J-LO
A tool for runtime-checking temporal assertions

J-LO
Ein Werkzeug zur Überprüfung temporaler Eigenschaften zur Laufzeit

von
Eric Bodden

Diplomarbeit

in Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im November 2005

angefertigt am
Lehrstuhl Informatik 2 für
Programmiersprachen und Programmanalyse
bei
Professor Dr. Klaus Indermark

Abstract

We present a Runtime Verification framework for Java programs called the *Java Logical Observer*, *J-LO* for short. 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. A novelty of our approach is that we provide a special form of LTL allowing free variables in propositions which can bind objects on the execution trace.

Zusammenfassung

In dieser Arbeit stellen wir ein *Runtime Verification Framework* für Java-Programme vor, den *Java Logical Observer*, kurz *J-LO*. Eigenschaften können in *Linear Time Logic* (LTL) über AspectJ Pointcuts spezifiziert werden. Diese Eigenschaften werden zur Laufzeit durch einen automatenbasierten Ansatz überprüft, in welchem Zustandsübergänge durch Aspekte ausgelöst werden. Unser Ansatz benötigt nicht notwendigerweise den Java-Quelltext der zu instrumentierenden Anwendung, da AspectJ auf dem Bytecode arbeitet und somit auch Anwendungen Dritter instrumentiert werden können. Die Hauptneuheit unseres Ansatzes besteht darin, dass wir eine spezielle Ausprägung der LTL bereitstellen, die es erlaubt freie Variablen in Propositionen zu definieren, die dann zur Laufzeit durch Objekte entlang des Ausführungspfades belegt werden.

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 31.10.2005 Eric Bodden

Acknowledgements

Much gratitude is due to the following people: My supervisor Volker Stolz, for the invaluable contributions he made and the immeasurable patience he showed throughout this work. Along with him, Michael Weber and Benedikt Bollig were tremendously helpful on the semantics side. I thank Prof. Klaus Indermark for supervising my thesis.

I am very grateful to the *abc* development team, for their support with their compiler framework and for the many fruitful discussions about my work.

Also I wish to thank all people of the Darmstadt Software Technology Group for giving me the chance to present my work and for their valuable comments which gave a first impulse to many considerations that were taken during the development of *J-LO*. I am particularly grateful to Mira Mezini for co-supervising this work.

I thank everyone who contributed to the fruitful discussions at the past conferences and workshops. Especially I want to thank everyone who was involved with the ACM Student Research Competition for their efforts.

I wish to thank the DFG for funding my trip to OOPSLA '04 and the European Network of Excellence on Aspect-Oriented Software Development for funding my attendance of AOSD '05.

Supporting graphics for this thesis were provided by Volker Stolz, Mik Kersten and the *abc* group for which I am very grateful.

Last but not least much gratitude goes to my friends and family who had to endure extensive discussion about various Runtime Verification techniques during the past year.

Eric Bodden

Aachen, November 2005

Contents

Organization of this thesis

This thesis is organized as follows:

Chapter 1 gives some reasons for the necessity of temporal assertions. Such assertions allow to verify temporal interdependency between events occurring during the execution of an application. We give examples from the literature as motivation and present problems which arise due to the constraint support for Runtime Verification in Java. We give an overview of how *J-LO* addresses those shortcomings.

Chapter 2 provides sufficient background knowledge to be able to follow the rest of this work. In section 2.1 introduce the basic techniques of formal verification including static approaches as Model Checking opposed to dynamic approaches from the field of Runtime Verification. We show up similarities and differences between both fields. Section 2.2 explains the concept of metadata in Java 5 and how it can be used to specify annotations in the Java source code. We introduce the various forms of annotations and justify our choice of metadata annotations in *J-LO*. Section 2.3 introduces the concept of aspect-oriented programming, its history and its most popular implementation in form of the programming language AspectJ. Section 2.4 explains how metadata annotations and aspect-oriented programming can be used in combination.

In chapter 3 we introduce the *Dynamic Linear Temporal Logic* (*DLTL*), a special kind of LTL with free variables in propositions which can be bound to objects along the execution trace at runtime. Section 3.1 explains its syntax, while in sections 3.2 and 3.3 we derive the declarative semantics. There might be formulae which are syntactically correct but whose static semantics are undefined. Section 3.4 introduces a static analysis for detecting such formulae. Section 3.5 explains the operational semantics and proves them equivalent to the declarational counterpart.

Chapter 4 is designated to the implementation of *J-LO*. We explain in detail how the operational semantics can be rendered into executable code. We give an overview of all employed tools and describe how we assure that despite the instrumentation *J-LO* performs we can guarantee that all runtime properties which can be specified in *DLTL* remain unaffected by this instrumentation.

In chapter 5 we explain how we tried to gain maximal confidence in this implementation by applying various metrics and give reasons for the theoretical performance of the algorithms implemented in *J-LO*. In addition we also provide benchmarks which allow to make assumptions about the overall actual performance of *J-LO*. In particular we discuss the problem of lock order reversal — a problem which may potentially lead to deadlocks.

Related work is discussed in chapter 6. Here we compare to related approaches in the fields of formal verification, Runtime Verification in particular as well as aspect-oriented programming. In case of the latter we focus on various new trace languages that have been proposed and compare those languages to *DLTL*.

We conclude this thesis in chapter 7.

The appendix lists own publications which were derived from this work, states some pitfalls we came across during our research, and contains a detailed explanation of the various types of pointcuts in the aspect-oriented language AspectJ. In addition, we briefly list the contents of the attached CD-ROM and give a list of symbols and notations.

Chapter 1

Motivation

1.1 Semantic Interfaces and Temporal Interdependencies

The goal of this project was to develop a tool which provides a convenient means of reasoning about the behaviour of an application at runtime. This raises the question how runtime behaviour is specified today, without such a tool.

A survey we conducted during this research showed that almost none of the programming languages around has rich support for verification built-in. The only concept Java provides are *assertions*: An assertion over a Boolean expression states that this expression has to hold during runtime, whenever the assertion is reached. It can be used for checking pre- and postconditions as table 1.1 shows. Line 10 implements the precondition `child != null`, which is informally stated in the documentation of the application interface (API) in line 7. If the application is started with the command line parameter `-enableassertions`, the control flow reaches this assertion statement, and `child` is `null`, an *AssertionError* is thrown by the Java runtime. If this command line parameter is not given, assertions are not taken into account. Disabled assertions impose no runtime overhead (cf. `http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html`).

With respect to software design, from the example from table 1.1 one can learn two important things: First of all, assertions in Java are restricted to localised reasoning. Without additional code, a single assertion can only refer to state which is visible to the currently executing object and available at the time the assertion itself is evaluated. As a result, temporal reasoning about the control flow of an application is impossible. Secondly, the assertion implements a check which is already informally stated in the API documentation just above the constructor declaration itself. Thus, the check is actually redundant. It could have been automatically inferred if the condition `child != null` had been

```
public class InnerNode implements TreeNode {

  private TreeNode child;

  /** Constructs a new inner node
   *  with child <code>child</code>.
   *  @param child The child node.
   *              May not be <code>null</code>. */
  public InnerNode(TreeNode child) {
    assert child != null;
    this.child = child
  }

  ...
}
```

Table 1.1: Java assertion checking for non-nullness

stated in the documentation in a formalized way. The tool we introduce will overcome both problems: It provides an expressive formalism which enables the notation of temporal assertions. Those assertions become part of the API documentation using Java 5 metadata annotations.

1.2 Motivating Examples

Motivating examples can for instance be found in [All02], which is an excellent article about temporal bug patterns. *Bug patterns* are patterns of recurring faults arising through common coding errors. *Temporal* bug patterns describe faults arising through misuse of objects or functions with respect to the time line. This will become clearer as we quote some of those bug patterns here.

1.2.1 A simple stack

The article [All02] states amongst others several requirements that typically have to be fulfilled when using a stack:

1. *Once `push(x)` occurs, `top()` will return `x` until a push or a pop occurs.*
   `Always{push(x) implies {{top()==x} until {push(y) || pop()}}`
2. *If the stack is empty, there should be no pops until a push occurs.*
   `Always{isEmpty() implies {{!pop()} until {push(x)}}}`

3. *Given we have a length operation, if the length is n, and a push occurs, then in the next step, the length will be n+1.*
   `Always{{length==n && push(x)} implies {Next{length==n+1}}}`

The above specification already tells us a lot about the behaviour of a stack. We want to use this small example as an example for what temporal conditions need to be checked in real world application and how *J-LO* addresses the implementation of the appropriate checks.

