StaRVOOrS — Episode II

Strengthen and Distribute the Force*

Wolfgang Ahrendt[1], Gordon J. Pace[2], and Gerardo Schneider[3]

[1] Chalmers University of Technology, Sweden.
ahrendt@chalmers.se
[2] University of Malta, Malta.
gordon.pace@um.edu.mt
[3] University of Gothenburg, Sweden.
gerardo@cse.gu.se

Abstract. Static and runtime techniques for the verification of programs are complementary. They both have their advantages and disadvantages, and a natural question is whether they may be combined in such a way as to get the advantages of both without inheriting too much from their disadvantages. In a previous contribution to ISoLA'12, we have proposed StaRVOOrS ('Static and Runtime Verification of Object-Oriented Software'), a unified framework for combining static and runtime verification in order to check data- and control-oriented properties. Returning to ISoLA here, we briefly report on advances since then: a unified specification language for data- and control-oriented properties, a tool for combined static and runtime verification, and experiments. On that basis, we discuss two future research directions to strengthen the power, and broaden the scope, of combined static and runtime verification: (i) to use static analysis techniques to further optimise the runtime monitor, and (ii) to extend the framework to the distributed case.

1 Introduction

The development of *lightweight* verification techniques in what concerns ease of use and automation is considered to be one of the major challenges being addressed by the verification community.

Runtime verification is one such technique: a monitor is usually automatically extracted from a property written in a formal language, and an executable program automatically synthesised. The monitor is then run in parallel with the monitored program, checking at runtime that its underlying property is being satisfied by the *current* run, and flagging a violation if this is the case. Though the overheads induced by runtime verification are small when compared to the

* Published in *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation - ISoLA'16 (1); Track: Static and Runtime Verification: Competitors or Friends?*, volume 9952 of LNCS, pages 402-415. Springer, Oct 2016. DOI: 10.1007/978-3-319-47166-2_28.

computational effort required by most static analysis and verification techniques, these can still be a problem in certain settings.

Static verification has the advantage of being used pre-deployment, coming with strong guarantees in what concerns correctness for *all* possible runs. This generality is, however, hard to achieve (if not impossible) automatically, in particular when verifying data-oriented properties. Among other things, *loop invariants* typically need to be provided by a human user. Verification systems therefore rely on code annotations, or interactive proof construction. With that, they can achieve a lot, however introducing the additional constraint of needing highly trained experts.

Another dimension, somewhat orthogonal to the above, are complementary issues with checking *data-oriented* and *control-oriented* properties. Data-oriented properties (e.g. *all the numbers stored in the array are positive*) are typically very costly to monitor fully at runtime. Control-oriented properties (e.g. *files can be read only between a login and a logout*), on the other hand, typically require (often manual, sometimes unsafe) abstractions before they can be efficiently verified statically.

In 2012 we introduced StaRVOOrS to the ISoLA community [3], a promise of a unified framework for the specification and verification of data- and control-oriented properties combining static and runtime verification techniques. Though the approach was sketched as tool- and language-independent, had discussed a possible implementation targeting Java programs based on the runtime verifier LARVA [10] and the static verifier KeY [5] .

That promise started to materialise in recent years in the form of two published papers. In [1] we introduced the automata-based formalism *ppDATE* which may be seen as an extension of *DATE* [9] (the underlying specification language of LARVA), extended with pre/post-conditions. We gave a high-level description of the algorithm to translate *ppDATE* into *DATE*. In [8] we presented the tool StaRVOOrS, a full implementation of this framework.

In this paper we report on our achievements concerning StaRVOOrS (Section 2), and we discuss two interesting extensions and research directions: (i) the use of static analysis techniques to further optimise our runtime monitors, in particular by using control-flow approaches (Section 3), and (ii) the extension of the framework to the distributed case (Section 4).

2 StaRVOOrS — Episode I

StaRVOOrS (STAtic and Runtime Verification of Object-ORiented Software) [3] is a framework for the specification of data- and control-oriented properties, and their verification using static and dynamic techniques. It combines the use of the deductive source code verifier KeY [5] with that of the runtime monitoring tool LARVA [10] to analyse and monitor systems with respect to a specification written in a formalism called *ppDATE*.

