Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 144 (2006) 109–124

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Temporal Assertions using AspectJ

Volker Stolz[1] and Eric Bodden[1]

Dept. of Computer Science
Programming Languages and Program Analysis
RWTH Aachen University
52056 Aachen, Germany

Abstract

We present a runtime verification framework for Java programs. Properties can be specified in Linear-time Temporal Logic (LTL) over AspectJ pointcuts. These properties are checked during program-execution by an automaton-based approach where transitions are triggered through aspects. No Java source code is necessary since AspectJ works on the bytecode level, thus even allowing instrumentation of third-party applications. As an example, we discuss safety properties and lock-order reversal.

Keywords: Runtime verification, LTL, AspectJ, aspect-oriented programming, alternating automata.

1 Introduction

To avoid misbehaviour, many software products include assertions which check that certain states on the execution path satisfy given constraints and otherwise either abort execution or execute specific error-handling. These assertions are usually limited to testing the values of variables. However, often it would be convenient not only to reason about a single state but also about a sequence of states. This enables the developer to reason about control flow and execution paths. In previous work [15], we discussed a symbolic checker for parametrised LTL formulae over finite paths which allowed us e.g. to reason about a problem found in multi-threaded applications commonly referred to as *lock order reversal*. At runtime, potential problems would be pointed out

[1] Email: {stolz,bodden}@i2.informatik.rwth-aachen.de

1571-0661/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2006.02.007

to the developer. We have implemented a working prototype with similar functionality for Java applications using a symbolic checker based on the code generation approach we describe in the following. The major contribution of this work is to propose an alternative, automaton-based solution which allows for even more expressiveness, though yielding better performance.

A major drawback of the former approach was that annotations driving the checker had to be inserted into the source code of the application under test. For practical purposes, we supplied an annotated replacement for the concurrency-library. Applications wishing to use this framework thus had to be recompiled.

Also, such source code-based hooks may lead to severe problems with respect to object-oriented properties such as behavioural subtyping: Any specification which is stated for a certain class should also hold for all its subclasses. The use of source code annotations within method bodies does not reflect such implicit rules. Thus we restrict our formalism in such a way that it only allows to reason about well-defined interfaces, meaning method calls and field accesses. The recent success of the utility Valgrind [11] shows that there is sufficient demand for better debugging support (in contrast to techniques like model checking).

In Section 2 we give a short introduction into temporal logic. Instead of symbolically checking a formula, we use a translation into an alternating automaton. Section 3 discusses instrumentation of (Java) applications and focuses on aspect-oriented programming in AspectJ. The combination of LTL and AspectJ in Section 4 yields a state-machine for each formula to be checked through a finite automaton where transitions are driven by an aspect. We discuss an extension to parametrised automata for handling recurring patterns with state and conclude in Section 5.

2 From LTL to alternating automata

In this section we give a finite path semantics for LTL and remind the reader on how to translate LTL formulae into alternating automata.

2.1 Path semantics for LTL

Linear-time temporal logic (*LTL*) [12] is a subset of the Computation Tree Logic CTL* and extends propositional logic with operators which describe events along a computation path. The operators of LTL have the following meaning:

- "Next" ($\mathbf{X}\ \varphi$): The property φ holds in the next step

- "Finally" ($\mathbf{F}\ \varphi$): φ will hold at some state in the future
- "Globally" ($\mathbf{G}\ \varphi$): At every state on the path ϕ holds
- "Until" ($\varphi\ \mathbf{U}\ \psi$): φ has to hold until finally ψ holds.
- "Release" ($\varphi\ \mathbf{R}\ \psi$): Dual of $\mathbf{U}$; expresses that the second property holds along the path up to and including the first state where the first property holds, although the first property is not required to hold eventually.

Usually LTL is defined over infinite paths. For runtime verification, we assume that the verification process is stopped at some point in time and correspondingly the LTL formulae have to be evaluated over a *finite* path. Thus we declare the semantics as follows.

