A Lightweight LTL Runtime Verification Tool for Java

Eric Bodden
Chair for Computer Science II
Programming Languages and Program Analysis
RWTH Aachen University
52062 Aachen, Germany
eric.bodden@rwth-aachen.de

ABSTRACT

Runtime verification is a special form of runtime testing, employing formal methods and languages. In this work, we utilize next-time free linear-time temporal logic (LTL\X) as formal framework. The discipline serves the purpose of asserting certain design-time assumptions about object-oriented (OO) entities such as objects, methods, and so forth. In this paper we propose a linear-time logic over joinpoints [4], and introduce a lightweight runtime verification tool based on this logic, J2SE 5 metadata [3] and an AspectJ-based [2] runtime backend. Implementations have been proposed so far for imperative and functional languages [5]. To our knowledge our approach is the first to allow addressing of entire sets of states, also over subclass boundaries, thus exploiting the OO nature.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: LOGICS AND MEANINGS OF PROGRAMS—*Specifying and Verifying and Reasoning about Programs*

General Terms

Reliability, Verification, Security, Documentation

Keywords

Runtime Verification, Linear-time temporal logic (LTL), Metadata, AspectJ, Joinpoints, Concurrent Systems

1. LINEAR-TIME TEMPORAL LOGIC

Linear-time temporal logic (LTL) is a calculus that provides the foundation for temporal reasoning about certain properties, based on formulas that use the usual logic connectives $\neg, \wedge, \vee, \rightarrow, ...$ and temporal operators.

In our model the notion of a *state* is modeled by so-called joinpoints, a *path* is a specific execution path of a program. Joinpoints are points in the execution flow of a program. We allow specifications of sets of such joinpoints by defining the universe of our logic as the set of all possible *pointcuts*. Pointcuts are AspectJ-specific language constructs that allow the specification of a set of joinpoints. For formulas ϕ and ψ, the temporal operators are:

- **X** ϕ - Next: ϕ holds at the next state.
- **G** ϕ - Globally: ϕ holds on the remaining path.
- **F** ϕ - Finally: ϕ holds eventually (somewhere on the subsequent path).
- ϕ **U** ψ - Until: ϕ holds until at some state ψ holds.

The *next* operator however is omitted in our logic LTL\X since the notion of a next state would allow for too much indeterminism to provide simple semantics.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2. J2SE 5 METADATA

In versions of Java up to 1.4, one would have needed to introduce certain comment tags in order to allow for the inclusion of LTL\X formulas. J2SE version 5 introduces a straightforward, standardized mechanism for inclusion of additional information, commonly being referred to as *metadata*. Metadata may be added to any declaration, such as for class, interface, field, method, parameter, constructor, enumeration and local variables [3]. This is done by defining a new *annotation type* and then using this annotation type in the source code. For our purpose of specifying LTL\X formulas, we create the parameterized annotation type LTL, holding the actual formula as a simple String object:

```
public @interface LTL{ String formula; }
```

A remarkable fact about metadata annotations is that they are automatically compiled into Java bytecode as code attributes in a standardized way. This allows for deployment of such annotations in sealed bytecode packages and for easy handling by tools, such as the one proposed here.

3. GOOD USABILITY

Our approach enables the user to specify formulas directly as annotations to the source code. For example, one would like to verify that certain sensitive transaction methods should never be called unless the current user is authenticated. In such a case one could introduce another annotation type `SensitiveMethod` for the sole purpose of tagging such sensitive methods, and then specify an LTL\X formula as follows. The String holding the formula was broken into two parts for improved readability.

```
@LTL("!execution(@SensitiveMethod * *.*(..)) " +
"U (call(User.logIn(..)) /\ F if(User.loggedIn))")
```

This fully captures the situation (logout neglected here for simplicity) in an easily understandable way. In this example the formula can be broken down into three pointcuts:

- `!execution(@SensitiveMethod * *.*(..))` matches every joinpoint except the ones that represent the execution of any method being annotated with the annotation type *SensitiveMethod*.
- `call(User.logIn(..))` matches calls to the method `User.logIn(..)`.
- `if(User.loggedIn)` matches all joinpoints where the proposition `User.loggedIn` holds.

