Efficient and Expressive Runtime Verification for Java

Eric Bodden[*]
Chair for Computer Science II
Programming Languages and Program Analysis
RWTH Aachen University
52056 Aachen, Germany

ericbodden@acm.org

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, Semantic Interfaces, Linear-time temporal logic (LTL), Metadata, Aspect-oriented programming, Concurrent systems

1. PROBLEM & MOTIVATION

One of the big advances of software engineering during the last decades was the development of new techniques to modularize code into functional units. Object-oriented programming (OOP) allows for separation of functionality and association of functionality with the state it alters. Aspect-oriented programming (AOP) goes a step further and allows for the separation of whole crosscutting concerns into single units, such concerns being code which builds a functional unit but is though scattered through the whole application in OOP.

As a result of this development, such units are often being developed by different teams, who do not necessarily share any details about each other's implementations, though they do share interfaces to the modules they provide. Thus, the need for clear specification of those interfaces is crucial for a smooth and safe software development process. In order to specify those interfaces, various techniques have been proposed. *Design by Contract* (DBC) [10] is probably the most famous one: The developer of a unit specifies its usage through interfaces and some form of specification, which was so far mostly restricted of pre- and postconditions. Such specification is usually not part of the programming language and often simply marked down as comment and thus not automatically asserted. Some of the available tools however already allow for *formal specification* of such localised conditions and *automated verification* of those during runtime of the application.

[*]ACM membership number: 4133682
Supervisor: Volker Stolz
Category: Undergraduate

Hitherto, those approaches were all restricted to local reasoning: Pre- and postconditions enable reasoning about one single event, e.g. a method invocation. However they do not allow for the specification of *temporal interdependencies* (TIs) between units. This is a clear restriction, since at design time, temporal restrictions are usually well known: If an initialization method exists on a given object, it should be clear that it has to be called before the object is used. Also, in an accounting application, certain operations must not be called until a log on event has taken place. Such TIs exist at design time and should be part of the interface, concretised in a formalised way, so that implementers who program to this interface can have automatically asserted that their implementation behaves as expected by the interface provider - not only locally but over the whole execution trace.

1.1 Contributions

The contributions of this work are a new formalism which allows to build expressive formulae over temporal traces in an intuitive way as well as a complete implementation of this formalism, which instruments any given Java application in bytecode form with appropriate runtime checks of those formulae.

- The formalism is based on Next-time free Linear-time temporal logic [11] over finite traces [7].
- The temporal operators of this logic are applied to a universe of pointcuts in the aspect-oriented language AspectJ. This approach is novel and unique in the field.
- Formulae are deployed either by the use of Java 5 metadata annotations or in an XML format.
- The generated instrumentation code is efficient and generally scalable up to large-sized applications.
- The instrumentation code can be proven to be side-effect free, meaning, that the base application is oblivious to this code in general.

Since our paper on OOPSLA'04 we have implemented a prototype which integrates the whole approach into one single tool. From the semantically annotated interface with formula annotations, the readily instrumented application is just one command away.

1.2 Overview

We proceed with section 2, which gives a bird's eye view of the architecture of our application and necessary background information. Here we also compare to related work. Section 3 describes our approach in detail, giving information about the formalism we provide and about the techniques we use to implement its evaluation. Subsection 3.5 gives some rational for why we call this approach efficient. Subsection 3.6 reports on some difficulties which arise from collecting and tracking state over time. Finally in section 4, we conclude stating the overall results and planned future work.

2. BACKGROUND & RELATED WORK

Our application allows for the specification of temporal interdependencies by the means of *Java 5 metadata annotations* in the formalism of a *Linear-time temporal logic over pointcuts.* A *pointcut* is a notion from aspect-oriented software development, specifying a set of events in the dynamic control flow of an application, *joinpoints.* Such a set can be written down by a certain kind of regular expressions matching on joinpoints, for instance method calls and field accesses. We decided to use the pointcut model of the premier Java-based AOP implementation *AspectJ* because its model matches our requirements very well. It allows to capture method calls and state changes, and this is exactly what specifies the intrinsic and extrinsic behavior of objects.

```
//match all state changes of public fields
//binding the new value to 'newValue'
pointcut publicStateChange(Object newValue):
    set(public * *.*) && args(newValue);

//match all calls to methods starting with 'get'
pointcut getMethodAccess():
    call(* *.get*(..));
```

