Combining Monitoring with Run-time Assertion Checking

Gouw, C.P.T. de

Author: Gouw, Stijn de
Title: Combining monitoring with run-time assertion checking
Issue Date: 2013-12-18

Related- and Future Work 7

In this chapter we present a direct comparison of our own framework and corresponding tool support with PQL [64], Jassda [16], LARVA [26] and MOP [20]. We model the properties of the Fredhopper case study described in the previous section in these respective tools. We consider the **learnability** of the frameworks: that is, does the framework provide a specification language with a surface syntax close to existing formal/modeling languages? Is the semantics of the specifications properly documented? We test how easily the frameworks can be **adopted** or integrated into the the software development cycle in an industrial context such as at Fredhopper. This includes operational steps like installation, execution, and documentation and support.

We now provide a brief overview of these three frameworks.

PQL Program Query Language, PQL, is developed by Martin et al. [64], it is a query language for pattern matching (possibly recursive) sequences of method invocations of Java programs. Unlike all the other approaches in the evaluation, PQL queries express invalid behavior rather than all possible valid behavior. Figure 7.1 shows a PQL query expressing part of the `Worker` property. Specifically, it matches invalid behavior of invoking `reg` after invoking `establish` with the input argument "LIST".

Jassda Java with assertions Debugger Architecture, Jassda, is a framework developed by Brörkens et al. [16]. Trace assertions are given in a CSP-like notation. The CSP-like notation maps invocations and returns of method calls of Java programs as events. For example, Figure 7.2 shows that the event $w.rg.begin$ is mapped to the invocation of method `Worker.reg` The framework takes such assertions, combined with the Java Debugger Architecture to monitor calls and returns of methods. It then generates a program for a state machine, and attaches to the Java Virtual Machine running the system under test that is accepting debug connections. The framework then monitors invocations and returns of method calls and output log messages to a separate file on state transitions.

LARVA Logical Automata for Runtime Verification and Analysis, LARVA, is developed by Colombo et al. [26]. LARVA provides a modeling language for specifying both valid and invalid method invocations of Java programs by enumerating transitions in a state transition function of an abstract machine. Each transition is conditioned on an event, where an event maps to one or more invocations and returns of method calls. The modeling language also permits declaring global variables of any visible Java types (class/interface), and at each transition an optional condition can be made about values of these variables against input arguments or return values of methods as well as any number of Java statements on these variables and values. Figure 7.3 shows how to partially model `Worker` property in LARVA.

JavaMOP Monitoring-Oriented Programming, MOP [20], is a run-time monitoring tool based on aspect-oriented programming which uses context-free grammars to describe properties of the control flow of histories. Properties on the data-flow are *predefined* built-in functions (basically AspectJ functions such as a 'target' to bind the callee and 'this' to bind the caller, comparable to built-in attributes of terminals in our setting). This limits the expression of data properties: there is no support for defining properties of *sequences* of terminals. To circumvent this limitation one may however hack general properties into the tool implementation. Figure 7.4 formalizes the protocol behavior of the `Worker`.

In contrast, our approach supports a general methodology to introduce systematically *user-defined* properties, by means of attributes of non-terminals. Furthermore SAGA supports conditional productions which are essential to specify protocols dependent on data in a declarative manner. Finally, JavaMOP does not directly support the specification of local histories (i.e. monitoring the messages sent and received by a single object).

Expressiveness

	`Snapshot`	`Coordinator`	`Worker`
PQL	yes	no	no
Jassda	yes	no	no
LARVA	yes	yes	yes
MOP	yes	yes	yes
SAGA	yes	yes	yes

Table 7.1: Comparison of Expressiveness

We investigated the expressiveness of the specification languages of these tools by attempting to express and check the SnapShot, Worker and Coordination properties (see Chapter 5). The resulting specifications are given in Figures 7.1 to 7.4.

Table 7.1 summarizes the results. Neither PQL nor Jassda can express the `Coordinator` and `Worker` properties since neither allows user-defined properties of data. In both `Coordinator` and `Worker` properties, the validity of a method invocation is dependent on the value of the input arguments as well as the return values. For example in PQL snippet in Figure 7.1, to model the invalid sequence of method invocations `establish("LIST")` followed by `reg(_)`, we had to encapsulate the string value `"LIST"` into the method `SyncServer.getList()` as PQL does support object manipu-

	Specification	Execution
PQL	5	2
Jassda	4	2
LARVA	2	1
MOP	5	1
SAGA	3	1

Table 7.2: Duration per Activity in hours

```
query main ()
uses object Worker w; object String s;
matches {
  w = Acceptor.getWorker();
  s = SyncServer.getList();
  w.establish(s); w.reg(s); }
executes Util.printStackTrace(*);
```

