and runtime verification into a verification method that is easier to use than static methods, while providing a higher level of confidence and a lower application slow down than runtime verification.
2. To let both of the underlying verification techniques (static and runtime) profit to a great extent from the analysis performed by the respective other.
3. To develop techniques which also allow exploiting partial results, like unfinished proofs, to strengthen the result of the unified verification method.
4. To provide a unified framework for specifying desired properties of the system to be verified. The user will not have to separate (sub-)properties that are subject to static vs. runtime verification during the process.
5. To support the combination of data centric and control centric properties, including real time constraints.
6. To provide a verification solution for sequential, concurrent, and distributed object-oriented applications.
7. To implement a powerful, integrated tool that achieves the aforementioned goals for Java applications in particular.

2 Survey of the Field

Because the combination of static verification and runtime verification is a central part of this project, we briefly summarise here the state-of-the-art of both areas. In addition, we discuss contemporary work on combining static and dynamic techniques.

2.1 Static Verification of Software

Static software verification is a formal technique for reasoning about properties of *all possible runs* of a program. There are basically two families of approaches, deductive verification and model checking.

Deductive program verification has been around for nearly 40 years [41], however, a number of developments during the last decade brought dramatic changes to how deductive verification is being perceived and used.

- The era of verification of individual algorithms written in academic languages is over: contemporary verification tools support commercial programming languages such as Java [22, 31, 13] or C# [11] and they are ready to deal with industrial applications [39, 45, 46, 38].
- Earlier, deductive verification tools used to be stand-alone applications that were usable effectively only after years of academic training. Nowadays, one can see a new tool generation that can be used after limited investment in training [1], and that is integrated into modern IDEs [11, 13]. However, full automation is still rarely achieved when verifying functional properties of programs with loops, for instance.

- Perhaps the most striking trend is that deductive verification is emerging as a base technology. It is not only employed for correctness proofs, but in automatic test generation [21, 35, 32], and bug finding [37].

Among the state of the art efforts is the KeY tool [2], co-developed by the main applicant. It differs from other projects in its close to complete coverage of the Java programming language [12]. In contrast to verifiers based on higher order logics, the prover of the KeY system provides a state-of-the-art user interface, high automation, and an easy mechanism for extending its rule base.

Apart from deductive verification, model checking has been applied extensively and successfully for the static verification of both hardware and software systems. The adaptation of this technique to object-oriented software is progressing but still in an early stage. The static verification side of StaRVOOrS will focus on deductive techniques rather than model checking. The reason is that deductive techniques are better suited for fully precise reasoning, whereas model checking normally requires abstractions. This is an advantage when connecting to run-time verification, where abstracting from the real data and execution model is neither necessary nor desirable.

2.2 Runtime Verification of Software

Runtime verification (RV) is a technique for monitoring the execution of a software system, detecting violations as they appear at runtime. In recent years researchers have implemented RV monitoring tools which usually compile high-level (temporal) properties into monitor implementations (e.g., [23, 40, 29, 30, 8, 27]). There are two main concerns when defining and using RV. First of all, in order to minimise the possibility of erring, it is desirable that monitors are automatically synthesised from formally specified properties. Secondly, though a minimal runtime overhead is acceptable, it is desirable to reduce them as much as possible.

The above concerns are obviously interdependent: properties should be written in a formal language that is expressive enough as to represent meaningful properties, but not too much as to avoid efficient monitoring.

Different solutions based on optimisations have been presented to alleviate the overhead problem ([15, 16]). Further approaches aim at obtaining small monitors by construction [44], or use some kind of overhead guarantee, as proposed in [25]. However, there is still need to improve runtime monitoring techniques as motivated by the development of specific techniques to improve monitor efficiency [20].

In the following, we give a brief overview over state-of-the-art runtime monitoring tools developed in recent years, without claiming completeness. ConSpec [6] inlines a runtime monitor into applications on mobile devices based on observed contract violations. JavaMOP [23] is a monitoring-oriented development environment where parts of the system's functionality are designed as monitor-triggered code. Java-MaC [40] enables automatic instrumentation to have access to system events. Higher-level activities are processed by the runtime checker to raise an alarm if any of the specified properties are violated. Eagle [34] is a runtime verification tool supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints and statistics. Lola [30] guarantees bounded memory to perform online monitoring, and differs from most other synchronous languages in that it is able to refer to future values in a stream. Tracematches [8] is an extension to AspectJ allowing the specification of trace patterns, also supporting parametrisation of events. This work has been extended in [17] to improve efficiency by making a temporal and spatial partitioning among collaborative

users.