KeY is a deductive verification system for data-centric *functional correctness* properties of Java source code that generates proof obligations from a Java

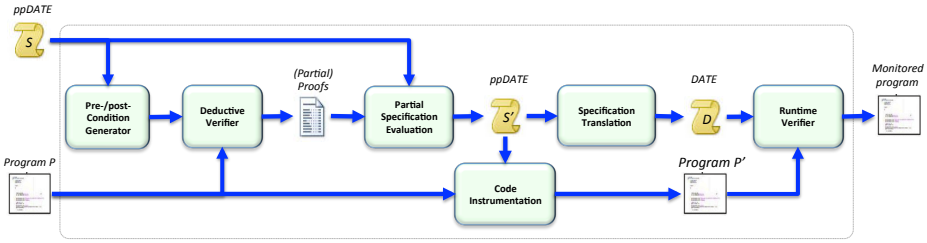

Fig. 1. High-level description of the StaRVOOrS framework workflow

program enriched with annotations written in JML (Java Modeling Language) [21]. These proof obligations are written in *dynamic logic*, a modal logic tailored to reason about programs.

LARVA (*Logical Automata for Runtime Verification and Analysis*) [10] is an automata-based tool for the runtime verification of Java programs. It automatically generates a runtime monitor from a property written in the automata-based specification formalism *DATE* (*Dynamic Automata with Timers and Events*). LARVA transforms the specification into monitoring code together with AspectJ code which links the system with the monitors.

In order to combine, and get advantage of, these two verification approaches, we have defined a specification language able to represent both data- and control-oriented properties. For the control-oriented part we rely on *DATE*s, which to a certain extent also allows for the specification of data. We extend *DATE* with pre/post-conditions (or more precisely, with *Hoare triples*) in order to get more elaborated ways to specify the data-oriented part.

In the rest of this section we briefly present the StaRVOOrS workflow, we describe *ppDATE* through an example, and we give an overview of the tool and some preliminary experiments.

The StaRVOOrS Workflow. The abstract workflow of the use of StaRVOOrS is given in Fig. 1. Given a Java program P and specification S of the properties to be verified, these are transformed into suitable input for the *Deductive Verifier* module (i.e. KeY) which attempts to statically prove the properties related to pre- and post-conditions. If any part of the specification is not fully verified by KeY, it will be left, in a specialised form, in the specification to be verified at runtime. The approach uses the *partial* proofs generated by KeY, which are used to generate conditions for execution paths not statically verified. The *Partial Specification Evaluator* module then rewrites the original specification S into S', refining the original pre-conditions with the path conditions resulting from partial proofs, thus covering only executions that are not closed in the static verification step. The *Specification Translation* then converts the *ppDATE* specification S' into an equivalent specification in *DATE* format (D) which can be used by the runtime verifier LARVA. The *DATE* specification language does

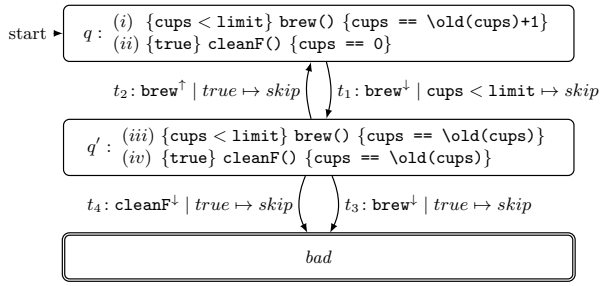

Fig. 2. A *ppDATE* controlling the brew of coffee

not support pre/post-conditions which thus have to be translated to use notions native to the LARVA input language. This also requires a number of changes to the system (through the *Code Instrumentation* module), in order to be able to distinguish different executions of the same code unit and adding methods which operationalise pre/post-condition evaluation. The instrumented program P' and the *DATE* specification D are then used by the *Runtime Verifier* LARVA, which generates a monitor M using aspect-oriented programming techniques capturing relevant system events and linking P' to M.

The monitor and the program are executed together after deployment, running P' in parallel with M. The instrumented system identifies violations at runtime, reporting error traces to be analysed.

