StaRVOOrS User Manual (release 1.7)

31 May, 2018

Jesús Mauricio Chimento

Contents

1 Introduction

Day by day the use of formal verification techniques to verify the correctness of programs is increasing. In general, verification tools use either static verification techniques (i.e., the verification is performed prior to program execution), or dynamic verification techniques (i.e., the verification is performed during program execution), in order to verify whether a program fulfils certain properties.

Nowadays, a new trend focused on the combination of static and dynamic verification techniques is starting to emerge. StaRVOOrS (STAtic and Runtime Verification of Object-ORiented Software) is a tool which aims at both the specification and verification of properties by combining the use of *Static Verification* and *Runtime Verification*. On the whole, StaRVOOrS is fed with a Java program and a *ppDATE* specification [5] describing properties which the program under scrutiny must fulfil, and it automatically generates a runtime monitor which will verify the specified properties (at runtime) whenever the provided program is executed.

This document is the user manual of StaRVOOrS. Its structure is as follows. Section 2 provides an intuitive description of the *ppDATE* specification language used by this tool. Section 3 gives a high level explanation about how this tool works. Section 4 shows how to write a *ppDATE* specification in the input language of the tool. Finally, section 5 provides a complete example on how to run this tool.

2 *ppDATE* Specification Language

Here, we briefly introduce the *ppDATE* specification language. However, both its complete (formal) description and its semantics can be found in [6].

ppDATE is an automaton-based formalism which, basically, consists of a labelled transition system whose states may include Hoare triples describing properties about the methods of the system under scrutiny.

Transitions in a *ppDATE* are labelled by a trigger (tr), a condition (c) and an action (a). Together, the label is written $tr \mid c \mapsto a$. A transition is *enabled* to be taken whenever its trigger is active and the condition guarding it holds. In addition, if a transition is taken, we say that it is *fired*. Whenever a transition is fired, its action is executed.

Regarding the triggers, they are activated by the occurrence of either a visible *system event* such as entering or exiting a method, or an *action event* generated by certain actions labelling other transitions. We use the notation $\texttt{foo}^{\downarrow}$, $\texttt{foo}^{\uparrow}$, $e?$, to represent the trigger which is activated whenever the method `foo` is entered, the trigger which is activated whenever the method `foo` is exited, and the trigger which is activated whenever the action event e occurs, respectively.

Regarding the conditions, they are expressions written using JML boolean expression syntax [10]. Conditions may depend on the values of *system variables* (i.e., of the system

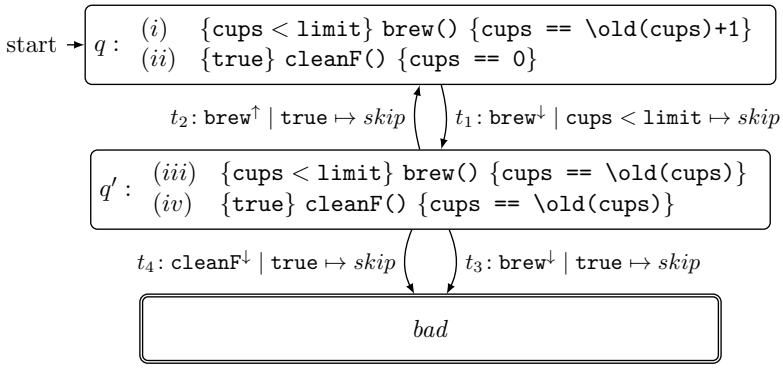

Figure 1: A *ppDATE* controlling the brew of coffee

under scrutiny) and the values of *ppDATE variables* (i.e., variables which belong to the *ppDATE*). The latter can be modified via actions in the transitions.

Regarding the actions, they consist on any number of the following: (i) assignments of the form $v = exp$, where v is a *ppDATE* variable and *exp* is an expression that may depend on system variables and *ppDATE* variables; (ii) an action *!* such that *e!* represents the generation of the action event *e*; (iii) an action `\create`, used to generate instances of a *ppDATE* template (see Sec. 4.3); (iv) IF-THEN conditional expressions whose branching condition depends on the valuations of system variables and *ppDATE* variables; (v) an action `\log` such that `\log`(*string*) adds *string* into the log file generated by the monitor; (vi) and (Java) programs. All the actions should end in a semicolon.

In relation to the Hoare triples on the states of a *ppDATE*, intuitively, if a Hoare triple $\{\pi\}$ `foo()` $\{\pi'\}$ is included in some state q, this property ensures that: if method `foo` is entered while the monitor is in state q, and pre-condition π holds, then upon reaching the corresponding exit from `foo`, post-condition π' should hold. Both pre-/post-conditions in the Hoare triples are expressed using *JML Boolean Expressions* syntax (see Sec. 4.5 for details about this syntax).

Now, let us introduce an example in order to give a better intuition on how a *ppDATE* is described.

ppDATE Specification Example

Let us consider a *coffee machine system* where, after a certain amount of coffee cups are brewed, its filters have to be cleaned. If the limit of coffee cups is reached, the machine should not be able to brew any more coffee. In addition, while the coffee machine is active (a coffee cup is being brewed), it is not possible to start brewing another coffee, or to clean the filters.

Fig. 1 illustrates a *ppDATE* describing this part of the system. In other words, whenever

the coffee machine is not active, i.e., the machine is not brewing a cup of coffee, and the method `brew` starts the coffee brewing process, then it is not possible either to execute this method again, or to execute the method `cleanF` (which initialises the task of cleaning the filter), until the initialised brewing process finishes.

The previous property can be interpreted as follows: initially being in state *q*, the state which represents that the coffee machine is not active, whenever method `brew` is invoked and it is possible to brew a cup of coffee (i.e., the limit of coffee cups was not reached yet), then transition t_1 shifts the *ppDATE* from state *q* to state *q'*. While in *q'*, the state which represents that the coffee machine is active, if either method `brew` or method `cleanF` are invoked, then transitions t_3 or transition t_4 shift the *ppDATE* to state *bad*, respectively. This indicates that the property was violated. On the contrary, if method `brew` terminates its execution, then transition t_2 shifts the *ppDATE* from state *q'* to state *q*. Note that the names used on the transitions, e.g. t_1, t_2, etc., are not part of the specification language. They are included to simplify the description of how the *ppDATE* works.

In addition to this, the Hoare triples in state *q* ensure the properties: (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. Property (i) has to be verified if, while the *ppDATE* is on state *q*, the method `brew` is executed and its precondition holds; and property (ii) has to be verified if, while the *ppDATE* is on state *q*, the method `cleanF` is executed and its precondition holds. Regarding state *q'*, the Hoare triples in this state ensure the properties: (iii) no coffee cups are brewed; (iv) filters are not cleaned. Property (iii) and (iv) are verified if either method `brew` and method `cleanF` are executed, and their preconditions hold, respectively. Here, remember that this state represents that the coffee machine is active. Thus, if it occurs that either the method `brew` or the method `cleanF` are executed while the *ppDATE* is on this state, then, as this would move the *ppDATE* to state `bad`, one would expect the value of the variable `cup` to remain unchanged. This is precisely what is verified when either property (iii) or (iv) are analysed.

Note that none of the Hoare triples makes reference to the state of the coffee machine, i.e., there is no information about whether the machine is active or not. This is due to fact that the state of the machine is implicitly defined by the states of the *ppDATE*. If the *ppDATE* is in state *q*, the coffee machine is not active. However, if it is in state *q'*, then the machine is active. Therefore, the Hoare triples are *context dependent*. This is the reason why we can describe properties with the same precondition, but with different postconditions depending on the state of the *ppDATE* in which they are placed.