Finally, LARVA [27], co-developed by the co-applicant, is a tool tailored to verify untimed and real-time properties of Java programs. Properties can be expressed in a number of notations, including timed-automata enriched with stopwatches (DATEs), Lustre, and a subset of the duration calculus. The tool has been successfully used on a number of case-studies, including an industrial system handling financial transactions. LARVA also performs analysis of real-time properties, and whenever possible to calculate an upper-bound on the memory and temporal overheads induced by monitoring.

2.3 Combining static and dynamic techniques

The combination of different verification techniques in order to get the best from each, is not new. In particular there have been some successful stories combining different static analysis techniques. This is the case for instance of the SLAM project [10] where symbolic model checking, program analysis and theorem proving are combined on a novel fashion to verify drivers written in C. Another example is InVeSt [14] integrating algorithmic and deductive verification techniques, using abstraction, to verify invariance properties.

In recent years there has been work combining static analysis (other than verification) and runtime verification in different ways. We mention a few of such works below and we discuss the main differences with our approach. Arhto and Biere describes an architecture based on JNuke where Java programs can be statically and dynamically analysed [9]. In this framework, a static analyser tries to detect faults which are manually checked by a user who writes test cases for each fault found. The program is then run many times against those test cases confirming, or not confirming, the failure. In the latter case, a log is kept for future runs of the static analyser. In [18] static analysis is used to improve the performance of runtime monitoring based on tracematches. The paper presents a static analysis to speed up trace matching by reducing the runtime instrumentation needed. The static analysis part is based on 3 stages in order to: rule out some tracematches, eliminate inconsistent instrumentation points, and finally further refine the analysis taking into account certain execution order. In [19] Bodden et al present ahead-of-time techniques to statically prove the absence of *all* program errors, or mark specific parts of the programs where such errors are likely to occur at runtime. The approach is based on tracematches.

The main differences of our approach with the above works are the following. Unlike [9] we are not concerned with testing faults found by a static analyser but to prove as much as we can with a static verifier, and only the non-provable parts are verified during the real execution of a program. Besides we do not extract test cases to test the system but perform runtime verification. Like [18] we also aim at improving the efficiency of runtime verification but our techniques are completely different. While Bodden et al use static *analysis* we use deductive *verification*. This distinction is crucial as the kind of properties we can prove is not the same. Moreover, what distinguishes our work from any of the above (and other similar work) is that we provide a unified language for specifying properties which may be used both for static and dynamic verification, and that our combination is unique in the sense of combing *deductive verification* (not *static analysis*) of Java programs (i.e., KeY) with timed runtime verification (i.e., LARVA).

Finally we would like to mention CLARA, a framework to statically optimise runtime monitoring [20]. Note that CLARA is not about combining static analysis/verification and runtime verification techniques to verify software, but rather about using static analysis techniques to operate on the monitors themselves with the aim of improving performance.