The Specification Language *ppDATE*. *ppDATE* [1] is a formalism for specifying both control- and data-oriented properties. *ppDATE*s are automata with transitions labelled by a trigger (tr), a condition (c) and an action (a). Together, the label is written $tr \mid c \mapsto a$. Transitions are *enabled* whenever their triggers are active and the conditions guarding them hold. Triggers are activated by the occurrence of either a visible system event, such as the calling or termination of a method execution[4], or a *ppDATE* internal event generated by specific actions executed when a transition *fires* (that is, the transition is taken). The conditions may depend on the values of *system variables* (i.e., variables of the program to be monitored) and the values of *ppDATE variables* (i.e., variables which belong to the *ppDATE*). The latter can be modified via actions in the transitions. States in *ppDATE*s are decorated with Hoare triples of the form {pre} method-name(·) {post}, where pre and post are predicates in first-order logic describing what is to hold after the method method-name(·) is called (post), provided that pre holds before making the call.

We will not present *ppDATE*s formally in this paper, but rather illustrate the formalism through an example. Let us consider a *coffee machine* in which the filters needs to be cleaned after a certain amount of coffee cups are brewed.

[4] $\sigma^{\downarrow}$ means that method σ has been called and $\sigma^{\uparrow}$ means that method σ has terminated its execution.

After this maximum number of brewed cups is reached the machine should stop brewing more cups until the filters are cleaned. The brewing process cannot be interrupted: no new coffee cup can be brewed nor the filters be cleaned until the brewing is done.

Fig. 2 illustrates a *ppDATE* describing part of the behaviour of the coffee machine. Among other things, the *ppDATE* specifies the property that *it is not possible to brew one more coffee cup or to clean the filters until the brewing process is done*. That is, whenever the coffee machine is not active (i.e. is not brewing) and the method `brew` starts the coffee brewing process, it is not possible to execute this method again or to execute the method `cleanF`, which initialises the task of cleaning the filter, until the brewing terminates.[5]

The *ppDATE* may be interpreted as follows: initially being in state q, whenever method `brew` is invoked, if it is possible to brew a cup of coffee (i.e. the machine is not active and the limit of coffee cups was not reached yet), then transition t_1 shifts the automaton from state q to state q'. While in q', if either method `brew` or method `cleanF` are invoked, then transitions t_3 or transition t_4 shifts to state *bad*, respectively, in which case the property is violated. On the other hand, if method `brew` terminates its execution, then transition t_2 is fired going from state q' to state q.[6] The Hoare triples in state q specify the following: (i) if the amount of brewed coffee cups has not reached its limit yet, then a coffee cup is brewed; (ii) cleaning the filters sets the amount of brewed coffee cups to 0. The Hoare triple in state q' ensures that: (iii) no coffee cups are brewed; (iv) filters are not cleaned. Note that the Hoare triples make reference to the state of the coffee machine, i.e. there is no information on whether the machine is active or not. This is because the machine's status is implicitly defined by the *ppDATE*'s states. If the *ppDATE* is in state q, the coffee machine is not active, and active if in state q': *ppDATE*s are *context dependent*. This allows us to describe Hoare triples with the same precondition but with different post-conditions, getting a different meaning depending in which state of the *ppDATE* they are defined. To clarify the semantics of *ppDATE*s, consider, for instance, if we are in state q and method `cleanF` is called, thus triggering the Hoare triple requiring the number of cups to be zero upon exiting from the method. This postcondition check is enforced even if, by the time method `cleanF` exits the *ppDATE* has changed state to q'.

Tool and Experiments. We have implemented the StaRVOOrS tool [8], supporting the specification language *ppDATE*. The tool implements the workflow given in Fig. 1, where KeY acts as the Deductive Verifier, and LARVA acts as the Runtime Verifier. At first, the Hoare triples from *ppDATE* are translated to JML, after which KeY attempts to prove them, without user interaction or additional assertions (like loop invariants). KeY cannot complete most proofs this way, but

[5] In what follows when we talk about a *method* we refer to the corresponding method name of a Java implementation of the coffee machine controller.