3 High-level Description of StaRVOOrS

StaRVOOrS takes three arguments: (i) the path to the main folder of the Java files to be verified; (ii) a description (as input language) of the *ppDATE* specification for the provided program; and (iii) the path of the output folder (the generated files are stored in

this folder). Then, it automatically generates (1) a runtime monitor; (2) an instrumented version of the Java files in (i); (3) a report summarising the results obtained by statically verifying the Hoare triples described in (ii); (4) and a refined version of (ii), when possible.

To generate such output, StaRVOOrS combines the use of the deductive source code verifier KeY [4] with the runtime monitoring tool Larva [8]. KeY is a deductive verification system for data-centric *functional correctness* properties of Java programs, which generates, from JML [10] and Java, proof obligations in *Dynamic Logic* (a modal logic for reasoning about programs) [9], and attempts to prove them by using a *sequent calculus* which follows the *symbolic execution* paradigm. Larva is an automata-based Runtime Verification tool for Java programs which automatically generates a runtime monitor from a property using the automaton-based specification language *DATE* [7]. Larva transforms such specification into monitoring code together with AspectJ code to link the system under scrutiny with the generated monitor.

In a nutshell, StaRVOOrS output is generated by following the steps enumerated below.

(a) The Hoare triples described in (ii) are translated into JML contracts, which are textually added to to the Java files in (i) as annotations of the respective methods;

(b) KeY attempts to (statically) verify all the JML contracts automatically. The result obtained for each contract is either a complete proof, or a partial proof where some parts of the contract are proved and others are not, or that KeY cannot prove any of the parts the contract. These results are stored in a XML file. In addition, a report summarising the content of this file, i.e., (3), is generated. Here, note that our tool does not support user interaction with KeY. It uses this prover in fully automatic mode;

(c) The *ppDATE* specification is refined based on the XML file, i.e., (4). Fully verified Hoare triples are removed from the specification, but those Hoare triples which are not fully verified, are left in the specification to be verified at runtime. However, the original pre-conditions of the remaining Hoare triples may be strengthen with the (path) conditions resulting from partial proofs, thus covering at runtime only executions that are not closed in the static verification step;

(d) The refined *ppDATE* specification is encoded into a *DATE* specification. In particular, the *DATE* specification language does not support pre/post-conditions which thus have to be translated to use notions native to this specification language. This also requires a number of changes to the system (through code instrumentation), in order to be able to distinguish different executions of the same code unit, and to evaluate the Hoare triples in the states of the refined *ppDATE* at runtime. i.e, (2). Regarding the former, method declarations get a new argument which is used as a counter for invocations of this method. Regarding the latter, not every condition in a pre/postcondition of

a Hoare triple can be directly written as a Java Boolean Expression, e.g., quantified expressions. Thus, methods which operationalise the evaluation of those conditions are added to the Java files in (i);

(e) The LARVA compiler generates a runtime monitor using aspect-oriented programming techniques, i.e., (1).

Once deployed, the runtime monitor and the instrumented version of the Java files are executed together, thus effectively running the monitor in parallel with the program. The runtime monitor identifies violations at runtime, reporting error traces to be analysed.

4 Composing a *ppDATE* Specification in the Input Language of StaRVOOrS

In this section we explain in detail how to write a *ppDATE* specification using the input language of StaRVOOrS. The files written in such language have extension *.ppd*, and their content may consist on 6 sections which are ordered as follows: `IMPORTS`, `GLOBAL`, `TEMPLATES`, `CINVARIANTS`, `HTRIPLES` and `METHODS`. Below, we describe the content of each one of these sections, show their syntax, and provide examples illustrating how to write them.

4.1 IMPORTS

Section `IMPORTS` lists the packages included in the system under scrutiny which are related to the properties to be verified (both the Hoare triples and the automata). Its syntax is described as follows:

```
IMPORTS { import package ; }
```

Each package listed in this section follows the usual Java syntax for imports. For instance,

```
IMPORTS {
  import main.Foo ;
  import other.sub.Goo ;
  import other.Hoo ;
}
```

4.2 GLOBAL

Section `GLOBAL` contains the description of the *ppDATE* specification. Its syntax, which is described below, is written as follows:

```
GLOBAL {
 VARIABLES { -- definition of the variables -- }

 ACTEVENTS  { -- definition of the action events -- }

 TRIGGERS { -- definition of the triggers -- }

 PROPERTY property_name1 {
  STATES {  -- definition of the states of the ppDATE -- }
  TRANSITIONS {  -- definition of the transitions of the ppDATE -- }
 }

 PROPERTY property_name2 {
   -- definition of states and transitions --
 }
 ...
}
```

Note that one may describe more than one `PROPERTY`. This would be the case when one is describing several *ppDATE*s in one single specification file, i.e., each property represents a *ppDATE*.

4.2.1 VARIABLES

Subsection `VARIABLES` allows to include as part of the specification the declaration of variables. These variables, which are referred to as *ppDATE* variables, may be freely used in the transitions of a *ppDATE*, both in their conditions and actions. For instance, one may use an integer variable as a counter to keep track of how many times a method is executed. Below, we illustrate how variables may be defined within this subsection.

```
VARIABLES {
  type var ;
  type var = initial_value ;
}
```

Such syntax follows the usual Java syntax for the declaration of variables. For instance,

```
VARIABLES {
  String s;
  int i = 0;
}
```

Note that whenever a variable is not initialised when it is defined, its initialisation has to be performed by the execution of an action. Otherwise, there is going to be an exception at runtime whenever the monitor attempts to manipulate such variable.

4.2.2 ACTEVENTS

Subsection `ACTEVENTS` includes the declaration of the different *action events* which may be generated by using the action *!*. Here, it is only necessary to list the names of these events, as illustrated in the example below for the action events `e1`, `e2`, and `e3`.

```
ACTEVENTS {
  e1 ; e2 ; e3 ;
}
```

4.2.3 TRIGGERS

Subsection `TRIGGERS` includes the declaration of the different triggers which may be used in the transitions of a *ppDATE*.

Triggers Associated to System Events

The triggers which are activated by the occurrence of a visible *system event*, i.e., entering or exiting a method, have the following signature:

```
name(args) = {varDecl.method(args')sysevent }
```

Here, `name` is a label which works as an identifier for the trigger; `method` is the name of the method generating the system event which activates the trigger; and `sysevent` represents whether the trigger is activated by a system event produced by entering or exiting a method, represented with the notation `entry` or `exit`, respectively.

In addition, each trigger may have a number of arguments `args` which act as binds for `args'` (i.e., `args'` are the arguments in `args`, but without their types). This allows the access at runtime to the arguments which are being provided to the method, and to the value returned by a method (see examples below).

Regarding `varDecl`, it is a variable declaration which has the following signature:

```
varDecl = * | identifier | type identifier
```

The symbol `*` means that the triggers can be activated by an event associated to `method`, no matter what class it belongs to; `identifier` is the target object (instance of the class `type`) on which `method` is being called. Note that `identifier` should be always associated to a class. Thus, if one uses only `identifier` as variable declaration, then `args` should include an argument of the form `type identifier`. In addition, one may use `identifier` to access at runtime the target object.

Below, by considering the Java classes depicted in Fig. 2, we give several examples illustrating the different manners in which triggers might be defined.

```
public class Foo {
   public void foo();
}
```

```
public class Goo {
   public void goo(int x,boolean b);
   public int hoo(int x, int y, int z);
}
```

Figure 2: Example of Java classes.

```
TRIGGERS {
  foo1()                 = {Foo f.foo()entry}
  foo2()                 = {*.foo()entry}
  goo1(int x, boolean b) = {Goo g.goo(x,b)entry}
  goo2(int x)            = {Goo g.goo(x,*)entry}
  foo3()                 = {*.foo()exit()}
  goo4(int x,boolean b)  = {Goo g.goo(x,b)exit()}
  hoo(int x, int ret)    = {Goo g.hoo(x,*)exit(int ret)}
}
```

On these definitions, whenever either the target object of the method or any of the arguments of the method are not necessary for the definition of a trigger, e.g., the definition of a trigger which is activated by the execution of a method `foo` where several classes have an implementation for this method, they can simply be omitted by replacing them with the symbol '$*$', which is used as a place holder. Triggers `foo2`, `goo2`, `foo3`, and `hoo` are examples illustrating these situations. In addition, in the definition of a trigger which is activated by a system event produced by exiting a method, it is possible to refer to (and later to access) the value (or object) returned by the method, by including in the arguments of the trigger an argument with an appropriate type to represent such value, and then including this argument in the notation `exit`, as it is illustrated in the definition of trigger `hoo`.

