Tracechecks: Defining Semantic Interfaces with Temporal Logic

Eric Bodden and Volker Stolz

Software Modeling and Verification (MOVES)
RWTH Aachen University, 52056 Aachen, Germany
{bodden, stolz}@i2.informatik.rwth-aachen.de

Abstract. *Tracechecks* are a formalism based on linear temporal logic (LTL) with variable bindings and pointcuts of the aspect-oriented language AspectJ for the purpose of verification. We demonstrate how tracechecks can be used to model *temporal assertions*. These assertions reason about the dynamic control flow of an application. They can be used to formally define the semantic interface of classes. We explain in detail how we make use of AspectJ pointcuts to derive a formal model of an existing application and use LTL to express temporal assertions over this model.

We developed a reference implementation with the *abc* compiler showing that the tool can be applied in practice and is memory-efficient.

In addition we show how tracechecks can be deployed as Java5 annotations, yielding a system which is fully compliant with any Java compiler and hiding any peculiarities of aspect-oriented programming from the user. Through annotations, the tracecheck specifications become a semantic part of an interface. Consumers of such a component can then take advantage of the contained annotations by applying our tool and have their use of this component automatically checked at runtime for compliance with the intent of the component provider.

1 Introduction

Existing programs, especially large-scale applications, do not only consist of their code base and documentation. In object-oriented programs, often there exist implicit constraints e.g. in library APIs on how methods or fields may be used. Apart from simple constraints like that certain parameters must never be null, there are more complex limitations that e.g. some methods may only be invoked in special circumstances, like in a specific order. Sometimes these constraints are already checked through assertions. But the unwary developer may be tripped up by many more patterns which are only informally documented and not enforced. For example in the Java5 libraries, if a collection is added to a hash set, the set does not notice changes to the elements themselves and may hence return unsound results.

In this work we present *tracechecks*, a formalism which we consider well suited to specify such temporal relations. The proposed semantic framework is based on *linear temporal logic* (LTL) [17], which is widely known in the field of formal verification, and is often used for static Model Checking [7].

W. Löwe and M. Südholt (Eds.): SC 2006, LNCS 4089, pp. 147–162, 2006.

The first step in Model Checking is usually to derive a formal semantic model from an existing application. This model is then checked for correctness with respect to some temporal specification (e.g. in LTL). Quite often it happens that the semantic model is unsound or incomplete with respect to the actual behaviour of the implementation.

Our approach is novel in the sense that we restrict ourselves to a partial model (the one induced by a *run*) and use AspectJ to derive this partial model. The primitives of our temporal logic are AspectJ pointcuts, picking out joinpoints in the dynamic control flow of a Java application. That way the model is known to match the implementation because they actually coincide at well-defined points—the joinpoints. Section 2 gives two motivating examples where tracechecks enforce temporal constraints on Java interfaces. In Section 3 we explain how we derive a semantic model of an existing application using AspectJ and how LTL can be used to state *temporal assertions* over this model. We show that the model is a system where transitions are triggered by pointcuts. In Section 4 we present the syntax of tracechecks and give their semantics by example. In particular, tracechecks can access and bind objects as the application runs, hence providing a means of instance-based reasoning. Section 5 discusses details about our reference implementation as well as performance and deployment issues important to component-based software development. We also comment on possible usage scenarios and conclude with a discussion of related work.

2 Motivation

Component based software has much evolved during the last years. Where some decades ago a piece of software often existed of few large chunks of code with little recognizable structure, today we have programming languages and tool support for properly maintaining independent components—modules—on their own. This modular reasoning has lead to safer software which is easier to maintain and easier to evolve.

Yet, we find that modules as they are today lack important specification features to be fully reusable, as they are frequently only syntactically defined through their programming language interfaces. This induces a purely static view. A feature f can be accessed through a module m if and only if f is in the signature of m and if it *can* be accessed, one can usually do so at *any time*. (Sometimes exceptions are used to forbid certain access patterns but we see this as quite a cumbersome low-level solution to the problem.)

We found that this static view can lead to trouble when software is actually run. Frequently it can happen that certain functionality is only available at certain points in time when an application executes, or in other words: at certain times at runtime, certain features should *not* be allowed to be accessed for the sake of a safe and stable application.

For example, nothing should be written to an output stream, if the stream has been closed before. Such errors may be documented in APIs in the form of comments, but still the user of the output stream component has to remember

```
1  tracecheck(Collection c, Iterator i) {
2
3   sym iterator(Collection c, Iterator i) after returning (i):
4       call(Collection+.iterator()) && target(c)
5   sym modify(Collection c) after returning:
6       (call(Collection+.add(..)) || call(Collection+.remove(..))) && target(c)
7   sym next(Iterator i) before:
8       call(Iterator.next()) && target(i)
9
10  G( iterator(c,i) -> G(modify(c) -> G(!next(i))) ) {
11    throw new ConcurrentModificationException ("Collection "+c+" modified!");
12  }
13 }
```

Fig. 1. Safe iterator tracecheck

to obey this rule in order to get a safely working application. With tracechecks, such temporal assertions can be specified *right in place* and *can automatically be checked* at runtime. To further emphasize this dynamic view we would like to give a code example.

