StaRVOOrS User Manual

16 Abril, 2015

Jesús Mauricio Chimento

Contents

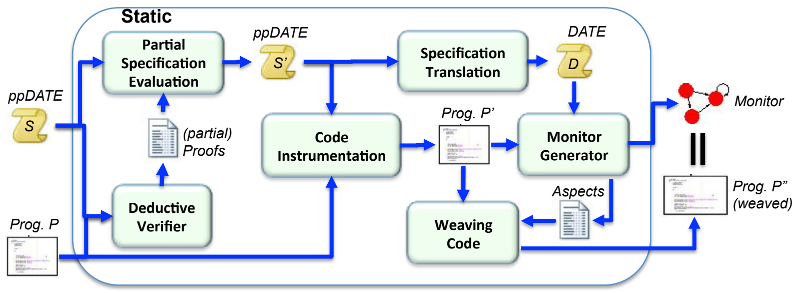

Figure 1: High-level description of the StaRVOOrS framework workflow

1 Introduction

Runtime verification and static verification are widely used verification techniques. *Runtime verification* is concerned with the monitoring of software, providing guarantees that observed executions of a program comply with specified properties. This approach can be used on systems of a complexity that is difficult to address by *static verification* such as systems with numerous interacting sub-units, heavy usage of mainstream libraries, and real world deployments. On the other hand, with runtime verification it is not possible to extrapolate about all possible execution paths. Furthermore, monitoring incurs runtime overheads which may be prohibitive in certain systems.

StaRVOOrS addresses these issues by combining runtime verification with static verification. This tool starts by statically verifying the system against a specification written in *ppDATE* [1], identifying parts which can either be verified automatically or partially resolved, thus leaving a simpler specification to check at runtime, in turn reducing the overheads induced by monitoring.

2 The StaRVOOrS Framework

The StaRVOOrS framework (STAtic and Runtime Verification of Object-ORiented Software) combines the use of the deductive source code verifier KeY [2] with that of the runtime monitoring tool Larva [3]. KeY is a deductive verification system for data-centric *functional correctness* properties of Java source code, which generates, from JML and Java, proof obligations in *dynamic logic* (a modal logic for reasoning about programs) and attempts to prove them. Larva (*Logical Automata for Runtime Verification and Analysis*) [3] is an automata-based Runtime Verification tool for Java programs which automatically generates a runtime monitor from a property using an automaton-based specification notation *DATE*. Larva transforms the specification into monitoring code together with AspectJ code to link the system with the monitors.

Fig. 1 gives an abstract view of the framework workflow. Given a Java program P and a specification S of the properties to be verified, given in the language *ppDATE* [1], these are transformed into suitable input for the *Deductive Verifier* module which, in principle, might statically fully verify the properties related to pre/post-conditions. What is not proved statically will then be left to be checked at runtime. Here, not only the completed but also the *partial* proofs are used to generate path conditions for not statically verified executions. The *Partial Specification Evaluator* module then rewrites the original specification S into S', refining the original pre-conditions with the aforementioned path conditions. Note that S' is no longer a full specification of the desired behaviour. Instead, it only specifies executions that are not covered by the static verification.

In a next step, the resulting *ppDATE* specification S' is, via *Specification Translation*, turned into a specification in *DATE* format (D), suitable for the runtime verifier. As *DATE* has no native support for pre/post-conditions, these are simulated by pure *DATE* concepts. This also requires changes to the code base (done by the *Code Instrumentation* module), like adding counters to distinguish different executions of the same code unit, or adding methods which operationalise pre/post-condition evaluation. The instrumented program P' and the *DATE* specification D are given to the *Monitor Generator*, which uses aspect-oriented programming techniques to capture relevant system events. Later on, the generated aspects are weaved (*Weaving Code*) into P'. The final step in the workflow is the actual runtime verification, which executes the weaved program P'' — running the original program in parallel with a monitor of the simplified property. In case of a runtime error, a trace is produced to be analysed.

3 Composing a *ppDATE* specification

In this section we will first introduce the *ppDATE* specification language and then we will go through a *ppDATE* specification file explaining the meaning of each one of its sections together with examples of syntax details.