Triggers Associated to Constructors

It is possible to define a special exit trigger which is activated when an object of a certain class is created. These triggers have the following signature:

```
name(type obj) = {type.new()exit(obj)}
```

where `obj` represents the created object for the class `type`. If the constructor has arguments, then they have to be included as part of the arguments of the trigger and the call to `new`. For instance, let us assume that the following signature for the constructor of a class `Foo`: `Foo(int n, boolean b)`. Then, one should define this kind of triggers as follows:

```
foo_new(int n, boolean b, Foo obj) = {type.new(n,b)exit(obj)}
```

4.2.4 PROPERTY

The core of this section is the subsection `PROPERTY`. It consists of the actual description of a *ppDATE*. This subsection is divided in two parts: `STATES` and `TRANSITIONS`. Note that there should be defined at least one property here.

STATES

Section `STATES` lists all of the states in a *ppDATE*. There are four kind of states: starting states, accepting states, bad states, and normal states. `STARTING` list the initial state of the *ppDATE*. There should be only one starting state listed. The accepting states, which are listed in `ACCEPTING`, represent the states in which it is desirable for the monitor to be in whenever the program under scrutiny terminates its execution. The bad states, which are listed in `BAD`, represent states which a monitor reaches whenever a property which is described with the transitions of the *ppDATE* is violated at runtime. Finally, normal states, which are listed in `NORMAL`, are neither accepting nor bad states, but simply possible states where a monitor may be in during the execution of a program.

In relation to the list of states, each entry consists on the name of a state, and a list of the names of the Hoare triples which have to be verified in that state (this is properly explained in Sec. 4.5). Entries in a list of states terminate in a semicolon.

Below you can see an example of a `STATES` subsection.

```
STATES {
 STARTING { q0 (h1) ; }
 ACCEPTING { q4 ; q5; }
 BAD { bad ; }
 NORMAL { q1 ; q2 (h2,h3) ; q3; }
}
```

Note that the order previously illustrated in the syntax (starting, accepting, bad, normal) should be preserved. In addition, it is mandatory to always include a starting state on a *ppDATE*. Otherwise, the monitor will not know from which state it should start. Finally, it is important to remark that the accepting states are considered *sink states*, i.e. they should not have outgoing transitions. Therefore, once a *ppDATE* reaches one of such states, it is deactivated (i.e. it stops running).

TRANSITIONS

Section `TRANSITIONS` contains the description of all the transitions in the *ppDATE*. For each *ppDATE* transition going from state `q` to state `q'` with trigger `tr`, (optional) condition `c`, and (optional) action `a`, this section includes a line of the form

```
q -> q' [tr \ c \ a]
```

Here, where `tr` should be either the name of a trigger defined in a `TRIGGERS` subsection (see Sec. 4.2.3), or an expression of the form `e?`, where `e` is an action event defined in the `ACTEVENTS` section (see Sec. 4.2.2); `c` is a boolean expression following JML syntax [10], which may depend on both system and *ppDATE* variables; and `a` is an action consisting on sequence of the following:

- assignments of the form $v = exp$, where v is a *ppDATE* variable and *exp* is an expression that may depend on system variables and *ppDATE* variables;
- an action `\gen(e)` which generates the action event `e` that activates the trigger `e?`;
- an action `\create`, used to generate instances of *ppDATE* templates (see Sec. 4.3);
- IF-THEN conditional expressions whose branching condition depends on the valuations of system variables and *ppDATE* variables;
- an action `\log` such that, `\log`(*string*) adds *string* into the log file generated by the monitor;
- a block of actions delimited by curly brackets;
- actions $++\,v$, $--\,v$, $++\,v$, $v--$, $v+=e$, and $v-=e$, with their standard (Java) meaning;
- (Java) programs.

All the actions should terminate in a semicolon. Regarding action `\gen(e)`, both in the theory ([5]) and in Sec. 2, this action is represented with the symbol !, i.e., `e!` generates the action event `e`. The main reason why we decided not to use the same notation in our input language as the one used in the theory is that both JML and Java use the symbol ! as the boolean negation. Thus, we consider that by introducing action `\gen` instead, we are avoiding the possible confusion which may arise in relation to whether `v!` refers to the negation of the boolean variable `v`, or the generation of the action event `v`. In addition, the trigger `e?` in the fourth transition represents the trigger which is activated whenever the action event `e` occurs.

Regarding the use of (Java) programs in the actions, the tool only supports the use of method calls, e.g., `inc(v)` in the first transition. However, it is possible to use the section METHODS (see Sec. 4.6) to define programs as (Java) methods. Then, one simply has to make a method call to them.

Below we list all the syntactically valid expressions which may be used within this section.

(1) `q -> q' [tr\c\a]`

(2) `q -> q' [tr\\]`

(3) `q -> q' [tr\]`

(4) `q -> q' [tr]`

(5) `q -> q' [tr\c\]`

(6) `q -> q' [tr\c ]`

(7) `q -> q' [tr\\a]`

Note that the expressions (2), (3), and (4) are equivalent. Similarly, expressions (5) and (6) are equivalent as well. In addition, when using a trigger in a transition it is not necessary to write their arguments in the trigger component of the transition. This does not affect the possibility of using such arguments in both the conditions, and the actions. For instance, given the trigger `t(int x) = {Goo g.goo(x)entry}`, one may define the transition `q0 -> q1 [t\x == 8]`, where `x` is the argument of the trigger `t`.

Now, let us illustrate some of the previous expressions with the following example:

```
TRANSITIONS {
  q0 -> q1 [tn \ c == 8 \ v = 0; \gen(e);]
  q0 -> q2 [f \ c == 2 \ ]
  q1 -> q3 [g \ \ IF (b) THEN inc(v); foo();]
  q2 -> q2 [ae?]
  q3 -> q2 [g \ true \ \log("info");]
}
```

4.3 TEMPLATES

In addition to *ppDATE*s which exist up-front, and 'run' from the beginning of a program's execution, new *ppDATE*s can be created by existing ones. For instance, one may want to create a separate 'observer' for each new user logging into a system. For that, one needs to be able to define parameterised *ppDATE*s, which we call *templates*, and allow *ppDATE*s to create new instantiations of them. Fig. 3 illustrates an example of a *ppDATE* template called *login-logout* which, given a user `u`, describes the property "the user has to log in to perform a deposit".

Section `TEMPLATES` lists tagged *ppDATE* templates. Below, we show the syntax of this section.

login-logout = λ u : User, tr : Trigger.

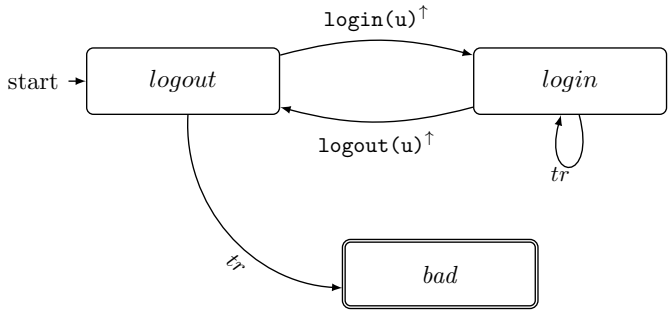

Figure 3: *ppDATE* template describing properties about the log in and log out of users.

```
TEMPLATES {
 TEMPLATE id_template (params) {
    VARIABLES { -- definition of the variables -- }
    TRIGGERS  { -- definition of the triggers -- }
    PROPERTY name { -- definition of the property -- }
 }
}
```

Each template is described within a subsection `TEMPLATE`, whose header is followed by a (unique) name `id_template` assigned to the template, and a list of parameters `params` used to generalise the definition of the templates. Each element in `params` has the form `Type var`, where `Type` is either a reference type (i.e. Java class), or one of the following special types: `Trigger`, `Condition`, `Action`, `HTriple`, and `MethodName`. These special types can be used to abstract triggers, conditions, actions, Hoare triples, and method names, respectively, in a template. Regarding `var`, it represents the abstraction of a value (of the corresponding type).