2.1 Safe Iterators

As a motivation, let us start with the *safe iterator*-pattern, which states that:

> *For each Iterator i obtained from a Collection c, there must never be an invocation of i.next() after the collection has been modified.*

This pattern is actually enforced in the Java5 library as follows. The *Iterator* implementation contains a mechanism to track modifications of the underlying collection by means of a modification counter. If the collection c is updated, the modification-count obtained by the iterator i on instantiation time and the current counter of the collection disagree and lead to an exception on the next access to the iterator. In this case, the specification has crept into the implementation of both the iterator and the collection.

With this work we introduce *tracechecks*, a formalism and tool to formulate such trace conditions and automatically check their violation at runtime. Java interfaces and classes (as well as AspectJ aspects) can be annotated with tracechecks to define their behaviour with respect to the execution timeline.

In our formalism the requirement from above can be specified *in a modular way* through the tracecheck in Figure 1. Line 1 declares the free variables c and i that *each* collection and iterator in question will be bound to. Lines 3–9 declare three symbols *iterator*, *modify* and *next*, which match the relevant joinpoints through pointcuts. The actual formula (expressed in LTL, see below) is stated in line 10, specifying through the outer "Globally" that this assertion should be checked on the whole execution path (and hence for *all* created iterators). For each iterator (left-hand side of the outer implication), we require of the

```
1  tracecheck(HashSet s, Collection c) {
2
3   sym add(HashSet s, Collection c) after returning:
4     call(HashSet+.add(..)) && target(s) && args(c))
5   sym modify(Collection c) after returning:
6       (call(Collection+.add(..)) || call(Collection+.remove(..))) && target(c)
7   sym remove(HashSet s, Collection c) after returning:
8     call(HashSet+.remove(..)) && target(s) && args(c))
9   sym contains(HashSet s, Collection c) before:
10      HashSet+.contains(..)) && target(s) && args(c))
11
12  G( add(s,c) -> G( modify(c) -> remove(s,c) R (!contains(s,c))) ) {
13    throw new ConcurrentModificationException (c+" modified while in "+s);
14  }
15 }
```

Fig. 2. Tracecheck detecting inconsistent use of collections and hash sets

remainder of the execution that after a call to *add* or *remove* no call to *i.next()* must occur. The body is executed for any instance that violates the formula. We have successfully validated this formula in practice. All examples are available on our project web-page `http://www-i2.informatik.rwth-aachen.de/JLO/`.

2.2 Unsafe Use of HashSets

Another practical application of our framework is based on an actual bug pattern observed by colleagues. When a collection is inserted into a *HashSet*, modifications to the contained collections influence the result of *HashSet.contains*-queries. This behaviour was not anticipated and led to unexpected results. While this is only arguably a bug but rather a mistake, the source code had to be screened for possible uses under the wrong assumptions. In this case, the JDK does not provide any builtin mechanism to detect such behaviour. We captured it in the following way:

For each HashSet s that contains a Collection c, there must be no invocation of s.contains(c) if the collection has been modified, unless the collection has been removed from the set in between.

With tracechecks, specifying this property is done by a translation into linear temporal logic (see Figure 2). Again, we define symbols matching the events of interest and then specify that globally (**G**) adding a collection to a set implies that from there on always the modification of this collection implies that the removal of the collection from the set releases (**R**) the property "not check if c is contained in s" from holding. The φ **R** ψ indicates that either ψ should hold on the whole path or at some point φ holds and in this case releases ψ from the obligation to hold any longer.

Unlike the tracechecks in those two examples, there may be application-specific tracechecks that require understanding and analysis of the application. The examples here shall demonstrate that in many cases tracechecks can be seen as a formalism to extend the interface of an aspect or class (which is currently mainly structure based) with semantic properties. Moreover those properties can automatically be checked, leading to higher confidence in the code in question.

In the next section we explain how we use AspectJ pointcuts to obtain a trace of the running application and present the underlying foundations for checking LTL formulae on a finite path. In particular we clarify the relation between the execution trace and the *model of the program*. We show how the pointcuts used as propositions in our formula influence the degree of abstraction of the model and thus the trace.

3 Introducing LTL

Linear temporal logic reasons about an *infinite path* in a *model* (usually a *Kripke* structure) [7]. It is thus an extension of *propositional logic*. A path is a sequence of states $\pi = \pi[0]\pi[1]\ldots$ such that each edge $(\pi[i], \pi[i+1])$ is contained in the transition relation of the model. Each state $\pi[i]$ is labelled with a *set of atomic predicates* (the *propositions*). Although this section focuses on concrete examples, we briefly wish to point the reader to Figure 3, which gives the grammars for tracechecks and LTL formulae.