[6] The names used on the transitions, e.g. t_1, are not part of the language; they are included only to simplified the description of how the *ppDATE* works.

the analysis of the partial proofs produces path conditions for those calls which need to be runtime checked. After refining the Hoare triples accordingly, the resulting *ppDATE* is translated to *DATE*, for which LARVA generates a runtime monitor. The StaRVOOrS tool is fully automatic, i.e., neither any component (KeY, partial proof analysis, specification transformations, LARVA), nor the workflow among the components require the user to interfere.

We have applied the tool to Mondex, an electronic purse application which has been used as a benchmark problem within the Verified Software Grand Challenge context [30]. Our variant is strongly inspired by a JML formalisation given in [29]. However, using *ppDATE*, we could more naturally represent the major 'status' of an observer as automata states, rather than in additional data. In that scenario, the combined approach makes monitoring up to 800 times faster than just using runtime verification [8].

3 Episode II, Trailer 'Control-Flow Optimisation'

Till now, in our framework we have emphasised the control-flow vs. data-flow dichotomy, arguing that although runtime verification can deal with control-flow properties in an effective manner, the approach can result in large overheads when dealing with data-flow. With this in mind, we have adopted static analysis techniques effective for data-flow properties in order to resolve expensive runtime analysis pre-deployment. This is the rationale behind the *ppDATE*s specification language — enabling specification of combined data- and control-flow properties.

Through the use of KeY, in StaRVOOrS we compositionally analyse the *ppDATE* specification without any control-flow information. The analysis looks at individual Hoare triples, either discarding them if a full proof is achieved, or refining their pre-conditions (such that they apply less often) if only a partial proof can be managed. Since *ppDATE*s deal with control-flow through the graph structure of the automaton, and the data-flow through the Hoare triples in the states, the static analysis leaves the *ppDATE* structure unchanged for runtime analysis. However, control-flow of the system might guarantee that parts of the *ppDATE* are not reachable, and thus, the Hoare triples for those states are unnecessary. The approach adopted in StaRVOOrS thus poses two challeges:

(i) Although static analysis is performed only once, pre-deployment, it can be an expensive process, and large specifications might require substantial resources to verify. However, the Hoare triples in the parts of the *ppDATE*s that are unreachable due to the system behaviour, need not be analysed.
(ii) The unreachable triples will result in additional code which dynamically verifies the system behaviour. Although unreachable, this will induce overheads in terms of the instrumented system's memory footprint and also result in additional checks when deciding which pre/post-conditions are applicable due to which *ppDATE* state the system resides in.

One solution is to adopt control-flow static analysis to reduce *ppDATE*s from a control structure perspective. A straightforward solution is to use the

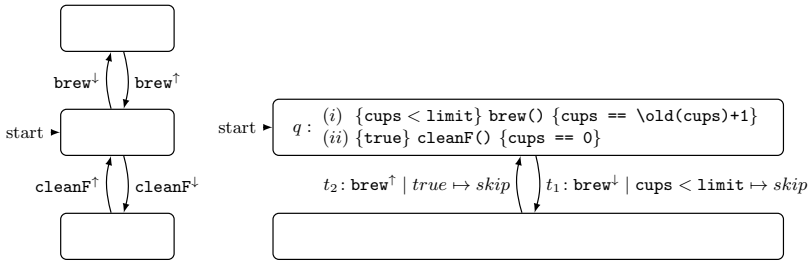

Fig. 3. (left) The control-flow graph of the system under scrutiny; and (right) An optimised *ppDATE* specification of brewing of coffee leaving out unnecessary checks

control flow graph of the system being analysed. For instance, reconsider the coffee-machine example given in Fig. 2. The information we extract from the system under scrutiny can be used to prune (i) transitions which can never be taken; (ii) states which are unreachable; and (iii) Hoare triples which can never be triggered in a particular state. Consider a sequential controller of the coffee-machine, which will never attempt to start cleaning the filter or brewing halfway during a coffee brewing or a filter cleaning, respectively. The control-flow graph extracted from the system would correspond to the graph given in Fig. 3(left). Such a graph can be automatically extracted from the system using standard techniques, which would guarantee that the language of traces described by the graph is an over-approximation of traces that the system can produce[7].

By simply composing the original *ppDATE* specification (Fig. 2) using a *quasi-synchronous composition*[8] with the control-flow graph (Fig. 3(left)), we can obtain a leaner specification (Fig. 3(right)). Further, albeit more sophisticated, analysis can also enable us to discard the bottom state.