Please note that all of the above specifications could very well be stated right in a formal interface documentation. In *J-LO* this is exactly the case: Formulae are part of the public interface of a class. Thus they form documentation and test both in one piece. Such specification is what we call *implementation specific*: It is specific to certain classes and/or interfaces which are part of our particular implementation. There are also *generic* specifications, which should be true in general. For example it is common practice that an application should never deadlock. In *J-LO*, such a generic specification needs to be written down only once and can be checked by the tool without writing any line of code.

In the following section, we provide an overview of the architecture we employ to implement checking of temporal assertions as stated above.

1.3 An overview of our solution

From the user's point of view, *J-LO* works mostly transparent (cf. figure 1.2): The user supplies formulae to the system by annotating source code in appropriate places. In particular we use Java 5 annotations (see section 2.2), which are automatically compiled into the bytecode by any Java 5 compliant compiler. *J-LO* works then at build time as a simple preprocessor: As input it takes the annotated bytecode. This can in particular be supplied as a third-party library. *J-LO* then instruments the bytecode with runtime-checks that check if the stated assertions hold. Thus for the user, *J-LO* is just another tool in the usual build chain. The temporal assertions can then afterwards be verified by simply running the instrumented application.

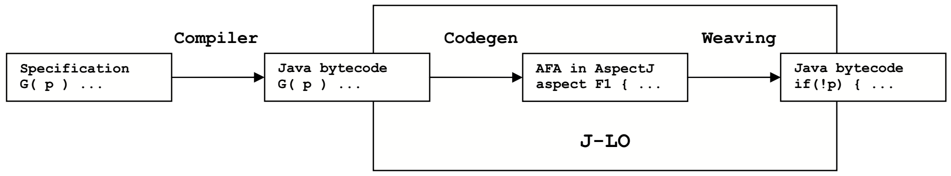


Figure 1.2: Workflow of *J-LO* usage

Chapter 2

Background

In this chapter we provide all the necessary background information that is required to understand the implementation details of *J-LO*. This comprises formal methods as Model Checking on the one hand as well as practical issues such as metadata and aspect-oriented programming on the other hand.

2.1 Formal verification

First we introduce the notion of *Model Checking*, which has similar goals as *runtime verification* and uses similar methods, however follows a purely static approach, which is more powerful in nature but unfortunately may often lead to performance problems one cannot easily cope with at the current time.

2.1.1 Static verification - Model checking

Clarke et al. define in [CGP99] the term *Model Checking* as

> *Model checking* [is a method] by which a desired behavioural property of a reactive system is verified over a given system (the model) through an exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviours that traverse through them.

So, as we learned, the input to a Model Checking process consists of two important parts:

1. The model. This shall here be given as a finite state system M.
2. A specification of a behavioural property, which shall here be given in the form of a finite set of temporal formulae $\Phi = \{\varphi_1, \dots, \varphi_n\}$.

The output of a Model Checking process is an answer `true` or `false` to the question *Does M satisfy Φ ?* or in other words: *Does $M \models \Phi$ hold?*

Depending on the kind of temporal formalism that is used, different Model Checking algorithms are applied. Here we want to focus on Model Checking for *linear temporal logic* (LTL) [Pnu77] in detail, since LTL is the formalism we employ for *J-LO*. Reasons for why we made this choice for LTL are given in the following subsections.

All such logics are typically defined over a *transition system* or *Kripke structure*. Thus we first want to introduce those notions.

2.1.1.1 Transition systems and Kripke structures

A *Kripke structure* over a set $\mathcal{P} = \{p_1, \ldots, p_n\}$ of propositions is a tuple

$$M = (S, R, L)$$

with

- S a finite set of states
- $R \subseteq S \times S$ a set of directed edges
- $L : S \to 2^{\mathcal{P}}$ a labeling function which labels each state with a (possibly empty) set of propositions.

The unlabeled structure (S, R) is a *transition system.*

For any vertex $s_i \in S$ with $L(s_i) = \{p_{i_1}, \ldots, p_{i_m}\} \subseteq P$ we say for each $p_{i_j} \in \{p_{i_1}, \ldots, p_{i_m}\}$ that p_{i_j} *holds in* s_i or short:

$$s_i \models p_{i_j}.$$

A *pointed Kripke structure* (M, s_0) is a Kripke structure M with a starting state $s_0 \in S$. Such a pointed Kripke structure typically represents the model which is to be verified by a Model Checking process. In the following when referring to the term *Kripke structure* we imply that this structure is pointed.

2.1.1.2 CTL*, CTL and LTL

Linear temporal logic [Pnu77] is a fragment of the richer *generalized computational tree logic CTL*.* Thus we first define CTL* and then derive the subset LTL and comment on its expressiveness.

CTL* is a propositional mathematical logic over Kripke structures as explained above. Its atoms are propositions reflecting the current state of a system. CTL* then combines those propositions using temporal and logical operators as well as path quantifiers. Our definition follows [TRW03].

Definition 2.1.1 (Syntax of CTL*)

- For each $p_i \in P$, p_i is a *state formula*.
- For state formulae φ and ψ, $\neg\varphi$, $\varphi \wedge \psi$ and $\varphi \vee \psi$ are state formulae.
- Each state formula is also a *path formula*.
- For a path formula φ, $\mathbf{E}\ \varphi$ and $\mathbf{A}\ \varphi$ are state formulae.
- For path formulae φ and ψ, we also have path formulae $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{X}\ \varphi$, $\mathbf{F}\ \varphi$, $\mathbf{G}\ \varphi$, $\varphi\ \mathbf{U}\ \psi$ and $\varphi\ \mathbf{R}\ \psi$.

All *state formulae* are valid CTL* formulae.

The above definition contains *path quantifiers* **E** (*Exists*) and **A** (*Always*), as well as *temporal operators* **X** (*neXt*), **F** (*Finally*), **G** (*Globally*), **U** (*Until*) and **R** (*Release*). Also we note a distinction between state formulae and path formulae. The former can be evaluated when focusing on a single state while the latter require one single path for evaluation. Temporal operators reason about states on a path. Thus they define the path formulae, whereas path quantifiers reason about sets of paths starting at a distinct state, and hence define state formulae. A CTL* formula is always evaluated at the starting state of a pointed Kripke structure. Hence only state formulae can be valid CTL* formulae.

Semantics of CTL*

The semantics of CTL* refer to the notion of a *path*. A path is defined as an infinite sequence of states

$$\pi = \pi[0]\pi[1]\ldots := (\pi[0], \pi[1], \ldots).$$

A path in a transition system $M = (S, R, L)$ adheres to the following condition:

$$\forall i \geq 0 : (\pi[i], \pi[i+1]) \in R.$$

For a clarified notation we also define π^i as the subsequence of π starting at the position $\pi[i]$:

$$\pi^i := (\pi[i], \pi[i+1], ...).$$

We define the semantics of CTL* inductively as follows:

For state formulae:

$(M, s) \models \mathbf{tt}$	(*true*)	
$(M, s) \not\models \mathbf{ff}$	(*false*)	
$(M, s) \models p_i$	iff	$p_i \in L(s)$
$(M, s) \models \neg p_i$	iff	$(M, s) \not\models p_i$
$(M, s) \models \varphi \wedge \psi$	iff	$(M, s) \models \varphi \wedge (M, s) \models \psi$
$(M, s) \models \varphi \vee \psi$	iff	$(M, s) \models \varphi \vee (M, s) \models \psi$
$(M, s) \models \mathbf{E}\ \varphi$	iff	$\exists \pi' : \pi'[0] = s \wedge (M, \pi') \models \varphi$
$(M, s) \models \mathbf{A}\ \varphi$	iff	$\forall \pi' : \pi'[0] = s \rightarrow (M, \pi') \models \varphi$

