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. In this paper we present the tool StaRVOOrS (Static and Runtime Verification of Object-Oriented Software), which combines static and runtime verification 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 runtime verification tool LARVA, and uses properties written using the ppDATE specification language which combines the control-flow property language DATE used in the runtime verification tool LARVA with Hoare triples assigned to states. The formalism enables KeY to attempt verification of the Hoare triples in the specification in fully automated mode, which often results in unfinished proofs. Through an analysis of such partial proofs, path conditions for the (statically) inconclusive executions are generated, and then used to refine the Hoare triples for runtime verification. Finally, the refined Hoare triples are translated into the DATE formalism, coding them as a combination of replicated automata and operationalised pre/post-conditions and LARVA is used to instrument the resulting DATE to monitor the code. StarRVOOrS effectiveness is demonstrated by applying it to the verification scenario Mondex, an electronic purse application.

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 [4, 3] and will not be repeated here. For this paper, however, it suffices to emphasise that this combination allows us to get a richer specification language able to express both data- and control-oriented properties, and to prove some properties once and for all statically, letting others to be checked at runtime, with the intended good side effect of having more efficient runtime monitors.

The tool is a fully automated implementation of the theoretical results presented in [4, 3]. Given a property specification and the original program, a fully automated tool chain produces a final, 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, among other steps, all to be described in the following sections.

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]. We do not present the details of the framework here but only give a brief overview of the deductive source code verifier KeY [5], and of the runtime monitoring tool Larva [6] on which the implementation of StaRVOOrS heavily relies, and give an overview of the specification language *ppDATE*.

The static verifier KeY. KeY is a deductive verification system for data-centric *functional correctness* properties of Java source code. 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 DL. At the core of KeY is a theorem prover for Java *dynamic logic* (DL), a modal logic for reasoning about programs [5]. 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. As with many other runtime verifiers, 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.

Consider the *ppDATE* shown in Fig. 1. 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

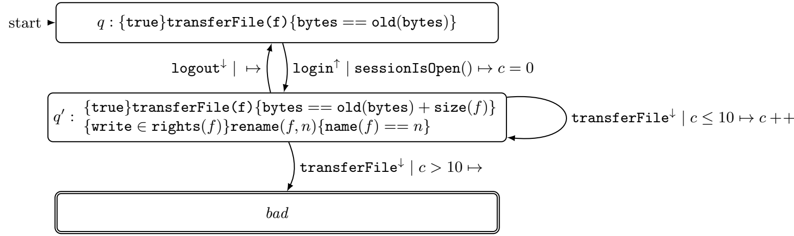

Fig. 1. A *ppDATE* limiting file transfers

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 also identified as *bad states*, denoted using a double-outline in the figure (see the bottom state), and used to indicate that if and when reached, the system has violated the property in question. The property represented in Fig. 1 can thus be understood to ensure that no more than 10 file transfers take place in a single login session.

The data-oriented features of the specification appear in *ppDATE*s 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. 1, 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. A complete formalisation of *ppDATE*s can be found in [3].

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 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).

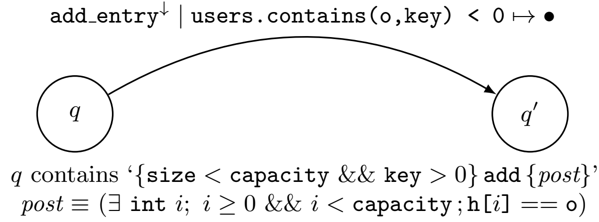

Fig. 2. *ppDATE* specification for adding a user.

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. 2 depicts the *ppDATE* specification. The property is written as the following script.

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

PROPERTY add {
```

```
  STATES {  NORMAL{q';}  STARTING{q (add_ok);}  }
  TRANSITIONS { q -> q' [add_entry\users.contains(o, key) < 0\] }
}
}
CINVARIANTS {
  HashTable {\typeof(h) == \type(Object[])}
  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, `q` and `q'`, the second being initial (`STARTING`). The syntax `q (add_ok)` assigns the Hoare triple `add_ok` to `q`.

