Runtime Verification

Lecture 7 : Temporal Logic in JavaMOP

http://www.runtime-verification.org/course

May 22, 2008

This seventh lecture introduces temporal logic and how it is supported by the JavaMOP system. Temporal logic has long been regarded as an important kind of specification language for specification and verification of computer systems, especially (but not limited to) concurrent systems. Temporal logic is for example often used when stating properties to be verified by model checkers. Here models are infinite traces. We shall see how temporal logic, past time as well as future time, can be used for monitoring finite traces corresponding to program executions.

At the end of these notes follow a collection of hints that are useful to know when using JavaMOP.

Installation

JavaMOP can be accessed from the website. There is no need to intall anything. On the website you can enter a specification, and with a press of a button generate an aspect, which you then cut-and-paste into your own environment to run with your own application. The slides will demonstrate this.

Reading

This week we will read the paper: *Efficient Monitoring of Safety Properties*, Klaus Havelund and Grigore Rosu.
This paper introduces the theory and monitoring algorithm for a past time temporal logic, very similar to the Past Time Linear Temporal Logic (PTLTL) implemented in JavaMOP. The website also contains some other optional-reading papers on future time temporal logic.

Assignments

Try to specify the three problems presented in lectures 2 and 3 (notes 2 and 3) using JavaMOP's *past time* temporal logic specification language. You may try future time logic, but it is less stable. Generate aspects, and try to compile and run them with a correct and a faulty program to demonstrate that they work.

JavaMOP hints

JavaMOP may contain a few surprises here and there. The following is a list of advices on how to avoid these issues.

1. General comments:

    (a) Call your aspect MonitorAspect.aj

    (b) Don't use ”it” as a parameter name to specifications, it will cause a name conflict in the generated aspect (bug).

    (c) Always use ”centralized” and ”scope=global”, as on slides.

    (d) Don't use ”violation” handlers together with ”partial”.

    (e) Only use the ”call” pointcut and not ”exec” pointcut. The ”exec” pointcut causes the called method to be instrumented, and if it is defined inside a Java library pakcage it will not get instrumented, since this library in general cannot be instrumented.

2. Context free grammars (CFG): be careful when using in combination with ”violation handlers”. Remember to think in terms of ”generating” events: the first ones occurring in the top rule. A monitor is only created when one of these are executed. This can cause surprises. Assume for example that we want to check the property that locks are taken and released in a matching pattern, like:

```
lock
    lock
        lock
        release
    release
release
```

    This can be specified with the rule:

```
productions : S -> lock S release | epsilon
```

    However, this definition causes only the ”lock” event to be monitor-generating whereas the ”release” event is not. Assume that we want to check conformance by using a violation handler as follows:

```
violation handler{
    System.err.println("*** bad lock pattern");
}
```

    Assume now the trace: ”release release”. This is obviously violating the specification from an *intuitive* point of view (our *intention* at least). No lock is taken but there are two releases. However, it will not get detected since a monitor is only generated when a ”lock” operation is detected, whereas the ”release” operation is completely ignored before that.

If it is acceptable that a CFG property only gets activated when one of the first events get detected, then CFG can be used with violation handlers. Otherwise it is suggested to use CFG with validation handlers, which is somewhat a counter intuitive use of CFG.

3. Temporal logic (PTLTL and FTLTL):

   (a) Temporal logic does not work with validation (bug). However, in temporal logic, checking a property with a validation handler is semantically equivalent to checking the negated property (put a not: ‘!’ in front) with a violation handler. Hence, if you want to state a property and get a handler to execute whenever it is validated, negate it, and define a violation handler.

   (b) Partial matching does not make sense in temporal logic (no need/effect of using the ”partial” keyword).

   (c) In future time temporal logic (FTLTL) one may have to insert a next-operator (”o”) to separate creating events from other events. For example, instead of just writing:

   ```
   formula : [](create -> [](updatesource -> [](!next)))
   ```

   meaning: “*it is always the case that when an enumerator is created from a vector, then from then on, if there is an update of the source vector, then from then on there are no calls of next*”, one will have to write:

   ```
   formula : [](create -> o [](updatesource -> [](!next)))
   ```

   Note the added next-operator ”o”. This next-operator is not needed in traditional temporal logic, but is required here due to internals of JavaMOP and the way it handles generating events, in this case ”create”, which it prefers to have standing alone.