For path formulae:

$$\begin{array}{lll}
(M,\pi) \models \varphi & \text{iff} & (M,\pi[0]) \models \varphi) \ (\varphi \text{ state formula}) \\
(M,\pi) \models \neg\varphi & \text{iff} & (M,\pi) \not\models \varphi \\
(M,\pi) \models \varphi \wedge \psi & \text{iff} & (M,\pi) \models \varphi \wedge (M,\pi) \models \psi \\
(M,\pi) \models \varphi \vee \psi & \text{iff} & (M,\pi) \models \varphi \vee (M,\pi) \models \psi \\
(M,\pi) \models \mathbf{X}\ \varphi & \text{iff} & (M,\pi^1) \models \varphi \\
(M,\pi) \models \varphi\ \mathbf{U}\ \psi & \text{iff} & \exists k \text{ s.th. } (M,\pi^k) \models \psi \wedge \forall l\ (l < k) \rightarrow (M,\pi[l]) \models \varphi \\
(M,\pi) \models \varphi\ \mathbf{R}\ \psi & \text{iff} & \forall k\ (M,\pi^k) \models \psi \vee \exists l\ (l < k) \text{ s.th. } (M,\pi[l]) \models \varphi \\
(M,\pi) \models \mathbf{F}\ \varphi & \text{iff} & (M,\pi) \models \mathbf{tt}\ \mathbf{U}\ \varphi \\
(M,\pi) \models \mathbf{G}\ \varphi & \text{iff} & (M,\pi) \models \mathbf{ff}\ \mathbf{R}\ \varphi
\end{array}$$

This definition identifies **R** as the dual operator to **U**. It holds that:

$$(M,\pi) \models \varphi\ \mathbf{U}\ \psi \quad \Longleftrightarrow \quad (M,\pi) \not\models \neg\varphi\ \mathbf{R}\ \neg\psi$$

Example 2.1.2 (Kripke structure)
Given the Kripke structure of figure 2.1, we evaluate the following formulae at the state s_0:

formula	result
p_1 **U** p_2	no valid CTL* formula because it is a path formula
$\mathbf{AX}(p_1\ \mathbf{U}\ p_2)$	not satisfied (e.g. for paths $(s_0, s_1, s_0, \dots)$)
$\mathbf{EF}(\mathbf{AX}(p_1\ \mathbf{U}\ p_2))$	satisfied (e.g. paths $(s_0, s_2, s_3, s_4, \dots)$)

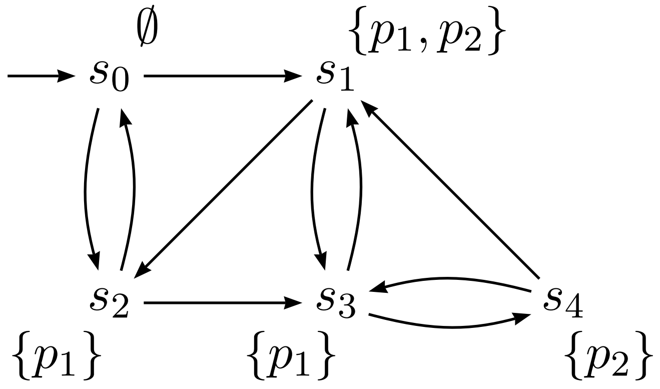


Figure 2.1: Example Kripke structure

For usual Model Checking, one identifies two fragments of CTL*, namely CTL and LTL which are strongly connected to the distinction between path formulae and state formulae above.

CTL is the *computational tree logic*. It is build up in the same way as CTL*, however temporal operators may not be cascaded. For instance *AGFp* is a valid formula in CTL* but not in CTL. As a consequence, such *fairness conditions* cannot be expressed in CTL. A CTL formula is evaluated at the starting state of a Kripke structure, just as in CTL*.

LTL is the *linear temporal logic*, the fragment of CTL* gained by removing the path quantifiers E and A. Thus, an LTL formula always reasons about the structure of a single path. A Kripke structure (M, s) satisfies an LTL formula φ if φ holds on *all paths* through M starting at s.

In the following we want to concentrate on LTL and see how Model Checking for an LTL formula can be performed.

2.1.1.3 LTL Model Checking

We define the LTL Model Checking problem as follows:

Given a Kripke structure (M, s) and an LTL formula φ, both over propositions $\{p_1, \dots, p_n\}$, check if $(M, s) \models \varphi$.

In addition, it is often desired that if $(M, s) \not\models \varphi$, the Model Checking process outputs a *counterexample*, a path through M which violates φ.

LTL Model Checking usually employs finite state machines called *Büchi automata*. A *Büchi automaton* is essentially an ordinary finite automaton but with an acceptance condition suitable for reading words of infinite lengths:

Definition 2.1.3 (Büchi automaton)
A nondeterministic Büchi automaton is a quintuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ with:

- Q finite set of states
- Σ finite alphabet
- $q_0 \in Q$ initial state
- $\Delta \subseteq Q \times \Sigma \times Q$ transition relation
- $F \subseteq Q$ a set of final states.

A *run* of $\mathcal{A}$ on an input word $\pi = (\pi[0], \pi[1], \dots) \in S^\omega$ of infinite length is an infinite sequence $\rho = (\rho_0, \rho_1, \dots) \in Q^\omega$ satisfying the following conditions:

- $\rho_0 = q_0$,
- $\forall i \geq 0 : (\rho_i, \pi[i], \rho_{i+1}) \in \Delta$.

We say that $\mathcal{A}$ *accepts* a path π if there exists a run ρ of $\mathcal{A}$ on π that visits states in F infinitely often.

Generally for any automaton $\mathcal{A}$, we define the *language recognized by* $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ as:

$\mathcal{L}(\mathcal{A}) := \{ \pi \mid \mathcal{A} \text{ accepts } \pi \} \subseteq S^\omega$

Based on those Büchi automata, the LTL Model Checking process can then be defined as follows:

1. Transform the given Kripke structure (M, s) into a Büchi automaton $\mathcal{A}_{(M,s)}$ recognizing the ω-language of all infinite paths through (M, s).

2. Transform the formula $\neg\varphi$ into an equivalent Büchi automaton $\mathcal{A}_{\neg\varphi}$.

3. Construct a product automaton $\mathcal{B}$ recognizing the language $\mathcal{L}(\mathcal{A}_{(M,s)}) \bigcap \mathcal{L}(\mathcal{A}_{\neg\varphi})$.

4. Check $\mathcal{B}$ for nonemptiness. If $\mathcal{L}(\mathcal{B}) \neq \emptyset$ then $(M, s) \not\models \varphi$ and every path $\pi \in \mathcal{L}(\mathcal{B})$ is a violating path for φ in (M, s). Otherwise (M, s) satisfies φ.

Step 1 is straightforward. The Kripke structure is simply interpreted as a Büchi automaton. Steps 3 and 4 are problems of basic automata theory. Step 2 however is nontrivial. Thus we will elaborate on the automaton generation a bit further.

The automaton generation usually happens in three steps: First the formula φ is converted to an *alternating automaton*. Then this alternating automaton is transformed into a *generalized Büchi automaton* which is then converted to an ordinary Büchi automaton in a last step.

The implementation of *J-LO* is entirely based on alternating automata, because Büchi automata represent a more abstract model which comes unhandy for our purposes (cf. appendix B). Thus we explain the conversion to alternating automata in detail. For the subsequent two conversions we point the interested reader to [Tho03].

2.1.1.4 From LTL to alternating automata

Our translation works similarly to the one described by Gastin and Oddoux [GO01]. First we want to define alternating automata in general. Then we define how we interpret such automata in our special setting.

Definition 2.1.4 (Alternating finite automaton)
An alternating finite automaton (AFA) is a quintuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ with

- Q finite set of states
- Σ finite alphabet
- $q_0 \in Q$ initial state
- $\delta : Q \times \Sigma \to 2^{2^Q}$ transition function

- $F \subseteq Q$ set of final states.

2^{2^Q} is here as usual the powerset of the powerset of Q. Those sets represent Boolean combinations in Disjunctive Normal Form (DNF). For instance $\{\{q_1, q_2\}, \{q_3\}\}$ represents the Boolean combination $(q_1 \wedge q_2) \vee q_3$. So a transition leading from q_0 to $\{\{q_1, q_2\}, \{q_3\}\}$ would mean a nondeterministic choice between moving simultaneously to q_1 and q_2 on the one hand or just to q_3 on the other hand. Using sets instead of Boolean expressions directly leads to easier semantics. In this representation, each *clause* (subset of Q) stands for one single run of $\mathcal{A}$. Note that although AFA allow for nondeterminism in this way, they are not actually nondeterministic themselves because δ is a *function*, mapping each state to a unique clause set of successor states.

A run on an AFA $\mathcal{A}$ is a directed acyclic graph over Q adhering to δ. $\mathcal{A}$ accepts an input path $\pi \in \Sigma^\omega$ if there *exists* a run on π, such that *all* branches of the run visit states of F infinitely often.

