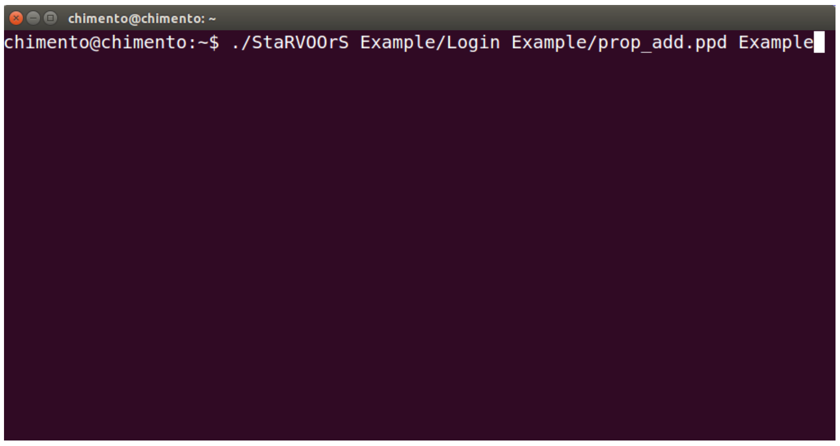
Tool Demonstration Script

Abstract. In this document we describe a demonstration on how to use the StaRVOOrS tool, using a working example. On `www.cse.chalmers.se/~chimento/starvoors`, in section *Tutorials*, a screencast showing this demonstration can be found in under the name *Tutorial on how to use the tool StaRVOOrS*. On the same website, in section *Downloads* you can find the files of the working example under the name *Login example files*.

1 Running StaRVOOrS

In order to run StaRVOOrS, as it is illustrated in Fig. 1, 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*).



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

Fig. 1. Runnig StaRVOOrS

A *ppDATE* specification is described on a file with extention *.ppd*. This file consists[1] of 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.

[1] Not all sections are mandatory.

- **Global**: Describes the automaton (events, automata variables, transitions, states, etc).
- **CInvariants**: Class invariants are described in this section.
- **Contracts**: Lists named Hoare triples.
- **Methods**: Definition of methods to avoid having a lot of code on the transitions of the automaton.

Fig.2 illustrates an example of such a file. We will use it as running example for this demonstration.

2 StaRVOOrS ouput

Fig.3 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 Hoare triples (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*.

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. 4 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. 5 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.

```
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. 2. *ppDATE* description of a property

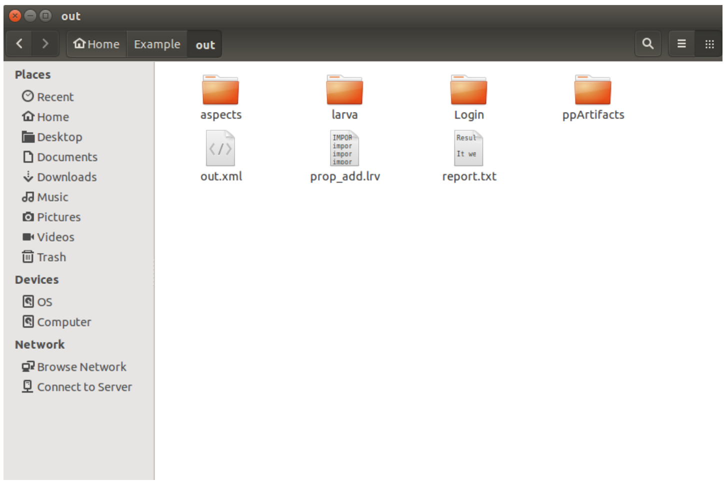
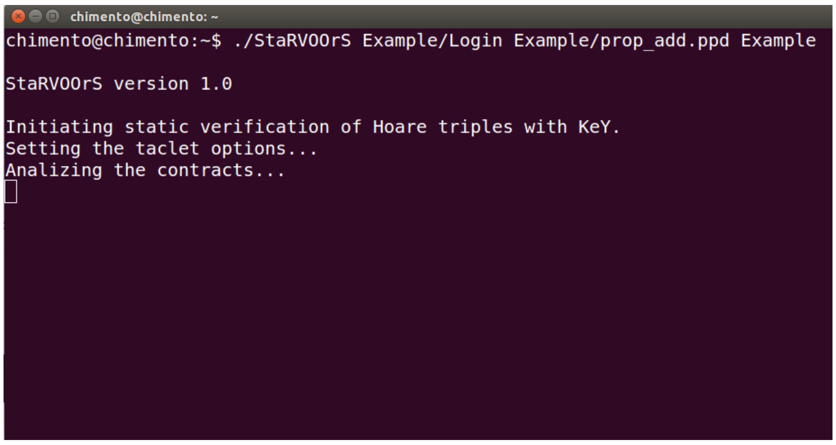
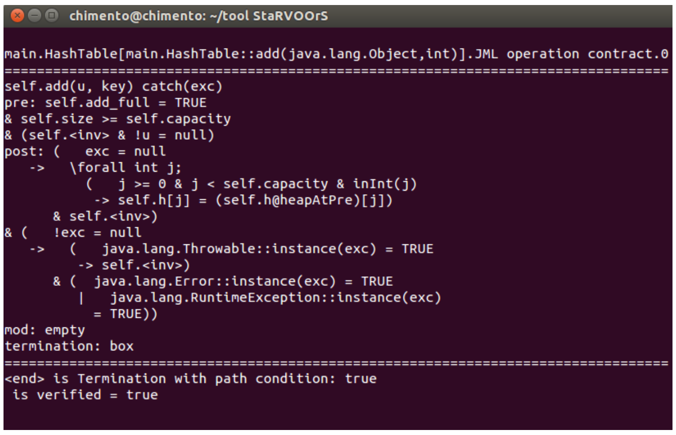


Fig. 3. StaRVOOrS output



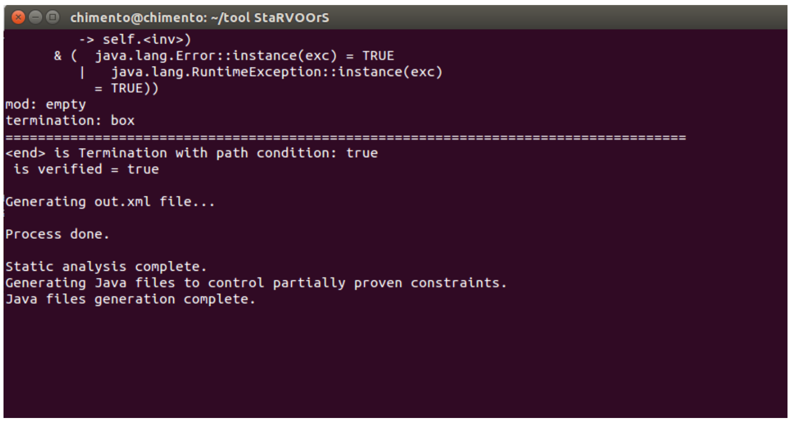
```
chimento@chimento:~$ ./StaRVOOrS Example/Login Example/prop_add.ppd Example