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 analysis, 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 contract, 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 typically some others are closed. After the (partial) proofs are constructed, they are analysed by our tool, and results are collected in an XML file. Most

importantly, this file contains, for each contract, *path conditions* for each goal of the (partial) proof, together with the open/closed-status of the corresponding goal. These path conditions are additional assumptions on the *pre*-state which guide the proof to the respective goal.

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 those path conditions corresponding to open goals.

In our example, the pre-condition of `add_ok` will be strengthened with the path 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 done, the tool syntactically analyses the specification for declarative assertions in pre/post-conditions which may need to be "operationalised" — that is, 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 to the Java code base, 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 w.r.t. 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'. We focus on analysing the transactions taking place between these purses, which follow a multi-step message exchange protocol: 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. We only summarise here some key aspects of the verification process; the analysis of the verification result is given in next section.

Initially, the *ppDATE* automaton consisted on 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 2.5 minutes on PC Pentium Core i7, where most time is used in KeY's static analysis of the Hoare triples (2.15 minutes). 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 summarise below the experimental results of applying our tool to the Mondex case study. In particular, we compare execution times of (a) the unmonitored implementation, (b) the monitored implementation using the original specification S, and (c) the monitored implementation using specification S', obtained from S via application of StaRVOOrS. The table below shows the execution time, on a PC Pentium Core i7, for these three scenarios when the system is run performing different numbers of transactions.

Transactions	(a) no monitoring	(b) monitoring S	(c) monitoring S'
10	8 ms	120 ms	15 ms
100	50 ms	3500 ms	90 ms
1000	250 ms	330000 ms	375 ms

As expected, the addition of a monitor causes an overhead on the execution time (b), as compared to unmonitored execution (a). However, this overhead is substantially reduced by using our approach (c). The entire saving comes from only triggering post-condition checks in states satisfying path conditions from open goals in KeY proofs.

5 Conclusions

In this paper we have presented an implementation of the StaRVOOrS framework combining (partial) static and (optimised) runtime verification. A key contribution 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.

The specification of the property is given as a *ppDATE* [3] which allows us to arbitrarily combine control-oriented (based on automata with event triggered transitions) and data-oriented (relating final and initial data values) properties in a single formalism, and thereby to describe a larger variety of applications. Another aspect of this combination is that data-oriented properties formulated in a pre/post style can be made dependent on the history of previous events.

To illustrate how this framework works, we have applied it to a variant of the Mondex case study [9, 8]. We have analysed the behaviour of the transaction protocol for transferring money between electronic purses, and have demonstrated how this protocol can be verified using our framework. We have also applied our framework to a login system (see [2] for the sources of this case study).

Our current experiments are encouraging as we do improve the time complexity of the runtime verifier Larva. As for memory complexity, we are aware that more work should be done to optimise the size of our generated monitors as the

current translation of *ppDATE* into *DATE* generates many replicated automata. We plan to apply heuristics to reduce the size of such automata.

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.

References

1. MasterCard International Inc. Mondex. `www.mondexusa.com/`.
2. StaRVOOrS. `www.cse.chalmers.se/~chimento/starvoors/files.html`.
3. W. Ahrendt, J. M. Chimento, G. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In *Formal Methods (FM'15)*, LNCS. Springer-Verlag, 2015. To appear. Available online at `http://www.cse.chalmers.se/~chimento/starvoors/publications.html`.
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-Verlag, 2012.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 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.
9. J. Woodcock. First Steps in the Verified Software Grand Challenge. In *SEW'06*, pages 203–206. IEEE Computer Society, 2006.

A Tool Demonstration Script

In this appendix we describe a demonstration on how to use the StaRVOOrS tool, using a working example. The files of the example can be found in `www.cse.chalmers.se/~chimento/starvoors/files.html` under the name *Login example files*.

A.1 Running StaRVOOrS

In order to run StaRVOOrS, as it is illustrated in Fig. 3, the following input should be provided: the address of the source code to be analysed (*Example/Login*), the address of the *ppDATE* specification describing the property to be verified (*Example/prop_add.ppd*), and an output directory where the files generated by the tool are going to be placed (*Example*).