Let $PROP$ be a set of atomic propositions and $w = w[1]...w[n] \in (2^{PROP})^n$ a finite path. For each path position $w[j]$ $(1 \leq j \leq n)$ and proposition $p \in \{p_1, ..., p_m\}$ and formulae φ and ψ:

$$
\begin{array}{llll}
w[j] & \models \mathbf{tt}, & w[j] \not\models \mathbf{ff}, & \\
w[j] & \models p & \text{iff} & p \in w[j] \\
 & \models \mathbf{X}\ \varphi & \text{iff} & j < n \text{ and } w[j+1] \models \varphi \\
 & \models \mathbf{F}\ \varphi & \text{iff} & \exists k\ (j \leq k \leq n) \text{ s.th. } w[k] \models \varphi \\
 & \models \mathbf{G}\ \varphi & \text{iff} & \forall k\ (j \leq k \leq n) \rightarrow w[k] \models \varphi \\
 & \models \varphi\ \mathbf{U}\ \psi & \text{iff} & \exists k\ (j \leq k \leq n) \text{ s.th. } w[k] \models \psi \\
 & & & \quad \wedge\ \forall l\ (j \leq l < k) \rightarrow w[l] \models \varphi \\
 & \models \varphi\ \mathbf{R}\ \psi & \text{iff} & \forall k\ (j \leq k \leq n) \rightarrow w[k] \models \psi \\
 & & & \quad \vee\ \exists l\ (j \leq l < k) \text{ s.th. } w[l] \models \varphi
\end{array}
$$

We write $w \models \varphi$ if $w[1] \models \varphi$. Furthermore we consider formulae in *negation normal form* where negations are pushed down to the propositions using basic equivalences. We call the set of all LTL formulae in this form LTL_{NN}. Our implementation will be built on *Next*, *Until* and *Release*, so we use the following equivalences to express *Finally* and *Globally*:

$$\mathbf{F}\ \varphi = \mathbf{tt}\ \mathbf{U}\ \varphi \qquad \mathbf{G}\ \varphi = \mathbf{ff}\ \mathbf{R}\ \varphi$$

2.2 Generating automata

In our previous approach, we successively check all propositions on top-level of the formula, i.e. those that are not shadowed by temporal operators. Then, the outer temporal operators are unrolled once. But nested temporal oper-

ators will lead to constantly expanding formulae. It is hard to reason what set of minimisation steps would be complete in order to counterbalance this tendency. To eliminate this *evaluate-unroll-minimise* cycle, we transform the LTL formulae to alternating automata.

This model allows convenient reasoning about such minimisations. Our translation works similarly to the one described by Finkbeiner and Sipma [5]. An alternating finite automaton (AFA) is a quintuple $\mathcal{A} = (Q, \Sigma, q_0, T, F)$ with

- Q finite set of states
- Σ finite alphabet
- $q_0 \in Q$ initial state
- $T : Q \times \Sigma \to B_+(Q)$ transition function
- $F \subseteq Q$ set of final states,

where $B_+(Q)$ is the set of all positive Boolean combinations over Q, recursively defined as the smallest set such that

- $Q \subseteq B_+(Q)$,
- $\mathbf{tt}, \mathbf{ff} \in B_+(Q)$ and
- $q_1, q_2 \in B_+(Q) \Rightarrow (q_1 \wedge q_2) \in B_+(Q), (q_1 \vee q_2) \in B_+(Q)$.

Note that this set contains equivalent but syntactically distinct formulae. Our implementation represents Boolean combinations by sets of sets of states. This leads to automatic identification of equivalent formulae.

A run on an AFA $\mathcal{A}$ is a directed acyclic graph over Q. $\mathcal{A}$ accepts an input $\mathcal{P} = (\mathcal{P}_1, ..., \mathcal{P}_n) \in (2^{PROP})^n$ if there exists a run on $\mathcal{P}$, such that all branches of the run lead to a final state after reading $\mathcal{P}$.

In our interpretation, the AFA are defined over LTL formulae, thus we have the following identities for an AFA $\mathcal{A}_\varphi$ for a given LTL formula $\varphi \in LTL_{NN}$. Let the *closure of a formula* $cl(\varphi)$ be the set of all sub-formulae of φ. Then

- $Q := cl(\varphi) \subseteq LTL_{NN}$
- $\Sigma := 2^{PROP}$
- $q_0 := \varphi$
- $F := \{q \in Q \ \mid \ q = (\Phi \ \mathbf{R} \ \Psi) \text{ for some } \Phi, \Psi\} \cup \{\mathbf{tt}\}$.

F is defined this way because a *Release* formula is always valid on the empty path whence an *Until* formula is not.