3.1 *ppDATE* specification language

Formalisms for specifying software generally fall into two categories — either *data-oriented* approaches (like first-order logic) are strong in describing data-related properties but only at specific points of the execution, while *control-oriented* property languages (such as LTL) specialise more in specifying legal control paths. StaRVOOrS uses the automata-based specification language *ppDATE*s as its property input language, which enables the combination of data- and control-based properties in a single formalism [1]. At its core, *ppDATE*s are a composition of the control-flow language *DATE* and more

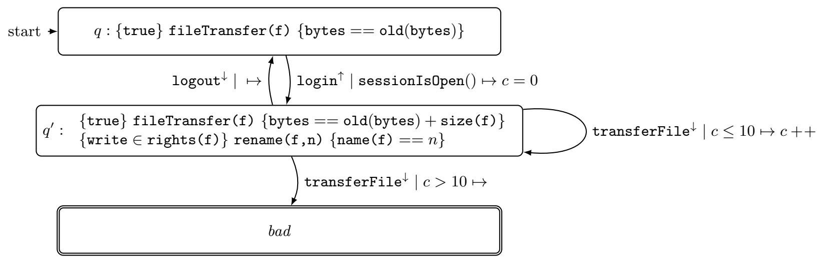
start
q : {true} fileTransfer(f) {bytes == old(bytes)}
logout$^{\downarrow}$ | $\mapsto$
login$^{\uparrow}$ | sessionIsOpen() $\mapsto c = 0$
q' : {true} fileTransfer(f) {bytes == old(bytes) + size(f)}
{write $\in$ rights(f)} rename(f,n) {name(f) == n}
transferFile$^{\downarrow}$ | $c \leq 10 \mapsto c{+}{+}$
transferFile$^{\downarrow}$ | $c > 10 \mapsto$
bad

Figure 2: A *ppDATE* limiting file transfers

data-oriented specifications in the form of Hoare triples with *p*re-/*p*ost-conditions, hence its name.

A *ppDATE* property is based on an alphabet of events which can be detected when the system performs them. The events are typically entry and exit points of methods and (named) blocks of code, and exception raising. We write $\mathtt{f}^{\downarrow}$ and $\mathtt{f}^{\uparrow}$ respectively to denote the moment of entering and exiting function or code block with name `f`. These detectable events, essentially points of interest along the control paths followed by the system, are used to drive the control-flow side of the specification.

Consider the *ppDATE* shown in Fig. 2. If we ignore the information given in the states, the property is depicted as an automaton, more specifically in the form of a *DATE*, in which transitions are tagged with triples: $e \mid c \mapsto a$ — indicating that (i) they are triggered when event e occurs and condition c holds; (ii) and apart from changing the state of the property, action a is executed. For instance, the reflexive transition on the middle state is tagged: $\mathtt{transferFile}^{\downarrow} \mid c \leq 10 \mapsto c{+}{+}$, indicating 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. 2 can thus be understood to ensure that no more than 10 file transfers take place in a single login session.

Ignoring, for the moment, the information stored in the states and block-related events, this formalism is identical to *DATE*s. Both formalisms support other features, including: (i) timers which may be used in the transition conditions or as events to trigger transitions; (ii) communication between automata using standard CCS-like channels with $x!$ acting as a broadcast on channel x and which can be read by another automaton matching on event $x?$; and (iii) replication of automata through which every time a particular event distinct from earlier ones (e.g. using a method's parameters or the target object) is received a new automaton is created (e.g. used to replicate a property for each instance of a class). To illustrate the use of the last feature, we note that in the case of a multi-user system, the property depicted in Fig. 2 does not work as expected, since logins and logouts of different users would interact with the property in undesirable ways. To resolve this issue, we would enclose the property within a replication clause to ensure that an instance of the property is replicated for each different user. By specifying which instance of the property is used when an event is identified (e.g. one can identify the user related to a login by identifying the target of the method call), the automata are created by need and effectively execute in parallel.

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 $\{\pi\}$ `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, then upon reaching the corresponding exit from `f`, postcondition π' should hold. Pre-/post-conditions in Hoare triples are expressed using JML boolean expression syntax [4], which is designed to be easily usable by Java programmers.