Note that as a template describes a *ppDATE*, the subsections `VARIABLES`, `TRIGGERS`, and `PROPERTY` are defined just like it is described in Sec. 4.2. In addition, the triggers defined in a template may have the same name as the triggers defined in a (non-template) *ppDATE*. Whenever this happens, the template will always refer to its own definition of the trigger. Below, we illustrate how the *ppDATE* template in Fig. 3 could be written using this syntax.

```
TEMPLATES {
 TEMPLATE login-logout (User u,Trigger tr) {
  TRIGGERS  {
    login_ex(String username, int pwd) = {u.login(username, pwd)exit()}
    logout_ex() = {u.logout()exit()}
```

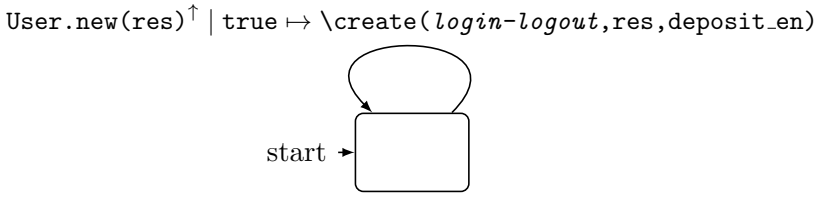

Figure 4: *ppDATE* in charge of creating instances of the template *login-logout*.

```
  }
  PROPERTY deposit {
    STATES {
       STARTING { logout ; }
       ACCEPTING { login ; }
       BAD { bad ; }
    }
    TRANSITIONS {
       logout -> login [login_ex]
       logout -> bad [tr]
       login  -> logout [logout_ex]
       login  -> login [tr]
    }
  }
 }
}
```

Regarding the instantiation of a template, it is accomplished by using the action create on the transition of a *ppDATE*. This action receives as arguments the name of the *ppDATE* template to be instantiated and a list of values to instantiate the parameterised arguments of the template, and it generates the instance of the template. For example, Fig.4 illustrates a *ppDATE* which creates an instance of the template *login-logout* (Fig. 3) upon declaration of an object of class `User`. Here, `res` represents the (concrete) object of class `User` which was created. In addition, the trigger `User.new`$^{\uparrow}$ is activated when such a creation occurs.

4.4 CINVARIANTS

Section `CINVARIANTS` lists the definitions of class invariants which may need to be considered during the verification of the properties. Its syntax is described as follows:

```
CINVARIANTS {
 class { invariant }
}
```

Here, `class` represents a Java class in the program under scrutiny whose implementation has to preserve the invariant definition described by `invariant`. Such invariants follow JML-like syntax and pragmatics. Below we illustrate an example of this section.

```
CINVARIANTS {
 Foo { v <= 10 }
 Foo { count >= 0 }
}
```

Note that if no class invariants are needed on a specification, then this section may be omitted. In addition, the actual version of the tool only uses the class invariants during the static verification of the Hoare triples. However, we are currently working to include the verification of class invariants at runtime as well.

4.5 HTRIPLES

Section *HTRIPLES* lists tagged Hoare triples. Its syntax is described as follows:

```
HTRIPLES {
  HT hoare_triple_name {
    PRE { -- precondition -- }
    METHOD { -- method to verify -- }
    POST { -- postcondition -- }
    ASSIGNABLE { -- variables modified -- }
  }
}
```

Each Hoare triple is described within a subsection `HT`, whose header is followed by the name assigned to the Hoare triple. This name is unique for each Hoare triple, and it is used to associate the Hoare triples with the states of a *ppDATE*. Subsection `HT` is composed by four parts: `PRE`, which describes the pre-condition of the Hoare triple; `POST`, which describes the post-condition of the Hoare triple; `METHOD`, which describes which is the method that has to fulfil the Hoare triple; and `ASSIGNABLE`, which lists the variables that might be modified when the method under scrutiny on the Hoare triple is executed. Here, `PRE`, `POST`, and `ASSIGNABLE` follow JML-like syntax and pragmatics.

Regarding `METHOD`, it is an expression of the form `file.method(types)`, where `method` is the name of the method related to the Hoare triple, `file` is the Java class where the previous method is implemented, and `types` is a list of arguments that can be used to differentiate methods with the same name. Note that the use of `types` is optional. Regarding `PRE`, `POST`, and `ASSIGNABLE`, these parts may be omitted. If so, they will have as default values `true`, `true`, and `\everything`, respectively.

Below, we provide an example illustrating the use of this section.

```
HTRIPLES {
  HT inc_ok {
    PRE { v }
    METHOD { Foo.inc }
    POST { count == \old(count)+1 }
    ASSIGNABLE { count }
  }
  HT inc_err {
    PRE { !v }
    METHOD { Foo.inc }
    POST { count == \old(count) }
    ASSIGNABLE { \nothing }
  }
  HT add1_ok {
    PRE { true }
    METHOD { Goo.add(int) }
    POST { x == \old(x) + n }
    ASSIGNABLE { \everything }
  }
  HT add2_ok {
    METHOD { Goo.add(int,int) }
    POST { x == n + m }
  }
}
```

Note that this section may contain several subsections `HT`, one per each Hoare triple. In addition, if no Hoare triples are included as part of a *ppDATE*, then this section may be omitted.

4.6 METHODS

Section *METHODS* is an optional section which allows to include method declarations as part of a specification. These methods will be included as part of the implementation of the monitor generated by the tool. Its syntax is described as follows:

```
METHODS {
  type method(arguments) { -- method implementation -- }
}
```

Methods are declared following standard Java notation. However, access modifiers (i.e., *public*, *protected*, *private*) are not necessary when declaring these methods. If a method is declared as *static* method, then monitor variables will not be accessible within that particular method. Below, we illustrate an example of this section.

```
METHODS {
  boolean compare (int x, int y) { return (x == y); }
  int four() { return 4 ; }
}
```

4.7 Remarks

4.7.1 Comment Lines

One may include comments in a *ppDATE* specification by writing %% followed by the comment.

4.7.2 Key Words

The key words are words appearing in the grammar of *ppDATE*. We strongly recommend not to use these words as part of your implementation. Otherwise, you may run into parsing issues. The key words used in *ppDATE* are the following:

| ACCEPTING | ACTEVENTS | ASSIGNABLE | BAD
| CINVARIANTS | FOREACH | GLOBAL | HT
| HTRIPLES | IMPORTS | METHOD | METHODS
| NORMAL | PINIT | POST | PRE
| PROPERTY | STARTING | STATES | TEMPLATE
| TEMPLATES | TRANSITIONS | TRIGGERS | VARIABLES
| call | entry | execution | exit | final | import
| uponHandling | uponThrowing | where

4.7.3 Private Variables in the Hoare triples

When writing a Hoare triple it is possible to include private variables both in its precondition, and its postcondition. StaRVOOrS will automatically annotate the source code with appropriate JML clauses (i.e. `spec_public`) preceding the (private) variables definition, at the time of statically verifying the Hoare triples. However, note that if later one of these Hoare triples has to be verified at runtime, then the monitor will require access to the private variables values. In such situations, the user will have to decide how to deal with these variables and modify the generated files accordingly. For instance, one can introduce getter methods in order to give access to the monitor to the values of the private variables, and then update the generated file HoareTriplesPPD.java (under ppArtifacts) to use those methods. See Sec. 5.4 for more details about the files generated by the tool.

