A Unified Approach for Static and Runtime Verification: Framework and Applications

Wolfgang Ahrendt[1], Gordon J. Pace[2], and Gerardo Schneider[1⋆]

[1] Dept. of Computer Science and Engineering, Chalmers | Univ. of Gothenburg, Sweden.
[2] Dept. of Computer Science, University of Malta, Malta.
ahrendt@chalmers.se, gordon.pace@um.edu.mt, gersch@chalmers.se

Abstract. Static verification of software is becoming ever more effective and efficient. Still, static techniques either have high precision, in which case powerful judgements are hard to achieve automatically, or they use abstractions supporting increased automation, but possibly losing important aspects of the concrete system in the process. Runtime verification has complementary strengths and weaknesses. It combines full precision of the model (including the real deployment environment) with full automation, but cannot judge future and alternative runs. Another drawback of runtime verification can be the computational overhead of monitoring the running system which, although typically not very high, can still be prohibitive in certain settings. In this paper we propose a framework to combine static analysis techniques and runtime verification with the aim of getting the best of both techniques. In particular, we discuss an instantiation of our framework for the deductive theorem prover KeY, and the runtime verification tool Larva. Apart from combining static and dynamic verification, this approach also combines the data centric analysis of KeY with the control centric analysis of Larva. An advantage of the approach is that, through the use of a single specification which can be used by both analysis techniques, expensive parts of the analysis could be moved to the static phase, allowing the runtime monitor to make significant assumptions, dropping parts of expensive checks at runtime. We also discuss specific applications of our approach.

1 Introduction

There is a significant quest from the software industry for *lightweight formal methods* — methods which achieve a high degree of confidence in desired (sub-) system properties, while satisfying high demands on usability and automation. There are various reasons for this increasing need in software development, including the following recent parallel trends:

– *Model driven development.* There is an ever more dominant role of models in the software development process.

⋆ Corresponding author.

- *Automated software engineering.* There is a trend to (partly) automate even more steps in the development cycle.
- *Exploding complexity of embedded software.* The demands on the safety of the increasingly complex embedded units is typically extremely high.
- *Concurrency and distribution.* Distributed architectures led to an increase in the possible causes of failure. On a more fine-grained level, concurrency is also becoming more important due to the rise of multi-core processors.
- *Software standards and certification.* In certain domains (e.g., avionics, automotive, medical), standards for architecture, interfaces, and processes are becoming very important.
- *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.

Even if *static verification* of software has become more relevant, effective and efficient, overcoming certain inherent limitations has proved to be hard. Certain static verification techniques have high precision, in which case powerful judgements are still too hard to achieve automatically, while others 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 *runtime verification* which, compared to static verification, has complementary strengths and weaknesses. Runtime verification combines the full precision of the execution model (even including the real deployment environment) with full automation. On the other hand, it only ever judges observed runs, and cannot judge alternative and future runs. Another drawback is the computational overhead of monitoring the running system which, although typically not very high, can still be prohibitive in certain settings.

In this paper, we propose a unified static and runtime verification framework for object-oriented software. The aim 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 paper is organised as follows. We first give some background on static and dynamic verification techniques/tools. In Section 3 we present our framework, and in Section 4 we provide an example to illustrate how our framework could be applied in practice. We briefly describe some application domains of our framework in Section 5. We discuss related work in Section 6 and we conclude in the last section.

2 Background

2.1 Static Verification of Software

Principles
Static software verification reasons 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 [20, 52, 29, 11] or C# [8] and they are ready to deal with industrial applications [39, 47, 51, 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 [8, 11]. On the other hand, 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 [19, 34, 30, 10], and bug finding [50, 36].

Among the state of the art efforts is the KeY tool [2], which it is close to complete coverage of the Java programming language [9]. 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. We describe KeY in more details below.

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.