For instance, the Hoare triple appearing in the top state of the property given in Fig. 2, ensures that any attempted file transfer when in the top state (when logged out), should not change the byte-transfer count. Similarly, while logged in (in the middle state of the property) (i) the number of bytes transferred increases when a file transfer is done while logged in; and (ii) renaming a file does indeed change the filename as expected if the user has the sufficient rights.

To ensure efficient execution of monitors, *ppDATE*s are assumed to be deterministic by giving an ordering in which transitions are executed. A complete formalisation of *ppDATE*s can be found in [1].

3.2 Composing the script of a *ppDATE* file

A *ppDATE* specification is described on a file with extention *.ppd*. This file consists in 5 sections (ordered as they are listed): *Imports*, *Global*, *CInvariants*, *Contracts* and *Methods*.

3.2.1 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. Each file name should be preceded by the word *import*.

The syntax for this section is as follows:

```
IMPORTS { import main.HashTable; }
```

3.2.2 Global

Describes the automaton (i.e. events, automata variables, transitions, states, etc). This section is an extension of the *Global* section of LARVA. In the user manual of LARVA, which can be accessed in `www.cs.um.edu.mt/svrg/Tools/LARVA/LARVA-manual.pdf`, you can find a detail explanation for this section. The only differences with *ppDATE Global* section is that in *ppDATE* the definition of the states include a list of contracts to verify in each state. This list appears right next to the name of the state. Furthermore, for the time being, we do not support the use of *Initial Code* on the states definition.

The syntax for this section is as follows:

```
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()}
}

PROPERTY add {

STATES
{
NORMAL { q2 ; }
STARTING { q (add_ok, add_full,hashfun_ok) ; }
}

TRANSITIONS {
q -> q2 [add_entry\hasht.contains(u, key) < 0\]
}
}
}
```

3.2.3 CInvariants

Class invariants are described in this section by `class_name {invariant}`.

The syntax for this section is as follows:

```
CINVARIANTS {
 HashTable {h.length == capacity}
 HashTable {capacity >= 1}
}
```

It is not mandatory to include this section in the script, i.e. no class invariants are needed for the static verification of the Hoare triples.

3.2.4 Contracts

Lists named Hoare triples. Each Hoare triple is described in a subsection *CONTRACT*, whose header is follow by the name assigned to the contract. On it, *PRE* describes the precondition of the Hoare triple, *POST* describes the postcondition of the Hoare triple, *METHOD* describes which is the moethod associated to the Hoare triple and *ASSIGNABLE* list the variables that might be modified when the method associated to the Hoare triple is executed. Both precondition and postcondition predicates follow JML-like syntax and pragmatics.

The syntax for this section is as follows:

```
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[*]}
 }
}
```

It is not mandatory to include this section in the script, i.e. it is not desire to perform static verification.

3.2.5 Methods

Definition of methods to avoid having a lot of code on the transitions of the automaton. 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.

The syntax for this section is as follows:

```
METHODS {
  int foo() {return 0;}
  boolean goo(int v) {return true;}
}
```

4 Using the StaRVOOrS tool

In this section we will show how to use StaRVOOrS by applying the tool to a working example.

4.1 Working Example: Login Scenario

We consider a scenario where users attempt to log in into a system. On this 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.

The *ppDATE* specification, which will only specify a small part of the system's behaviour, will be focused on the method `add`. Whenever this method is executed, on one transition of the automaton it will be verified that the user to be added is not already logged in the system. Regarding the pre/post-condition of the method `add`, it will be analysed two contracts: *add_ok* specifies that, if there is room for an object u in the hash table, then after adding that object into the hash table it is found in one of the entries of the array h ; *add_full* specifies that, if there is no room for an object in the hash table, then the hash table should not be modified when an attempt to add a new user is performed. Besides, it is necessary to include a Hoare triple related to the hash function: *hashfun_ok* specifies that the hash function should receive a positive value.

Note that both the `HashTable` class file and the *ppDATE* specification are provided in Sec.5.