Table 1: Example pointcuts in the AspectJ language

Earlier runtime verification approaches (cf. subsection 2.2 and [12]) made use of special source code annotations providing hooks to trigger state changes as soon as the dynamic control flow reaches one of them. Those hooks were then picked up by the verification environment. This breaks the encapsulation, which is natural to OOP, taking behavioral subtyping into account: A condition specified for a class should also be valid for all of its subclasses. All earlier approaches for trace-based verification we are aware of do not adhere to this rule. By restricting ourselves to the interception of communication and state changing events, we capture exactly the object-oriented nature and provide a formalism that is exactly as expressive as it should be. Also, at compile time the formulae only *specify* where hooks are to be inserted (by using pointcuts). The actual instrumentation can be performed in a later stage and can even be deffered to load time, instrumenting even classes the system was not aware of at specification time.

Related work dates back to the early days of Design by Contract which became famous with its implementation in the programming language Eiffel. Eiffel allows for the specification of localised pre- and postconditions as well as invariants.

2.1 Contract4J

Contract4J [5] is a tool implementing DBC for Java: It extracts Java 5 metadata annotations from source code. Those annotations hold pre- and postconditions, which are then transformed into AspectJ code, implementing their verification semantics. This approach suffers from several drawbacks: Firstly for annotation extraction, Sun's Annotation Processor Tool (part of the JDK) is being used. This allows extraction from source code only. We allow extraction of annotations from bytecode which is much more flexible since it is applicable also when source code is not available. The specification language comprises no pointcuts, which makes sense, given that the reasoning is purely localised. No temporal interdependencies are taken into account whatsoever.

2.2 Java PathExplorer

One tool which does provide a way to reason about traces, is the JavaPathExplorer [8] due to Havelund and Roşu, which uses a similar approach of specifying runtime behavior. They use the same semantics of LTL over finite paths. However, their approach is not AOP based and thus does not benefit of any optimizations through static approximations built into current compilers for AOP languages. Also, it cannot provide any pointcut language. Instead they use the aforementioned source code annotations as hooks in the program to match on, with the previously mentioned problems of breaking encapsulation.

2.3 Tracecuts and tracematches

Walker and Viggers [13] 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. This issue is currently being addressed by Allan et al. [1], who implement a variation of tracecuts, called *tracematches* as a plugin to the *abc* compiler - the same compiler we use. We might indeed benefit from this work when it comes down to associating state with formulae. We elaborate on this in section 3.6.

3. APPROACH AND UNIQUENESS

The described approach is unique in various ways.

It is the first approach we are aware of, which allows for specification of expressive temporal formulae, still using a convenient method for deployment by the means of Java 5 metadata.

It is the first implementation of runtime verification which employs pointcuts, enabling the user to reason about sets of events rather than single points on the time line.

It is the best-optimizing runtime verification approach we are aware of at the time, given the expressiveness we provide.

3.1 Architecture

The tool we developed is a plugin to the AspectBench Compiler toolkit *abc* [3], which is an alternative, extensible, highly optimizing AspectJ implementation (opposed to the reference implementation *ajc*).

First we *extract formulae* stated in the form of Java 5 metadata annotations *from bytecode.* This allows for deployment of such formulae with readily compiled applications. So for instance a framework provider could ship its framework with such formulae contained in the bytecode to have clients automatically check their implementation for correctness. In this sense, we allow the specification of semantically enriched interfaces. Table 2 gives an example for the automatic assertion of initialization. It makes use of the keywords *thisMember* and *thisType* which are automatically substituted by the according signatures.

```
@LTL("!call(thisMember) U call(thisType.init())")
void someMethodRequiringInitialization() {...}
```

Table 2: Java 5 metadata annotation

Those formulae are then *parsed* by the abc parser we have extended with our plugin. Syntax errors are recognised at once. If syntactically correct, the formula is transformed into an automaton-based evaluator (see chapter 3.4). The generated verification code is modular in the sense, that we generate one single aspect unit per formula. Those aspects are then woven into the original application by subsequent passes, which is so being instrumented to trigger evaluation of the formulae during runtime.