KeY: A System for Static Verification of Java Programs
KeY is a deductive verification system for data centric *functional correctness* properties of Java source code. From Java code augmented with specifications given in JML (Java Modelling Language [43]), KeY generates proof obligations in a program logic, called *dynamic logic* (DL) for Java [11]. DL extends first-order logic with two additional operators, $\langle p \rangle \phi$ and $[p]\phi$, where p is a program and ϕ is another DL formula. A formula $\langle p \rangle \phi$ is true in a state s if there *exists* a terminating run of p, started in s, which results in a state where ϕ is true. As for the other operator, a formula $[p]\phi$ is true in a state s if *all* terminating runs of p, started in s, result in a state where ϕ is true. For deterministic programs p, the difference between $\langle p \rangle \phi$ and $[p]\phi$ is only termination. Hoare logic [37], can be seen as a special case, as the Hoare triple $\{\phi\}p\{\psi\}$ is equivalent to the DL formula $\phi \rightarrow [p]\psi$.

The core of KeY is a theorem prover for validity of Java DL formulas, using a sequent calculus. We cannot introduce the calculus here, but we mention a typical pattern of sequents. If Γ is a list of formulas, the sequent $\Gamma \vdash \langle p \rangle \phi$ means that p, if started in a state fulfilling all Γ, terminates in a state fulfilling ϕ. For instance, $\texttt{x} < \texttt{y} \vdash \langle \texttt{tmp:=x;x:=y;y:=tmp;} \rangle\ \texttt{y} < \texttt{x}$ is a valid sequent. The calculus uses the *symbolic execution* paradigm. For that, DL is extended by 'explicit substitutions'. During symbolic execution of p, the effects of p are *gradually*, from the front, turned into explicit substitutions. Meaning that after some proof steps, a certain prefix of p has turned into a substitution σ, representing the effects so far, while a 'remaining' program p' is yet to be executed. While verifying p, an intermediate proof node may look like $\Gamma \vdash \sigma\langle p' \rangle \phi$, telling that, if Γ was true before p, and σ is the accumulated effect up to now, then ϕ will be true after executing the remaining program p'. Note that most proofs branch over case distinctions, largely triggered by Boolean decisions in the source code. The branching happens by applying rules like the following, simplified[3] if rule:

$$\textsf{if}\ \frac{\Gamma, \sigma(b) \vdash \sigma\langle s_1\ \ \omega \rangle \phi \qquad \Gamma, \sigma(\neg b) \vdash \sigma\langle s_2\ \ \omega \rangle \phi}{\Gamma \vdash \sigma\langle \texttt{if}\ b\ s_1\ \texttt{else}\ s_2\ \ \omega \rangle \phi}$$

Unlike the explicit substitutions preceding the diamond modalities "$\langle \ldots \rangle$", the notation $\sigma(b)$ indicates that σ is *applied* to b, and thereby resolved. Similar for $\sigma(\neg b)$. Through rules like the above, the left side of any sequent, on any branch of the symbolic execution proof, lists conditions for the current execution path to be taken (in addition to the original precondition, in case there is any).

2.2 Runtime Verification of Software

Principles

Runtime verification is a technique for monitoring the execution of a software system, detecting violations as they appear at runtime. In recent years researchers have implemented monitoring tools which usually compile high-level (temporal) properties into monitor implementation (e.g., [21, 40, 27, 28, 5, 26]). There are two main concerns when using runtime verification:

1. In order to minimise the possibility of erring it is desirable that monitors are automatically synthesised from formally specified properties.
2. Though a minimal runtime overhead is acceptable, it is of course 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.

There are two main flavours of runtime verification — *synchronous monitoring*, in which, after each performed action, the system does not proceed further until the monitor confirms that the action did not violate the specification, and *asynchronous monitoring*, in which the system logs all relevant events, which are processed independently by the monitor, possibly on a separate address-space.

[3] The simplified rule ignores side effects or exceptions possibly caused by b.

