StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java*

Jesús Mauricio Chimento[1], Wolfgang Ahrendt[1], Gordon J. 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

Abstract. We present the tool StaRVOOrS (Static and Runtime Verification of Object-Oriented Software), which combines static and runtime verification (RV) of Java programs. The tool automates a framework which uses partial results extracted from static verification to optimise the runtime monitoring process. StaRVOOrs combines the deductive theorem prover KeY and the RV tool LARVA, and uses properties written using the ppDATE specification language which combines the control-flow property language DATE used in LARVA with Hoare triples assigned to states. We demonstrate the effectiveness of the tool by applying it to the electronic purse application Mondex.

1 Introduction

In this paper we present StaRVOOrS, a tool for the specification and verification of data- and control-oriented properties combining static and runtime verification techniques. A detailed motivation for the combination along these two dimensions (data- vs. control-oriented, and static vs. dynamic verification) has been reported in [3, 4] and will not be repeated here. For this paper, we only emphasise that this combination allows us to get a richer specification language able to express both data- and control-oriented properties, proving some properties once and for all statically, letting others to be checked at runtime.

The tool is a fully automated implementation of the theoretical results presented in [3, 4]. Given a property specification and the original program, our tool chain produces a statically optimised monitor and the weaved program to be monitored. This includes the automated triggering of numerous verification attempts of the underlying static verification tool, the analyses of resulting partial proofs, and the monitor generation.[4]

* Supported by the Swedish Research Council under the *StaRVOOrS* project (*Unified Static and Runtime Verification of Object-Oriented Software*), no. 2012-4499.

[4] The implementation of StaRVOOrS, its user manual, and a video showing how to use StaRVOOrS, are available from [2].

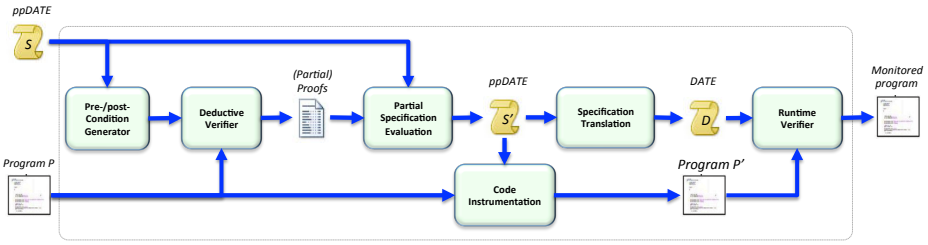

Fig. 1. High-level description of the StaRVOOrS framework workflow

2 The StaRVOOrS Framework

The StaRVOOrS framework (STAtic and Runtime Verification of Object-ORiented Software) was originally proposed in [4] and its theoretical foundations further developed in [3]. Object oriented software provides an abstract manner in which we replicate properties for every instance of a class, but many of the features in the framework that we discuss in this paper are not specific to object oriented software. The workflow of StaRVOOS is shown in Fig. 1, and is explained in detail in [3]. Here we give a brief overview of the deductive verifier KeY [5], the runtime monitoring tool Larva [6], and the specification language *ppDATE*.

The static verifier KeY. KeY is a deductive verification system for data-centric *functional correctness* properties of Java programs [5]. It features (static) verification of Java source code annotated with specifications written in the *Java Modelling Language* (JML) [7]. JML allows for the specification of pre/post-conditions of methods, and loop invariants. KeY translates the different parts of the specification to proof obligations in Java *dynamic logic* (DL). At the core of KeY is a theorem prover for Java DL, a modal logic for reasoning about programs. KeY uses a *sequent calculus* following the *symbolic execution* paradigm.

The runtime verifier Larva. Larva (*Logical Automata for Runtime Verification and Analysis*) [6] is an automata-based runtime verification tool for Java programs. Larva automatically generates a runtime monitor from a property written in a formal language, which in the case of Larva is an extension of timed automata called *DATE*s (*Dynamic Automata with Timers and Events*). At their simplest level *DATE*s are finite state automata whose transitions are triggered by system events and timers. Further details and the formalisation of *DATE*s can be found in [6]. Given a system to be monitored (a Java program) and a set of properties written in terms of *DATE*s, Larva generates monitoring code together with AspectJ code to link the system with the monitors.

***ppDATE*: A Specification Language for Data- and Control-Oriented Properties.** StaRVOOrS uses *ppDATE*s as its property input language, which enables the combination of data- and control-based properties in a single formalism. ppDates are a composition of the control-flow language *DATE*, and of data-oriented specifications in the form of Hoare triples with *p*re-/*p*ost-conditions.