In the following, we want to adopt this automaton model to linear temporal logic. In order to do so, it is crucial to know that any LTL formula can be brought into *negation normal form (NNF)*. In this form, negations only occur in front of propositions. We define the function *nnf* as follows:

$$\begin{array}{rcl}
nnf : LTL & \to & LTL^{nnf} \\
\neg \mathbf{tt} & \mapsto & \mathbf{ff} \\
\neg \mathbf{ff} & \mapsto & \mathbf{tt} \\
\neg p & \mapsto & \neg p \\
\neg\neg\varphi & \mapsto & nnf(\varphi) \\
\neg(\varphi \wedge \psi) & \mapsto & nnf(\neg\varphi) \vee nnf(\neg\psi) \\
\neg(\varphi \vee \psi) & \mapsto & nnf(\neg\varphi) \wedge nnf(\neg\psi) \\
\neg \mathbf{X}\, \varphi & \mapsto & \mathbf{X}\, \neg nnf(\varphi) \\
\neg(\varphi\ \mathbf{R}\ \psi) & \mapsto & (nnf(\neg\varphi)\ \mathbf{U}\ nnf(\neg\psi)) \\
\neg(\varphi\ \mathbf{U}\ \psi) & \mapsto & (nnf(\neg\varphi)\ \mathbf{R}\ nnf(\neg\psi))
\end{array}$$

Here we assume that the operators $\mathbf{F}$ and $\mathbf{G}$ have already been reduced according to their semantics:

$$\begin{array}{rcl}
\mathbf{F}\, \varphi & \equiv & \mathbf{tt}\ \mathbf{U}\ \varphi \\
\mathbf{G}\, \varphi & \equiv & \mathbf{ff}\ \mathbf{R}\ \varphi
\end{array}$$

LTL^{nnf} is defined as the set of all LTL formulae in NNF. Given this negation normal form, we can now proceed with the specialization of our automaton model.

Definition 2.1.5 (AFA for an LTL formula φ)
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 $\varphi \in LTL^{nnf}$ over propositions in $\mathcal{P}$. Let the *closure of a formula*, $cl(\varphi)$, be the set of all sub-formulae of φ (plus the formulae **tt** and **ff**). Then

- $Q := cl(\varphi) \subseteq LTL^{nnf}$
- $\Sigma := 2^{\mathcal{P}}$
- $q_0 := \varphi$
- $F := \{q \in Q \mid q = (\varphi \mathbf{R} \psi)$ for some $\varphi, \psi \in LTL^{nnf}\} \cup \{\mathbf{tt}\}$.

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 transition function δ is derived directly from the definition of the CTL*/LTL semantics and recursively defined as follows:

Let $\mathcal{P}$ some finite set of propositions, $p \in \mathcal{P}, \varphi, \psi \in LTL^{nnf}$ formulae over $\mathcal{P}$. Then

- $\delta(\mathbf{tt}, \mathcal{P}) = \{ \emptyset \}$
- $\delta(\mathbf{ff}, \mathcal{P}) = \emptyset$
- $\delta(p, \mathcal{P}) = \begin{cases} \delta(\mathbf{tt}, \mathcal{P}) & \text{if } p \in \mathcal{P} \\ \delta(\mathbf{ff}, \mathcal{P}) & \text{otherwise} \end{cases}$
- $\delta(\neg p, \mathcal{P}) = \begin{cases} \delta(\mathbf{tt}, \mathcal{P}) & \text{if } p \notin \mathcal{P} \\ \delta(\mathbf{ff}, \mathcal{P}) & \text{otherwise} \end{cases}$
- $\delta(\varphi \wedge \psi, \mathcal{P}) = \delta(\varphi, \mathcal{P}) \bigotimes \delta(\psi, \mathcal{P})$
- $\delta(\varphi \vee \psi, \mathcal{P}) = \delta(\varphi, \mathcal{P}) \bigcup \delta(\psi, \mathcal{P})$
- $\delta(\mathbf{X}\ \varphi, \mathcal{P}) = \{\{\varphi\}\}$
- $\delta(\varphi\ \mathbf{U}\ \psi, \mathcal{P}) = \delta(\psi\ \vee\ (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi)), \mathcal{P})$
- $\delta(\varphi\ \mathbf{R}\ \psi, \mathcal{P}) = \delta(\psi\ \wedge\ (\varphi \vee \mathbf{X}(\varphi\ \mathbf{U}\ \psi)), \mathcal{P})$

Here $\bigotimes$ is defined as the clause product (derived by the laws of De Morgan): For two sets $s = \{s_1, \ldots, s_n\}$ and $t = \{t_1, \ldots, t_m\}$ of sets, we define

$$s \bigotimes t := \{\ s_i \bigcup t_j \mid 1 \leq i \leq n, 1 \leq j \leq m\ \}.$$

Also it should be noted that the calculation of $\delta(\varphi, \mathcal{P})$ is well-founded and all leaves are labelled with a subformula of φ. In particular, any AFA based on this definition is known to be *weak* as defined in [Roh97].

A weak automaton has a partially ordered state set, meaning there exists a partial order relation $\preceq$ over $Q = \{q_1, \ldots, q_n\}$ and a permutation $i_1, \ldots, i_n$ of $\{1, \ldots, n\}$ such that $\{q_{i_j} \preceq q_{i_{j+1}} \mid 1 \leq j < n\}$.

This weakness property is caused by the fact that each successor state of a state φ of $\mathcal{A}$ can only either be φ itself or a subformula of φ. In particular this means that there can be no nontrivial cycles during the evaluation of δ.

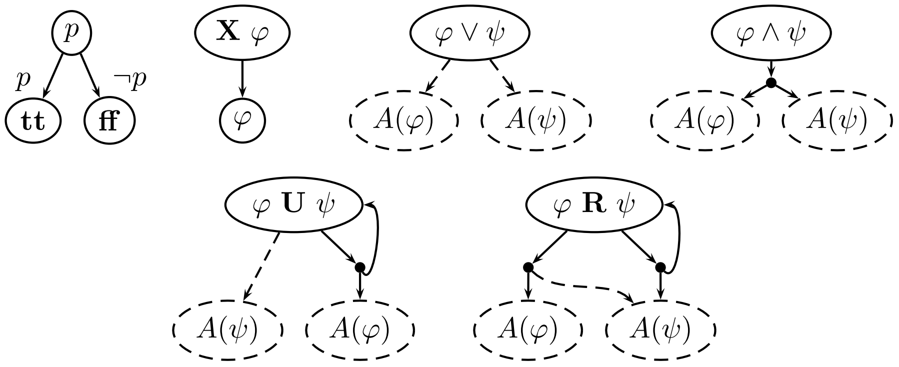


Figure 2.2: Transition function of an AFA over LTL

Figure 2.2 shows an informal graphical representation of how the transition function is calculated. Bullets represent conjunctive edges. The dashed nodes $A(\varphi)$ and $A(\psi)$ represent the automata which are gained through recursive application of the transition function to φ respectively ψ.

An example AFA for the formula **ff** $\mathbf{R}$ $(\neg p \ \vee\ (\mathbf{tt}\ \mathbf{U}\ q))$ is shown in figure 2.3. (Note that edges from propositions to **tt** respectively **ff** were omitted for a better overview.) In particular, the figure shall reflects the partial ordering of the state set, inducing the tree-like structure of the automaton.

2.1.1.5 From alternating automata to Büchi automata

The conversion from an AFA $\mathcal{A}$ to a Büchi automaton commences in two steps:

1. Create a *generalized Büchi automaton* $\mathcal{G}_{\mathcal{A}}$ equivalent to $\mathcal{A}$.

2. Create a Büchi automaton $\mathcal{B}_{\mathcal{A}}$ equivalent to $\mathcal{G}_{\mathcal{A}}$.

Those conversions are straightforward and out of the scope of this work. We point the interested reader to [Tho03].