While the latter is attractive in that it induces minimal overheads on the system, the *a posteriori* nature of the analysis makes it useless if one wants to discover and address problems in real-time. Although in-between solutions have been proposed (e.g., see [24]), they are far from being universally applicable, and thus, if one wants to have a guarantee that the system does not proceed beyond a violation, one has no choice but to pay the cost in terms of overheads induced by synchronous monitoring.

Different solutions based on optimisations have been presented to alleviate the overhead problem, e.g. [13, 14]. Further approaches aim at obtaining small monitors by construction [45], or use some kind of overhead guarantee, as proposed in [23]. Despite the advance of the state-or-the-art with such approaches there is still need to improve runtime monitoring techniques as motivated by the development of specific techniques to improve monitor efficiency [18].

In the following, we give a brief overview over state-of-the-art runtime monitoring tools developed in recent years, without claiming completeness. ConSpec [3] inlines a runtime monitor into applications on mobile devices based on observed contract violations. JavaMOP [21] 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 [33] 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 [28] 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 [5] is an extension to AspectJ allowing the specification of trace patterns, also supporting parametrisation of events. This work has been extended in [15] to improve efficiency by making a temporal and spatial partitioning among collaborative users.

Larva: A Runtime Verification tool for Java

Larva (*Logical Automata for Runtime Verification and Analysis*) [26], 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 —*Dynamic Automata with Timers and Events*), Lustre, and a subset of the duration calculus.

As an example of the kind of properties one can express in DATEs and verify with Larva let us consider a system where one needs to monitor the number of successive bad logins and the activity of a logged in user. By having access to *badlogin*, *goodlogin* and *interact* events, one can keep a successive bad-login counter and a clock to measure the time a user is inactive. Fig. 1 shows the property that allows for no more than two successive bad logins and 30 minutes of inactivity when logged in, expressed as a DATE. Upon the third bad login or 30 minutes of inactivity, the system reverts to a bad state. In the figure, transitions are labelled with events, conditions and actions, separated by a backslash. It is assumed that the bad login counter is initialised to zero.

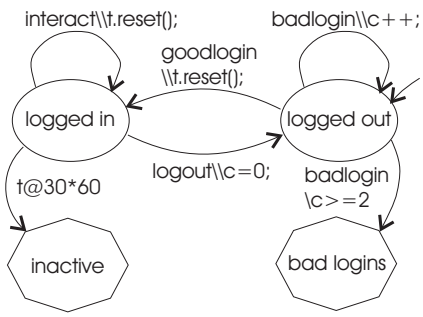

Fig. 1. The DATE of the bad logins scenario

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.

3 A Proposed Framework for Integrated Static and Runtime Verification

In this section we present our framework. We start by discussing a unified language for specifying both static and runtime properties, we then describe our framework in general terms, and we finally discuss some interesting features that could be added to enhance our framework.

Though the conceptual model underlying our framework is general and tool-independent, we use LARVA and KeY as a basis to the proposed unified language, and as instances of some of the modules to be used in the framework.

3.1 A unified specification language for static and dynamic verificaion

In order to explore the proposed framework, we are investigating a concrete instantiation — combining the deductive verification tool KeY with the runtime verification tool LARVA. One of the first main challenges is that of identifying a unified specification language. While KeY uses pre- and postconditions for specifications, LARVA uses DATEs — essentially symbolic automata, with timers (allowed to be used also as stopwatches) and the means for dynamic replication of properties (for instance, a property which will be replicated for each user in the system).