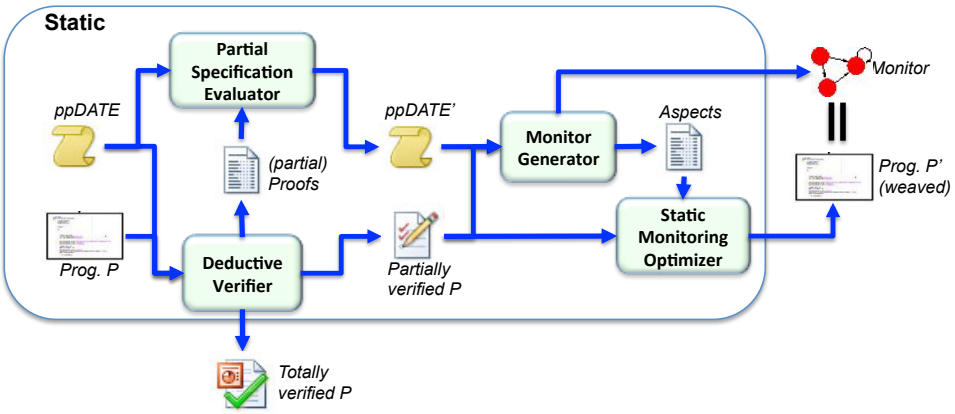

Figure 1: High-level description of the StaRVOOrS framework

3 Project Description

3.1 Research Group

The project will be performed by a research team consisting of the main applicant, Wolfgang Ahrendt (with 20% of full time), and the co-applicant Gerardo Schneider (with 10% of full time), plus a Ph.D. student (with 80% of full time). Wolfgang Ahrendt and Gerardo Schneider come into the project with a strong background in static verification (see [2] and Appendix B+C) and runtime verification (see [26] and Appendix B+C), respectively. Note that, even if Ahrendt and Schneider are affiliated with different universities (Chalmers and the University of Gothenburg), they work at the same department, Computer Science and Engineering, which is a joint department of both universities.

3.2 The StaRVOOrS framework

A visualisation of the proposed framework is depicted in Fig. 1. To keep that picture simple, some elements are left out, and certain work units are hidden in simple arrows, but we will explain that on the way.

The framework takes as input an object-oriented program P (just called P in the following) and a specification of the desired properties of P. The working title for the format of the specification is ppDATE, to be developed in the project. The format will be designed for co-specifying data centric, control centric, and real time aspects of P, in a unified way. Conceptually, ppDATE describes communicating automata with with event-triggered transitions, timers, and functional unit specifications. Events are actions on objects (foremost method calls), timer events, primitives for synchronising with different automata, or a combination thereof. In addition, events are potentially augmented with conditions, actions, plus logic based, data centric specifications of the pre-post behaviour of the called method. In this way, ppDATE will be an extension of the DATE format [26] with data centric, logic based **p**re/**p**ost specifications of methods, which is dominant in specification languages like JML (Java Modeling Language [42, 43]). Other elements of data centric specifications, like class invariants, will also be integrated.

The fact that Fig. 1 takes ppDATE as a starting point should *not* indicate that ppDATE is the final and only input format for scenarios where StaRVOOrS is used. In fact, in a later phase of the project, we will investigate suitable front end formats for ppDATE, with the goal to achieve highest usability and intuitive usage in an integrated development process and environment. For instance such a front end format could potentially be closer in spirit to JML, in the form of annotations to P itself, than to an explicit automata view.

The first step in the depicted workflow is the extraction of those parts of the specifica-

tion that are in scope for being verified by the deductive verification unit. Here, we aim at static program verification techniques targeting data-centric properties. In particular, the verifier can attempt to prove the correctness of pre/post-conditions of methods, relative to the methods' implementations in P. In our own implementation of the StaRVOOrS, the deductive verifier will be based on the KeY tool [2, 13], a state-of-the-art tool for Java source code verification (see Sect. 2.1), co-developed by the main applicant. Accordingly, the object-oriented language to be supported by our own implementation will be Java.

Deductive verification tools typically rely on user input to difficult proof steps, like finding loop invariants. However, our framework is designed for fully automated usage of the verifier. Therefore, not all prove attempts will lead to complete proofs. The workflow makes use of both, complete and partial, proofs, when specialising the ppDATE specification.

The purpose of the *partial specification evaluator* for ppDATE is to spare the runtime verification (in the end of the workflow) from checking properties that were proved statically. For instance, post-conditions that were completely proved (relatively to a certain method and pre-condition) do not need to be checked at runtime at all. The more interesting question is how to still make use of the information contained in *partial* proofs for the run-time verification phase. Here, the basic idea is to construct, from the open proof goals, specialisations of the pre-condition to the cases where the post-condition could, respectively could not, be proved. For instance, suppose the original ppDATE automaton features a transition $s \xrightarrow{pre/m()/post} s'$ (where *pre* and *post* are the pre- and post-condition of calling method m). Suppose further the deductive verifier produces a partial, i.e., unfinished proof for $\{pre\}m()\{post\}$. (We ignore s for simplicity.) Then, it is possible, by analysis of the open proof goals, to construct two specialisations pre_1 and pre_2 of pre, with $pre_1 \wedge pre_2 \leftrightarrow pre$, such that pre_1 corresponds to the open and pre_2 to the closed proof branches, respectively, and $\{pre_2\}m()\{post\}$ is a consequence of the partial proof. This can be used by the partial specification evaluator to replace s' with two clones s'_1 and s'_2, and instead of the above transition have $s \xrightarrow{pre_1/m()/post} s'_1$ and $s \xrightarrow{pre_2/m()/true} s'_2$. Thereby, during runtime verification, only the transition to s'_1 will trigger a checking of the postcondition *post*, but not the other transition, as *post* is ensured there statically. This is just to give an impression of what can be done; there will be other usages of (partially) proved results in the specialisation of the ppDATE automaton.