Figure 7.1: PQL

```
trace worker {
 eventset w { class="Worker" }
 eventset r { method="reg"}
 ...
 process main() {
  w.ls.begin -> w.ls.end ->
  w.et.begin -> w.et.end -> STOP
  []
  w.os.begin -> w.os.end ->
  w.et.begin -> w.et.end ->
  w.rg.begin -> STOP }}
```

Figure 7.2: JASS

lation. LARVA and MOP, on the other hand, support executing arbitrary Java statements when an event occurs, hence it is possible to define data-oriented properties such as `Coordinator` and `Worker`. As such, user-defined properties of the data of a single event are possible to express. It is not possible to di-

```
IMPORTS{ import java.util.*; }
GLOBAL {
 FOREACH (Worker w) {
  VARIABLES { String c = null; ArrayDeque q = null;}
  EVENTS{
   et(String s, Worker w1) = {
     w1.establish(s);} where {w = w1;}
   is(String s, List is, Worker w1) = {
     w1.reg(s)uponReturning(is);} where {w = w1;}
   tr(Item i, Worker w1) = {
     w1.transfer(i);} where {w = w1;}}
  PROPERTY workers{
   STATES {
    STARTING{ start{} }
    BAD{ regL{} transW{} }
    NORMAL{ est{} regS{} transC{} }}
   TRANSITIONS{
    start -> est [et()\\c = s;]
    est -> regL [is()\"LIST".equals(c)]
    est -> regS [is()\! "LIST".equals(c)\q =
                           new ArrayDeque(is);]
    regS -> transW [tr()\q.pop() != i]
    regS -> transC [tr()\q.pop() == i] }}}}
```

Figure 7.3: LARVA

rectly express properties of *sequences* of events (i.e. the dataflow of the history). In LARVA, non-regular context-free protocols cannot be expressed *directly*: one would have to write the parser for a context-free grammar oneself. The user would then essentially be writing their own run-time checker in Java, bypassing MOP and Larva. This is clearly unfeasible, and the resulting specifications are not declarative anymore. Most importantly, in that degenerative sense of expressiveness, AspectJ (on which MOP and LARVA are based) would already be sufficient.

Moreover, since LARVA supports the manipulation of arbitrary Java objects as global variables, it is as expressive as SAGA. However, unlike SAGA, LARVA requires the specifi-

```
import java.io.*; import java.util.*;
suffix HasNext(Worker w) {
   event et before(Worker w):
      call(* Worker.establish(String)) && target(w) {}
   event rg before(Worker w):
      call(* Worker.reg(String)) && target(w) {}
   event is after(Worker w) returning(List result):
      call(* Worker.reg(String)) && target(w) {}
   event tr before(Worker w):
      call(* Worker.transfer(int)) && target(w) {}
   cfg : S -> epsilon | et U, U -> epsilon | rg V,
         V -> epsilon | is W, W -> epsilon | tr W
   @fail { System.err.println("Protocol violation"); }}
```

Figure 7.4: MOP

cation to be explicit on both valid and invalid method call sequences. For example, the specification in Figure 7.3 would allow `reg()` to be invoked immediately at the start state, as the transition from the start state is not defined, it is simply ignored by the monitoring framework.

Learnability

Learnability is the capability of a software product to enable the user to learn how to use it. Table 7.2 shows the number of hours spent on activities to specify and monitoring properties defined in Figure 6.5.

The most time spent at specification was for PQL; PQL defines a new specification language for expressing queries for (recursively) matching sequences of method invocations. We find the language to be counter-intuitive as it does not match any existing modeling or programming languages. Moreover, it requires the user to specify *invalid* behavior rather than valid ones and it is unclear how to specify method invocations with specific input values. Similarly Jassda lacks an integration into the general context of assertion checking, which is needed to specify properties of variable values.

	Documentation	Maintenance	Support
PQL	1 paper, examples	2006	Minimal
Jassda	papers, (German) thesis, examples	2006	Minimal
LARVA	papers, manuals, examples	2011	Immediate
MOP	papers, manuals, examples	2011	Immediate
SAGA	papers, examples	2012	Immediate

Table 7.3: Adoptability

LARVA provides an intuitive language for specifying regular protocols. Specifications are finite state automata with optionally actions (arbitrary Java code) on the transitions of the automaton. Actions can be used to express data-oriented properties, though in an imperative style. Context-free protocols are however much more cumbersome to express as noted previously. Despite the fact that the Worker property has only been formalized partially in LARVA due to requirements to express *all invalid* sequences of method invocations, the full specification in SAGA is much more concise.