As briefly described in the previous section, while KeY addresses the behaviour of a method as a relation, DATEs are less structured and are triggered with events happening in the system, which may refer to method entry and exit points (although with no native notion of identifying the entry and related

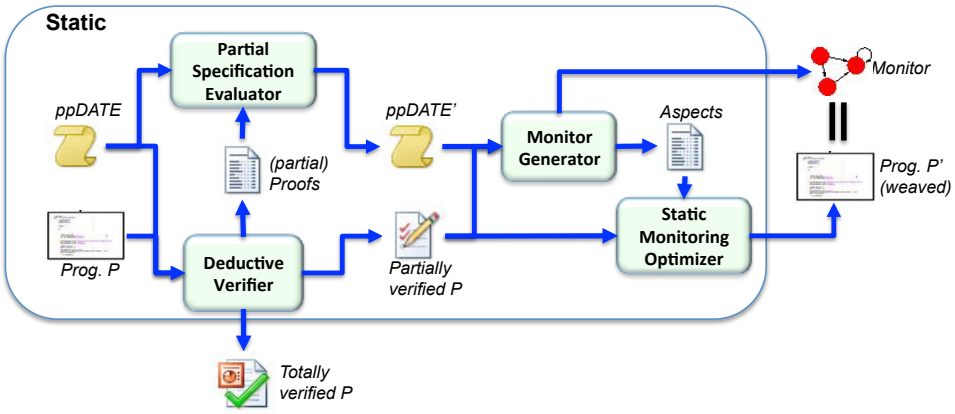

Fig. 2. High-level description of the framework

exit of a single call), but also to exceptions raised by the system. To combine the approach, we propose *ppDATEs*, an extension of DATEs with the notion of pre- and postconditions, enabling transitions from a state to another whenever a method call satisfying the precondition terminates with the postcondition holding. Notationally, pre-/postconditions are additional elements of transition labels, put in the front/end of the other label elements: $s \xrightarrow{pre \backslash \ldots \backslash post} s'$

ppDATEs enable the co-specification of data centric, control centric, and real time aspects of a system in a unified way. Concretely, ppDATE describes communicating automata 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.

3.2 Description of the framework

An abstract view of the proposed framework, taking as input an object-oriented program P and a specification of the desired properties, can be found in Fig. 2. We describe our framework in what follows based on the figure.

Deductive verification tools typically rely on user input to difficult proof steps, like finding loop invariants. However, the proposed framework is designed for fully automated use of the verifier, represented in the figure by the *Deductive Verifier* module. Therefore, not all proof 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* is to spare the runtime verification (at the end of the workflow) from checking properties that were

proved statically. For instance, postconditions that were completely proved (relatively to a certain method and precondition) 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 precondition to the cases where the postcondition could, respectively could not, be proved. For instance, suppose the original ppDATE automaton features a transition $s \xrightarrow{pre\backslash m()\backslash post} s'$ (where *pre* and *post* are the pre- and postcondition of calling method m). Suppose further the deductive verifier produces a partial, i.e., unfinished proof for $pre \vdash \langle m() \rangle post$ (ignoring 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 $pre_2 \vdash \langle m() \rangle post$ 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 $s \xrightarrow{pre_1\backslash m()\backslash post} s'_1$ and $s \xrightarrow{pre_2\backslash m()\backslash true} s'_2$. 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.

The *Monitor Generator* takes as main input the specialised specification, ppDATE′. From such a specification, it uses aspect-oriented programming techniques to capture relevant system events, and implements runtime checks to ensure no violation takes place. The current approach to automata based monitor generation used in Larva [25], cannot deal in a satisfactory manner with the 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 [22].

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 [18] is a good candidate to base our static monitor optimizer. 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. 2).

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 future work, 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.

3.3 Additional features

In addition to what is discussed above, a crosscutting concerns is the treatment of real-time properties. On the runtime side, LARVA already supports timers. On the static verification side, there is recent research on loop bound analysis using a combination of KeY and COSTA [4]. Yet, these two are very different aspects of real-time.

The framework has further potentials 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 [31] or testing [35], and the proposed framework could well be ideal for dynamic-to-static feedbacks of similar kind. Another issue is the broadening of our current deductive test case generation approach [30] to control related aspects, like call-graph related test coverage criteria.