4.2 Running StaRVOOrS

In order to run StaRVOOrS, the tool should be provided with 3 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.

e.g. StaRVOOrS Example/Login Example/prop_add.ppd Example

4.3 StaRVOOrS ouput

The output produced by StaRVOOrS consists on: the monitor files generated by Larva (folder *aspects* and folder *larva*), the files generated by StaRVOOrS to runtime verify partially proven contracts (folder *ppArtifacts*), an instrumented version of the source code (folder *Login*), the xml file used by StaRVOOrS to optimized the *ppDATE* specification (*out.xml*), a report explaning the content of the .xml file (*report.txt*) and the *DATE* specification obtained as a result of translating the (optimized) *ppDATE*.

4.4 StaRVOOrS execution insights

StaRVOOrS is a fully automated tool. However, in order to have a better understanding on what is going during its execution, below we will explain it in three stages. Note that during each one of this stages StaRVOOrS will produced some text output on the terminal which inform the user on which stage the tool is working at.

The first stage corresponds to the static verification of the Hoare triples using KeY. During this stage, first the KeY (taclet) options are set. This options are parameters that tell KeY how it should proceed during the verification process. For the time being, we are just using the standard options. Then, KeY is ran.

While KeY analyses all the contracts (i.e. the Hoare triples), every time KeY is done with one of them some information related to this analysis is given as output in the terminal. 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 optimized the *ppDATE* specification. However, in order to provide the user with 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 *ppDATE* specification of our working example, KeY proves the contracts *add_full* and *hashfun_ok*, but it only partially proves the contract *add_ok*. As it can be read on the report, this contract will be strengthen by adding to its precondition the predicate `!(h[hash_function(key)]== null)`

Whenever it is necessary to runtime verify partially proven contracts, StaRVOOrS instruments the source code by adding a new parameter to the method(s) associated to the contract(s). This new parameter is used to distinguish different method calls. This change is introduced in the *ppDATE* specification too. Besides, StaRVOOrS generates two files (both within a folder name *partialInfo*): *Contracts.java* and *Id.java*. The former contains methods which are generated from the pre-postconditions of the contracts, which will be use by the monitor when verifying the respective contract. The latter will be used to deal with new parameter added to the methods.

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*. Once the execution of Larva is completed, StaRVOOrS execution is completed too.

4.5 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*).

5 Example Files

- *ppDATE* script (*prop_add*):

```
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}
}
}
```

- HashTable.java file:

```
public class HashTable {

// Open addressing Hashtable with linear probing as collision resolution.

public Object[] h;

public int size;

public int capacity;

HashTable (int capacity) {
   h = new Object[capacity];
   this.capacity = capacity;
   size = 0;
}

public int hash_function (int val) {
   return (val % capacity);
}

// Add an element to the hashtable.
public void add (Object u, int key) {

    if (size < capacity) {

    int i = hash_function(key);

        if (h[i] == null) {
            h[i] = u;
                size++;
                return;
                }
        else {
            int j = 0;

            while (h[i] != null && j < capacity)
            {
              if (i == capacity-1) i = 0;
              else {i++;}

              j++;
            }

            h[i] = u;
            size++;
            return;
           }

    } else {
           return;
           }
}

// Removes an entry from the hashtable.
public void delete (Object u, int key) {
   int i = hash_function(key);
   int j = 0;

   while (!u.equals(h[i]) && (j < capacity))
   {
   if (i == capacity-1) i = 0;
   else {i++;}

   j++;
   }

   if (u.equals(h[i])){
      size = size - 1;
      h[i] = null;
   }
}

// check if an intry is in hashtable.
//   If it is, then returns the position in the hashtable where it is.
//   If it isn't, returns -1.
public int contains (Object u, int key) {
   int i = hash_function(key);
```

```
      int j = 0;

      if (u == null) return -1;

      while (!u.equals(h[i]) && (j < capacity))
      {
      if (i == capacity-1) i = 0;
      else {i++;}

      j++;
      }

      if (u.equals(h[i]))
         return i;
      else {return -1;}
   }

   // access to the entry of hashtable in the position idx.
   public Object foo (int idx) {
      return h[idx];
   }
   }
```

References

[1] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109 of *LNCS*, pages 108–125. Springer-Verlag, 2015.

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

[3] 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.

[4] 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.