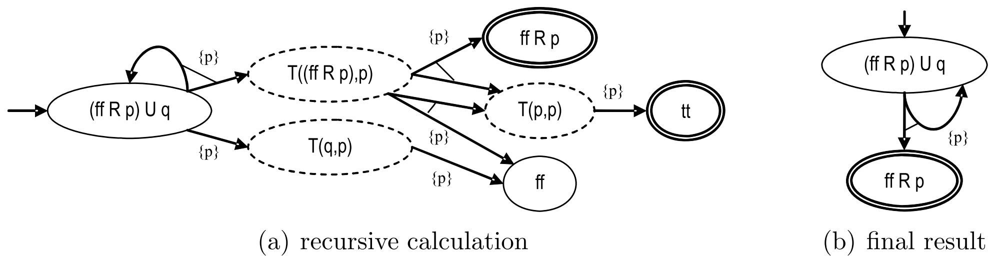
Note that all states of the AFA are valid LTL formulae. The reader should keep this in mind since in the following we use both the words *state* and *formula* in exchange, whatever might be more appropriate for the given

purpose. The transition function T is recursively defined as follows: Let $\mathcal{P} \subseteq PROP, p \in PROP, \varphi, \psi \in LTL_{NN}$. Then

- $T(p, \mathcal{P}) = \mathbf{tt}$ resp. $\mathbf{ff}$ if $p \in \mathcal{P}$ resp. $p \notin \mathcal{P}$
- $T(\neg p, \mathcal{P}) = \mathbf{tt}$ resp. $\mathbf{ff}$ if $p \notin \mathcal{P}$ resp. $p \in \mathcal{P}$
- $T(\mathbf{tt}, \mathcal{P}) = \mathbf{tt}$ and $T(\mathbf{ff}, \mathcal{P}) = \mathbf{ff}$,
- $T(\varphi \wedge \psi, \mathcal{P}) = T(\varphi, \mathcal{P}) \wedge T(\psi, \mathcal{P})$
- $T(\varphi \vee \psi, \mathcal{P}) = T(\varphi, \mathcal{P}) \vee T(\psi, \mathcal{P})$
- $T(\mathbf{X}\varphi, \mathcal{P}) = \varphi$
- $T(\varphi \mathbf{~U~} \psi, \mathcal{P}) = T(\psi, \mathcal{P}) \vee (T(\varphi, \mathcal{P}) \wedge \varphi \mathbf{~U~} \psi)$
- $T(\varphi \mathbf{~R~} \psi, \mathcal{P}) = T(\psi, \mathcal{P}) \wedge (T(\varphi, \mathcal{P}) \vee \varphi \mathbf{~R~} \psi)$.

In particular it should be noted that the calculation of $T(\varphi, \mathcal{P})$ is well-founded and the leaves are labelled with a formula, $\mathbf{tt}$, or $\mathbf{ff}$.

Figure 1 (a) shows how the successor state of $(\mathbf{G}\ p) \mathbf{~U~} q = ((\mathbf{ff} \mathbf{~R~} p) \mathbf{~U~} q)$ for the input $\{p\}$ (where p holds, but q does not hold in the current state) is calculated. Solid nodes represent states while dashed nodes represent intermediate steps in the recursive calculation according to the definition of T. Edges sharing the same origin represent conjuncts. The final result is shown in Figure 1 (b). It is derived by Boolean evaluation of the transition structure with respect to idempotency and commutation.



(a) recursive calculation (b) final result

Fig. 1. Calculation of $T(((\mathbf{ff} \mathbf{~R~} p) \mathbf{~U~} q), \{p\})$

The generated AFA are known to be *weak*, meaning that there exists a partial order on Q. Thus there can be no non-singular loops. This is an important property since it means that when reading an input, states can only be switched to states "further down" in the automaton or remain unchanged. As a result, it suffices to evaluate the transition relation without taking nested Boolean combinations into account: Cases such as $\varphi_1 \wedge (\varphi_2 \wedge (\varphi_1 \wedge \varphi_3))$, where φ_1 reoccurs on a deeper level, are impossible.

In order to calculate the full AFA for a given formula and a given input, one has to calculate all successors of all states in this way. The calculation of one successor is linear in $|cl(\varphi)|$. For a set $\mathcal{P}_\varphi$ of propositions in φ with

$|\mathcal{P}_\varphi| = n$, there are 2^n possible successors of each state. Thus, full calculation of the full transition function leads to an exponential blowup. This might not be desired, especially taking into account that not all states (sub-formulae) are reachable on a given input. At runtime, the input is known and it suffices to calculate the successor states *under this given input*. Hence, there is a natural trade-off between on-the-fly generation of successor states as the input is read and static pre-calculation of the whole transition table for the full state space.