4 An Illustrative Example

Let us reconsider the login-scenario, extending on Fig. 1, introduced in section 2.2 when describing the tool LARVA. We assume now that whenever a user logs in, she is added to a set of current users. A specification of this scenario will consist of a number of parallel ppDATEs specifying different properties. Such ppDATEs could be activated by mutual synchronisation or by events. For instance the high level property about the login will consist of a modified version of the DATE shown in Fig. 1, where the `good-/bad-login` events are augmented with the user's identity as a parameter. Additionally we will have, in parallel, another ppDATE which includes pre/post-conditions of data sensitive operations, like the method call `users.add(u)` which is activated whenever the event `goodlogin(u)` happens. An according transition will look like $s \xrightarrow{pre\backslash \texttt{add()} \backslash post} s'$.

Before giving details about *pre* and *post*, let us discuss the Java implementation of the set of users. We assume the set is implemented with help of an array `arr`, using hashing for fast look-up. Hash conflicts are resolved by open addressing, meaning the method `add` first tries to put the object into `arr` at the position of the computed hash code. If that index is occupied, however, `add` searches for the nearest following index which is free. The set has a capacity limited by the length of `arr`. To enable easy checking whether or not the capacity is reached, a field `size` keeps track of the number of stored objects.

The ppDATE specifying the behaviour of the user set will therefore contain a transition $s \xrightarrow{\texttt{size<arr.length} \backslash \texttt{add()} \backslash \exists\, i.\, \texttt{arr[}i\texttt{]=o}} s'$, among others. To deal with the postcondition, the runtime verifier may use a technique known as 'runtime assertion checking', where logical formulas are operationalised [22]. For our example, $\exists\, i.\, \texttt{arr[}i\texttt{]} = \texttt{o}$ would be turned into a loop walking through the array.

Checking the postcondition needs to be done each time the transition fires. However, we can optimise away this runtime check for certain cases, using static verification with KeY. If one tries to statically prove, with KeY, that `add`'s implementation is correct with respect to some JML specification, KeY first will gener-

ate proof obligations in form of DL sequents. One of them could look like the following: $\texttt{size} < \texttt{arr.length} \vdash \langle \texttt{add(o)} \rangle \exists i.\ \texttt{arr}[i] = \texttt{o}$
When constructing a proof of this sequent, KeY will branch over case distinctions in the code of `add`, such as whether the initial hash index is free or occupied. The two sequents resulting from that branching look like:

$$\texttt{size} < \texttt{arr.length},\ \texttt{arr[o.hashCode()\%arr.length]} = \texttt{null} \ \vdash \ \ldots$$

and

$$\texttt{size} < \texttt{arr.length},\ \neg\, \texttt{arr[o.hashCode()\%arr.length]} = \texttt{null} \ \vdash \ \ldots$$

The first branch will be easier to automatically close by KeY than the second, which requires handling a loop searching for the next free index. Therefore, if KeY runs in auto-mode, excluding using interaction, it might only close the first branch.

As we have managed to prove one of the branches, there will be no need to monitor the case when the initial hash index is free, and only the branch when this is not the case will be monitored. Therefore, we can replace the transition given above by two transitions:

$$s \xrightarrow{\texttt{size<arr.length},\ \texttt{arr[o.hashCode()\%arr.length]=null}\backslash \texttt{add()}\backslash true} s'_1$$

and

$$s \xrightarrow{\texttt{size<arr.length},\ \neg\, \texttt{arr[o.hashCode()\%arr.length]=null}\backslash \texttt{add()}\backslash \exists i.\, \texttt{arr}[i]=\texttt{o}} s'_2$$

Thereby, during runtime verification, only the transition to s'_2 will trigger a checking of the postcondition, but not the other transition, as *true* is ensured trivially. Given a hashtable that is well dimensioned, the cheaper case will be more frequent at runtime.