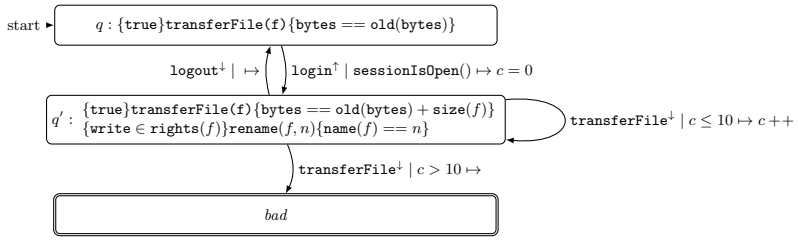

Fig. 2. A *ppDATE* limiting file transfers

Consider the *ppDATE* shown in Fig. 2. The structure of the automaton, less the information given in the states, provides the control-flow aspect of the property in the form of a *DATE*, in which transitions are tagged with triples: $e \mid c \mapsto a$ — indicating that (i) they are triggered when event e occurs and condition c holds; (ii) and apart from changing the state of the property, action a is executed. For instance, the reflexive transition on the middle state is tagged: $\texttt{transferFile}^{\downarrow} \mid c \leq 10 \mapsto c{+}{+}$, means that if the automaton is in the middle state when the system enters the function named `transferFile` and counter variable c does not exceed 10, then the counter is incremented by 1. Some states are identified as *bad states*, denoted using a double-outline in the figure, and used to indicate that if and when reached, the system has violated the property in question. The property represented in Fig. 2 can thus be understood to ensure that no more than 10 file transfers take place in a single login session.

In *ppDATE*s the data-oriented features of the specification appear in the states. A state may have a number of Hoare triples assigned to it. Intuitively, if Hoare triple $\{\pi\}\texttt{f}\{\pi'\}$ appears in state q, the property ensures that: if the system enters code block `f` while the monitor lies in state q and precondition π holds, upon reaching the corresponding exit from `f`, postcondition π' should hold. Pre-/post-conditions in Hoare triples are expressed using JML boolean expression syntax [7], which is designed to be easily usable by Java programmers. For instance, the Hoare triple appearing in the top state of the property given in Fig. 2, ensures that any attempted file transfer when in the top state (when logged out), should not change the byte-transfer count. Similarly, while logged in (in the middle state of the property) (i) the number of bytes transferred increases when a file transfer is done while logged in; and (ii) renaming a file does indeed change the filename as expected if the user has the sufficient rights.

To ensure efficient execution of monitors, *ppDATE*s are assumed to be deterministic by giving an ordering in which transitions are executed.

3 The StaRVOOrS Tool Implementation

StaRVOOrS takes three arguments: (i) The Java files to be verified (the path to the main folder), (ii) A description of the *ppDATE* as a script (a file with

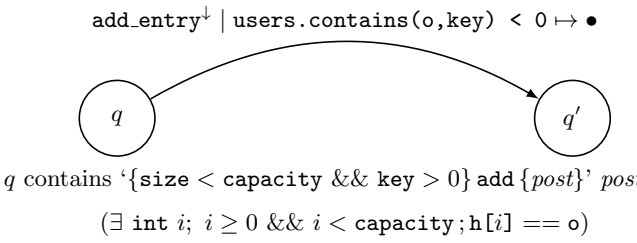

Fig. 3. *ppDATE* specification for adding a user.

extension `.ppd`), and (iii) The path of the output folder. The output of the tool is the runtime monitor (this file is placed in the output folder together with an instrumented version of the Java files).

To describe our implementation, we use as working example a *login scenario*, where users attempt to login into a system. The set of logged *users* is implemented as a `HashTable` object, whose class represents an open addressing hash table with linear probing as collision resolution. The method `add`, which is used to add objects into the hash table, first tries to put the corresponding object at the position of the computed hash code. However, if that index is occupied then `add` searches upwards (modulo the array length) for the nearest following index which is free. Within the hash table object, users are stored into a fixed array `h`, meaning that the set has a capacity limited by the length of `h`. In order to have an easy way of checking whether or not the capacity of `h` is reached, a field `size` keeps track of the number of stored objects and a field `capacity` represent the total amount of objects that can be added into the hash table.

In a nutshell, the tool works following these steps: (1) A property is written using our script language for *ppDATE*s; (2) Hoare triples are extracted from the specification of the property, are translated into JML contracts to be added to the Java files; (3) KeY attempts to verify all JML contracts, generating (partial) proofs, the analysis of which results in an XML file, (4) The *ppDATE* is refined based on the XML file; (5) Declarative pre/post-conditions are operationalised; (6) The code is instrumented with auxiliary information for the runtime verifier; (7) The *ppDATE* specification is encoded into *DATE*s; (8) The LARVA compiler generates a runtime monitor. We will now describe some of the above steps in more detail by describing them using our running example.