2.3 Determinisation

The AFA can be determinised as follows:

For a given AFA $\mathcal{A}_\varphi = (cl(\varphi), 2^{PROP}, \varphi, T, F)$, we define the deterministic finite automaton (DFA) $\mathcal{B}_\varphi$ as $\mathcal{B}_\varphi := (Q_\mathcal{B}, 2^{PROP}, \varphi, \delta, F_\mathcal{B})$ where

- $Q_\mathcal{B} := \{\varphi\} \cup \{b \in B_+(cl(\varphi)) \mid b = T(\psi, \mathcal{P})$ for some $\psi \in cl(\varphi), \mathcal{P} \subseteq PROP\}$ the set of positive Boolean combinations of sub-formulae of φ occurring in the transition relation of $\mathcal{A}_\varphi$,
- $F_\mathcal{B} := \{q \in Q_\mathcal{B} \mid \mathit{final}(q) = \mathit{true}\}$ where for all $\varphi_1, \varphi_2 \in Q_\mathcal{B}$:
  - · $\mathit{final}(\varphi_1 \wedge \varphi_2) = \mathit{final}(\varphi_1) \wedge \mathit{final}(\varphi_2)$,
  - · $\mathit{final}(\varphi_1 \vee \varphi_2) = \mathit{final}(\varphi_1) \vee \mathit{final}(\varphi_2)$,
  - · $\mathit{final}(\varphi_1) = \begin{cases} \mathit{true} & \text{if } \varphi_1 \in F \\ \mathit{false} & \text{else} \end{cases}$
- and $\delta(q, \mathcal{P}) = T(q, \mathcal{P})$ for all $q \in Q_\mathcal{B}$.

The definition of δ makes clear that essentially the DFA takes the very same transition as the corresponding AFA. The only difference is that a transition of the AFA results in a Boolean combination of states. In the corresponding DFA, this Boolean combination itself is defined as the *deterministic* successor state for this input. Since the AFA is weak, there can only be a finite set of such positive Boolean combinations. Hence, the state set of the DFA is finite.

2.4 Soundness on finite paths

When initially evaluating the topic and possible implementations, we considered modelling LTL using Büchi automata. A Büchi automaton is a natural concept from the point of view of model checking, where traces are usually infinitely long and the automaton correspondingly accepts infinite traces. As we found out in experiments and as Havelund mentions in [13], Büchi automata are difficult to adapt to reason about finite traces, because of their accepting condition. In particular there may be several minimal Büchi automata all recognising the same language and even with the same transition table but with different sets F of accepting states. Depending on the LTL

to Büchi automata translation, the resulting automaton might or might not be suited for evaluation of finite paths, where it is essential which states *exactly* are final. Thus we opted for alternating finite automata, which provide a sound model and can, as presented, easily be transformed to ordinary DFA over finite words.

3 Instrumenting the program with AspectJ

Aspect-oriented programming in general has the aim of separating *crosscutting concerns* into separate modules of development and deployment. Such concerns typically build a logical and functional unit but are still scattered throughout the whole application due to limitations of expressiveness of the base programming language. Aspect-oriented languages typically come as an extension to a base language, adding a layer of quantification, that allows code to be modularised into a single unit by employing declarative constructs telling the runtime when and where to apply the code as the application runs.

AspectJ is today the most widely used aspect-oriented programming (AOP) language, based on the core language Java. It was originally developed by Xerox PARC in the late 90's. Various companies and researchers contributed to its development. It is integrated into large-scale production environments like IBM Websphere.

In the aspect-oriented notion a core application exposes a set of identifiable points in its dynamic control flow at runtime, called *joinpoints*. An aspect in the AspectJ language employs *pointcuts*, which can be seen as second-order predicates over those joinpoints. The language provides the following pointcuts as well as their Boolean combinations (other pointcuts match e.g. on the static structure or on exceptions, but are not of interest for us):

pointcut	matches
`execution(`*MethodSignature*`)`	execution of a method
`call(`*MethodSignature*`)`	call to a method (caller side)
`set/get(`*FieldSignature*`)`	field accesses
`cflow(`*Pointcut*`)`	control flow of any joinpoint matched by the inner pointcut
`if(`*BooleanExpression*`)`	whenever the expression is satisfied; the expression has generally access to static Java members as well as values exposed by the formula

Pointcuts allow to pick out points in the dynamic control flow. We employ them simply as the propositions of our logic: A pointcut proposition is valid at a given point in time iff this pointcut matches the current joinpoint.

