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

Jesús Mauricio Chimento[1], Wolfgang Ahrendt[1], Gordon Pace[2] and Gerardo Schneider[3]

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

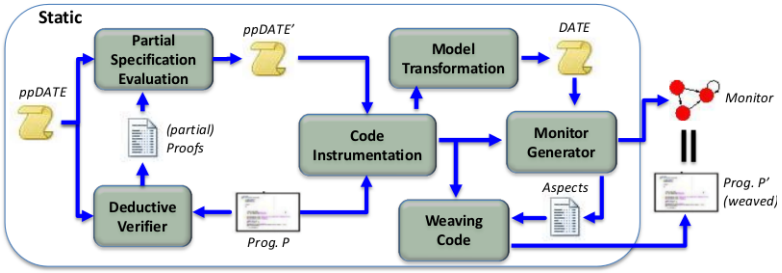
1 Introduction

Runtime verification techniques are concerned with the monitoring of software, providing guarantees that observed runs comply with specified properties. These methods are strong in analysing systems of a complexity that is difficult to address by *static verification*, like systems with numerous interacting sub-units, heavy usage of mainstream libraries, real (as opposed to abstract) data, and real world deployments. On the other hand, the major drawbacks of runtime verification are the impossibility to extrapolate correct observations to all possible executions, and that monitoring introduces runtime overheads. In the work we present here, the authors address these downsides by combining runtime verification with static verification, such that: (i) Static verification attempts to 'resolve' those parts of the properties which can be confirmed statically with low overhead (like, e.g. fully automatically); (ii) the static results, even if only partial (see below), are used to change the property specification such that generated monitors will not check dynamically what was confirmed statically.

In addition to combining static and runtime verification, we propose a specification language, and an according verification methodology, that captures both *control-oriented* properties (like the *DATE* language used in the runtime verification tool Larva [4]) and *data-oriented* properties (like the *Java Modelling Language* JML [3, 5]).

2 The StaRVOOrS Framework

Below we show an abstract view of the framework, which was initially sketched in [1].



Given a Java program P and a specification π of the properties to be verified (given in the language *ppDATE*, see Sec. 3), these are transformed into suitable input for the *Deductive Verifier* module which, in principle, might statically fully verify the properties related to pre/post-conditions. What is not proved statically will then be left to be proven at runtime. Here, not only the completed but also the *partial* proofs will be used by the *Partial Specification Evaluator* module in order to rewrite the original π into π' which triggers runtime checks for the parts which were not possible to prove statically. To achieve this, the original pre-conditions from π are refined to express also path conditions for not statically verified executions.

In order to use *aspects*, the program P is subject to *Code Instrumentation* and π' is modified according to this instrumentation. Next, the modified π' specification is transformed into a specification suitable for the runtime verifier by *Model Transformation*. The instrumented program, together with the previously obtained specification, is given to the *Monitor Generator*, which uses aspect-oriented programming techniques to capture relevant system events. Later on, the generated aspects are weaved (*Weaving Code*) into the instrumented version of P. The final step in the work-flow 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.

To realise the StaRVOOrS framework, we combine, initially, the use of the deductive source code verifier KeY [2] and the runtime monitoring tool Larva [4]. KeY is a deductive verification system for data-centric *functional correctness* properties of Java source code, which generates proof obligations in *dynamic logic* (DL), a modal logic for reasoning about programs. Larva[1] is an automata-based runtime verification tool for Java programs which automatically generates a runtime monitor from a property written in the formal language *DATE*. Larva transforms the set of properties into monitoring code together with AspectJ code to link the system with the monitors.

3 A Specification Language for Static and Runtime Verification of Data and Control Properties

Due to the fact that we want to specify both control-oriented and data-oriented properties in a single language, we are in the need of formalizing a language which supports both.

For data-oriented property specification, KeY supports the *Java Modelling Language* (JML) [5]. JML allows for the specification of pre- and post-conditions of method calls, and class invariants.

The control-oriented specification language of Larva is called *Dynamic Automata with Timers and Events* (DATEs). DATEs are essentially an extension of timed automata. At their simplest level, they are finite state automata, whose transitions are triggered by system events (primarily entry points $f^{\downarrow}$ and exit points $f^{\uparrow}$ of methods) and timers, but augmented with: (i) A symbolic state which may be used as conditions to guard transitions and can be modified via actions also specified on the transition; (ii) Replication of automata, through which a new automaton is created for each discovered instance of an object; (iii) Communication between automata using standard CCS-like channels with $c!$ acting as a broadcast on channel c and which can be read by another automaton matching on event $c?$.

Hence, we want that our specification language allow us to generate, from a specification written on it, suitable artifacts for both JML and *DATE*. With this idea in mind we have

[1] *Logical Automata for Runtime Verification and Analysis.*

formalized the language *ppDATE*, which is essentially an extension of *DATE*s with Hoare-triples on states. Therefore, in addition to transitions of the form $q \xrightarrow{event|condition \mapsto action}_M q'$, the *ppDATE* automaton has a mapping τ associating sets of Hoare-triples $\{pre\}\,\texttt{event}\,\{post\}$ to states. Intuitively, while in a state q with $\{pre\}\,\texttt{event}\,\{post\} \in \tau(q)$, any invocation of `event` satisfying precondition *pre* on entry, should satisfy postcondition *post* upon exit.

4 Ongoing and Future Work

We have formalised the language for combining (partial) static and (optimised) runtime verification, which we called *ppDATE*, and we have applied our framework to two small real-life scenarios: a *login management system*, and an implementation of *binary semaphores*. In both cases, we analyse properties which should be guaranteed by the respective implementations, and we demonstrate how these properties can be partly statically, partly dynamically verified using our framework.

We will continue this work as follows: (i) Prove soundness of *ppDATE*s transformation. (ii) Implement *ppDATE* as a script language. (iii) Automate the usage of KeY, the analysis of its (partial) proofs, and the integration of the result in optimised *ppDATE*s. Towards this direction, we have so far implemented a facility which given a Java program annotated with JML specifications, returns an .xml file that shows which specifications hold, which specifications do not hold and, in case of a branching, which branches were closed, which branches remain as open goals and which was the condition that produced the branching. (iv) Integrate KeY and Larva. (v) Address the automation of the 'operationalisation' of pre/post-conditions containing algorithmic content. (vi) Apply our framework to case studies.

References

[1] Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. A unified approach for static and runtime verification: Framework and applications. In *ISOLA '12*, LNCS 7609. 2012.

[2] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[3] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *SERP'02*, pages 322–328. CSREA Press, 2002.

[4] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva - a tool for runtime monitoring of java programs. In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.

[5] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual. Draft 1.200*, 2007.