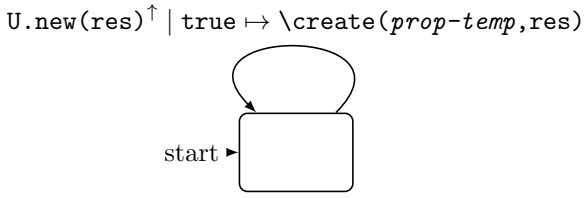

Figure 5: *ppDATE* in charge of creating instances of the template *prop-temp*.

4.8 Extra Features

This section describes some extra features added to the tool and its input language, which are not covered by the semantics described in [6]. Note that in Sec. 4.8.7 we provide arguments regarding how the soundness of the *ppDATE* semantics is preserved when considering these features.

4.8.1 PINIT Definition in Section PROPERTY

When describing a *ppDATE* specification, it is quite common to have some *ppDATE*s only focus on creating instances of a template upon declaration of an object. Such *ppDATE*s would look like the *ppDATE* illustrated in Fig. 5, which creates an instance of the template `prop-temp` every time an object of the class `U` is created. Here, `\result` represents the (concrete) object of class `U` which was created. In addition, the trigger `U.new`$^{\uparrow}$ is activated when such a creation occurs.

Therefore, we decided to include a special subsection `PINIT` as part of the section `PROPERTY`, which can be used to specify these kinds of *ppDATE*s in a simple manner. The syntax of this subsection is as follows:

```
PROPERTY property_name {
   PINIT { (template_name,Class) }
}
```

where `property_name` is the name of the property (as described in Sec.4.2.4), `template_name` is the name of a *ppDATE* template defined in section TEMPLATE, and `Class` is the name of the class associated to the declared object. Below, we illustrate how the *ppDATE* from Fig.5 is described using this special subsection.

```
PROPERTY example {
   PINIT { (prop-temp,U) }
}
```

Alike Fig. 5, property `example` describes a *ppDATE* which has a single state with only a loop transition which is fired every time an object of the class `U` is created, leading to an instantiation of the template `prop-temp` using the created object as argument.

4.8.2 Where clause

When declaring a trigger, any of its arguments can be bound to a variable which is not directly related to the method arguments. For instance, let us assume that we have to perform some processing on a particular value, and that we want that, depending on its result, a *ppDATE* fires a transition (or not). Then, by using a where clause right after a trigger definition one can use one argument of the trigger as a bound for that particular value. Consider the next example:

```
TRIGGERS {
 goo(int x,boolean y) = {Goo g.foo(x)entry} where {y = g.IsValid();}
}
```

Here, we do not have any interest in the whole object `g`, but we simply need to know if its a valid object or not, fact which can be computed using the method `IsValid()`, in order to send the *ppDATE* to either the state `q2`, or `bad`, respectively. Then, one can use the boolean argument `y` of the trigger for binding the result of that method. This would allow us to write transitions like the following ones:

```
TRANSITIONS {
  q1 -> q2  [goo\ y]
  q1 -> bad [goo\ !y]
}
```

Here, remember that it is not necessary to write the arguments of a trigger in the trigger component of a transition, but one can refer to them in both the conditions and the actions.

Furthermore, any variable which is not directly bound to the method arguments is initialized in the where clause. This is done by checking that there is at least one assignment statement with the unbound variable on the left-hand side.

Note that the statements in the where clause can be any valid JAVA statements and these can call any relevant method from imported packages, and that the use of curly brackets in this clause is compulsory.

4.8.3 Foreach construct

The `FOREACH` construct can be used as a simplistic alternative to the use of *ppDATE* templates. Consider the following *ppDATE*:

```
GLOBAL {
  TRIGGERS {
    log(User user) = {Interface f.login(User user)entry}
    out(User user) = {Interface f.logout(User user)entry}
```

```
  }
  PROPERTY example {
    STATES {
      ACCEPTING { logout ; }
      BAD { bad ; }
      STARTING { login ; }
    }
    TRANSITIONS {
       logout -> login [log\\ \create(deposit-temp,user)]
       logout -> bad [out]
       login  -> logout [out]
       login  -> bad [log]
    }
  }
}

TEMPLATES {
 TEMPLATE deposit-temp (User u) {
  TRIGGERS {
    dep(int amount) = {u.deposit(amount)entry}
   }
   PROPERTY deposit { --- }
 }
}
```

On this *ppDATE*, every time a user logs in the interface, an instance of the template `deposit-temp` is created in order to runtime verify the property `deposit` for that user.

Now, let us introduce a similar *ppDATE* to the one described above, but written using the foreach construct:

```
GLOBAL {
  TRIGGERS {
    log(User user) = {Interface f.login(User user)entry}
    out(User user) = {Interface f.logout(User user)entry}
  }
  PROPERTY example {
    STATES {
      ACCEPTING { logout ; }
      BAD { bad ; }
      STARTING { login ; }
    }
    TRANSITIONS {
       logout -> login [log]
```

```
        logout -> bad [out]
        login  -> logout [out]
        login  -> bad [log]
    }
  }

  FOREACH (User u) {
   TRIGGERS {
    dep(int amount) = {User u1.deposit(amount)entry } where {u = u1;}
   }
   PROPERTY deposit { --- }
  }
}
```

In this version of the *ppDATE*, as soon as an object of the class `User` is created, a *ppDATE* verifying the property `deposit` is generated. Here, remember that this is not what happen the template version of the *ppDATE*, where the *ppDATE* verifying property `deposit` is only created when a user logs in.

Using a foreach construct may seem simpler than using a *ppDATE* template. However, one have to consider the following points when using it:

(i) This construct introduces a context to the *ppDATE*, i.e., the triggers, variables and transitions will now be in a particular context. Hence, each trigger should specify its context so that the *ppDATE* which will be affected will only be the one belonging to that particular context. This is done by using the where clause associated to each trigger. In addition, variables may be affected for the introduction of a context. This happens when different contexts have variables with the same name. By default, variables are match to the innermost context. However, it is possible to indicate which one is the context of the variable by using the following special notation: `::amount`, `::u::amount`. The former notation refers to the variable `amount` in GLOBAL (i.e. top level), whereas the latter refers to the variable `amount` in the context of the foreach (i.e. `Foreach (User u)`).

(ii) *ppDATE*s for verifying the properties within a foreach are always going to be generated upon creation of an object, even if the execution of the program does not require to verify them.

(iii) This construct can only refer to reference types.

(iv) *ppDATE* templates are much more expressive than this construct.

Anyhow, note that we mainly decided to include this construct in our language because:

(I) in the case where one just wants to generalise a specification regarding objects of a particular reference type, this construct may be simpler to use compared to the use of a template;

(II) *DATE* users can start writing *ppDATE* specifications right away, without being limited to learn to use templates first;

(III) it allows to migrate *DATE* specifications to *ppDATE* specifications in a simple manner.

4.8.4 Channel Communication

ppDATE offers a simplistic manner for automata communication by using action *!*. However, in certain situations it would be desirable to send information (i.e. an object) from one automaton to the other. Thus, we introduce the use of channels for accomplishing such communications.

A channel will broadcast its messages to all the *ppDATE*s listening to it at the moment of broadcasting. In order to use them, one have to include in the `VARIABLES` section of `GLOBAL` (at top level in the case a foreach construct is used) a declaration of the following form:

```
GLOBAL {
  VARIABLES {
    Channel channelName ;
  }
  ....
}
```

One should use in a transition the action `channelName.send(o)` to send a message (i.e. object `o`) through the channel. In order to receive the massage, the receiver *ppDATE*s should include a trigger similar to the following one:

```
TRIGGERS {
   rec(type obj) = {channelName.receive(obj)entry}
}
```

These triggers are activated by the events generated when an object is send through the channel, i.e. by the action `channelName.send`. Then, in the transition enabled by the occurrence of `rec` one can access to the sent object by referring to `obj` (target object instance of the class `type`).