Important is that in combination one has a method to calculate a Büchi automaton $\mathcal{B}_{\varphi}$ for each LTL formula φ. As pointed out in section 2.1.1.3, this

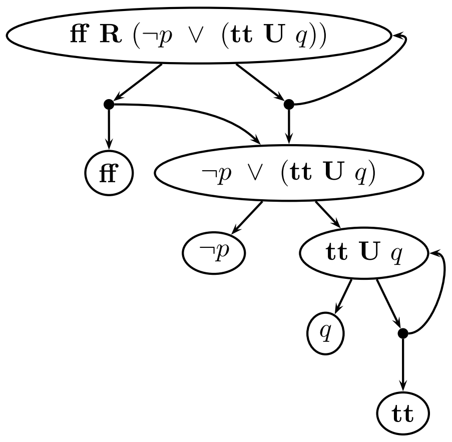


Figure 2.3: Example AFA for an LTL formula

procedure is then employed to calculate an automaton for the negated specification, $\mathcal{B}_{\neg\varphi}$, which is then combined with the model using a usual product construction.

We now take a quick excursion and have a look of what this looks like in terms of a fully-flavoured model checker.

2.1.1.6 LTL Model Checking in Spin

Spin [Hol04] is probably the most widely used LTL model checker today. Spin uses a process oriented modelling language, PROMELA – the *Process Meta Language*. The Spin book [Hol04] states:

> The specification language is intended to make it easy to find good abstractions of system designs. PROMELA is not meant to be an implementation language but a systems description language. To make this possible, the emphasis in the language in on the modeling of process synchronization and coordination, and not on computation. The language is also targeted to the description of concurrent *software* systems, rather than the description of hardware circuits (which is more common for model checking applications).
>
> The basic building blocks of SPIN models are asynchronous processes, buffered and unbuffered message channels, synchronising statements, and structured data. Deliberately, there is no notion of time, or of a clock: there are no floating point numbers, and there are only a few computational functions. These restrictions make it relatively hard to model the computation of, say, a square root in this language [...], but relatively easy to model and verify the behaviour of clients and servers in networks of processors [...].

```
 1 #define a 1
 2 #define b 2
 3 #define seen_a (last_seen == a)
 4
 5 chan ch = [1] of { byte };
 6 byte last_seen = a;
 7
 8 proctype A() {
 9     do
10     :: ch!a
11     of
12 }
13
14 proctype B() {
15     do
16     :: ch!b
17     of
18 }
19
20 proctype C() {
21             do
22             :: if
23 read_a:        :: ch?a -> last_seen = a
24 read_b:        :: ch?b -> last_seen = b
25                fi
26             of
27 }
28
29 init { atomic { run A(); run B(); run C() }
30 }
```

Table 2.4: A simple Spin example program

Promela features asynchronous communication via channels, deterministic and nondeterministic choice, continuous loops, guards and process abstraction.

Table 2.4 gives a small example of a model definition in Promela syntax. Line 5 defines a typed channel of size 1. Lines 8-12 and 14-18 define templates *A* and *B* for two processes which repeatedly write the output *a* respectively *b* to the channel. Lines 20-27 define a process template *C* for a process which repeatedly reads incoming input on the channel. When an *a* is read, *C* stores the read element in the variable *last_seen*. This will later be queried with the proposition *seen_a*. The *init* statement in line 29 is a special statement used to fork particular instances of the process templates defined before.

In Spin, propositions are Boolean expressions as e.g. defined by *seen_a* above. Those can be used in LTL formulae.

For example calling Spin with `spin -f '[]<> seen_a' model.prm` [1] specifies that proposition *seen_a* should *not* hold infinitely often, which is equal to the fact that a is not received infinitely often. Note that Spin already takes the negated formula $\neg\varphi$ as input.

In Spin, state labels such as *read_a* and *read_b* in process C represent propositions for LTL: At each state s, if s is labelled with $\{l_1, \ldots, l_n\}$, this means that in state s the propositions $\{l_1, \ldots, l_n\}$ hold. It is also possible to mark states as accepting: $s \in F \iff \exists\, l \in \{l_1, \ldots, l_n\} : l$ *starts with* `accept`. This allows for general queries of liveness conditions (*some accepting state has to be seen infinitely often*) and similar.

As stated in the previous sections, LTL Model Checking works by translating LTL to Büchi automata and then performing a product construction with the model. Spin does exactly this: For the LTL formula `[]<> seen_a` from above, it generates a so-called *never claim* as shown in table 2.5.

This never claim directly models a Büchi automaton: It consists of states `T0_init` and `accept_S1`. In each of those states, if `seen_a` is valid, the automaton switches to `accept_S1`. Otherwise (`(1)` stands for **true**), it switches back to the initial state. The state `accept_S1` is accepting, while `T0_init` is not, so in the end this automaton accepts exactly the language of state sequences where `seen_a` holds again and again. As the name suggests, a never claim must never become true for the specification to be fulfilled. This is consistent with the fact that Spin takes the negated formula as input.

When executed, Spin then combines all generated or explicitly stated never claims with the rest of the system specification and performs an exhaustive search over the state space (the emptiness check mentioned in section 2.1.1.3). The specification is violated if in any never claim it is possible to reach an *acceptance cycle*, a cycle holding an accepting state. When such a cycle is reached, it is clear that an accepting state is visited again and again. Thus the never claim becomes true. Spin outputs a path to the entry of the acceptance cycle as well as the cycle itself. This is called a *counterexample* for the specification.

2.1.1.7 From Model Checking to Runtime Verification

After this small digression about LTL in the field of static verification we now want to turn over to the evolving field of Runtime Verification (RV). What follows is a motivation of RV in general, followed by some peculiarities induced by the fact that RV usually is performed in a purely dynamic context.

[1] Spin follows a syntax where `[]` represents **G** and `<>` represents **F**.

```
never { /* GFseen_a */
T0_init :    /* init */
  if
  :: (seen_a) -> goto accept_S1
  :: (1) -> goto T0_init
  fi;

accept_S1 :    /* 1 */
  if
  :: (seen_a) -> goto accept_S1
  :: (1) -> goto T0_init
  fi;
}
```

Table 2.5: Never claim for *GF seen_a*

Apparently, Model Checking does not actually verify a real application. Rather it is a method of verifying a finite-state system (*model*) of such an application. Hence it is important that the model approximates the behaviour of the actual implementation as closely as possible. This however is where the usual complexity problems arise: While CTL Model Checking can be done in polynomial time, the Model Checking problems for LTL and CTL* are PSPACE hard [SC85, CES86], which means that the verification time is exponential in the size of the specification. Hence, although verification is usually linear to the size of the model, even checking relatively small models can take a long time while smaller models can be checked faster. Of course, the closer the model approximates reality, the less abstract it is and hence the more states it comprises. As a result, finite-state systems used in Model Checking are often either model small systems such as hardware controllers or they model an application on a very abstract level. The latter case of course makes the method incomplete: An application can still contain errors on a fine-grained level, which cannot be detected by verifying an abstract, coarse-grained model. This is where Runtime Verification comes into play.

2.1.2 Runtime Verification

Runtime Verification (RV) is a special field of runtime *testing* where test cases are generated from a formal specification. Thus RV shares with Model Checking the properties that both assume a given specification and check something for compliance with this specification. The major differences are the following:

1. While static approaches usually work on a model of a piece of software, Runtime Verification requires the actual application, simply because the

specification is checked against a running program (or a trace which was recorded at an earlier time).

2. As a consequence, paths in the world of Runtime Verification are usually finite, because the runtime of an application is finite: At some point the test is finished as the application is shut down. Reasoning abut infinite execution paths only makes sense if one is able to detect cycles in the execution flow, which is usually not possible with the Runtime Verification techniques that are available today.

3. While approaches like Model Checking usually aim to verify the behaviour of an application on all possible execution paths, in Runtime Verification one only observes *the one and only* execution path and checks the specification against it.

4. As a consequence, Runtime Verification is not actually real verification: Even when the specification is satisfied on all inspected paths, there might still be another violating path, which has simply not been tested yet. Therefore good path coverage is necessary to achieve a high reliability. As a consequence, Runtime Verification usually collaborates with unit testing [Bec00], which is aimed at good path coverage as well.

Due to those properties, *J-LO* takes specifications of the form of LTL formula with finite path semantics. LTL is suitable here, since it reasons about one single path, which is the current execution path in our case. Because the application is checked over a finite running time, the path is always finite and the semantics are defined accordingly.

J-LO instruments the actual application in such a way, that each given formula is checked during runtime. Formulae which state liveness conditions (such as *something happens again and again*) are evaluated over the whole path and their final state is reported during application shutdown.

Specification of those formulae takes place using *metadata annotations*. That way they become part of the public interface. The annotations are being introduced in the following section.