Though it is no so difficult in MOP to formalize the protocol behavior of the Worker, Figure 7.4 (data-oriented properties are more problematic, as these cannot be expressed directly as mentioned), the meaning of the grammars in MOP is unclear: the failure handler was triggered by MOP even for correct programs. Whether this is due to misunderstanding on our part of the meaning of MOP specifications, or due to a bug in MOP remains unclear even after a thorough reading of the documentation.

For PQL, most time is spent identifying which Java statements are supported and how variables can be manipulated. The actual set-up of the run-time checking (compilation, instrumentation etc.) are carried by mirroring the setting in the toy examples provided by the installation package. For Jassda, time is spent at understanding the Java Debugger Architecture, and in particular the proper settings in the configuration files.

We evaluated how easily the frameworks can be **adopted** or integrated into the the software development cycle in an industrial context such as at Fredhopper. This includes operational steps like installation, execution, and documentation and support. The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. This type of testing requires a running FAS instance and can be augmented with run-time assertion checking techniques. Lack of support and maintenance (Table 7.3) reduces the confidence in PQL and Jassda.

Future Work

The concurrent version of our run-time checker which is described in Chapter 6 currently supports only regular grammars and all attributes must be inherited. As described, this restriction allows efficient run-time checking since one does not need to store the full history (just storing the attribute values of the 'previous' history suffices), and one does not need to re-parse the full history when a message is added to it (instead, one simply executes a single step in a finite automaton). The single threaded version allows more general grammars, but at the cost of a more expensive parsing process (the entire history is stored, and completely re-parsed whenever an event occurs). This suggests a possible direction of future work: investigate if and how more general grammars can be parsed incrementally, and implement efficient parsers for that class of grammars. Some initial theoretical work has already been done in this direction by Hedin [43].

Another direction of future work concerns error reporting (by error, we mean here that a history has been reached during execution which violated a specification as given by the grammars). If the run-time checker detects an error, what information is reported, and how is it presented? Clearly it is cumbersome to read through long stacktraces and low-level details of Java virtual machines that one gets by simply executing the program inside a debugger. On the other hand, the reported error should be sufficient to isolate the incorrect part of the source-code if it is to be of use for fixing the error. As a first step, the current version of the run-time checker outputs a UML sequence diagram depicting an invalid history upon detection, but much work remains to be done. For instance, it is clearly infeasable to visualize large histories (or even any history containing more than 10.000 messages). Thus suitable abstractions must be found. In particular, the question arises: which of the messages in the history are actually relevant for the error that was found? Once this is known, the other messages can simply be filtered away.

A third opportunity for future work concerns off-line monitoring. The current run-time checker parses the current history during execution of the program in real-time and stops (or possibly corrects) the original program once a violation of the specification is caught. Therefore the run-time checker induces an overhead during execution. This can partly be alleviated on multicore machines by running the run-time checker in a separate virtual machine (this is done by default if one uses the Java debugger, as explained in Chapter 4), and running that virtual machine on another processor. However there will still be some communication to report the next method call to be executed between the two virtual machines, and communication between two processes generally decreases performance. An alternative would be to develop a run-time checker which writes the history to disk. This allows to investigate any potential errors at another time, potentially even on a physically completely separate machine! One possible downside of this alternative is that since errors are only detected at a later time, the running system is not prevented from unsafe behavior. To keep the sizes of the stored histories manageable, it is in this regard clearly also important to find efficient representations or abstractions of the history.

While our formalism was based on context-free grammars (extended with attributes and assertions), there are more expressive grammar formalisms, such as Boolean grammars [71] or context-sensitive grammars (see for example [63]). These formalisms still have a decidable parsing problem. Future work in this direction can be done by investigating if (and how) these formalisms can be extended by some form of attributes, similarly to how attribute grammars are an extension of context-free grammars.

A perhaps simpler line of future work would be to extend the tool, for example with wildcards in communication views, or associating some form of time to each communication event. Wildcards are useful for specifying patterns (or sets) of method calls. For instance, if a class contains multiple overloaded methods, and one wants identify all of them in the attribute grammar, using a wildcard as the list of parameters would be a simple solution which avoids explicitly listing all variants of the overloaded method in the comunication view. As another feature, one could store the time at which a method call occured as an additional built-in attribute in the grammar terminals. This additional attribute could be used to specify non-functional properties such as resource requirements. For instance, it allows to express properties like 'the method m should not be called within 1 second after n was called'. In this respect it would be interesting to compare the resulting attribute grammars with existing temporal logics.