5 Applications

5.1 Electronic and legal contracts

The term 'contract' has mostly been used in software systems as a metaphor and not according to the common meaning of the word. The first use of the term in connection with software programming and design was done by Meyer in the context of the language Eiffel (*programming-by-contracts*, or *design-by-contract*) [46]. Software contracts can appear as integrated part of a programming language, like in Eifel, or phrased in a special contract language, like JML [42] (supported by KeY) and Code Contracts [44]. Similarly, as a metaphor, the term has been used to describe interfaces of component-based systems or service-oriented architectures.

In the following, however, the word *contract* will be used as a general term to describe any kind of normative document, including contracts in the legal sense and other agreements where the different parties involved engage on certain obligations. Here, we are primarily interested in *electronic agreements*, which form an electronic version of legal contracts, and the role of static and dynamic verification plays in their analysis.

As a simple example consider agreements in the context of installing application on a mobile phone. In such cases the contract is shown in natural language

(e.g., English) and the user must accept the terms and conditions stipulated there; otherwise the application is not installed. Ideally, each user would read and understand the agreement, and foresee the consequences in case of violation of certain clauses. Reality is different, though. So how else could one validate a contract between two parties before accepting it? This requires that the contract is fully formalised, which is not an easy task as witnessed by a number of current research papers on the topic (see for instance [48, 49] and references therein).

One solution is to monitor the system using a contract as the specification, possibly giving a notification before allowing actions which may lead to a violation to go through. Our proposal is that a third party would statically verify the application software against the agreement, leaving as little as possible to be verified at runtime in the form of a monitor provided with the application. This ensures contract adherence, whilst keeping overheads low.

5.2 Transaction-handling systems

Systems which handle transactions (such as financial payments) are becoming increasingly prevalent. Although various design patterns are used to control the complexity of their development, they still pose various challenges to their verification:

- *Concurrency.* Typically, a transaction-handling system handles many transactions concurrently, and unless appropriate design principles are adhered to, this can lead to an overload in complexity. The main principle is to structure transactions to act in a manner which externally is perceived to be atomic — the operation takes place without the possibility of interference. While being a sound design principle, ensuring that the transactions are internally built to satisfy this is rarely easy.
- *Long-lived transactions.* Whenever transactions have to communicate with real-life systems, they may end up with a substantial increase of their lifespan. This means that the complete locking of all necessary resources for the duration of the transaction is not an option. This requires the use of more complex design patterns to engage and use resources in a rational way.
- *Handling failure.* Various transactions interact with external systems (e.g., a payment transaction may have to communicate with a bank) over which the system has no control. One of the side-effects of these interactions is that the external systems may fail. Handling such failure is not trivial, and although approaches such as compensations have been proposed and used, entwining the logic of normal and exceptional (failing) behaviour induces an overhead in complexity.
- *Varying system loads.* Although typically transactions are not individually computationally expensive, the concurrent nature of the handling of transactions introduces a varying load on the machine processing them. In many domains, this load frequently features regular surges in activity, during which performance becomes an important issue. For example, in an online betting system, the number of incoming bets per minute may peak in the last minutes before an important football match.

The first three issues indicate that such systems are ideally verified, even if runtime verification may introduce unacceptable overheads. We thus believe that transaction-handling systems can be an ideal domain on which to apply our proposed framework.

6 Related Work

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 [7], 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 [12] integrating algorithmic and deductive verification techniques, using abstraction, to verify invariance properties.

More recently, some research has been conducted aiming at a combination of static *analysis* (other than *verification*) and runtime verification in different ways. Arhto and Biere describes an architecture based on JNuke where Java programs can be statically and dynamically analysed [6]. 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 [16] 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 [17] 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.

CLARA is a framework to statically optimise runtime monitoring [18], which uses static analysis techniques to operate on the monitors themselves with the aim of improving performance, as opposed to the combination of static analysis and verification with runtime verification techniques to verify software.

As opposed to [6], 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 [16] our proposed approach 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.

A related structure to ppDATEs has already been explored in [32], a work supervised by one of the authors of this paper.