The whole tool is fully integrated. From the original application to the instrumented code there is just one single command line invocation necessary or a button click in Eclipse respectively.

In the following we briefly describe the provided formalism and its semantics.

3.2 LTL over pointcuts

We implemented a Linear-time temporal logic over pointcuts. LTL is a subset of the computation tree logic CTL*. It allows for reasoning about one single path of a computation tree. Its syntax and semantics are defined as follows:

For any formulae φ and ψ:

- **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: ψ holds somewhere and φ holds up till then.

Also we allow the usual logical connectives $\vee$,$\wedge$,$\neg$ and $\rightarrow$. Note that in our interpretation, a *state* is actually any *joinpoint* of the dynamic control flow of the application, which AspectJ exposes to the runtime.

To avoid ambiguous semantics we do not expose the *Next* operator because of severe semantically difficulties, which arise for the user when reasoning about the *next joinpoint*, and in order to get the logic resistant with respect to stuttering [8]. However, the operator is internally used for evaluation.

3.3 Examples

Consider as example the implementation of an ATM, having specified requirements with respect to authentication.

Always, the user is logged in or we see no call to `debit`.

```
G( if(User.isLoggedIn())
 || !call(* Account.debit()) )
```

We also allow for the collection of state, which then can be reasoned about:

For all users `u`*, we do not see a call to* `login` *until at least 60 seconds after the last login attempt.*

```
User u:
(!call(* u.login(Credentials))) U
(if(Time.getTime() > u.lastLogin()+60000))
```

Note that this formula accumulates state. As described in chapter 3.6, this feature is a real challenge for the implementation.

As noted earlier, those formulae are being transformed into an automaton, which is explained in the next subsection.

3.4 Automaton-based evaluation

For maximum efficiency of the instrumented application, we transform the LTL formulae to *non-deterministic Büchi automata* (NBA). It is known [6] that an efficient transformation into *non-deterministic* BA exists:

A non-deterministic Büchi automaton is a quintupel $A = (Q, \Sigma, q_0, \Delta, F)$ with

- Q finite state set,
- Σ finite alphabet (here set of contained pointcuts),
- q_0 initial state,
- $\Delta \subseteq Q \times \Sigma \times Q$ transition relation and
- F a set of final states.

A accepts an *infinite* input word w if there exists an infinite run of A on w such that a state in F is visited *infinitely often.* Figure 1 gives an example.

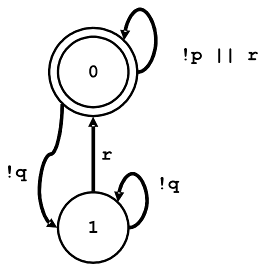

Figure 1: NBA for formula $G(p \rightarrow (\neg qUr))$

Such NBAs are of little help in our case since an implementation needs to behave deterministically. Unfortunately, determinization of BAs in general, is not possible without loss of expressiveness. However, it can easily be shown that in our case a simple power set construction suffices and yields sound results. The result is a deterministic BA (DBA) as shown in figure 2.

We maintain soundness because we perform *runtime* verification (opposed to static model checking) and thus can constrain the input of the automaton to *finite* words generating finite runs. There are only two situations, in which a BA can be known to accept an infinite input after reading a finite prefix: In a minimised DBA there may only exist two *sink states*, the positive sink **TT** and the negative sink **FF**. **TT** has a loop to itself, no other outgoing edge and is final, **FF** is non-final respectively.