3.1 *ppDATE* Property: Adding a User

For simplicity we do not present the full specification for the login example but rather focus on the operation of adding a user to the hash table. Fig. 3 depicts the *ppDATE* specification. The property is written as the following script.

```
EVENTS {
 add_entry(Object o,int key) = {HashTable users.add(o, key)}
}
```

```
PROPERTY add {
 STATES {  NORMAL{q2;}  STARTING{q1 (add_ok);}  }
 TRANSITIONS { q1 -> q2 [add_entry\users.contains(o, key) < 0] }
}
CINVARIANTS {
 HashTable {h.length == capacity}
 HashTable {h != null}
 HashTable {size >= 0 && size <= capacity}
 HashTable {capacity >= 1}
}
CONTRACTS {
 CONTRACT add_ok {
   PRE {size < capacity && key > 0 }
   METHOD {HashTable.add}
   POST {(\exists int i; i>= 0 && i < capacity; h[i] == o)}
   ASSIGNABLE {size, h[*]}}
}
```

Invariants (section `CINVARIANTS`) are described by `class_name {invariant}`. Section `CONTRACTS` lists named Hoare triples (`CONTRACT`). The predicate in the post-condition follows JML-like syntax and pragmatics. The second semicolon is semantically an 'and', but conveys a certain pragmatics. It separates the 'range predicate' (`i >= 0 && i < capacity`) from the desired property of integers in that 'range', (`h[i] = o`). The constraint `add_ok` specifies that, if there is room for an object `o` in the hash table and the received key is positive, then after adding that object into the hash table it is found in one of the entries of the array `h`. Finally, the `PROPERTY` section represents the entire automata, which in this tiny example has only two states, `q1` and `q2`, the second being initial (`STARTING`). The syntax `q1 (add_ok)` assigns the Hoare triple `add_ok` to `q1`.

3.2 Proof Construction and Partial Proof Analysis

The first step in our work-flow is to annotate the Java sources with JML contracts extracted from the Hoare triples specified in the *ppDATE*. We automatically generate such JML annotations and insert them just before the corresponding method declaration. Once the JML annotations are in place, the tool performs static verification, checking whether, or to which extent, the various JML contracts (each corresponding to a Hoare triple in *ppDATE*) can be statically verified. KeY is used to generate proof obligations in Java DL for each contract, and attempts to prove them automatically. Although we could have allowed for user interaction (using KeY's elaborate support for interactive theorem proving), we chose to use KeY in auto-mode, as StaRVOOrS targets users untrained in theorem proving.

For each Hoare triple KeY's verification attempt will result in either a full proof, where all goals are closed, or a partial proof, where some goals are open while others are closed. Partial proofs are analysed by our tool, and results are collected in an XML file. Most importantly, this file contains, for each Hoare triple specifying a method, say m, additional assumptions on the state in which m

is called, telling whether or not this Hoare triple needs to be checked at runtime for executions of m.

3.3 *ppDATE* Transformation: Hoare Triple Refinement

Our tool uses the output of our previous step for refining, in the *ppDATE*, all Hoare triples based on what was proved/unproved. Hoare triples whose JML translation was fully verified by KeY are deleted entirely. On the other hand, each Hoare triple not fully proved by KeY is refined. The new precondition is a conjunction (`&&`) of the old precondition and a disjunction of new preconditions corresponding to open proof branches.

In our example, the precondition of `add_ok` will be strengthened with the condition for the one goal not closed by KeY, `!(h[hash_function(key)] == null)`. The Hoare triple will thus be refined as follows:

```
CONTRACT add_ok {
  PRE {size < capacity && key > 0
       && !(h[hash_function(key)] == null)}
  METHOD {HashTable.add}
  POST {(\exists int i; i>= 0 && i < capacity; h[i] == o)}
  ASSIGNABLE {size, h[*]}  }
```

Once all Hoare triples in the original *ppDATE* are refined this way, reflecting the results from static verification, the tool will translate the resulting *ppDATE* into the pure *DATE* formalism, to be processed by Larva further on.

3.4 Translation to *DATE* and Monitor Generation with Larva

Once the refinement is performed, the tool syntactically analyses the specification for declarative assertions in pre/post-conditions which may need to be *operationalised* i.e. transformed into algorithmic procedures. This includes, for instance, transforming existential and universal quantification into loops. The next step in the work-flow is to instrument the source code by adding identifiers to each method definition and additional code to get fresh identifiers. These identifiers will be used to distinguish between different calls to the method.

After these modifications, the statically refined (see section 3.3) *ppDATE* specification is translated into the pure *DATE* formalism, enabling monitor generation by Larva. The control part of the *ppDATE* is already in automaton form, and can be interpreted directly as a *DATE*, but we still have to encode the Hoare triples into *DATE*. We refer to [3] for details of this translation.