⟨TRACECHECK⟩ ::= [**perthread**]
  tracecheck (⟨VAR DECL⟩ **) {**
    ⟨SYMBOL DECL⟩+
    ⟨LTL FORMULA⟩
    ⟨METHOD BODY⟩
  }

⟨SYMBOL DECL⟩ ::=
  sym [**(** ⟨VAR DECL⟩ **)**]
    ⟨NAME⟩ ⟨KIND⟩**:** ⟨POINTCUT⟩**;**

⟨KIND⟩ ::= **before**
 | **after**
 | **after returning** [**(** ⟨VARIABLE⟩ **)**]
 | **after throwing** [**(** ⟨VARIABLE⟩ **)**]
 | ⟨TYPE⟩ **around** [**(** ⟨VARIABLES⟩ **)**]

(a) Tracecheck grammar

⟨ARG⟩ ::= ⟨SYMBOL⟩
 | ⟨LTL FORMULA⟩
 | **(** ⟨ARG⟩ **)**

⟨LTL FORMULA⟩ ::=

! ⟨ARG⟩	$\neg\ \varphi$	(*not* φ)
X ⟨ARG⟩	$\mathbf{X}\ \varphi$	(*neXt* φ)
F ⟨ARG⟩	$\mathbf{F}\ \varphi$	(*Finally* φ)
G ⟨ARG⟩	$\mathbf{G}\ \varphi$	(*Globally* φ)
⟨ARG⟩ **U** ⟨ARG⟩	$\varphi\ \mathbf{U}\ \psi$	(φ *Until* ψ)
⟨ARG⟩ **R** ⟨ARG⟩	$\varphi\ \mathbf{R}\ \psi$	(φ *Releases* ψ)
⟨ARG⟩ **&&** ⟨ARG⟩	$\varphi \wedge \psi$	(φ *and* ψ)
⟨ARG⟩ **||** ⟨ARG⟩	$\varphi \vee \psi$	(φ *or* ψ)
⟨ARG⟩ **->** ⟨ARG⟩	$\varphi \rightarrow \psi$	(φ *implies* ψ)
⟨ARG⟩ **<->** ⟨ARG⟩	$\varphi \leftrightarrow \psi$	(φ *iff* ψ)

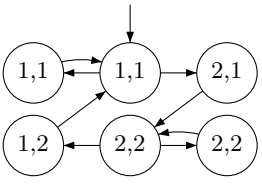
(b) Syntax of LTL formula

Fig. 3. Tracecheck and LTL grammar

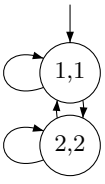
They consist of the set of Boolean operators as well as the temporal operators *Next*, *Finally*, *Globally*, *Until* and *Release*, which can be used to temporally combine propositions or sub-formulae.

```
x:=1; y:=1;
while (p1) {
  f(x,y);
  if (p2) then
    { x:=1; y:=1; }
  else
    { x:=2; y:=2; }
} /* while */
```

(a) Pseudo-source



(b) General model



(c) Abstracted model

Fig. 4. Simple *while*-loop with branching

For the verification of programs, these atomic propositions could be abstracted from each program state, i.e. the complete program state with heap, program counter, local variables, and call stack. Usually the program counter and a projection of parts of the state would be used to limit the model to the relevant propositions for the task at hand (cf. for example the specification language PROMELA [15]). The *model of a program* is defined by the generally undecidable set of all computation paths. We limit ourselves to reasoning about an *actual execution trace* of the program to overcome the inherent limitation of Model Checking on obtaining an appropriate model to existing source code.

Throughout the paper, the atomic propositions of our framework are pointcut expressions that select the matching joinpoints as the states of our abstract model. Each state is labelled with the set of *active propositions*, i.e. the propositions which match the current joinpoint. For example in the case of Figure 1, each state where an iterator i is created for a collection c would be labelled with a superset of $\{iterator(c, i)\}$. Although our examples only use `call` and `if`-pointcuts, any other pointcut, e.g. `cflow`, may be used.

3.1 Temporal Assertions

Reasoning about *one* such state is closely related to *assertions*. An assertion is the check of a predicate over the *current state* of a system (identified through the position in the source code). We can further abstract this to a model where we retain only those states in which assertions are actually checked by a tracecheck.

As an example, consider the program in Figure 4(a). It contains two predicates *p1, p2* that decide (possibly non-deterministically if for example I/O is involved) the number of iterations and which branch to take. Figure 4(b) shows the model we obtain if we are interested in the variables x, y. (We do not show the edges leading out of the loop.) Note that it contains two states labelled $(1, 1)$ or $(2, 2)$, but all are distinguishable from each other since they have different predecessors and successors. Figure 4(c) shows the abstracted model obtained if we are only interested in the values of the arguments of the method invocation f.