The `cflow` pointcut provides a quasi-temporal notion: For another pointcut p, `cflow(`p`)` matches all joinpoints which occur between entering a joinpoint matched by p and leaving this joinpoint. E.g. `cflow(execution(void C.f()))` matches all joinpoints between start and end of the execution of `f()`.

Pointcuts can contain wildcards to abstract from certain parts of a signature that are to be matched. Thus, any pointcut is able to pick out entire sets of joinpoints as the application runs. In order to actually contribute behaviour to the core application, *pieces of advice* can be attached to each pointcut. The semantics state that whenever the pointcut matches a joinpoint in the dynamic control flow of the application, the piece of advice is to be executed at the specified time: either *before*, *after* or *in substitution of* (*around*) the original joinpoint. Figure 2 gives an example for logging authentication events.

```
pointcut auth(User u):
  call(* Authentication.login(User)) && args(u);

after returning(): auth(User user) {
  SecurityLog.log("User " + user.getId() + " logged in");
}
```

Fig. 2. AspectJ pointcut and advice logging authentication events

As the example shows, the additional pointcuts `this`, `target` and `args` expose runtime state to a piece of advice. Our approach is also capable of handling such runtime state. The focus of this paper however is the general approach. We will explain handling of such state in future work.

AOP implementations like AspectJ are *weaving compilers*. This means that they implement an aspect by statically weaving it into the core application using bytecode transformations. Runtime checks for dynamic type checking etc. are automatically inserted where necessary.

4 Putting it all together

In the following, we describe how we interpret LTL over a Java application and how the aforementioned model leads to a complete implementation.

4.1 LTL in the state space of a Java program

Each run of a Java program defines an execution trace where each state on the trace corresponds to a virtual machine state. The state includes e.g. the program counter, the current stack, and heap (for a detailed discussion of the Java Virtual Machine see [10]). Each bytecode instruction (field access, method invocation, etc.) triggers a state transition.

This model is too fine-grained (e.g. usually there is no need to reason about the value of the program counter) and we limit ourselves to states in the execution addressable through AspectJ. Pointcut expressions select a (possibly non-contiguous) set of states from the trace, e.g. through Boolean composition of pointcuts. In general, there is no one-to-one correspondence between joinpoints and neither bytecode-instructions nor source code. This fine distinction however is not detrimental to our approach.

We assume that the corresponding pieces of advice do not alter the original control-flow of the application except when triggering assertions, although aspects may be (ab-)used to alter the behaviour of the application substantially.

Let us consider the following example for a temporal property: Every call to a method `C.f()` is finally followed by a call to `C.g()`. In LTL we would express this in the following way:

$$\mathbf{G}\ (\texttt{call}(\texttt{C.f}()) \longrightarrow \mathbf{F}\ \texttt{call}(\texttt{C.g}())\ )$$

The outer *Globally* assures that this formula is verified indeed for every occurring call to `C.f()`, since otherwise we would only match the trace where exactly the first state corresponds to the method invocation.

A solution using pure AspectJ must instrument both method invocations and introduce a new variable which tracks if we are looking for a call to `C.g()`. A `cflow`-pointcut does not help here, because the call to `C.g()` does not have to be within the control flow of `C.f()`.

Instead of programming the state transitions manually, our implementation of an LTL-checker gives the developer a way of deriving a verification aspect from the formula. The state-tracking variables are implicitly contained in the generated automaton. By combining aspects with temporal reasoning, we obtain a powerful way of expressing temporal inter-dependencies.

4.2 Can we see the code, please?

Consider the following example: The method `requiresInit()` shall not be called unless the method `init()` has been invoked before:

```
(!call(void SomeClass.requiresInit()))
```

```
U (call(SomeClass.init()))
```

We employ the *abc* compiler [3] to parse each such formula. When generating the abstract syntax tree (AST) we can statically check for syntax- and type-errors. We then transform this AST into an aspect in the AspectJ language. This aspect consists of two major parts:

(i) A state variable holding the current state. This is first assigned the initial state of the DFA, which is represented by the original formula.