The final step is the generation of the monitor by the Larva compiler, taking as input the *DATE* obtained in the previous step. The compiler not only generates the monitor but also generates aspects, and weaves the code with the Java programs subject to verification. See [6] for further explanation on Larva.

4 Case study: Mondex

Mondex is an electronic purse application for smart cards products [1]. We consider a variant of the original presentation, strongly inspired by the JML formalisation given in [8]. One of the main differences with respect to the the original presentation is that we consider a Java implementation working on a desktop instead of the Java Card one for smart cards. The full specification and code of this case study can be found from [2].

Mondex essentially provides a financial transaction system supporting transferring of funds between accounts, or 'purses'. Whenever a transaction between two purses is to take place, (i) the source and destination purses should (independently) register with the central fund transferring manager; (ii) then a request to deduct funds from the source purse may arrive, followed by (iii) a request to add the funds to the destination purse; and (iv) finally, there should be an acknowledgement that the transfer took place, before the transaction ends.

Besides specifying the protocol, one has to specify the behaviour of the involved methods, which obviously changes together with the status of the protocol. For instance, transfer of funds from a purse to another should succeed once both purses have been registered, but should fail if attempted before registration or if an attempt is made to perform the transfer multiple times. This behaviour is encoded by different Hoare triples assigned to different S states.

The control-oriented properties ensure that the message exchange goes as expected. In contrast, the pre/post-conditions (in total, there are 26 Hoare triples in the states of the *ppDATE*) ensure the well-behaviour of the individual steps.

We feed StaRVOOrS with the above *ppDATE* and the source code of Mondex. Our tool automatically produces a runtime monitor which is then run in parallel with the application. Initially, the *ppDATE* automaton consisted of only one automaton with 10 states and 25 transitions. Except for two Hoare triples related to the initialisation and termination of a transaction which were fully proven by KeY, all the other 24 triples are only partially verified by KeY. The automated analysis of these proofs leads to a refined *ppDATE* as explained in section 3.3. Besides, it is necessary to deal with the operationalisation of the JML operator `\old`. This is done by adding a fresh variable at the automaton level, saving the value of the variable annotated with `\old` before the method (associated to its Hoare triple) is executed. Then, when analysing the postcondition, if the value of the variable has changed, it can be compared with its previous value store in the automaton level variable. The obtained *DATE* (following the procedure explained in section 3.4) consists on 25 automata, one automaton to control the main property and 24 replicated automata to control postconditions, with 106 states and 196 transitions in total. Also, due to the operationalisation of `\old`, it were added four new variables at automata level in the main automaton.

The whole process to generate the monitor for Mondex took our tool 2 minutes 30seconds on PC Pentium Core i7, where most time is used in KeYs static analysis of the Hoare triples (2 minutes 15 seconds). Our original implementation of Mondex weighted 23.5 kB. After, running the tool, the total weight of all the new generated files related to the implementation of the monitor is 177.8 kB.

We have compared the execution times of: (a) the unmonitored implementation, (b) the monitored implementation using the original specification S and translating it unoptimised into a DATE, and (c) the monitored implementation using the specification S', obtained from S via application of StaRVOOrS. The concrete performance numbers of this experiment are the same as the ones reported in [3, sect. 5]. To summarise the results, the addition of a monitor (case (b)) causes an overhead on the execution time w.r.t. the unmonitored version (a), between 15 and 1000 times. However, this overhead is dramatically reduced by using our approach (case (c)), only doubling the execution time (again w.r.t. (a)). The saving comes from only triggering post-condition checks in states satisfying pre-conditions from open branches in KeY proofs.

5 Conclusions

A key feature of our work is that everything is done fully automatic: StaRVOOrS is a push-button technology taking as input a specification and a Java program and given as output a partially verified program running in parallel with a runtime monitor. Our current experiments are encouraging as we drastically improve the time complexity of the runtime verifier Larva. Both the efficiency gain for monitoring and the confidence gain can only increase along with future improvements in the static verifier used. For instance, if ongoing work on loop invariant generation in KeY leads to closing some more branches in typical proofs, this will have an immediate effect that is proportional to the frequency of executing those loop at runtime. For related work on the combination of static verification and static verification, we refer the reader to [3].

Acknowledgements. We would like to thank C. Colombo and M. Henschel for their support concerning implementation issues about Larva and KeY respectively.

References

1. MasterCard International Inc. Mondex. `www.mondexusa.com/`.
2. StaRVOOrS. `www.cse.chalmers.se/~chimento/starvoors`.
3. W. Ahrendt, J. Chimento, G. 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.
4. 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.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. 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.
7. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft 1.200*, 2007.
8. I. Tonin. Verifying the Mondex case study. The KeY approach. *Technical Report 2007-4, Universität Karlsruhe*, 2007.