Temporal assertions use LTL path formulae as a means of reasoning about a *sequence of states*. They allow us to specify that states have to occur in a special order, e.g. that a call to a method f must eventually be followed by a method

call to g, expressed by the LTL formula $\mathbf{F}(f \rightarrow \mathbf{F}g)$. The operator $\mathbf{F}$ is often pronounced "Finally" because of its meaning.

Another important LTL operator is called "Globally". It specifies that a property should hold on every state of the model. E.g. it might be desirable to confirm that in each state the values of the variables x, y are equal: $\mathbf{G}\ (x = y)$.

We observe some differences between the models above (where LTL formulae have to hold on *all* paths) and a specific path. For the aforementioned program, $\mathbf{F}(x = y)$ holds on any infinite path in both models. $\mathbf{G}(x = y)$ does not hold in the general model because of the states $(2, 1)$ and $(1, 2)$. If we are evaluating a formula at *runtime*, we might observe a path where these states are not visited and the formula might hold *on this run*. Consequently we use a finite path semantics, i.e. over a single finite *unwinding* of a model.

We conclude that the level of abstraction the model provides is essential to its validity. By the appropriate use of pointcuts as propositions in our LTL formulae unimportant intermediate states can be filtered away, hence leading to an abstracted model as in Figure 4(c), where we filtered for method calls. Specifically, the abstract model is defined through the propositions in the formula. Hence we can now formulate the query "On all invocations of f, do x and y have the same value?": $\mathbf{G}\ (f \rightarrow (x\ =\ y))$. In our implementation, we would use a *call*-pointcut to select the method invocation and an *if*-pointcut to evaluate the predicate over the variables.

In the following, we discuss the remaining temporal operators which reason about intervals and need a more thorough discussion.

Until, Release and Next. The binary operators "Until" and "Release" can be considered the low-level operators of our temporal logic. The aforementioned operators "Finally" and "Globally" can be expressed using "Until" and "Release":

$$\begin{array}{rl|rl} \mathbf{F}\ \varphi & \equiv \mathbf{tt}\ \mathbf{U}\ \varphi & \mathbf{G}\ \varphi & \equiv \mathbf{ff}\ \mathbf{R}\ \varphi \\ \neg(\varphi\ \mathbf{U}\ \psi) & \equiv \neg\varphi\ \mathbf{R}\ \neg\psi & \neg(\varphi\ \mathbf{R}\ \psi) & \equiv \neg\varphi\ \mathbf{U}\ \neg\psi \\ \varphi\ \mathbf{U}\ \psi & \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi)) & \varphi\ \mathbf{R}\ \psi & \equiv \psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi)) \end{array}$$

The "Until"-operator $\mathbf{U}$ states that a formula $\varphi\ \mathbf{U}\ \psi$ holds in a state if the sub-formula φ holds from this state on until a state is reached where ψ holds. ψ is required to hold eventually, that is before the end of the program.

The dual operator "Release", $\varphi\ \mathbf{R}\ \psi$, specifies that either ψ should hold indefinitely or that ψ holds up to and including the state where φ holds. We already used this operator in the *HashSet*-example in the previous section: $\mathbf{G}(\mathit{modify}(c)\ \rightarrow\ \mathit{remove}(s, c)\ \mathbf{R}\ \neg\mathit{contains}(s, c))$.

A detailed discussion of the application of these specific operators is out of the scope of this paper and we point the interested reader to [18].

The last temporal operator is $\mathbf{X}$, the "Next"-operator. A formula $\mathbf{X}\ \varphi$ holds if φ holds in the next state, e.g. we might require that after pushing the start-button the engine should turn on through $\mathit{start} \rightarrow \mathbf{X}\ \mathit{running}$.

While LTL is only arguably an appropriate specification language, we consider it appropriate for a prototype. In the static verification community, several other specification languages like SUGAR [4] and FORSPEC [3] exist, which also contain additional syntactic sugar hiding the temporal logics in the semantic layer to make the input languages more user-friendly.

A comprehensive survey of existing verification patterns and how to express them in various specification formalism including LTL can be found in [11]. It can serve as a starting-point into specifying properties. The *HashSet*-requirement for example can be identified as a combination of the "Universality After"-pattern and a variant of the "Absence of P after Q until R"-pattern, where P is the *contains*, Q the *modify* and R the *remove*-action.

4 Tracechecks

The introductory examples show that tracechecks use an LTL formula with free variable bindings in order to specify conditions over the dynamic execution trace. Figure 3(a) gives the formal syntax of tracechecks. A tracecheck consists of a declaration of free variables which can be bound during evaluation, a nonempty list of symbol (proposition) declarations, an LTL formula declaration (cf. Figure 3(b)) and a body. The keyword *perthread* causes a thread-local instantiation of the formula, if a property should be checked for each thread separately.