2.2 Java 5 Metadata

The *metadata* facility was introduced to Java in version 5, which corresponds to version 3 of the Java Language Specification [GJSB05]. Its specification took place in the Java Community Process (JCP) with the Java Specification Request 175 [JSR]. This document states:

> This facility allows developers to define custom annotation types and to annotate fields, methods, classes, and other program elements with annotations corresponding to these types. These annotations do not directly affect the semantics of a program. Development and deployment tools can, however, read these annotations and process them in some fashion, perhaps producing additional Java programming language source files, XML documents, or other artifacts to be used in conjunction with the program containing the annotations.
>
> Every annotation has an annotation type associated with it. In order to create an annotation type, you must declare it with an annotation type declaration. In addition to enabling a family of annotations, declaring an annotation type creates an interface that can be used to read those annotations. Annotation types can also be used in the definition of other annotation types, giving rise to annotation types with deep structure, and allowing substructures to be reused. Annotation types share the same namespace as ordinary class and interface types.

In *J-LO* we use the annotation type shown in table 2.6.

```
1 @Retention(CLASS)
2 @Target({CONSTRUCTOR,METHOD,TYPE,FIELD})
3 public @interface LTL {
4     String value();
5 }
```

Table 2.6: LTL annotation type in *J-LO*

The annotation type itself contains annotations which alter its applicability: Line 1 defines that the *Retention Policy* for annotations of this type shall be `CLASS`. This means that such annotations are persistently stored in the bytecode however are not to be made available to the reflection framework at runtime. Since *J-LO* instruments the application statically, this is fully sufficient. There are other Retention Types `SOURCE` and `RUNTIME`. The former advises the compiler to not even compile annotations into bytecode, while the latter leads to annotations which are still available in the bytecode and during the runtime of the application. The application can use reflection to retrieve annotations over a generated interface and alter its behaviour according to the semantics of the annotation.

Line 2 states the possible *Element types* which annotations of this type can be attached to. In *J-LO*, annotations can be attached to constructors, methods, types and fields. Other available Element Types are `PACKAGE`, `PARAMETER`, `ANNOTATION_TYPE` and `LOCAL_VARIABLE`, all of which seem not to be very suitable locations for the purpose of specifying formulae. It should be noted that annotations on local variables can only have `SOURCE` retention, since Java bytecode is stack based and thus local variables are not retained.

Lines 3 to 5 define the actual annotation type. Its name is `LTL` and its only parameter is a `String` of name `value`.

The annotation type can then be used as shown in table 2.7.

```
 1 class Foo {
 2     @LTL(value="<someFormula>")
 3     int field = 23;
 4
 5     @LTL("<someOtherFormula>")
 6     void bar() {
 7         ...
 8     }
 9     ...
10 }
```

Table 2.7: Example LTL annotation in *J-LO*

In line 2, the field is annotated with a value of `"<someFormula>"`. This could generally be any constant string. In order to be correctly parsed by *J-LO*, it will in our case have to adhere to the LTL syntax we define. `value` is actually a special label for an annotation parameter: If an annotation has only one parameter and its name is `value`, then this name can be omitted when instantiating an annotation. An example of this is shown in line 5.

So in our scenario, the user annotates constructors, methods, types and fields with formulae, which are then being compiled to bytecode using a standard compiler. *J-LO* extracts those annotations and applies the appropriate verification semantics to the application.

In order to do so, *J-LO* needs to employ instrumentation techniques. We use the aspect-oriented language AspectJ for this purpose, which we explain in the next section.

2.3 Aspect-oriented programming

The purpose of *Aspect-oriented programming* (AOP) is to separate crosscutting concerns. Such concerns are typically technical features that scatter throughout a given program and whose implementation is usually not part of the application core.

Such concerns typically include *tracing/logging*, *authentication*, *caching*, *transactioning* and so forth [Lad03, CCHW05]. Figure 2.8 [CCBH03] shows how the implementation of logging is originally implemented in the *Apache Tomcat* [Goo01] servlet container.

The dark horizontal bars mark regions of code where logging is implemented, while white regions comprise other source code. Classes shown in gray implement no logging at all.

As one can easily see, the logging code is scattered through about half of the classes. Using an aspect, all code concerned with logging could be separated into *one single* unit of code.

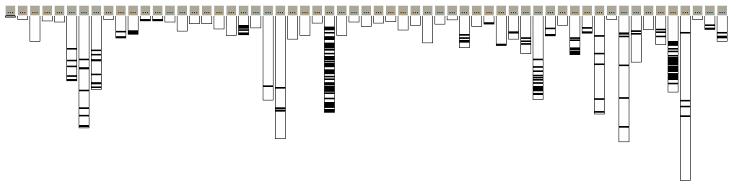


Figure 2.8: Logging as crosscutting concern in Tomcat

Aspect-oriented programming is performed using an AOP language, which is usually built as a language extension on top of one of the traditional functional, imperative or object-oriented programming languages.

Functional languages like LISP and SCHEME [CI91] tend to have support for AOP virtually built-in [Cos03] and indeed Gregor Kiczales, one of the pioneers of AOP, mentions [Lem04] that the idea of AOP originates from experiences with MacLisp.

However, the most widely used aspect-oriented programming language today is *AspectJ*, which is built on top of Java. It was originally developed by Xerox PARC[2] in the late 90's. Later on various companies and researchers contributed to its development. Especially IBM keeps pushing forward AspectJ till this date and provides powerful tool integration for several IDEs such as Eclipse[3], which originated from IBM and is now an independent open source project. At IBM, AOP is today in wide use in a production environment for application middleware products. According to Bill Gates, also Microsoft started embracing AOP technology during the last years [SFS02].

In the following we want to introduce the basic concepts of aspect-oriented programming by giving some examples in AspectJ. Afterwards we explain the basic semantics of AspectJ as they are necessary to understand the internal workings of *J-LO*.

[2]Palo Alto Research Center

[3]`http://www.eclipse.org/`

2.3.1 The anatomy of an aspect

In AspectJ, an aspect is an implementation of a *crosscutting concern*. Each aspect can be understood as a reactive unit: It consists of *pointcuts*, which tell *when* to react, and pieces of *advice* which tell *how* to react. Aspects interact with a base application during runtime at well-defined interaction points.

2.3.1.1 Joinpoints

Those points are called *joinpoints*. As we will see later on, a joinpoint is not actually a point but rather an interval in the dynamic control flow of an application. Examples are the execution of a method, the initialization of a class or write access to a field.

Sets of joinpoints can be described by pointcuts.

2.3.1.2 Pointcuts

A *pointcut* is a predicate over joinpoints. In AspectJ one can distinguish between the following classes of pointcuts:

- Context binding pointcuts (`this`, `target`, `args`)
  Those are used to expose context (objects) to the aspect for internal use.
- Kinded pointcuts (`call`, `set`, ...)
  Primitive pointcuts picking out join points of a certain kind (e.g. method calls, field accesses).
- Lexical pointcuts (`within`, `withincode`)
  When conjoined with other pointcuts, those pointcuts can restrict the set of matched joinpoints by lexical scopes.
- Control flow-based pointcuts (`cflow`, `cflowbelow`)
  Those restrict matching to joinpoints inside a certain control flow.
- Expression-based pointcuts (`if`)
  Those pointcuts can evaluate Boolean expressions and match based on the evaluation result.
- Boolean combinations
  For each two pointcuts `pc1` and `pc2`, `!pc1`, `pc1 || pc2` and `pc1 && pc2` are valid pointcuts as well. Their semantics correspond to finite set inversion, union and intersection respectively.

Each pointcut consists of a keyword (as above) depicting its kind, such as `call`, `execution` etc., and a set of brackets holding a pattern. This pattern can be of different types, depending on the kind of the pointcut:

- *TypePattern* This stands for a pattern over an arbitrary Java type signatures. Such patterns can contain the wildcard '`*`', which stands for an arbitrary sequence of characters allowed inside an identifier. Also they can use Boolean combinations of simple TypePatterns or the modifier '`+`' which matches all subclasses of a type. For instance `(Foo* && !foo.Bar+)` matches all types whose name starts with `Foo` and which are not a subtype of `foo.Bar`.

- *IDPattern* This describes a pattern over Java identifiers. This can contain wildcards as noted above.

- *FieldPattern* A field pattern describes a set of fields. Hence it has the form *ModifiersPattern TypePattern IDPattern*. For instance `public !static Number+ num*` would mean the set of all fields which are public but not static, of a subtype of `Number` and whose name starts with `num`.