The soundness of the optimisation rests on (i) the fact that the control-flow graph provides an over-approximation of possible system behaviour; (ii) taking a quasi-synchronous composition of a *ppDATE* with a control-flow graph effectively results in a *ppDATE* which represents the conjunction of the original property and the property that the system's behaviour remains within the control-flow graph; and (iii) if we know that a system satisfies a property C (the control-flow graph), then verifying a property π is equivalent to verifying $\pi \wedge C$.

[7] Note that, any event not appearing on any outgoing transition from a state is taken to mean that while in that state, that event is guaranteed not to occur. This visual notation contrasts with *ppDATE*s, in which, the semantics entail event not triggering any outgoing transition may occur, and leave the *ppDATE* in the same state.

[8] By quasi-synchronous composition, we mean the restriction of a *ppDATE* with an automaton, such that a *ppDATE* transition triggered by event e synchronises with a transition labelled e on the automaton, no matter what the condition and action are. Furthermore, the synchronisation is unidirectional, in that we limit the behaviour of the *ppDATE*, obtaining a *ppDATE* which is necessarily smaller, rather than the Cartesian product of the states of the *ppDATE* and the automaton.

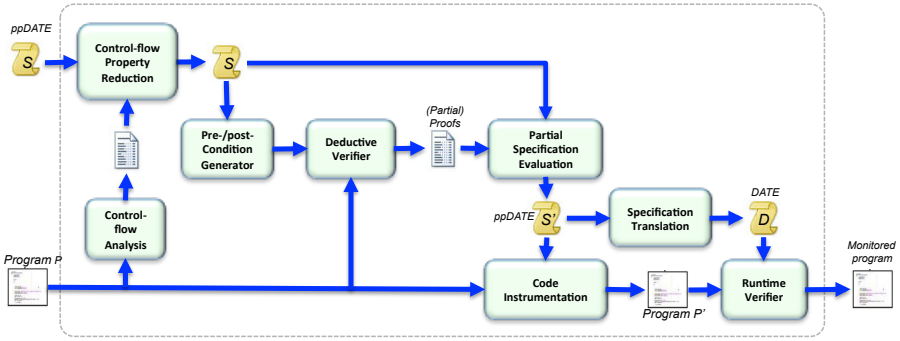

Fig. 4. High-level description of the StaRVOOrS framework workflow enriched with control-flow analysis

This approach is closely related to the optimisations used in Clara [7, 6], and we could introduce control-flow optimisation before the data-based static analysis is applied, as depicted in Fig. 4.

4 Episode II, Trailer 'Distributed StaRVOOrS'

The days of stand-alone software applications are largely over. Cloud solutions and mobile applications are perhaps the most prominent instances of a development towards ever more distributed computing. But this trend is equally dominant in areas less visible to end users. For instance, instead of singular embedded systems interacting largely with their physical environment, modern vehicles carry internal networks of interacting programmed units. Distributed software is ubiquitous. The overwhelming combinatorial complexity of possible interactions and interleavings makes distributed software systems particularly prone to unforeseen, unintended behaviour of multiple criticality. This makes system analysis and verification efforts even more important than in the stand-alone case. At the same time, distributed computational scenarios pose enormous challenges to static analysis and verification. There exist many approaches in the literature, partly supported by tools. But in general, sufficiently powerful methods tend to be heavy from a developer's perspective. We believe that the key to significantly advancing the state-of-the-art lies in a carefully designed interplay of static and runtime techniques both on the local and the global level of the distributed system. On either level, properties which are a bottleneck for static verification shall be addressed by runtime verification. On the other hand, properties which require too much overhead for runtime checking shall be addressed by static verification. This way, we can increase both the scope and the feasibility of verification in the realm of distributed systems. To achieve this, we will exploit the potential of *compositional assume-guarantee* (AG) reasoning [23, 18, 26], so far only used in the realm of static verification, in the context of combined static and runtime verification.