(ii) A transition table implemented by $n + 1$ pieces of advice where n is the number of different propositions (pointcut definitions) employed in the formula. In particular we have
   - one such piece of advice for any pointcut proposition recording that this pointcut matches (the proposition is then active at the current state)
   - one advice that matches any joinpoint where one of the former matches, triggering the state change for the recorded set of propositions.

It is important to note that in the case where several pieces of advice match a single joinpoint, they are executed in the textual order in which they occur in the AspectJ compilation unit.

Figure 3 shows an outline of the aspect we would generate for the example from above. First we label pointcuts as propositions. Then we define the initial state as the given formula. Two pieces of advice gather all valid propositions for a given joinpoint. The last piece of advice is executed whenever the pointcuts $p1$ or $p2$ match, and importantly *after* all other pieces of advice, since it is the last one in the textual order. It finally triggers the state transition of the automaton. Here we demonstrate the approach employing lazy state-space generation. A state (sub-formula) is capable of determining its own successor state under a given set of valid propositions.

The generated aspects are then woven into the application, resulting in a version instrumented for runtime verification. Figure 4 gives a graphical overview of the different steps involved.

4.3 Running the application

In order to have the specification checked, it is only necessary to run the instrumented application. The instrumentation/weaving does not have to take place prior to load time. AspectJ generally supports load-time weaving through a special classloader. This allows instrumentation of classes as they are loaded into the system, which may affect classes that were not available at specification time. In particular, formulae can also reason about interfaces. This implies that all loaded classes implementing such an interface will automati-

```
aspect InitPolicy {
  pointcut p1(): call(void SomeClass.requiresInit());
  pointcut p2(): call(SomeClass.init());

  int p1 = 1; int p2 = 2;
  Formula state = Until( Not(p1), p2 );

  Set<int> currentPropositions = new Set<int>();

  after(): p1() { currentPropositions.add(p1); }
  after(): p2() { currentPropositions.add(p2); }

  after(): p1() || p2() {
    state = state.transition(currentPropositions);
    if(state.equals(Formula.TT)) {
      // report formula as satisfied
    } else if(state.equals(Formula.FF)) {
      // report formula as falsified
    }
    state.clear(); //reset proposition vector
  }
}
```

Fig. 3. Aspect implementing the initialisation example from page 9

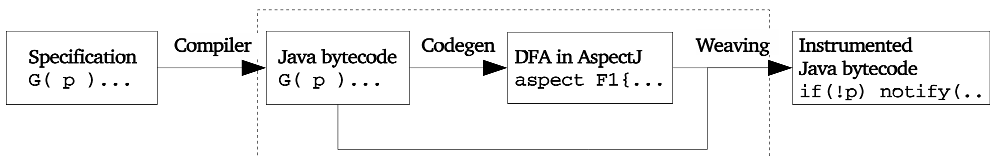


Fig. 4. Instrumentation steps

cally be checked for compliance with the given formulae at runtime.

4.4 Overhead

Runtime Verification in general must introduce some kind of overhead in the instrumented application because of the hooks that trigger the evaluation of the assertions. This overhead is linear to the number of formulae, and linear to their length. `call`, `execution`, and `set` pointcuts are triggered only when necessary by the AspectJ implementation. An `if(expensiveFunc())` pointcut may result in an arbitrarily large overhead. Thus, such expensive evaluations should not be performed within formulae. As long as `if` point-

cuts are restricted to field matches like `if(User.loggedIn)`, the overhead is considerably low.

By making use of AspectJ as an implementation strategy, we automatically inherit all the powerful optimisations which come with the AspectJ implementation. In particular and opposed to earlier approaches, there is no overhead at all for joinpoints which do not lead to a further evaluation of one of the given formulae. For instance, a formula `G(!call(C.f()))` will only be evaluated when `C.f()` is ever really called.

We believe that our implementation is efficient, given the expressiveness of our formalism, except the usual overhead introduced by the usage of aspect-oriented programming, which is known to be in the region of not more than two percent compared to an implementation based on non-modular instrumentation [2]. This overhead might still be reduced by ongoing improvements in the *abc* compiler.

The AspectJ weaver makes sure that necessary code is only inserted at those places which require instrumentation in order to trigger the evaluation. With respect to this, the implementation has an optimal low overhead. When statically precomputing the full transition table, the cost we pay at each such instrumentation point can be split into

- the cost of a virtual method call to the aspect instance,
- a state change of the associated automaton, consisting of one integer comparison and one integer assignment.