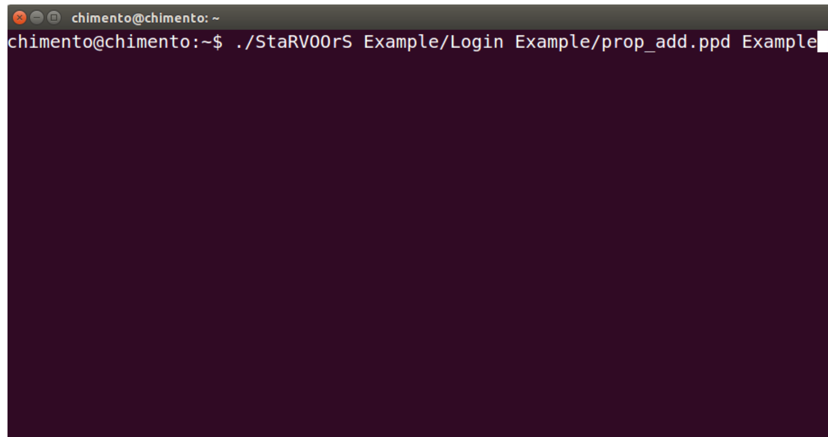

Fig. 3. Runnig StaRVOOrS

A *ppDATE* specification is described on a file with extention *.ppd*. This file consists[4] in 5 sections:

- **Imports**: Lists any packages (or files) which will be used in any of the other sections. At least there should be an import of a package of the system to be monitored.
- **Global**: Describes the automaton (events, automata variables, transitions, states, etc).
- **CInvariants**: Class invariants are described in this section.
- **Contracts**: Lists named Hoare triples.

[4] Not all sections are mandatory.

– **Methods**: Definition of methods to avoid having a lot of code on the transitions of the automaton.

Fig.4 illustrates an example of such a file. We will use it as running example for this demonstration.

A.2 StaRVOOrS ouput

Fig.5 illustrates all the files generated by StaRVOOrS when it is used to analise the *Login* example. This output consists of: the monitor files generated by Larva (folder *aspects* and folder *larva*), the files generated by StaRVOOrS to runtime verify partially proven contracts (folder *partialInfo*), an instrumented version of the source code (folder *Login*), the xml file used by StaRVOOrS to optimise the *ppDATE* specification (*out.xml*), a report explaining the content of the .xml file (*report.txt*) and the *DATE* specification obtained as a result of translating the (optimised) *ppDATE*.

A.3 StaRVOOrS execution insights

StaRVOOrS is a fully automated tool. However, in order to have a better understanding on what happens behind the scenes, we will explain it in three stages.

The first stage is the static verification of the Hoare triples using KeY. Fig. 6 shows the output produced by the tool on the terminal during this stage. At first, KeY (taclet) options are set, which tell KeY how it should proceed during the verification process. For the time being, we are just using the standard options. Then, the KeY prover attempts verifying all the contracts (i.e. the Hoare triples), one by one.

Every time a proof attempt is saturated, some information related to this analysis is given as output in the terminal. Fig. 7 illustrates an example of such a situation for the contract *add_full*.

All the information given as output in the terminal is sum up in the generated file *out.xml*. This file is not intended for the user, it is used by StaRVOOrS to optimise the *ppDATE* specification for runtime checking. However, in order to give to the user some understandable feedback about what happened during the static verification of the contracts, StaRVOOrS generates a file *report.txt* which briefly explains the content of the .xml file.

The second stage correspond to the previously mentioned optimization. On this stage, all the contracts which were proven are removed from the *ppDATE* specification and those which were only partially proven are modified to include the conditions which lead to unclosed path on a proof.