The monitor generator takes as main input the specialised specification, ppDATE'. It actually not only generates a monitor, *performing* the very runtime checks, but also aspects (in the sense of aspect-oriented programming) for *triggering* the runtime checks, to be weaved into the application to be monitored. Our own implementation of the StaRVOOrS framework will partly be based on LARVA [26], a monitor generator for Java programs specified with the DATE format (see Sect. 2.2), co-developed by the co-applicant. To current approach to automata based monitor generation needs to be extended, in the project, to cope with those data centric parts of the specification that could not be (fully) ensured statically. This is particularly true for postconditions as in most cases they involve some kind of procedural checking (e.g., to check that an array is indeed sorted). Here we will pursue alternative approaches, like dedicated nested automata (for checking postconditions), and logic based runtime assertion checking of the kind done for JML specifications [24], possibly using a hybrid of those techniques in the end.

Before weaving the generated aspects into the code to be monitored, further static optimisations will be applied in the *Static Monitoring Optimizer* module, using, and expanding on, recent results in the area of combining static analysis (other than verification)

with runtime verification. In particular, CLARA [20] is a good candidate to base our static monitor optimizer on (see Sec. 2.3). Note that the optimisations can also affect the monitor itself giving the possibility to reduce its size and thus enhancing performance (this part is not shown in Fig. 1).

The final step in the workflow is the actual runtime verification, which executes the weaved program P' in parallel with the resulting monitor. Suitable forms of reporting and analysing the results of runtime verification, in certain cases including error recovery mechanisms, are natural extensions of the framework. They will be addressed in the project, without aiming at full generality, however. Rather, these issues are specific for the demands of a deployment scenario and application area, and will be tailored for specific deployments and case studies.

In addition to what is discussed above, a crosscutting concerns is the treatment of real-time properties. On the runtime side, DATE and LARVA already support timers. On the static verification side, there is recent research on loop bound analysis using a combination of KeY and COSTA [7]. Yet, these two are very different aspects of real-time. Within StaRVOOrS, we will develop a uniform way to specify real-time properties, together with a combined static and runtime verification.

The project will also explore potentials of the framework outside the main workflow as sketched above. One is the possibility of a feedback loop from the runtime verification to the (static) deductive verifier. For instance, there is work on discovering likely invariants by dynamic analysis [33] or testing [36], and StaRVOOrS could well be an ideal framework for dynamic-to-static feedbacks of similar kind. Another issue is to broaden of our current deductive test case generation approach [32] to control related aspects, like call-graph related test coverage criteria.

Clearly, there are further research directions highly relevant in our context, but probably outside of the scope of the project we are applying for herewith. One such is is the usage of both deductive verification and model checking on the static side. This is natural as we are mixing data and control centric aspects in the unified specification language, but it goes beyond what we commit to here.

3.3 Project plan

During the first year, we plan to develop a version of the syntax and semantics of the language ppDATE which already features the combination of data and control centric expressiveness. In parallel, as a motor for further developments, we will develop a prototype of the static specification evaluator, performing, at first simple, transformations triggered by input from the deductive verifier. The end of year two will see a prototypical implementation of the entire framework (excluding yet the static monitoring optimiser), not yet using static verification in all ways ever possible. In year three, the prototype will be applied to a simple but realistic case study. Also, we will work on the static monitoring optimiser, and examine deeper, further reaching exploitation of partial proofs in the partial evaluator, including changes to the core deductive machinery itself. In year four, more challenging case studies will be performed, and front end formats to ppDATE will be studied. In parallel, the focus will turn to real-time properties. In year five, all of the above will consolidate, and we will develop a solid integration into a common development environment. In addition, the aforementioned dynamic-to-static feedback loop will be explored, and the usage of the framework for test case generation.

3.4 Measures of Success

Our project will be successful if the following is achieved:

1. A completed PhD thesis finished by the end of the project.
2. At least 5 papers published in high-level conferences, and at least 2 journal papers submitted.
3. A complete, well defined, integrated specification language, with high usability, for combined data centric, control centric, and real-time properties.
4. The fully developed, and implemented, StaRVOOrS framework.
5. Successful application of the framework to realistic case studies.