In addition to that, we propose the syntactic constructs `thisMethod`, `thisField`, `thisClass`... that match the item, that was annotated with the formula.

One can easily see that this use of pointcuts (and implicitly propositions) leads to a simple format, however maintains powerful expressiveness of our logic: We allow to reason about entire sets of states, which might be spread over the whole lifecycle of the application, by just using one single pointcut. The temporal operators give access to an easy specification of dependencies between those states.

4. PROCESSING THE LTL\X FORMULAS

As we demonstrated above, the annotations to the source code, enabling runtime verification, are minimal and J2SE 5 compliant. We require *no language extension* rather making use of parameterized annotations. They are passed through to the bytecode by any J2SE 5 compliant compiler.

In order to process those annotations, we propose a lightweight preprocessing engine that extends and makes use of the AspectJ compiler. This engine has the purpose of generating pieces of advice (pieces of code to be executed at a given joinpoint) that implement the actual check of the behavior specified by the formula. For instance the example from above could be translated into three pieces of advice:

```
enum StateType =
  {matchLhsOfF, matchRhsOfF, allMatched};
StateType state = matchLhsOfF;

after returning():
  if(state == matchLhsOfF) && call(User.logIn(..))
    { state = matchRhsOfF; }

after returning():
  if(state == matchRhsOfF) && if(User.loggedIn)
    { state = allMatched; }

before():
  if(state != allMatched) &&
  execution(@SensitiveMethod * *.*(..))
    { /* report violation of formula */ }
```

Our implementation strategy has several advantages over earlier approaches. Firstly it allows reasoning about sets of states through pointcuts. The usual approach is to annotate each single state, that is to be monitored, separately. Apart from that, pointcuts capture the OO nature by allowing direct reasoning about subclasses as well. Also we require no additional logic to process the formulas as such, since we directly transform formulas to pieces of advice, making use of the aspects in order to propagate state information from one joinpoint to another. This makes the implementation straightforward because during parsing of the formula, the parsing of the pointcuts is delegated to the AspectJ compiler. The generation of the appropriate pieces of advice also only requires minimal transformations of pointcuts. As well, the use of AspectJ provides type checking of pointcuts for free. Finally another advantage is that AspectJ allows for *unweaving* woven code, allowing easy removal of any verification code. Regarding efficiency we note that the proposed implementation would not yield any additional complexity compared to earlier approaches.

Our technique can be applied to all kinds and scales of Java applications. However, it is a general opinion that best use cases lie in the field of application middleware and concurrent systems. Those tend to show a higher need for verification of safety properties, handling of mutex problems and similar. Temporal logic perfectly aids such reasoning.

5. PREREQUISITES

As noted above, the implementation of our proposal would make use of the AspectJ compiler which would have to be J2SE 5 compliant and metadata-aware in order to be able to process the metadata tags. Unfortunately, this is not yet the case, however is already on the current plan for version 1.3 [1]. As soon as this technology is available, implementation of our preprocessing engine will be possible at once.

6. CONCLUSIONS

We have presented a lightweight tool that allows inclusion and verification of next-time free linear-time temporal logic formulas into Java applications. The formulas can be easily specified in the source code by metadata annotations. Thus, no enhancement of the Java language as such is required and the specified formulas can be deployed within Java bytecode. In order to express assumptions about the execution flow of a program, we allow AspectJ pointcuts to be part of those formulas. Initial tests revealed only minor implementation issues, however as prerequisite, J2SE 5 compliance and metadata awareness of the AspectJ compiler are required, which are expected for the next major release. From the resulting Java application, the runtime verification code can be easily removed for deployment by making use of the *unweave* feature of the AspectJ compiler.

7. REFERENCES

[1] Adrian Colyer (IBM), AspectJ project lead. Personal communication, June 2004.

[2] AspectJ website. http://www.eclipse.org/aspectj/.

[3] Java specification request for metadata annotations (JSR175). http://jcp.org/en/jsr/detail?id=175.

[4] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[5] V. Stolz and F. Huch. Runtime verification of Concurrent Haskell programs. In *Proceedings of the Fourth Workshop on Runtime Verification*, to appear in ENTCS. Elsevier Science Publishers, 2004.