The former is likely to be eliminated as the weaver implementation improves. As soon as inlining is fully supported, even this virtual method call will disappear. The latter is unavoidable. When formulae require state, some additional bookkeeping has to take place. This is the topic of the next section.

4.5 Lock-order reversal

We use a more complicated example to motivate the need for parametrised formulae: To avoid the problem of lock-order reversal (cf. [6], [15]), we would like to assert through an LTL formula that if two locks are taken in a specific order (with no unlocking in between), the system should warn the user if he also uses these locks in swapped order because in concurrent programs this would mean that two threads could deadlock when their execution is scheduled in an unfortunate order.

Notice that we do not want to abort the execution in this example: we are here also interested in mere warnings, as a violation of the formula might not

[2] Personal communication with M. Webster, IBM Hursley Performance Labs, 2003

coincide with a deadlock. To observe the behaviour of the whole execution-path (of which the erroneous behaviour might only be a sub-path), we wrap the formula into the temporal *Globally*.

Thus, if we consider a class `Lock` with explicit `lock` and `unlock` methods like we might find them in any programming language, we obtain for two threads p_i, p_j and two locks l_x, l_y the formula (it is arguable if LTL is an appropriate specification language):

$$\neg lock_{(p_i,l_y)}\ \mathbf{U}\ (lock_{(p_i,l_x)} \wedge (\neg unlock_{(p_i,l_x)}\ \mathbf{U}\ lock_{(p_i,l_y)}))$$
$$\rightarrow \mathbf{G}\neg(\neg lock_{(p_j,l_x)}\ \mathbf{U}\ (lock_{(p_j,l_y)} \wedge (\neg unlock_{(p_j,l_y)}\ \mathbf{U}\ lock_{(p_j,l_x)}))),\ i \neq j, x \neq y$$

Notice that the formula has four parameters: two locks and two threads. Using the current implementation, this means that we would have to pre-generate all possible formula-instantiations. This pre-generation could either be based on a fixed maximum number of locks and threads or on lock/thread creation at runtime. Both approaches have drawbacks. The first one will obviously miss anything with values larger than the specified upper bounds. The latter requires additional instrumentation of object-creation and the current trace, which for memory reasons will usually be limited to a suffix of the run.

In the example, the thread-identifier shall be passed as an argument, although in a typical implementation it might be implicit (e.g. stored in a special variable or obtainable through an API). In order to express this specification in our formalism, we employ variable bindings. The implementation of this additional feature is work in progress and will be presented in a future publication. Bindings can easily be modelled by constraints (additional conjuncts) in the AFA where the updating and checking of constraints is closely tied to the on-the-fly evaluation. Translating the above formula into a new variant of the LTL-syntax allowing variable bindings results in the following expression (we distinguish between logical operators in LTL and AspectJ for clarity):

```
pointcut lock(Thread t, Lock l):
  call(Lock.lock(Thread)) && args(t) && target(l);

pointcut unlock(Thread t, Lock l):
  call(Lock.unlock(Thread)) && args(t) && target(l);

Thread i,j; Lock x,y;
¬lock(i,y) U (lock(i,x) ∧ (¬unlock(i,x) U (lock(i,y) ∧ ¬target(x))))
→ G ¬ (¬lock(j,x)
        U (lock(j,y) ∧ ¬args(i) ∧ (¬unlock(j,y) U lock(j,x) )))
```

Variable binding works similar to Prolog: The first invocation of an aspect containing `target` or `args` will bind the variable to the current value. All other occurrences referring to the same variable are then turned into predicates. New

bindings for different matching parts of the trace will cause a new incarnation of the related subformulae. This has the desired effect of generating a separate formula for each behaviour observed by the left-hand-side of the implication.

5 Related work

JavaPathExplorer [7] due to Havelund and Roşu reasons about traces and uses a similar approach of specifying runtime behaviour. They use similar semantics of LTL over finite paths, although their approach is not AOP based.

Walker and Viggers [16] proposed a language extension to AspectJ, *tracecuts*. Tracecuts do not match on events in the execution flow as pointcuts do, but instead match on traces of such events. Those traces are specified by means of context-free expressions over pointcuts. Since this approach provides a language extension, it cannot be used in combination with ordinary Java compilers. Tracecuts do not provide automatic tracking of state. Inspired by this work, Allan *et al.* [1] extended the *abc* compiler with *tracematches* which allow to bind free variables in pointcut expressions. This solves the problem of binding the above formula for every valid combination of parameters strictly on the basis of those values actually occurring during execution. The implementation in *abc* follows similar thoughts as the approach we proposed in [15]. Allan *et al.* however do not employ alternating automata in their model. We have the feeling that using AFA as underlying representations, bindings can be integrated in a very natural way.