4.1 Static Verification of Distributed Software

The two main schools of static software verification are model checking and deductive verification. Of those, model checking has been extensively applied to distributed scenarios. We refrain from giving an overview here, but mention the SPIN model checker [15] as an archetypal tool for model checking (asynchronous) distributed scenarios. However, our next steps will not necessarily be based on model checking on the static side. One of the reasons is that model checking is used to verify *abstractions* of concrete systems, whereas runtime verification verifies runs of concrete systems. In addition, we aim at also verifying *data-oriented, functional properties* of distributed systems. For those, deductive methods are better suited.

Concerning deductive methods for distributed systems, we have process calculi and contract based methods. Process calculi are still rather abstract for the targeted combination with runtime analysis, and mostly lack integration to real world paradigms (like object-orientation). Highly relevant, however, for our project are *contract* based deductive methods for distributed systems, in particular the *compositional 'assume-guarantee'* (AG) approach to verification of distributed systems, first introduced by Misra and Chandy [23]. Compositionality means that the implementation of each component in the distributed system can be verified independent of the implementation of other components, against local contracts which state 'assumptions' on the environment and 'guarantees' of the component itself. This technique builds on principles of Hoare logic, and thereby can be instantiated for many concrete programming language of interest. The difference is that the contracts do not (only) talk about pre/post-states of some code, but also about the in- and outgoing communication *during* the execution of a component's implementation. Verifying each component's local compliance with its own contract, while assuming the other component's contracts (but not their implementation), proves correctness of the entire system.

Given a system which is composed by components communicating via (some form of) message passing, the implementation of each component can be specified by, and verified against, a local contract which states: a) *assumptions* about the messages and data sent from the environment, and b) *guarantees* about messages and data sent to the environment. Some variants of AG, including the work in [2], do not distinguish between assumption resp. guarantee formulas, but represent both in one invariant over the communication history. Intuitively, a component has to guarantee that outgoing messages maintain the invariant, given that incoming messages do so. In the case of object-oriented distributed systems, messages are method calls (with parameters) and method returns (with return values). *Assumptions* talk about incoming messages, i.e., method calls from callers of `this` object, and method returns from callees of `this` object. Similarly, *guarantees* talk about outgoing messages, i.e., method calls to callees of `this` object, and returns to callers of `this` object. This is true for both synchronous and asynchronous method execution.

When this principle is applied to modern software artefacts, it has to also cope with information hiding, by refining conditions on the communication to

conditions on the internal (object) state. For instance, a positive account balance can be expressed externally in terms of summing up parameters of deposit resp. withdrawal messages, without reference to the internal state. An internal invariant can then refine the status of the event history to the internal state representation. For a comprehensive account on assume-guarantee style reasoning, see [11].

Among the recent contribution to integrating assume-guarantee style (static) verification of distributed software into contemporary verification technology are extensions [2, 13] of the KeY verifier to the asynchronous distributed languages Creol [17] and ABS [16].

4.2 Runtime Verification of Distributed Software

Concerning runtime verification of distributed systems, some of the issues discussed in the literature are: (i) characteristics of properties and systems such that the former are *monitorable* on the latter [22]; (ii) dedicated formalisms tailored for distributed runtime monitoring, [27, 28]; (iii) the choice of location of the runtime monitors [14].

Concerning formalisms for writing properties about distributed systems, a reference is *past-time Distributed Temporal Logic* (ptDTL) introduced by Sen et al. [28], and the more recent logic DTL [27]. DTL combines the three-valued linear temporal logic (LTL_3 [4]) with ptDTL, and is able to express more properties than ptDTL, like Boolean combinations of safety properties.

The choice of locations of the monitors is quite an important issue because communication across locations is usually expensive and information-sensitive. A good discussion about this choice is presented in [14], where a theoretical framework is presented for comparing those choices. Studying this aspect is not an exclusivity from the runtime verification community; it has been studied in other communities before, as for instance in security. The papers [20, 25] provide a clear survey of those techniques for usage control.

From the practical side, a taxonomy of software-fault runtime verification tools is presented in [12], including some targeting distributed and parallel systems. Among those, it is worth mentioning the Java Runtime Timing-constraint Monitor (JRTM) [24]. JRTM monitors timing properties (written in Real Time Logic —RTL) of distributed, real-time systems written in Java. Zhou et al. [31] presents DMaC, a distributed monitoring and checking platform built upon: (i) the Monitoring and Checking (MaC) framework (providing means to monitor and check running systems against formal requirements), and (ii) a declarative domain-specific approach for specifying and implementing distributed network protocols. DMaC uses a formal specification language called MEDL, similar to past-time LTL, in which it is possible to specify safety properties of a distributed system.