- *MethodPattern* A method pattern describes a set of methods accordingly. It is of the form *ModifiersPattern TypePattern TypePattern '.' IDPattern '(' TypePattern ',' ... ')' ['throws' TypePattern]*. For instance the pattern `public boolean *.equals(Object)` matches all `public` methods returning a `boolean` defined in any (`*`) class ans taking a single argument of type `Object`.

- *ConstructorPattern* Such a pattern is meant to match a set of constructors. It is similar to the MethodPattern: The only differences are that there is no TypePattern for the return type and that the method identifier is fixed to `new`. So the pattern `protected Cloneable+.new(..)` matches all `protected` constructors of implementors of the `Cloneable` interface which take an arbitrary set of arguments. ('`..`' stands for a list of '`*`' of arbitrary length.)

The full list of available pointcuts in AspectJ is given in the appendix on page 132. Further information can be found in the AspectJ programming guide [Asp]. We will refer to those pointcuts in the rest of this work.

2.3.1.3 Example

Assume we have an implementation of the aforementioned stack, of which an excerpt is given by table 2.9.

In line 23, the `main` method constructs a new Stack with an initial capacity of `-1` which is immediately overriden by the value of `INITIALSIZE` inside the constructor at line 10. In line 25, `main` invokes `pop`, which raises an exception at line 14 due to the fact that the Stack is still empty. This exception is then handled in line 27 by dumping it to `System.err`.

```
class Stack extends ArrayList {

    final static int INITIALSIZE = 1;

    static int initialSize () {
        return INITIALSIZE;
    }

    Stack(int initialSize ) {
        super(initialSize>=0?initialSize: initialSize ());
    }

    Object pop() {
        return this.remove(this.size()−1);
    }

    ...
}

class Main {

    public static void main(String[] args) {
        Stack s = new Stack(−1);
        try {
            s.pop();
        } catch(Exception ex) {
            System.err.println(ex);
        }
    }
}
```

Table 2.9: Example implementation of a stack

Figure 2.10 shows a sequence diagram of the full execution of the `main` method. Classes (static context) are drawn as curved boxes while objects have sharp edges. The numbers are used to label points or regions (intervals) in the control flow. The next paragraph describes them in detail.

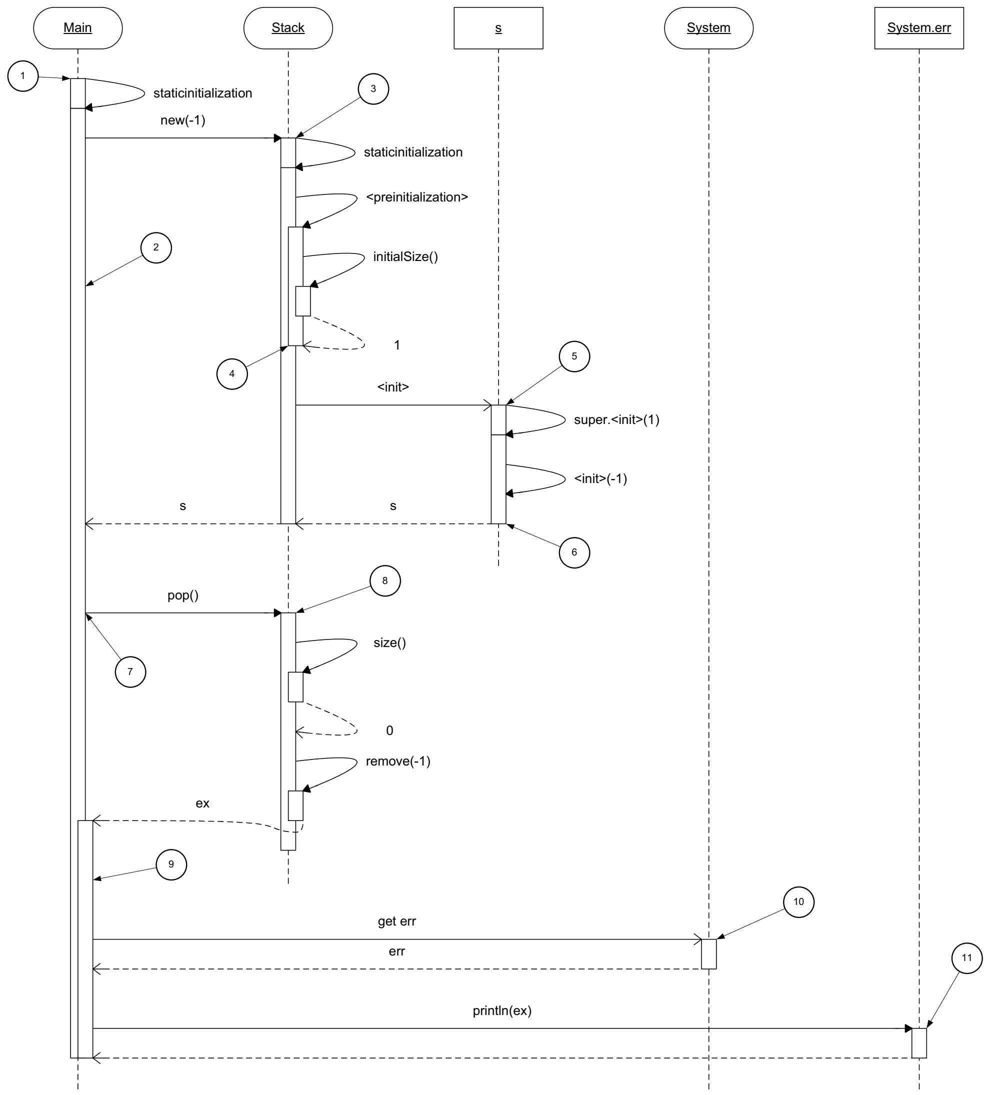


Figure 2.10: Example sequence diagram

We assume that we use AspectJ to instrument the two classes `Main` and `Stack` only. Then the following holds.

- The pointcut `staticinitialization(*)` matches regions 1 and 3.
- Region 2 is for instance matched by `execution(* *.main(String[]))`.

- The pointcut `cflow(execution(* *.main(String[])))` matches everything in the control flow of region 2, which includes any region shown here except region 1.
- The pointcut `cflowbelow(execution(* *.main(String[])))` matches the same as the last pointcut except region 2 itself.
- `preinitialization(Stack.new(*))` matches region 4.
- `initialization(Stack.new(*))` matches regions 5 and 6, while pointcut `execution(Stack.new(*))` matches 6 only.
- At region 5, `args(a)` binds `a` to the value `1`, while at region 6, it is bound to `-1`, because this is the argument value of the constructor of `Stack`.
- Since regions 5 and 6 both lie inside the context of `s`, here both, `this(t)` and `target(t)` binds `t` to the object `s`.
- Point number 7 is e.g. matched by `call(Object Stack.pop())`. Here `target(t)` binds `t` to the Stack instance `s`. `this(m)` does not match for any Object `m`, since point 7 lies in a static context and hence there is no executing object.
- Region 9 is e.g. matched by `handler(Throwable+)`, since `Exception` is a subtype of `Throwable`. At this region, `args(e)` binds `e` to the thrown exception `ex`.
- The pointcut `get(PrintStream *)` matches region 10.

Boolean combinations and invalid pointcuts

The aforementioned examples all comprised simple *kinded* pointcuts. Boolean combinations of pointcuts are used to quantify over joinpoints even further. For example, conjoining a pointcut `pc1` with a pointcut `pc2` narrows the set of matched joinpoints to the intersection of both:

```
pointcut pc(Foo f):  call(* *.foo(..))  && target(f);
```

This pointcut matches all calls to methods `foo` on types which are an instance of `Foo`. The call target is bound to `f`. Not all combinations, however, are valid. The following pointcut is invalid because it tries to bind f under negation. Since the semantics are equivalent to *all joinpoints where either not `foo` is called or the call target is no instance of `Foo`*, so there are joinpoints where `f` cannot be bound to an instance of `Foo`.

```
pointcut notpc(Foo f):  !(call(* *.foo(..))  && target(f));
```

The AspectJ semantics demand that all parameters must always be bound. Thus, this pointcut is invalid and an error is be given at compile time. The reader should keep this in mind, since it is also a crucial point of the semantics of *J-LO*.