A definition of the formal declarative semantics of tracechecks is out of the scope of this work and can be found in [5] where we also prove them equivalent to our operational semantics. In the following we want to explore the semantics *by example*, recalling the initial specification of the *iterator* requirement.

4.1 Quantification

The formula (with free variables c, i) can be written as:

$$\mathbf{G}(\mathit{iterator}(c,i) \rightarrow \mathbf{G}(\mathit{modify}(c) \rightarrow \mathbf{G}(\neg \mathit{next}(i))))$$

The informal requirement specification states that the condition $\mathbf{G}(\mathit{modify}(c) \rightarrow \mathbf{G}(\neg \mathit{next}(i)))$ should hold *for each pair* (i, c) *of iterator and collection*. With tracechecks, quantification *over objects* can be expressed by quantifying *over events*. Global quantification over a variable x can be modelled by wrapping a formula $\varphi(x)$ with a "Globally"-formula of the form $\mathbf{G}(\mathit{create}(x) \rightarrow \varphi(x))$. Likewise, existential quantification can be modelled by "Finally"-formulae of the form $\mathbf{F}(\mathit{create}(x) \wedge \varphi(x))$.

Tracechecks always specify a language of *valid traces*. That means that we are naturally interested in traces which *violate* the LTL formula of a tracecheck. A tracecheck body is executed whenever a formula is *falsified*. In cases where this falsification took place under a certain binding, this binding can be referred to by the variables declared in the tracecheck body (cf. Figure 6, line 9).

It may happen that no such binding is available. For instance the formula $\mathbf{F}(\mathit{create}(x))$, which states that at some point in time, some object x is created,

can only be falsified at application shutdown time. If it is falsified, this means that *create*(x) did not occur. Consequently, x cannot be bound. In such cases, x will be *null* in the tracecheck body. Future versions of our implementation will use static analysis in order to avoid accidental unchecked use of such variables.

4.2 Annotation Style Syntax

In addition to the tracecheck syntax, our implementation offers an inlined "annotation" style that can be used to deploy specifications as annotations in interfaces of ordinary classes (cf. Section 5). For the iterator example, this allows to directly attach the formula to the *iterator()* method of the *Collection* interface as shown in Figure 5. Note how the keywords *thisMethod* and *thisType* can be used to refer to the member respectively type the annotation is attached to. That way, formulae can be directly attached to the components they reason about in a reusable way. (Like in the AspectJ semantics, pointcuts over interfaces specify behaviour over *all classes implementing that interface*.)

```
1  interface Collection {
2
3  @LTL("thisType c, Iterator i:
4   G( exit(call(thisMethod) && target(c)) returning(i) ->
5    G( exit( (call(thisType+.add(..)) || call(thisType+.remove(..))) && target(c))
6      -> G(! entry(call(Iterator.next()) && target(i)) ) ) )
7  ")
8  Iterator  iterator ();
9
10 //remaining interface code
11 }
```

Fig. 5. Annotation style definitions in our prototype tool J-LO

Since tracechecks in annotation style have no body, if an error is detected, the implementation issues a message to a set of user definable observers. These may simply output an error message or apply some more sensible error handling, depending on the property. Also, such annotations are currently not automatically documented by Sun's `javadoc` API documentation tool. Future versions will likely support such a feature.

Using annotations as a means of deployment, the specification literally forms a (semantic) part of the public interface of a class. This can be useful for several purposes, comprising documentation, runtime checking (through our tool) but also static verification by third party tools. In particular, the *designer* of an interface, class or component can attach such semantic annotations to its code and have them compiled into Java bytecode. People *using* this class or component or implementing this interface respectively can then in a second, independent step simply apply our tool to have their implementation instrumented to be

checked for compliance with this semantic interface. We believe that this is a unique feature which has not been provided before in practice and that it is a major contribition to the modular deployment of components.

4.3 History Access Through *if*-Pointcuts

This syntax imposes one problem: Since there is no body available, one cannot perform any further computation on the bound values. In particular, one cannot filter for unwanted valuations. As a solution, tracechecks implement an extended semantics for *if*-pointcuts, giving them access not only to valuations at the current joinpoint but also to variables which have been bound earlier on the path. Figure 6 shows a tracecheck enforcing the *Singleton* design pattern [14].

```
1  tracecheck(Singleton s1, Singleton s2) {
2
3    sym create(Singleton s) after returning (s):
4      call(static Singleton Singleton+.inst ());
5    sym createAnother(Singleton s, Singleton t) after returning (s):
6      call(static Singleton Singleton+.inst ()) && if(s!=t);
7
8    G(create(s1) -> XG !createAnother(s2,s1) ) {
9     throw new SpecViolationException ("Two singletons detected:"+s1+","+s2);
10   }
11 }
```

Fig. 6. Tracecheck enforcing *Singleton* pattern

Note that the symbol *createAnother* gets a parameter t passed in (lines 5–6), which is not provided by the symbol itself. This raises the question what happens when one must decide if a condition such as $s \neq t$ actually holds at the current joinpoint, but one of the variables has not yet been bound. Indeed such formulae are forbidden. In [5] we explain a static analysis based on abstract interpretation which assures the validity of given formulae at compile time.