4.8.5 AspectJ Features

When a trigger is defined, it is possible to use either the *call*, or the *execution* matching modality from aspectJ in its definition. Basically, *call* captures all callers (i.e. the sources of method calls), whereas *execution* captures the calls themselves, no matter where they have originated. In addition, *call* does not intercept super calls to non-static methods. This occurs due to the fact that super calls are different in Java, since they do not behave via dynamic dispatch as other calls to non-static methods would do.

Below, we extend the signature for trigger definitions with this new feature:

```
name(args) = { aspectjMod varDecl.method(args')sysevent }
```

where `aspectjMod := call | execution`

4.8.6 Clocks

Clocks can be used in StaRVOOrS as timers which produce (internal) events once a certain time interval has elapsed. In particular, clocks introduce the possibility of defining real-time properties using *ppDATE*.

In order to use them, one has to, first, declare them in the `VARIABLES` section (see Sec. 4.2.1) of the *ppDATE* specification. For instance, one would declare a clock `c` in the following manner: `Clock c = new Clock();`. In addition, one has to define a special trigger to capture the timeout (event) produced by the clock. Below, we illustrate the syntax for these kind of triggers.

```
name() = { clock@time }
```

Here, `name` is a label which works as an identifier for the trigger; `clock` is the name of the declared clock, e.g. in our previous declaration `c` is the name of the clock; `time` is the amount of seconds after which the clock will produce the timeout captured by this trigger, each time the clock is reseted. For simplicity, one can also define the clock as `clock@%time`, meaning that the clock will automatically reset after producing the timeout.

Note that clocks are contextual, i.e. if one declares a clock within either a `FOREACH` (see Sec. 4.8.3), or a `TEMPLATE` (see Sec. 4.3), a spare clock will be created for each monitored object (or template instance). Moreover, if a clock is declared in the `GLOBAL` section (see Sec. 4.2), then only one clock is created, and it will be available in all the inner contexts of the specification.

In addition, clocks are automatically started as soon as their contexts start existing. For instance, whenever the monitor starts working, all of the clocks declared in the `VARIABLES` section, within the context of the `GLOBAL` section, are started.

Clocks implementation

Clocks are implemented as Java objects to simplify their usage. Below, we provide the signature of the different methods which are part of the *Clock* class, and we briefly describe what they do. All these methods can be used in the transitions of the *ppDATE*s, e.g. `c.reset()` can be used to reset clock `c`.

- `void reset()`: as its name indicates, resets the clock, i.e. the clock is restarted;
- `double current()`: returns the number of seconds which have elapsed since the clock was started (or reseted);
- `int compareTo(double seconds)`: compares `seconds` to the current value of the clock, i.e. the value returned by method `current`. If they are the same value, then this method returns zero. If the latter is bigger than the former, then this returns a positive integer. Otherwise, this method returns a negative integer;
- `void off()`: Switches off the clock;
- `void on()`: Turns (back) on the clock;
- `void pause()`: Pauses the clock;
- `void resume()`: Resumes the clock.

Comparison to Timed Automata

Properties described using timed automata can also be described by using a *ppDATE* with clocks. For instance, let us assume we have two clocks, $\mathtt{c}_1$ and $\mathtt{c}_2$, such that $\mathtt{c}_1$ triggers after 42 seconds, and $\mathtt{c}_2$ triggers after 10 seconds. In addition, the trigger of one clock causes the reset of the other one. Fig. 6 illustrates this example as a timed automaton. This automaton can also be written as the following *ppDATE* (in the StaRVOOrS input language):

```
GLOBAL {
  VARIABLES {
    Clock c1 = new Clock();
    Clock c2 = new Clock();
  }
  TRIGGERS {
    c1Trigger() = { c1@42 }
    c2Trigger() = { c2@10 }
  }
  PROPERTY example {
```

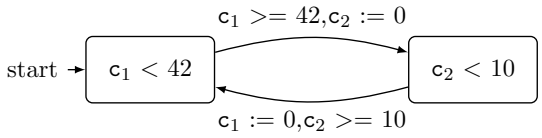

Figure 6: A timed automaton example.

```
    STATES {
       STARTING { start }
       NORMAL { normal }
    }
    TRANSITIONS {
       start -> normal [ c1Trigger \ c1.compareTo(42) >= 0 \ c2.reset();]
       normal -> start [ c2Trigger \ c2.compareTo(10) >= 0 \ c1.reset();]
    }
  }
}
```

Note that this example is an adaptation from an example provided in [**?**].

4.8.7 Towards the Semantics of the Extra Features

As mentioned above, all of the extra features depicted in this section are not covered by the *ppDATE* semantics described in [6]. Here, we do not provide any formal treatment for the semantics of these features. Still, we foresee that their use is safe, i.e., the soundness of the specifications is preserved. Below, we provide some arguments regarding why the soundness of *ppDATE* semantics would not be alter when these extra features are used.

Regarding the `PINIT` definition, as it is simply syntactic sugar to simplify the writing of some particular *ppDATE*s, its semantics would be the same as the one for the *ppDATE*s written in the ordinary manner, i.e., it preserves soundness.

Regarding the `where clause`, it could only break the soundness of a specification if a variable is bound to a particular value returned by a method, and this method throws an exception. However, in [6] programs are assumed to be side-effect free, i.e., they are restricted to not have any effect on the system which could in turn be observed by the monitor. Thus, the use of this clause preserves soundness.

Regarding the `FOREACH` construct, as it is intended to be a simplistic alternative to the use of *ppDATE* templates, their semantics could be described in a similar manner to the one described in [6] for the templates. Thus, we can argue that the use of this construct preserves soundness.

Regarding the use of `channels` and `clocks`, in order to guarantee that their use preserves the soundness of *ppDATE* semantics, the semantics described in [6] should be

extended to handle these features properly. In order to do so, we can use as a base the semantics described in [7] for *DATE*. *DATE* provides the use of `channels` and `clocks` as well. In fact, we have implemented these features in the StaRVOOrS input language by following their implementation in *DATE*. Thus, as these features are considered to be sound in *DATE*, by considering similar semantics for them as depicted in [7] for *ppDATE*, we can argue that the use of these features preserve soundness.

5 Using StaRVOOrS

In this section we depict how StaRVOOrS works by running the tool on the coffee machine example introduced in Sec. 2. Both the *ppDATE* specification written in the input language of the tool, and a simplistic implementation of the coffee machine system, together with two big case studies based on Mondex [1] and SoftSlate (a real Java cart application) [2], can be found in [3], under the section *Downloads*.

5.1 Coffee Machine Specification and Implementation

This section illustrates both the *ppDATE* specification written in the input language of StaRVOOrS, and a (simplistic) implementation for the coffee machine system, which were informally introduced in Sec. 2.