4.3 Combined Static and Runtime Verification of Distributed Software

Our work on combining static and runtime verification of distributed software will be based on the following existing approaches, methods, and tools:

- The assume-guarantee paradigm for (static) distributed systems verification in general [23, 26, 11], and for (static) distributed *objects* verification in particular [2].
- Approaches to the scope and placement of runtime monitors in a distributed system [14].
- The results of our StaRVOOrS (Episode I) project for combined static and runtime verification of *sequential* object-oriented programs [1, 8]. In particular, we will extend to the distributed case:
  - The general principle of using complete *and incomplete* static proofs, analysing the latter to refine the original specs by path conditions which prevent runtime verification of statically verified cases [3];
  - The language *ppDATE*, combining automata-style control-flow oriented specification with data-oriented specification in form of (state-dependent) Hoare triples [1];
  - Experience gained in implementing and using the StaRVOOrS tool [8].

We are convinced that compositional assume-guarantee (AG) specification and reasoning, so far only used in the realm of static verification, has enormous potential in the context of combined static and runtime verification. We will exploit this potential in a number of ways. AG was conceived and used solely as a means for *static* verification. One bottleneck of AG is that the reduction of properties of the outer communication to properties of the inner state can require smart proof engineering. In our future work, however, we will refer sub-properties which are difficult to establish statically to runtime verification. Another, very severe bottleneck for practical applicability of AG is that it requires full access to the implementation of all components. Even if the implementation of individual components can be verified without knowledge of the other components' implementation (after all, the method is compositional by design), still the implementation of all components must be verified to establish the correctness of the overall system. But in real distributed scenarios, we often only know the internals of certain components, not of others. (Those may be legacy systems, binaries, or remote proprietary services.) We can, however, formalise the documented external behaviour of such closed components with AG contracts. Actual compliance of closed components with such contracts can then be checked by *runtime verification*. At the same time, these contracts can be *used*, as assumptions, in the verification of open components interacting with the closed ones. The latter can be done statically, or at runtime, or with a combination.

5 Conclusions

In this paper we have reported on our previous results concerning StaRVOOrS, a framework for the combination of static and runtime techniques for the verification

of data- and control-oriented properties. We have also identified two main research directions: i) optimisation of our framework by using static analysis techniques to reduce runtime overheads, and ii) extending StaRVOOrS to a distributed setting. We briefly present here a roadmap for achieving this endeavour.

Optimisation using control-flow static analysis. As described in Section 3, the runtime monitor may be further optimised by considering additional constraints of the program being analysed. In particular, we will use standard techniques to get an automata based on the control-flow of the program and apply quasi-synchronisation to compose it with the *ppDATE*. We will explore the connection, and eventual combination, with techniques like the one used in Clara [6].

Control- and data-oriented property language for distributed components. Any formalism for stating assumptions/guarantees/invariants has to be capable of expressing conditions on the history of communication events, including the carried data. The formalisms typically used are either of too limited expressiveness or too difficult to use for formalisation and reasoning. We will extend and adapt the control- and data-oriented property language *ppDATE* [1] to the distributed setting. The native support for properties of data *and* events will be even more profitable in the distributed setting than it already is in the sequential setting, because typical AG contracts require characterisation of *event* histories together with the carried *data*.

Identify and adapt static verification methods and tools. Neither the method nor the tool will be developed from scratch (one starting point can be [2]), but serious adaptions need to be made.

Identify and adapt a runtime verification method and tool. Neither the method nor the tool will be developed from scratch. The prime candidate is LARVA [10] (which employs aspect-oriented programming), but extended to the distributed setting. Among the issues will be strategies for placing (or even moving) runtime monitors within the distributed system, see [14].

Integrating static and runtime verification of distributed components. Develop a methodology and corresponding tool support which identifies sub-properties where static verification will be tried, analyses the result, and deploys the system for runtime monitoring of sub-properties which are not statically verified.