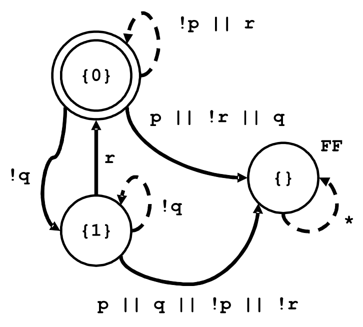

Figure 2: DBA for formula $G(p \rightarrow (\neg qUr))$

After a finite amount of time, at the latest after the whole runtime of the application, three situations can occur:

1. The formula is satisfied. This is exactly the case when the DBA is in **TT**, since then a final state is known to be visited infinitely often.
2. The formula is falsified. This is exactly the case when the DBA is in **FF**, since in this case no final state can be reached any more.
3. The formula could not be fully evaluated. This is exactly the case when the DBA is in a productive state but not **TT**. In the optimised automata this is equivalent to being neither in **TT** nor **FF**.

If neither **TT** nor **FF** exist in an optimised DBA, we can already decide statically, that the original formula cannot be evaluated during runtime and emit an error message at once. In all other cases we know that evaluation after a finite amount of time, and thus runtime verification, is effectively possible.

The aspect code that we generate from such an automaton looks as follows:

```
aspect DBA1 {
  int state = 0;
  pointcut p(): ...;  pointcut r(): ...;
  ...

  after(): if(state=={0})
   && (!p() || r()) {
      state = {0};
  }

  after(): if(state=={1})
   && (p() || q() || !p() || !r()) {
      state = {}; /* report error */
  }
}
```

As the example demonstrates, each transition of the automaton is implemented as so-called *advice* in AspectJ. An advice is a piece of code that is executed whenever the connected pointcut matches. The first advice, is superfluous in this example, since it would only switch the state from {0} to {0}. This behavior is reflected by trivial loops in the automaton. Thus, in a last step we eliminate such loops before we proceed to the actual code generation. We call the result of this process a *Runtime Verification Automaton.*In figure 2, the dashed edges would be removed.

3.5 Efficiency

Runtime Verification in general must introduce some kind of runtime overhead on the instrumented application because of the hooks that trigger the evaluation of the system. This overhead is generally linear to the number of formulae, and linear to their length. In our case however, the propositions contained in the formulae may be expensive to evaluate. While `call`, `execution`, `set` pointcuts are triggered only when necessary by the AspectJ implementation, an `if(someExpensiveEvaluation())` pointcut may result in an arbitrarily large overhead. Thus, such expensive evaluations should simply not be performed within formulae. As long as `if` pointcuts are restricted to field matches like `if(User.loggedIn)`, the overhead is considerably low.

Since we make use of AspectJ as an implementation strategy, we automatically inherit all the powerful optimizations which come with the AspectJ implementation. In particular and opposed to earlier approaches, every formula is *not* evaluated at every single joinpoint but only at those which are necessary for its evaluation. AspectJ filters the state space by the use and implementation of pointcuts. For instance, a formula `G(!call(A.foo()))` will only be evaluated when `A.foo()` is ever really called.

By using abc as AspectJ implementation, which is probably the best-optimizing AspectJ implementation around [4], we can assure that our implementation generates code that is as efficient as it can be at the current time.

We believe that our implementation is maximally 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 [9]. 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 necessarily need instrumentation in order to trigger the evaluation. With respect to this, the implementation has an optimally low overhead. 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, which is composed 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.

3.6 Exposing state

When exposing state, as shown in the second example of section 3.3, we need to perform some bookkeeping. This is done by associating distinct aspect instances with each tuple of exposed objects:

Assume we have given a formula $\varphi(T_1 o_1, ..., T_n o_n)$ where $o_1, ..., o_n$ are bound by pointcut predicates $p_1(o_1), ..., p_n(o_n)$. We denote the set of all automata containing p_i as $A^{(p_i)}$. Whenever a pointcut p_i associated with o_i matches the current joinpoint, *for each* automaton in $A^{(p_i)}$, we generate a new automaton instance $A_\varphi(T_1, ..., T_{i-1}, o_i, T_{i+1}, ..., T_n)$, where the position p_i is bound to o_i.