StaRVOOrS version 1.0

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

Fig. 4. 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. 5. 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.
```

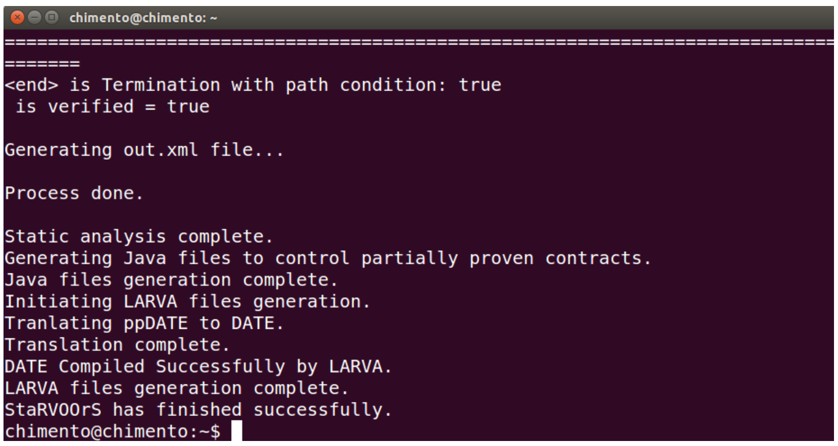
Fig. 6. Optimization and files generation after static verification

When analysing the specification shown in Fig. 2, KeY fully verifies the contracts *add_full* and *hashfun_ok*, but it only partially proves the contract *add_ok*. Fig. 7 illustrates how the *ppDATE* specification introduced in Fig. 2 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 `!(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.

```
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. 7. *ppDATE* description of a property

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 given as new parameters added to the methods. After that, the terminal will look like Fig. 6.
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. 8.



```
==========================================================================
=======
<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 contracts.
Java files generation complete.
Initiating LARVA files generation.
Tranlating ppDATE to DATE.
Translation complete.
DATE Compiled Successfully by LARVA.
LARVA files generation complete.
StaRVOOrS has finished successfully.
chimento@chimento:~$
```

Fig. 8. Monitor Generation

4 Running the application with the generated monitor

Once StaRVOOrS finishes its execution, the simplest way to run the application together with the generated monitor is to create an *AspectJ* project within *Eclipse* (or an application alike) and then use it to deal with the Java files. On this project, the instrumented files have to replace their old version (i.e. none instrumented) in the source code and the folders *aspects*, *larva* and *ppArtifacts* have to be copied in the main folder where the source code is placed. Then, everything is set to run the application within *Eclipse* or to export a *.jar* file.