4 Significance

By achieving our goals we will be extending the state-of-the-art of the usage of lightweight formal methods for software, in so far as a much higher degree of confidence in desired system properties can be achieved, while still satisfying high demands on usability and automation. This will enable a better adoption into industrial practise than contemporary methods with a comparative level of achieved confidence, in particular in the growing segment of safety critical software production and certification.

5 Preliminary Results by the Applicants

Wolfgang Ahrendt has a strong record in deductive (static) software verification. The particular focus of his work is logics, calculi, and systems for the verification of object-oriented programs in general, and Java in particular. Research results have constantly been implemented in (branches of) the KeY system [2], which will also serve as the basis for the 'Deductive Verifier' unit in our implementation of the StaRVOOrS. Related to the exploitation of partial proofs in the 'Partial Specification Evaluator' is the work on generating test cases from partial proofs ([32, 5]). Related to the real-time aspects of our project, we have collaborated with the Radboud University Nijmegen (see 6) on statically verifying Java loop bounds [47], using a combination of KeY and COSTA [7]. W. Ahrendt has co-developed a program logic and calculus which identifies the logically quantifiable objects with the actually created ones [3], addressing a gap between logical and and runtime models of execution. He also has developed a logic and calculus for compositional verification of locally concurrent, globally distributed objects [4]. This is related to StarVOOrS as it allows to connect an internal, data centric verification of classes with an external, more control centric verification of interfaces.

Schneider's work on runtime verification is also important to the current application. This includes the development of a runtime monitoring framework based on rich automata [26], techniques for runtime verification of real-time properties [28], and its realisation on the tool LARVA for monitoring Java programs [27].

6 Research Collaborations

The project leader and the co-applicant have contact with the following international well-known research groups conducting research on related topics to our proposal:

- Karsruhe Institute of Technology (Germany). Research contacts: Prof. Peter H. Schmitt and Prof. Bernhard Beckert. Long term, and future, collaborators in the KeY project.
- University of Darmstadt (Germany). Research contact: Prof. Reiner Hähnle. Long term, and future, collaborator in the KeY project.
- University of Malta (Malta). Research contact: Dr. Gordon Pace (Associate Professor). Our collaboration with Dr. Pace will be based on our previous work on run-time verification.

- University of Oslo (Norway). Research contacts: Prof. Olaf Owe and Prof. Einar Broch Johnsen. We will collaborate on the topic of compositional object-oriented verification.
- Company aicas (Karslruhe, Germany). Research contact: Dr. James Hunt, the scientific coordinator of the ARTEMIS project CHARTER, in which Chalmers is participating with W. Ahrendt as principal investigator. With the JamaicaVM, aicas is a leading supplier of realtime Java technology for embedded systems. We collaborate on deductive verification of, and test case generation for, realtime Java.
- Radboud University Nijmegen (Netherlands). Research contact: Prof. Marko van Eekelen. We collaborate on the verification of resource bounds in Java, based on the systems COSTA and KeY, in the frame of the the ARTEMIS project CHARTER.
- Centrum Wiskunde & Informatica (CWI, Amsterdam, Netherlands). Research contacts: Prof. Frank de Boer. We collaborate on closing semantic gaps between models of object-orientation in program logics vs. the runtime machinery.

7 Other Grants

The co-applicant of this proposal, Gerardo Schneider, is applying, in the same round, for a project research grant, title *SAMECO: Specification, Analysis and Monitoring of Electronic Contracts*, project number *2012-9463-96085-18*. The project is about the definition of a logic-based formal language for computer mediated transactions ("contracts") including the development of techniques and tools for model checking and monitoring such contracts. Schneider's project does not overlap with the current proposal.

References

[1] W. Ahrendt. Using KeY. In Beckert et al. [13], pages 409–451.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract object creation in dynamic logic – to be or not to be created. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands*, volume 5850 of *LNCS*, pages 612–627. Springer-Verlag, 2009.

[4] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010.

[5] W. Ahrendt and G. Paganelli. Verification driven test gerneration. Deliverable D7.1 of the ARTEMIS project CHARTER (ARTEMIS 2008-1-100039).