5.1.1 *ppDATE* Specification for the Coffee Machine

```
IMPORTS { import main.CMachine; }

GLOBAL {
 EVENTS {
   brew_entry() = {CMachine cm.brew()}
   brew_exit() = {CMachine cm.brew()uponReturning()}
   cleanF_entry() = {CMachine cm.cleanF()}
 }
 PROPERTY prop {
  STATES {
    BAD { bad ; }
    NORMAL { q2 (brew_error,clean_filter_error) ;}
    STARTING { q (brew_ok,clean_filter_ok) ; }
  }
  TRANSITIONS {
    q -> q2 [brew_entry \ cm.cups < cm.limit ]
    q2 -> q [brew_exit ]
    q2 -> bad [ brew_entry ]
    q2 -> bad [ cleanF_entry ] }
```

```
 }
}

HTRIPLES {
 HT brew_ok {
   PRE {cups < limit}
   METHOD {CMachine.brew}
   POST {cups == \old(cups)+1}
   ASSIGNABLE {cups} }
 HT brew_error {
   PRE {cups < limit}
   METHOD {CMachine.brew}
   POST {cups == \old(cups)}
   ASSIGNABLE {cups} }
 HT clean_filter_ok {
   PRE {true}
   METHOD {CMachine.cleanF}
   POST {cups == 0}
   ASSIGNABLE {cups} }
 HT clean_filter_error {
   PRE {true}
   METHOD {CMachine.cleanF}
   POST {cups == \old(cups)}
   ASSIGNABLE {cups} }
}
```

5.1.2 Coffee Machine Implementation

```
public class CMachine {
   public int cups;
   public int limit;
   public boolean active;

   CMachine(int limit) {
        this.limit = limit;
        cups = 0;
        active = false;
   }

   public void cleanF() {
       if (!active)
          cups = 0;
   }
```

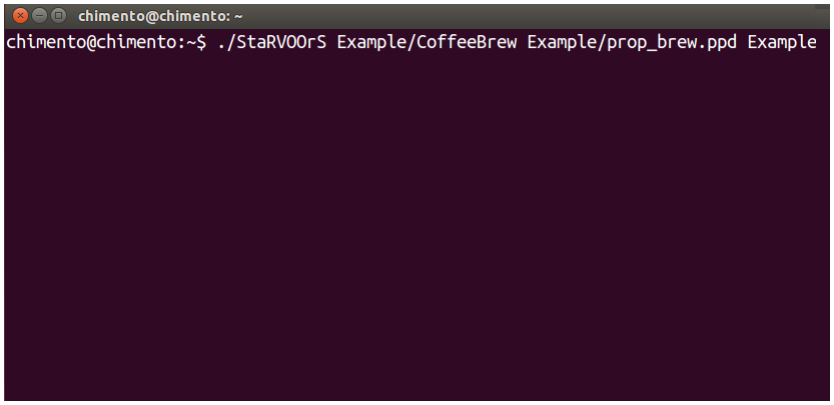

Figure 7: Running StaRVOOrS

```
    public void brew() {
        if (!active && cups < limit)
            cups++;
    }
}
```

5.2 Running StaRVOOrS

In order to run StaRVOOrS, as it is illustrated in Fig. 7, the following input should be provided:

(i) the path to the main directory of the Java files to be verified, for instance, `Example/CoffeeBrew`.

(ii) a description of the *ppDATE* specification for the provided program written in the input language of the tool, for instance, `Example/prop_brew.ppd`.

(iii) the path of the output directory where the files generated by the tool are going to be placed, for instance, `Example`.

5.2.1 Flags

When running StaRVOOrS, one may include flags to indicate different options. Most of these flags are use as follows:

```
StaRVOOrS [-OPTIONS] <java_source_files> <ppDATE_file> <output_add>
```

Below, we list them.

- -n (or --none_verbose)
  None verbose monitor generation.
- -x (or --xml)
  The .xml file generated by KeY is not removed.
- -r (or --only_rv)
  Monitor generation without performing deductive verification with KeY.
- -k (or --killbad)
  All the ppDATEs which have reached a bad state are killed, i.e. terminated.
- -d (or --distributed)
  Improves the output provided by the monitor in the context of active objects.[1]
- -v (or --version)
  Shows StaRVOOrS version number. Usage: `StaRVOOrS -v`
- -p (or --only_parse)
  Only parse the ppDATE file. Usage: `StaRVOOrS -p <ppDATE_file>`
- -h (or --help)
  Describes the different flags available. Usage: `StaRVOOrS -h`

Note that the flags -p, -h, and -v, have to be used in isolation. All of the other flags can be combined.

5.3 StaRVOOrS ouput

Fig.8 illustrates all the files generated by StaRVOOrS when it is used to analise the running 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 `ppArtifacts`), an instrumented version of the source code (folder `CoffeMachine`), a report summarising the results obtained during the static verification of the Hoare triples (*report.txt*), the optimised version (if any) of the provided *ppDATE* specification (*prop_brew_optimised.ppd*), and the *DATE* specification obtained as a result of translating the (optimised) *ppDATE* (*prop_brew.lrv*). Note that StaRVOOrS does not modify the provided source code, it creates an instrumented version of it. Thus, at the time of monitoring the code, the instrumented version of the source code is the one which should be used.

5.4 StaRVOOrS execution insights

StaRVOOrS is a fully automated tool. However, in order to have a better understanding of its execution, below we will explain it in three stages. Note that during each one of this stages,

[1]This feature was only tested in ProActive.

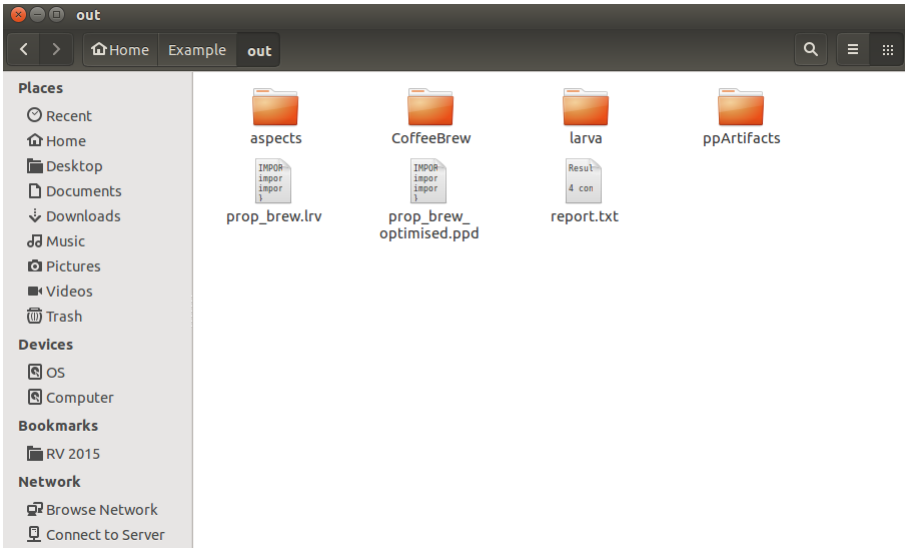

Figure 8: StaRVOOrS output

StaRVOOrS will produced some output on the terminal. We will illustrate such an output through figures.

The first stage corresponds to the static verification of the Hoare triples using KeY. Fig. 9 shows the output produced by the tool on the terminal during this stage. At first, KeY (taclet) options are set. These options are parameters which, for instance, indicate to KeY which rules of its sequent calculus it is able to use during the verification of a property. For the time being, we are just using the standard options. Then, KeY is ran.

While KeY analyses all the Hoare triples, every time a proof attempt is saturated, some information related to this analysis is given as output in the terminal. Fig. 10 illustrates this. Once KeY is done verifying all the Hoare triples, it generates a temporary file *out.xml*, which is removed once StaRVOOrS is done[2], describing its results. This file 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 corresponds to the refinement of the specification. In this stage, all the Hoare triples which were fully proven are removed from the *ppDATE*, and those which were only partially proven are modified by strengthening their pre-conditions including the conditions which lead to an unclosed path on a proof. For instance, in our running example the report would include the information that the pre-condition of the Hoare triple `brew_ok` is strengthen with the addition of the condition `active == TRUE`.

Whenever it is necessary at runtime to verify partially proven Hoare triples, StaRVOOrS

[2]Note that one may use the flag -x to include this file as part of the output.