Klose and Ostermann [9] discuss how temporal relations can be expressed in Gamma, an aspect-oriented language on top of a minimal object-oriented core language. Pointcuts are specified in a Prolog-like language and include timestamps that can be compared using the predicates `isbefore/after`. Their prototype requires an already existing trace to apply aspects to and is not applicable to an existing language.

Java-MaC [8] is a runtime-assurance tool for Java. The Meta Event Definition Language (MEDL) is used to specify safety properties. As the MaC architecture should be language-independent, a Primitive Event Definition Language (PEDL) provides the binding to the target language, here Java. While Java-PEDL has been designed to closely correspond to Java, it is not as comfortable to use as AspectJ where expressions are not *modelled after* Java, but in fact are Java expression. Also, state in MEDL seems to be limited to primitive types.

Valgrind [11] is a system for profiling x86 programs by instrumenting them at runtime. Tools for detecting memory management and threading bugs are provided. Extending Valgrind should be the natural choice if applications

compiled to native code (e.g. from C or C++) should be instrumented. In fact, an earlier version contained a tool implementing the Eraser-algorithm which detects data-races in multi-threaded programs [14].

Temporal Rover [4] is a commercial product by Time Rover, Inc. It handles LTL assertions embedded in comments by source-to-source transformation.

6 Conclusion and future work

We have presented a runtime verification framework based on aspect-oriented programming with AspectJ. LTL formulae with pointcuts as properties are translated into automata. By means of aspects we "walk" through the automaton during execution, detecting violations by entering an error-state. Our current implementation supports the full formalism, yet without access to runtime state. We discuss the improvement to parametrised formulae which overcome this issue. Once this implementation is finished, the resulting tool will provide a formalism which is more expressive and more natural to object-oriented reasoning than any former approach we are aware of.

One limitation of LTL is that it restricts us to reason about *regular* properties only, while often *context-free* properties are interesting as well. Reconsider the example from Section 4.1 with the variation of wanting to specify that "every call of `C.f()` is finally followed by a *matching* call to `C.g()`". I.e. it should no longer be possible for a single `C.g()` to release multiple "waiting" calls to `C.f()`. Raising the expressiveness to the power of context-free properties is non-trivial, though [2]. Although our approach should be able to handle such issues by tracking state, we yet lack a suitable language-extension to LTL to be able to take advantage of this.

In the context of concurrent systems, synchronisation points are crucial to the behaviour of the application and accordingly our logic should provide a convenient way to reason about those. AspectJ already allows matching of `synchronized` methods, although not of `synchronized(`*obj*`){...}`-blocks. Unfortunately, it is not generally possible to statically determine matching `monitorenter` and `monitorexit` instructions at the bytecode level after compilation. This can be addressed dynamically, though. Our prototype of the *Java Logical Observer* ***JLO*** is available from
http://www-i2.informatik.rwth-aachen.de/JLO/.

References

[1] C. Allan, P. Avgustinov, A.S. Simon, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching to AspectJ (*submitted to OOPSLA'05*). abc Technical Report abc-2005-01, McGill University, 2004.

[2] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD'05: Proceedings of the Fourth international conference on Aspect-oriented software development*. ACM Press, 2005.

[4] D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, Stanford, USA, 2000. Springer.

[5] B. Finkbeiner and H.B. Sipma. Checking Finite Traces using Alternating Automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

[6] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, USA, 2000. Springer.

[7] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[8] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O.V. Sokolsky. Java-MaC: A Run-time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[9] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Foundations of Aspect-Oriented Languages workshop (FOAL'05)*, Chicago, USA, 2005.

[10] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1997.

[11] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Third Workshop on Runtime Verification (RV'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[12] A. Pnueli. The Temporal Logics of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[13] G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical Report 01-15, Research Institute for Advanced Computer Science (RIACS), 2001.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[15] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell Programms. In K. Havelund and G. Roşu, editors, *Fourth Workshop on Runtime Verification (RV'04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, Barcelona, Spain, 2004. Elsevier Science Publishers.

[16] R.J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R.N. Taylor and M.B. Dwyer, editors, *12th International Symposium on the Foundations of Software Engineering*. ACM, 2004.