[6] I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. *Electr. Notes Theor. Comput. Sci.*, 197(1):45–58, 2008.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'08*, number 5382 in LNCS, pages 113–133. Springer, 2007.

[8] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40:345–364, October 2005.

[9] C. Artho and A. Biere. Combined static and dynamic analysis. In *AIOOL'05*, volume 131 of *Electr. Notes Theor. Comput. Sci.*, pages 3–14, 2005.

[10] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[11] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[12] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[13] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[14] S. Bensalem, Y. Lakhnech, and S. Owre. InVeST: A tool for the verification of invariants. In *CAV'98*, volume 1427 of *LNCS*, pages 505–510. Springer, 1998.

[15] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Trans. Software Eng.*, 28(2):129–145, 2002.

[16] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. In *Runtime Verification (RV)*, volume 4839 of *LNCS*, pages 22–37. Springer, 2007.

[17] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. *J. Log. Comput.*, 20(3):707–723, 2010.

[18] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549. Springer, 2007.

[19] E. Bodden, P. Lam, and L. J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT FSE'08*, pages 36–47. ACM, 2008.

[20] E. Bodden, P. Lam, and L. J. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *RV'10*, volume 6418 of *LNCS*, pages 183–197, 2010.

[21] A. D. Brucker and B. Wolff. Interactive testing with HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Proc. Workshop on Formal Aspects of Testing, FATES*, volume 3997 of *Lecture Notes in Computer Science*, pages 87–102. Springer-Verlag, 2005.

[22] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[23] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[24] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[25] C. Colombo. Practical runtime monitoring with impact guarantees of Java programs with real-time constraints. Master's thesis, University of Malta, 2008.

[26] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 135–149, L'Aquila, Italy, September 2009. Springer-Verlag.

[27] C. Colombo, G. J. Pace, and G. Schneider. LARVA –A Tool for Runtime Monitoring of Java Programs. In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, pages 33–37, Hanoi, Vietnam, 23–27 November 2009. IEEE Computer Society.

[28] C. Colombo, G. J. Pace, and G. Schneider. Safe runtime verification of real-time properties. In J. Ouaknine and F. Vaandrager, editors, *The 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'09)*, volume 5813 of *LNCS*, pages 103–117, Budapest, Hungary, 13-16 September 2009. Springer.

[29] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[30] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME)*, pages 166–174. IEEE Computer Society, 2005.

[31] X. Deng, J. Lee, and Robby. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. 21st IEEE/ASM Intl. Conference on Automated Software Engineering, Tokyo, Japan*, pages 157–166. IEEE Computer Society, 2006.

[32] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich and B. Meyer, editors, *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*. Springer, 2007.

[33] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[34] A. Goldberg and K. Havelund. Automated runtime verification with eagle. In *Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS)*. INSTICC Press INSTICC Press, 2005.

[35] W. Grieskamp, N. Tillmann, and W. Schulte. XRT — exploring runtime for.NET architecture and applications. In B. Cook, S. Stoller, and W. Visser, editors, *Proc. Workshop on Software Model Checking (SoftMC 2005), Edinburgh, UK*, volume 144(3) of *Electr. Notes Theor. Comput. Sci*, pages 3–26, 2006.

[36] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–276, 2009.

[37] R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated software engineering, Antwerp, Belgium*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM.

[38] J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In M. Rochard-Foy and A. Wellings, editors, *Proc. of the 4th Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2006*, pages 107–116. ACM Press, 2006.

[39] B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proc. 10th Conf. on Algebraic Methodology and Software Technology (AMAST), Stirling, UK*, volume 3116 of *LNCS*, pages 241–257. Springer-Verlag, July 2004.

[40] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[41] J. C. King. *A program verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[42] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[43] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft Revision 1.200*, Feb. 2007.

[44] P. O. Meredith, D. Jin, F. Chen, and G. Rosu. Efficient monitoring of parametric context-free patterns. *Autom. Softw. Eng.*, 17(2):149–180, 2010.

[45] W. Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In M. Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer-Verlag, Apr. 2005.

[46] P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In M. Hinchey and T. Margaria, editors, *Proc. 5.IEEE Int.Conf. on Software Engineeging and Formal Methods (SEFM)*, pages 47–56. IEEE Press, 2007.

[47] M. van Eekelen et al. Loop bound analysis. Deliverableg D6.5 of the ARTEMIS project CHARTER (ARTEMIS 2008-1-100039).