5 Reference Implementation

In this section we discuss some implementation details and how well tracechecks can be used in practice. We comment on the runtime overhead and explain possible deployment outside of aspects by using annotations. Our reference implementation is based on an adaption of alternating automata [13] with free variable bindings. It allows to implement the LTL semantics quite directly. Details are given in [5], so we only briefly outline important details.

Generally, for each tracecheck with n symbols we generate $n+1$ pieces of advice where the first n construct the set of propositions holding at a joinpoint and the

last advice triggers the automaton transition function. In addition, we generate a method containing the tracecheck body, which is called by the backend with the appropriate binding whenever the tracecheck fails. (The interested reader might want to have a look at the output of our tool during instrumentation as the prototype prints the generated aspects to the commandline.)

On startup of the instrumented application, the initial automaton configuration is installed in the runtime environment and then updated every time the aspect triggers a transition.

As mentioned, *if*-pointcuts in symbol declarations of a tracecheck can refer to variables which were bound earlier on in the trace. In order to evaluate an *if*-pointcut within the execution history, the compiler extracts the Boolean expression and constructs a closure which is attached to the defining proposition. The proposition is then passed in the correct variable binding at runtime through the evaluation of the transition function.

We have successfully tested our implementation with various assertions over data structures as well as on an instance of the lock order reversal pattern [18], where threads obtain locks in a way which may lead to a deadlock.

The work of Allan et al. [2] discusses an instance of the *safe iterator* pattern in *JHotDraw* (a Java drawing package, available at `http://www.jhotdraw.org/`) as use case. We were able to reproduce their results by executing a sequence of events violating the pattern in the graphical user interface. The error was properly picked up. If no instrumentation had been present, the error would probably have gone unnoticed. Step-by-step instructions along with all related code are available on our website.

5.1 Memory Overhead

With respect to memory leaks we made sure that our implementation uses strong references only where necessary, that is when variables are used within a tracecheck body (e.g. Figure 6, line 9). In those cases, strong references need to be kept in order to make sure that the object is still available when the tracecheck body is executed. For variables which are *not* referenced in such a way, each object bound to this variable can be garbage collected as if no tracechecks were present. All related propositions (weakly) referencing such objects are then automatically removed in the next application of the transition function. Their semantics is equal to those of *false*, because a proposition referencing an object that was garbage collected can never hold again.

As a result, when variables are not used within a tracecheck body (i.e. they are *collectable*), we observe only a constant memory overhead, since any bound object can be freed as usual. In particular this is always the case when using the annotation style syntax, because those specifications consist only of a formula and have no body which would require strong references to bound objects.

Figure 7 shows the memory consumption for our iterator example in *JHotDraw*. The left graph shows memory consumption for the version without instrumentation, when animating an object. Consumption is constantly around 1.3 MB. (Note that this is code compiled with the *abc* compiler. Code generated

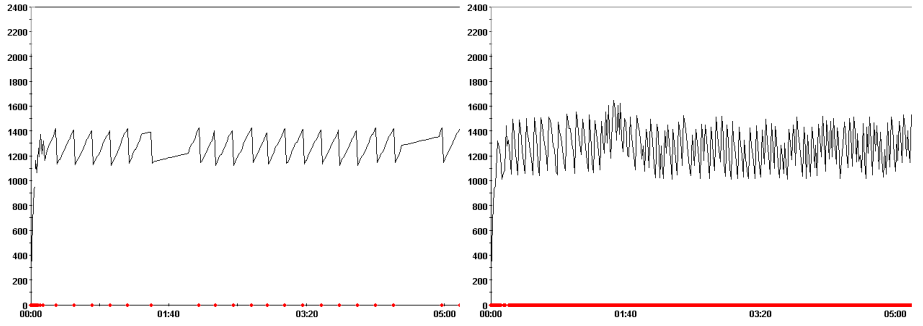


Fig. 7. Memory usage for the iterator example in JHotDraw; left: uninstrumented program (memory consumption in KB over time in minutes), right: instrumented version of the program; dots indicate garbage collection

by *javac* takes up about 2 MB.) The instrumented version shown on the right hand side shows the same constant memory consumption, however, it is triggering much more garbage collections than the original one. This is due to the fact that we have not yet optimised our application for speed. As a result, we still observe a considerable runtime overhead. Future versions will try to mitigate this problem by standard techniques such as caching. Yet the graph shall give proof of the fact that there are no memory leaks caused by our implementation.