Tune the balance of static vs. runtime verification of distributed behaviour. The 'effort level' for static verification can be guided by the mixed criticality levels of components and their services in the distributed system. And it can be guided by limits in time, budget, and education in the software ecosystem using our method. Note that, in particular, we will support the effort level 'full automation', resulting in many unfinished proofs. Still, our current results show that even that can limit the runtime overhead by a factor of up to 800 [8] (through automated analysis of unfinished proofs).

Investigate synchronous vs. asynchronous communication. Crosscutting the above concerns, we aim to investigate both synchronous and asynchronous communication. The choice has implications for all of the above. In terms of target languages/architectures, we will use Java-RMI (remote method invocation) for the synchronous case, and ABS [16] (an extension of Creol) or Active Objects [19] for the asynchronous case.

Case Studies. Will will also have running case studies, to experiment with, and evaluate. When more machinery is in place, we will use a bigger, realistic scenario to evaluate the overall approach. A possible candidate is from the automotive domain in connection with a big car manufacturer.

Acknowledgements

This research has been partially supported by the Swedish Research Council *(Vetenskapsrådet)* under the project *StaRVOOrS: Unified Static and Runtime Verification of Object-Oriented Software*, no. 2012-4499. We would like to thank Jesús Mauricio Chimento, for his substantial contributions to the the work we recapitulate in Sect. 2 (StaRVOOrS — Episode I), in particular the StaRVOOrS tool and the experiments.

References

1. W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109 of *LNCS*, pages 108–125. Springer, 2015.
2. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010.
3. W. Ahrendt, G. Pace, and G. Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISoLA'12*, LNCS 7609. Springer, 2012.
4. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. E. Bodden and P. Lam. Clara: Partially evaluating runtime monitors at compile time - tutorial supplement. In *RV'10*, volume 6418 of *LNCS*, pages 74–88, 2010.
7. 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.
8. J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java. In E. Bartocci and R. Majumdar, editors, *Runtime Verification*, volume 9333 of *Lecture Notes in Computer Science*, pages 297–305. Springer International Publishing, 2015.
9. 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.

10. 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.
11. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, Nov. 2001.
12. N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Software Eng.*, 30(12):859–872, 2004.
13. C. C. Din, S. L. Tapia Tarifa, R. Hähnle, and E. B. Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In M. Butler, S. Conchon, and F. Zaïdi, editors, *Proc. 17th International Conference on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2015.
14. A. Francalanza, A. Gauci, and G. J. Pace. Distributed system contract monitoring. *The Journal of Logic and Algebraic Programming*, 82(57):186 – 215, 2013. Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11).
15. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
16. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects, FMCO, Graz, Austria. Revised Papers*, LNCS. Springer, 2010.
17. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
18. C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, UK, 1981.
19. R. G. Lavender and D. C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
20. A. Lazouski, F. Martinelli, and P. Mori. Usage control in computer security: A survey. *Computer Science Review*, 4(2):81–99, 2010.
21. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual. Draft 2344*, 2013.
22. S. Malakuti Khah Olun Abadi, M. Akşit, and C. M. Bockisch. Runtime verification in distributed computing. *Journal of Convergence*, 2(1):1–10, June 2011.
23. J. Misra and K. Chandy. Proofs of networks and processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.
24. A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *RTAS'97*, pages 252–262. IEEE Computer Society, 1997.
25. Å. A. Nyre. Usage control enforcement - A survey. In *ARES'11*, volume 6908 of *LNCS*, pages 38–49. Springer, 2011.
26. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*. Springer, 1985.
27. T. Scheffel and M. Schmitz. Three-valued asynchronous distributed runtime verification. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 52–61, Oct 2014.
28. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 418–427, 2004.

29. I. Tonin. Verifying the Mondex case study. The KeY approach. *Technical Report 2007-4, Universität Karlsruhe*, 2007.
30. J. Woodcock. First Steps in the Verified Software Grand Challenge. In *SEW'06*, pages 203–206. IEEE Computer Society, 2006.
31. W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *DMaC*: Distributed monitoring and checking. In *RV'09*, volume 5779 of *LNCS*, pages 184–201. Springer, 2009.