7 Conclusion

We have presented the conceptual model of a framework for the verification of object-oriented systems. The proposed framework is based on a suitable combination of static and dynamic verification techniques, in particular based on the underlying approaches of KeY and Larva. We have proposed ppDATEs as a unified specification language for describing both static and dynamic properties, and we have shown an example to illustrate how our approach could be used. We have also described two application domains that we believe could benefit from our approach.

This is a position paper and as such much work is still to be done, starting with a formal definition of ppDATEs and ending with a full implementation of the framework, including proving interesting properties about the approach and applying it to real case studies.

References

1. W. Ahrendt. Using KeY. In Beckert et al. [11], 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. I. Aktug and K. Naliuka. Conspec: A formal language for policy specification. In *FLACOS '07*, pages 107–109, Oslo, Norway, October 2007.
4. 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.
5. 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.
6. 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.
7. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.
8. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In *CASSIS'05*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
9. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In *Java Card 2000*, LNCS 2041, pages 6–24. Springer, 2001.
10. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In *TAP'07*, LNCS 4454. Springer, 2007.
11. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
12. 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.
13. 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.

14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. A. D. Brucker and B. Wolff. Interactive testing with HOL-TestGen. In *FATES'05*, volume 3997 of *LNCS*, pages 87–102. Springer-Verlag, 2005.
20. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *FM´03*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
21. F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.
22. Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *SERP'02*, pages 322–328. CSREA Press, 2002.
23. C. Colombo. Practical runtime monitoring with impact guarantees of Java programs with real-time constraints. Master's thesis, University of Malta, 2008.
24. C. Colombo, G. J. Pace, and P. Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 40:1–26, 2012.
25. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS'08*, volume 5596 of *LNCS*, pages 135–149. Springer-Verlag, September 2009.
26. C. Colombo, G. J. Pace, and G. Schneider. Larva - a tool for runtime monitoring of java programs. In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.
27. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.
28. 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 *TIME'05*, pages 166–174. IEEE Computer Society Press, June 2005.
29. X. Deng, J. Lee, and Robby. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE'06*, pages 157–166. IEEE Computer Society, 2006.
30. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *TAP'07*, volume 4454 of *LNCS*. Springer, 2007.
31. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
32. K. Falzon. Combining runtime verification and testing techniques. Master's thesis, University of Malta, 2010.
33. A. Goldberg and K. Havelund. Automated runtime verification with eagle. In *MSVVEIS*, 2005.

34. W. Grieskamp, N. Tillmann, and W. Schulte. XRT — exploring runtime for.NET architecture and applications. In *Proc. Workshop on Software Model Checking (SoftMC 2005), Edinburgh, UK*, volume 144(3) of *Electr. Notes Theor. Comput. Sci*, pages 3–26, 2006.
35. 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.
36. R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *ASE'10*, pages 143–146, New York, NY, USA, 2010. ACM.
37. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, Oct. 1969.
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 *JTRES'06*, 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 *AMAST04*, volume 3116 of *LNCS*, pages 241–257. Springer-Verlag, 2004.
40. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A runtime 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*, 2007.
44. F. Logozzo. Our experience with the code contracts static checker. In *VSTTE'12*, volume 7152 of *LNCS*, pages 241–242. Springer, 2012.
45. 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.
46. B. Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.
47. W. Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In *FASE'05*, volume 3442 of *LNCS*, pages 357–371. Springer-Verlag, 2005.
48. G. J. Pace and G. Schneider. Challenges in the specification of full contracts. In *iFM'09*, volume 5423 of *LNCS*, pages 292–306, 2009.
49. C. Prisacariu and G. Schneider. A dynamic deontic logic for complex contracts. *J. Log. Algebr. Program.*, 81(4):458–490, 2012.
50. P. Rümmer. Generating counterexamples for Java Dynamic logic. In *Prel. Proc. of Workshop on Disproving at CADE'05*, pages 32–44, 2005.
51. P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In *SEFM'07*, pages 47–56. IEEE Press, 2007.
52. K. Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.