When analysing the specification shown in Fig. 4, KeY fully verifies the contracts *add_full* and *hashfun_ok*, but it only partially proves the contract *add_ok*. Fig. 9 illustrates how the *ppDATE* specification introduced in Fig. 4 would look like after the previous optimization. Note that in the section *CONTRACTS* only *add_ok* remains and that its precondition is strengthened with the predicate

```
IMPORTS { import main.HashTable; }

GLOBAL {
EVENTS {
add_entry(Object u,int key)={HashTable hasht.add(u, key)}
add_exit(Object u,int key)={HashTable hasht.add(u, key)uponReturning()}
hfun_entry(int val)={HashTable hasht.hash_function(val)}
hfun_exit(int val,int ret)={HashTable hasht.hash_function(val)uponReturning(ret)}
}
PROPERTY add {
STATES
{
NORMAL { q2 ; }
STARTING { q (add_ok, add_full,hashfun_ok) ; }
}
TRANSITIONS {
q -> q2 [add_entry\hasht.contains(u, key) < 0\]
}}
}

CINVARIANTS {
HashTable {\typeof(h) == \type(Object[])}
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] == u)}
ASSIGNABLE {size, h[*]}
}
CONTRACT add_full {
PRE {size >= capacity}
METHOD {HashTable.add}
POST {(\forall int j; j >= 0 && j < capacity; h[j] == \old(h)[j])}
ASSIGNABLE {\nothing}
}
CONTRACT hashfun_ok {
PRE {val > 0}
METHOD {HashTable.hash_function}
POST {\result >= 0 && \result < capacity}
ASSIGNABLE {\nothing}
}
}
```

Fig. 4. *ppDATE* description of a property

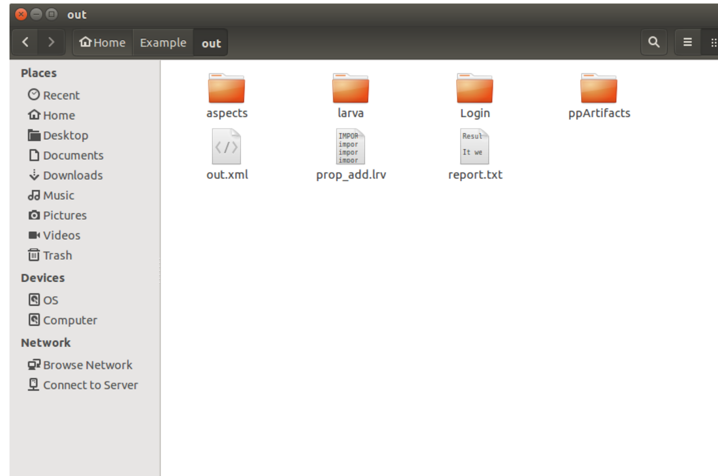

Fig. 5. StaRVOOrS output

```
chimento@chimento: ~
chimento@chimento:~$ ./StaRVOOrS Example/Login Example/prop_add.ppd Example

StaRVOOrS version 1.0

Initiating static verification of Hoare triples with KeY.
Setting the taclet options...
Analizing the contracts...
```

Fig. 6. Initiating Static Verification

```
chimento@chimento: ~/tool StaRVOOrS

main.HashTable[main.HashTable::add(java.lang.Object,int)].JML operation contract.0
================================================================================
self.add(u, key) catch(exc)
pre: self.add_full = TRUE
& self.size >= self.capacity
& (self.<inv> & !u = null)
post: (    exc = null
   ->    \forall int j;
           (    j >= 0 & j < self.capacity & inInt(j)
            -> self.h[j] = (self.h@heapAtPre)[j])
      & self.<inv>)
& (   !exc = null
   ->    (    java.lang.Throwable::instance(exc) = TRUE
          -> self.<inv>)
      & (  java.lang.Error::instance(exc) = TRUE
         |    java.lang.RuntimeException::instance(exc)
           = TRUE))
mod: empty
termination: box
================================================================================
<end> is Termination with path condition: true
 is verified = true
```

Fig. 7. Output shown on the terminal during static verification

```
chimento@chimento: ~/tool StaRVOOrS
          -> self.<inv>)
      & (  java.lang.Error::instance(exc) = TRUE
         |    java.lang.RuntimeException::instance(exc)
           = TRUE))
mod: empty
termination: box
================================================================================
<end> is Termination with path condition: true
 is verified = true

Generating out.xml file...

Process done.

Static analysis complete.
Generating Java files to control partially proven constraints.
Java files generation complete.
```

Fig. 8. Optimization and files generation after static verification

`!(h[hash_function(key)]== null)` (as it is stated in the file *report.txt*) and that in the list of properties to be verified in the starting state q the name of the proved Hoare triples were removed.