The implementation of this functionality is still unfinished work. Unfortunately, AspectJ does not directly support the necessary quantification over such tuples of objects: The iteration *for each automaton in* $A^{(p_i)}$ cannot be directly expressed in the language. This feature was discussed with the developers of AspectJ on the AOSD'05 conference [2] and in the end we found a way to define a reusable aspect that encapsulated this bookkeeping in a modularised way. The proposal by Allan et al. [1], however seems even better suited and might provide a solid base for our effort. Naturally, there is an additional overhead introduced by the procedure of assembling state, but again, this overhead is due to an intrisic complexity of the problem, not due to our implementation strategy.

4. RESULTS AND CONTRIBUTIONS

Initial evaluation of our prototype implementation indeed show the expected *low runtime overhead*. As long as only cheap pointcuts, matching on method call/execution or field access is used, the overhead is negligible. The use of the *if*-pointcut however, may lead to a slowdown, depending on the type of the boolean expression which is passed in.

To our best knowledge we present the first Java based tool for runtime verification which is expressive enough to reason about the temporal order in which *sets* of events in the dynamic control flow occur.

Also we were the first to provide a tool which is able to generate instrumented code based on a specification marked down as Java annotations extracted *from bytecode*. The use of bytecode allows for the separation of compilation and verification into two distinct phases. One party may specify formulae at compile time. Another party may have the application instrument at some lateron, even still at load time, spreading the instrumentation even over previously unknown, incoming classes - still in a well-defined way.

Opposed to some former research prototypes, we can also give rationale for why our application is highly optimized with respect to time consumption at runtime. For each formula only a linear overhead is added.

The provided tool assures that the instrumentation is side-effect free in most of the cases. In the remaining situations, where runtime behavior may be altered, a warning is issued at instrumentation time. Thus, verification results obtained by our tool are fully comprehensible and reproducable.

Also, with our work we have taken a first initial step into the field of semantically enriched interfaces. Such interfaces specify behaviour rather than plain lexical structure as it is usual for ordinary Java interfaces. Since our approach covers the well-known pre- and postconditions as well as all kinds of arbitrary temporal interdependencies between object interactions, we believe that our formalism is actually rich enough to cover most of the semantic specifications one could think of.

5. FUTURE WORK

Part of the future work will be a formal proof of correctness, conducted by prooving the equivalence of our declarative and operational semantics. Also we will elaborate on what exact performance overhead to expect under each possible given circumstances. We plan to apply our tool to a large-scale example application in order to test scalability and expressiveness.

Also some studies with respect to usability seem quite useful. LTL is a formalism which is very-well known in the field of verification by formal methods. However, many average software developers might find LTL not so easy to comprehend. A field study would shine some light on the optimality of this notation. An additional layer of abstraction might seem suitable for better comprehensibility.

6. ACKNOWLEDGEMENTS

Much gratitude is due to the following people: Volker Stolz, for the invaluable contributions he made and the immeasurable patience he showed throughout this work; the *abc* team, especially Oege de Moor, for a lot of help with their compiler framework and everyone who contributed to the fruitful discussions at the AOSD '05 conference.

7. REFERENCES

[1] C. Allan et al. Adding Trace matching to AspectJ. Technical Report abc-2005-01 at http://aspectbench.org/.

[2] 4th International Conference on Aspect-Oriented Software Development (AOSD'2005), Chicago, IL, USA, March 2005.

[3] P. Avgustinov et al. abc: An extensible AspectJ compiler. In *AOSD '05, Chicago, IL, USA*, pages 87–98. ACM Press, 2005.

[4] P. Avgustinov et al. Optimising AspectJ. In M. Hall, editor, *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[5] Contract4J. http://www.contract4j.org/.

[6] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001.

[7] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416. IEEE Computer Society, 2001.

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

[9] Matthew Webster, IBM UK, personal communication.

[10] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[11] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[12] V. Stolz and F. Huch. Runtime verification of Concurrent Haskell programs. In *Proceedings of the Fourth Workshop on Runtime Verification*, volume 113 of *ENTCS*. Elsevier Science Publishers, 2004.

[13] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. N. Taylor and M. B. Dwyer, editors, *Twelfth International Symposium on the Foundations of Software Engineering (SIGSOFT FSE)*, pages 159–169. ACM, 2004.