When variables are used within a tracecheck body, Allan et al. [2] suggested a static pointer analysis which is able to identify such cases and hence may warn the user at instrumentation time. That way the user has the possibility to decide for himself whether he wants to pay for this debug information with the unavoidable memory overhead. This analysis is implemented in *abc* and can thus be reused for tracechecks.

If an application is instrumented with expensive tracechecks, they should only be active in internal debugging builds of the software, and disabled on deployment due to their performance-penalty at runtime. Test-case generation and path-coverage are essential to effectively use tracechecks. Despite the perceived overhead, with growing computer performance there is a trend to continuously monitor applications and eventually combine results of different runs to obtain asymptotic verification of analysed programs [6].

6 Related Work

Specifying aspects based on the execution history of a program has been recognized as a desirable feature for aspect oriented programming under the name of Event-based AOP (EAOP) [10]. The AOP approach which is closest to the one of tracechecks are *tracematches* introduced by Allan et al. [2].

6.1 Tracematches

They propose a matching language based on regular expressions. Those are quite different compared to LTL formulae in a way that regular expressions are well suited for *existential patterns*, i.e. patterns which anticipate certain behaviour to exist and then match this behaviour. While this is useful for the purpose of tracematches, which is using them as an *implementation language* where additional behaviour is attached to *existing* paths, the *absence of faulty behaviour* can consequently often only be expressed in a cumbersome way—by enumerating the language of all possible paths leading to an error state.

The use of LTL as a specification formalism allows here to translate *safety conditions* ("something bad never happens") in a more direct way. Such patterns are essential to checking and verifications as [11] shows.

Table 8 shows a comparison of tracematches and tracechecks: While tracechecks can be deployed in an AspectJ-like syntax, they can also be deployed as Java annotations, forming a real part of a Java interface.

Also, while tracechecks (i.e. LTL) allow the user to express negation and conjunction, this is not possible with tracematches (i.e. regular expressions). Even if the respective operations "intersection" and "complement" were added, tracematches would still not be equally expressive: For example, $a^* \cap b^*$ is only satisfied by the empty trace, while the property $\mathbf{G}(a) \wedge \mathbf{G}(b) = \mathbf{G}(a \wedge b)$ is also satisfied by the trace $[\{a,b\}\{a,b\}]$. This is due to the fact that regular expressions always have to be interpreted over *strict sequences of events*. That means that the aforementioned trace would be interpreted as a trace $[a\ b\ a\ b]$ or similar, which is not matched by $a^* \cap b^*$. Since LTL is a *propositional* logic, it can distinguish such *overlapping* events. Pure LTL in turn cannot detect patterns which require modulo counting (e.g. $(aa)^*$). We believe that such patterns are seldom useful in the context of verification.

Table 8. Comparing tracematches and tracechecks

	Tracematches	Tracechecks
Formalism	regular expressions	linear temporal logic
Deployment	AspectJ language ext.	AspectJ language ext. or Java annotations
Input symbol	$p \in \Sigma$	$\{p_1, \ldots, p_n\} \in 2^\Sigma$
Semantics	sequential	interleaving
Negation	implicit, through def. of Σ	explicit
Conjunction	no	yes
Concatenation	yes	no
Quantification	$\exists x$, implicit	$\exists x, \forall x$, through LTL
Shutdown	explicit	implicit

Also, the "Globally" operator, as we use it here, provides a means to universally quantify over variables (cf. Section 4). With tracematches this is not possible, since regular expressions are implicitly existentially quantified.

Last but not least, the shutdown event of an application needs to be explicitly modelled in tracematches, while our tool installs a shutdown hook, automatically notifying the verification runtime, when the application shuts down.

We conclude that tracematches and tracechecks show indeed some similarities, but in the end are both each better suited for their particular purpose.

6.2 Other Approaches

Temporal logics have already been used together with AOP: In [1], rules based on temporal logics are used to describe sequences of instructions where events should be inserted. The instrumentation happens on a static level and does not consider free variables.

Douence, Fradet and Südholt [8] developed an aspect calculus where advice can be triggered not only via a single joinpoint but via *sequences*. Although their work is targeted towards a formal model of joinpoint matching and advice execution and less on an actual implementation, there are clear similarities to our work. Their formalism describes regular sequences of joinpoints, so it can rather be compared to the sequential model of tracematches than to our's. Consequently, they cannot express overlapping events. It is implemented in the Arachne system [9], a dynamic weaver for C applications.

Other work by Südholt and Farias [12] discusses the use of explicit protocols in the interfaces of components in order to satisfy a certain notion of correctness. Hence the goal of their work is certainly similar to ours. Yet, they use another specification formalism (finite state machines) and do not employ and aspect-oriented programming. Consequently, they are unable to exploit the crosscutting nature of pointcuts, an essential stength of the formalism presented hete. Also, they provide no implementation.