StaRVOOrS instruments the source code by adding a new parameter to the method(s) associated to the contract(s), to be used for runtime verification. This new parameter is used to distinguish different executions of the same method. This change is introduced in the *ppDATE* specification too. Besides, StaRVOOrS generates two files (both within a folder named *ppArtifacts*): *Contracts.java* and *Id.java*. The former contains methods which operationalise the pre-/post-conditions of contracts, which will be use by the monitor when verifying the corresponding contract. The latter will be used to generate unique values to be

```
IMPORTS { import main.HashTable; }

GLOBAL {
EVENTS {
add_entry(Object u,int key)={HashTable hasht.add(u, key)}
add_exit(Object u,int key)={HashTable hasht.add(u, key)uponReturning()}
hfun_entry(int val)={HashTable hasht.hash_function(val)}
hfun_exit(int val,int ret)={HashTable hasht.hash_function(val)uponReturning(ret)}
}
PROPERTY add {
STATES
{
NORMAL { q2 ; }
STARTING { q (add_ok) ; }
}
TRANSITIONS {
q -> q2 [add_entry\hasht.contains(u, key) < 0\]
}}
}

CINVARIANTS {
HashTable {\typeof(h) == \type(Object[])}
HashTable {h.length == capacity}
HashTable {h != null}
HashTable {size >= 0 && size <= capacity}
HashTable {capacity >= 1}
}

CONTRACTS {
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] == u)}
ASSIGNABLE {size, h[*]}
}
}
```

Fig. 9. *ppDATE* description of a property

given as new parameters added to the methods. After that, the terminal will look like Fig. 8.

The third stage corresponds to the generation of the monitor files. In order to do so, the *ppDATE* specification is translated by StaRVOOrS to a *DATE* specification. Then, it is used Larva to generate the monitor files from the previous *DATE*. When the execution of Larva is completed, which means that StaRVOOrS execution is completed too, the terminal will reflect the output illustrated in Fig. 10.

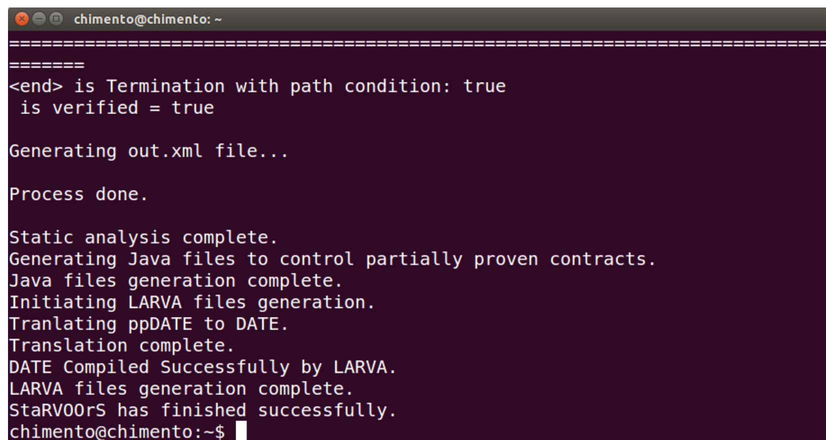

Fig. 10. Monitor Generation

A.4 Running the application with the generated monitor

Once StaRVOOrS finishes its execution, in order to run the application together with the generated monitor the instrumented files have to replace they old version (i.e. none instrumented) in the source code, the folders *aspects*, *larva* and *ppArtifacts* have to be copied in the main folder where the source code is placed and finally all these files must be compiled using an AspectJ compiler (e.g. *ajc*).