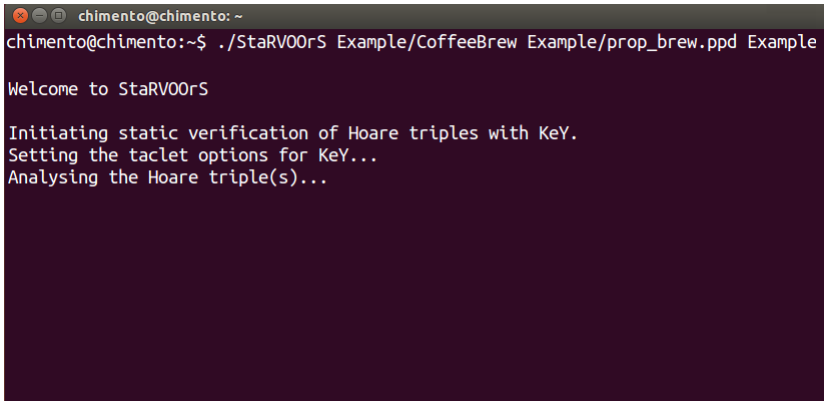

Figure 9: Initiating Static Verification

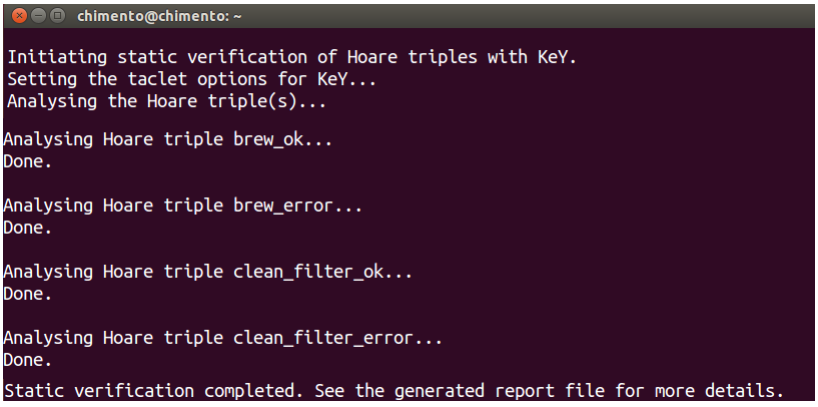

Figure 10: Output shown on the terminal during static verification

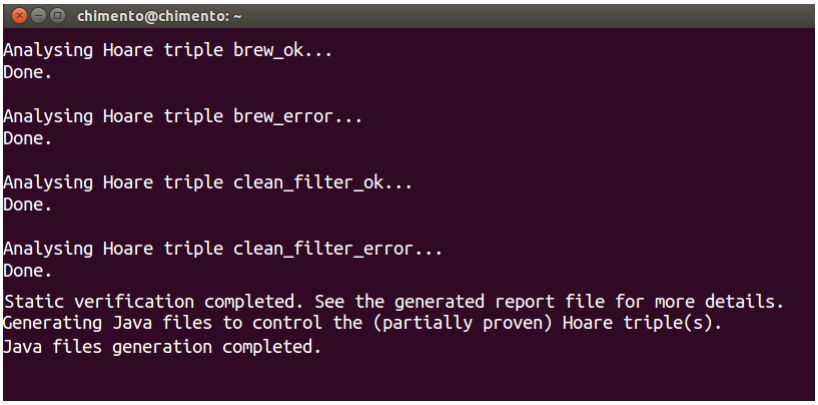

Figure 11: Optimization and files generation after static verification

instruments the provided Java files by adding a new parameter to the method(s) associated to the Hoare triple(s). This new parameter is used to distinguish different calls to the same method. This change is introduced in the refined *ppDATE* specification as well. Besides, StaRVOOrS generates several files within folder the `ppArtifacts` which are used to runtime verify the Hoare triples. For instance, the file *HoareTriplesPPD.java* contains the implementation of the methods which are used to verify the pre- and post-conditions of the Hoare triples. In addition, file *IdPPD.java* will be used to generate the value of the new parameter added to the methods. Once this stage is over, the terminal will look like Fig. 11.

The third stage corresponds to the generation of the runtime monitor. In order to do so, the refined *ppDATE* specification is translated by StaRVOOrS to a *DATE* specification (file *prop_brew.lrv* in our running example). Then, Larva is used to generate the monitor files from the *DATE*. After the execution of Larva is completed, leading to the generation of the files in the folders `aspects` and `larva`, StaRVOOrS execution is completed as well. The terminal will reflect this, as it is illustrated in Fig. 12.

5.5 Running the application with the generated monitor

To run the (generated) instrumented version of the program, let us call it P, together with the monitor, one can generate an executable jar file, and then run it on a Java virtual machine. We will use Java 1.7 to compile the Java files. However, due to compatibility issues with Larva, when compiling the aspects one has to use the version 1.5. Note that the aspects have to be compiled using an AspectJ compiler. We recommend the *ajc* compiler. In addition, to run the jar file one has to use *aj5* (like command *java*, but with support for AspectJ), or similar. Below, we provide a short script explaining how to create such a jar file.

First, go inside the output directory.

cd Example/out

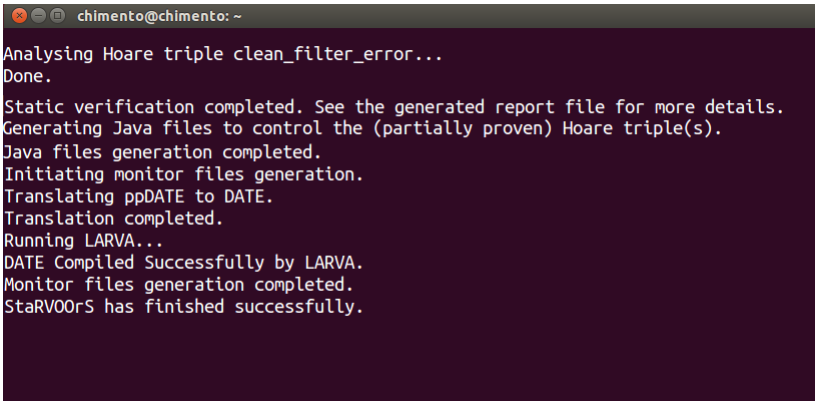

Figure 12: Monitor Generation

Second, copy the folders `larva` and `ppArtifacts` into the main folder of P.

cp -r larva CoffeeBrew
cp -r ppArtifacts CoffeeBrew

Third, create a directory named `Build`, and compile P using the option *-target 1.7* in such a way that the compiled files are placed within `Build`.

mkdir Build
*javac -target 1.7 $(find CoffeeBrew -name *.java) -d Build*

Next, create an executable jar file from the files in `Build`.

jar cfe coffeeM.jar main.CMachine -C Build .

Now, one has to weave the aspects into the jar file. In order to do so, the files in folder aspects have to be compiled using an AspectJ compiler. Here, we use *ajc*. Note that it is usually recommend to generate a new jar file when the aspects are weaved.

ajc -1.5 -sourceroots aspects/ -inpath coffeeM.jar -outjar coffeeM_asp.jar

Finally, this weaved executable jar file corresponds to the compilation of P together with the monitor. Therefore, running it would mean the one is running a monitored version of P. To execute the weaved jar file one can use *aj5* as follows:

aj5 -jar coffeeM_asp.jar

References

[1] MasterCard International Inc. Mondex. `www.mondexusa.com/`.

[2] SoftSlate Commerce. `www.softslate.com/`.

[3] StaRVOOrS web page. `cse-212294.cse.chalmers.se/starvoors`.

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016. to appear.

[5] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A Specification Language for Static and Runtime Verification of Data and Control Properties. In *FM'15*, volume 9109 of *LNCS*. Springer, 2015.

[6] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, pages 1–66, 2017.

[7] Christian Colombo, Gordon J. Pace, and Gerardo 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.

[8] 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.

[9] David Harel, Dexter C. Kozen, and Jerzy Tiuryn. *Dynamic logic*. Foundations of computing. the MIT Press, Cambridge (Mass.), London, 2000.

[10] 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.