Vanderperren et al. [19] propose the stateful pointcut language JAsCo also based on the above model. Pointcuts trigger transitions in a deterministic finite automaton and advice can be attached to each pointcut. JAsCo does not provide a means of quantification or bindings. These have to be implemented in the declaring aspect by hand.

In their work [20], Walkers and Viggers proposed the *tracecuts* formalism. As tracechecks and tracematches, tracecuts provide an AspectJ and pointcut based formalism for temporal reasoning. The authors describe an implementation by an AspectJ compiler extension, which gave of course some insights for our work. With respect to the formal model, the obvious difference to our approach is that tracecuts use context-free grammars for the specification of trace languages.

Additionally to context free languages, the set of languages recognisable by tracecuts is however even larger, since the implementation allows for the attachment of custom action blocks to each matching symbol. Such an action block has access to the whole execution history observed so far and can based on this decision reject the current symbol using a *fail* keyword, resuming as if the symbol had *not* just been read. A stack based implementation imposes additional overhead, although this might be optimised for regular traces.

Klose and Ostermann [16] discuss how temporal relations can be expressed in GAMMA, an aspect-oriented language on top of an object-oriented core language. Pointcuts are specified in a Prolog-like language and include timestamps that can be compared using the predicates *isbefore* or *isafter*. Their prototype requires a stored trace to analyse and is not applicable to an existing language.

In the field of annotation-based property checking, there are many tools around, such as Contract4J, JML, etc. but actually all of them support pure "Design by Contract", i.e. only pre- and postconditions and invariants. Each can be expressed through tracechecks, but tracechecks are more powerful as they allow to reason about the whole execution trace and not just a single point in the execution flow.

7 Conclusion

We have presented a specification framework for formal reasoning about object-oriented programs. The implementation is based on the *abc* compiler. Formulae in a temporal logic can be used to reason about the dynamic execution trace of a running application. The application is observed by an automaton where transitions are triggered by an aspect. Formulae can bind free variables to exposed objects on the execution path and can refer to those objects through a redefined scope of *if*-pointcuts during matching. They can be deployed using a language extension to AspectJ or by the means of Java5 annotations, yielding a fully Java compliant solution.

Through such annotations, the tasks of specification and verification is split into two parts: The designer of a component or interface adds annotations to his code representing the dynamic semantics of how the component is to be used. The annotations can then be compiled into bytecode and shipped to the user.

The user can then take advantage of the annotations for the purpose of documentation or automated runtime checking—by applying our tool. That way the user can make sure that his access to a component or implementation of an interface is compliant with the original intent of the component provider.

Our prototype, the *Java Logical Observer* ***J-LO***, together with all presented examples is available from `http://www-i2.informatik.rwth-aachen.de/JLO/`.

Acknowledgements. We thank the whole *abc* group for their useful comments on this work and on extending the AspectBench compiler in general.

References

1. R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. of Automated Software Engineering (ASE'03)*. IEEE, 2003.
2. C. Allan, P. Avgustinov, A. Simon, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA '05, San Diego, California, USA*, October 2005.

3. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A new Temporal Property-Specification Language. In P. S. Joost-Pieter Katoen, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*. Springer, 2002.
4. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*. Springer, 2001.
5. E. Bodden. J-LO, a tool for runtime checking temporal assertions. Master's thesis, RWTH Aachen University, Germany, 2005. Available from `http://www-i2.informatik.rwth-aachen.de/JLO/`.
6. T. M. Chilimbi. Asymptotic Runtime Verification through Lightweight Continous Program Analysis (invited talk). In *Fifth Workshop on Runtime Verification (RV'05)*. To be published in ENTCS, Elsevier, 2005.
7. E. Clarke Jr, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
8. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In G. C. Murphy and K. J. Lieberherr, editors, *Proc. of the 3rd intl. conf. on Aspect-oriented software development (AOSD'04)*. ACM, 2004.
9. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proc. of the 4th intl. conf. on Aspect-oriented software development (AOSD'05)*. ACM Press, 2005.
10. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the 3rd. intl. conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer, 2001.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st intl. conf. on Software engineering*. IEEE Computer Society Press, 1999.
12. A. Farías and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *In International Symposium on Distributed Objects and Applications (DOA)*. LNCS, 2002.
13. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. *Formal Methods in System Design*, 24(2):101–127, 2004.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *ECOOP'93—Object-Oriented Programming*, volume 707 of *LNCS*. Springer, 1993.
15. G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, Boston, Massachusetts, USA, September 2003.
16. K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Foundations of Aspect-Oriented Languages workshop (FOAL'05)*, 2005.
17. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symp. on the Foundations of Computer Science*. IEEE Computer Society Press, 1977.
18. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*. To be published in ENTCS, Elsevier, 2005.
19. W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In T. Gschwind and U. Aßmann, editors, *Workshop on Software Composition 2005*, volume 3628 of *LNCS*. Springer, 2005.
20. R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *Proc. of the 12th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering*. ACM Press, 2004.