2.3.1.4 Advice

As mentioned above, a piece of *advice* [Tei66] (or simply *advice* for short) is the functional unit of an aspect. An advice tells, *what* to do at a particular joinpoint. Hence, each advice consists of a pointcut specifying when the advice should apply, and an advice body that executes code at each joinpoint matched by the pointcut. In AspectJ there are five different kinds of advice:

Before advice This advice executes before each matched joinpoint. If the before advice throws an exception this can prevent the original joinpoint from executing.

After advice This advice executes after each matched joinpoint, regardless the fact whether the joinpoint returned normally or an exception was thrown.

After returning advice This advice executes after each matched joinpoint in the case where no exception was thrown. If there is a return value available, this can be exposed to the advice.

After returning advice This advice executes after each matched joinpoint in the case where an exception was thrown. The thrown exception can be exposed to the advice.

Around advice This advice can execute code before any matched joinpoint, then *may* optionally *proceed* with the original joinpoint and execute code after the original joinpoint has finished execution.

Example 2.3.1 (Advice)
Coming back to our example of a stack, one may find that it would be desirable to have a somewhat more precise exception message for the case where `pop` is invoked on an empty stack. The old code would just issue an inappropriate `ArrayIndexOutOfBoundsException`. The advice shown in table 2.11 shows how a more precise error message could be provided.

Line 1 declares an *around advice* returning an Object. This around advice captures joinpoints matched by `call(* Stack.pop()) && target(s)`, with `s` being bound to the call target.

In line 4, the advice invokes `proceed(..)`. This either calls the next advice matching the same joinpoint or the original joinpoint if there is no further

```
 1 Object around(Stack s):
 2   call(* Stack.pop()) && target(s) {
 3     try {
 4         return proceed(s);
 5     } catch (ArrayIndexOutOfBoundsException e) {
 6         if(s.size()==0) {
 7             throw new IllegalStateException(
 8                 "Don't use pop() when Stack is empty!"
 9             );
10         } else {
11             throw e;
12         }
13     }
14 }
```

Table 2.11: Advanced exception handling by the means of an around advice

matching advice as in our example. `proceed(..)` here gets the parameter `s`, which leaves the original call unchanged. One could have rerouted the call to another Stack by exchanging the parameter for another Stack object.

If this call returns without throwing an exception, the advice simply propagates the return value as stated in line 4.

However, every `ArrayIndexOutOfBoundsException` thrown by the stack implementation is caught in line 5. The stack, which has been bound to `s` is inspected further: If the size is `0`, it throws an *appropriate* semantic exception (lines 7-9). Otherwise, the original exception is thrown (line 11).

In that way, an aspect can handle the concern of exception handling as a separate, modular unit.

As mentioned in the description of `proceed(..)`, sometimes multiple pieces of advice can apply to the same joinpoint and in that case it may be important, which advice is executed first respectively last. This problem can be solved by defining an *advice precedence*. Understanding advice precedence is crucial to understanding *J-LO*. Thus we explain this mechanism in the following section.

2.3.1.5 Advice precedence

Advice precedence in AspectJ is defined in two layers:

The first layer defines what aspect precedes what other aspects. This is defined by a `declare precedence` statement. Such a statement takes a sequence of type patterns (see page 28) as arguments.

The following statement shows a `declare precedence` statement that gives any aspect matching the type pattern `*CachingAspect` higher precedence than any subclass of `LoggingAspect`.

declare precedence: ∗CachingAspect, LoggingAspect+;

The second layer defines precedence of pieces of advice inside one and the same aspect. Here precedence is based on the order in which pieces of advice are written down inside the aspect. The applied rules are indeed far from straightforward. Details can be found in the AspectJ documentation [Asp].

In the case of *J-LO*, we only employ *before* and *after advice*. If one limits pieces of advice to those and makes sure that all *before* advice precede all *after* advice in the textual ordering of each aspect, then those are executed in exactly this order at each joinpoint.

Apart from declaring precedence, in AspectJ one can also declare other things: For instance AspectJ supports open classes in a way that an aspect can declare members (fields or methods) on other classes. In the upcoming AspectJ 5, one can even declare Java 5 annotations on members or types. This is an interesting feature for *J-LO* and thus a feature we briefly want to explain.

2.4 AspectJ and metadata

Ramnivas Laddad's tutorial at JavaOne 2004 [Jav04] was titled *AOP and metadata: It takes two to tango.* This sentence is not just an empty shell as he explains: Aspects may well be used to both supply and consume metadata in a modular way.

2.4.1 Supplying metadata

In AspectJ 5 any aspect can supply metadata to any class by using the `declare annotation` statement. The following statement for example marks any method inside a class `Account` as `Authenticated`:

declare annotation: ∗ Account.∗(..)
: @Authenticated(permission="banking");

For *J-LO* this means that in the future specifications could be supplied from an arbitrary aspect declaring appropriate formulae to arbitrary classes.

2.4.2 Consuming metadata

On the other hand, aspects can consume metadata: The pointcut language was enhanced to allow matching on entities carrying a specific annotation. The following pointcut for example matches the execution of any method annotated with an `Authenticated` annotation.

pointcut authenticatedOps():
    execution(@Authenticated $*$ $*.*$(..));

Further details can be found in Laddad's DeveloperWorks article [Lad05]. This article was published within the series *AOP@Work* which is certainly an enjoyable reading for everyone who wants to get a practical insight into AspectJ and similar AOP languages.

This shall conclude our excursion to Metadata and aspect-oriented programming. A further note shall be given on *weak references*, special references which are used within the *J-LO* implementation. In order to be able to evaluate formulae with free variables being bound to objects, we have to reference such objects in the *J-LO* implementation. In order to not to prevent such bound objects from being garbage collected, we use weak references whenever possible.

2.5 Weak references in Java

An important feature of *managed code* environments such as the Java Runtime Environment is *garbage collection* (GC). Garbage collection assures that no memory is being held by the virtual machine for objects which are no longer accessible. Today there are various efficient garbage collection algorithms around (see [Jon96] for an overview). In the case of *J-LO* however, we wanted to make sure that the runtime verification does not interfere with GC: Objects should not be prevented from being garbage collected because this could lead to scalability problems.

Usual references in Java are *strong*. A strong reference to an object prevents this object from being garbage collected. Fortunately, since version 1.2, Java supports *weak* and *soft* references. Those are special container types that can hold one single strong reference to an object. Soft references only differ from weak references in some minor details which are not of concern for the implementation of *J-LO*. Hence in the following, we refer only to weak references as a shortcut.

If an object `o` references an object `p` over a weak reference, and `p` is not referenced by another *strong* reference, then `p` is available for garbage collection (it is assumed to be unreachable).

Additionally, each weak reference can be queried, if the referenced object is *still available*, has not been garbage collected yet. That way *J-LO* can track objects without strongly referencing them. Hence, monitored objects can be garbage collected as if *J-LO* was not present. If this occurs, *J-LO* treats this event in a well-defined way as we explain in section 4.4.2.

This concludes our presentation of background information. Based on this knowledge, we are now going to proceed with a detailed description of the syntax and semantics of our formalism.

Chapter 3

The syntax and semantics of *DLTL*

This chapter is organized as follows:

First we introduce the syntax that *J-LO* provides to define LTL formulae. Here we introduce free variables, which can be bound to objects during runtime.

In the next section, we introduce general finite path semantics for LTL. This is necessary, since formulae in Runtime Verification reason about a finite execution path. Our definition follows [HR01c] and has been frequently used in Runtime Verification literature.

In section 3.2.3 we explain in detail, why those semantics are insufficient in the case of *J-LO* where free variables may occur in propositions. We give reasons for why in *J-LO*, free variables are bound *over time*.

This leads to the necessity to partition an LTL formula φ into two parts $now(\varphi)$ and $next(\varphi)$, where the former has to hold at the current state and the latter on the subsequent path. This transformation is explained in detail in section 3.2.4.

Based on this notion of *now* and *next*, in section 3.3 we then define our full declarative semantics including a description of how free variables are handled.

The aforementioned definition of *now* and *next* assumes that in a given formula no variable is used before it is defined. Hence, section 3.4 presents a static analysis that allows to decide whether or not a formula fulfills this assumption.

Eventually, section 3.5 introduces the operational and prove them equivalent with the declarative semantics that have been described before.

So let us begin with the definition of the syntax. Given that we have a distinct semantics for our LTL formalism which is quite different compared to earlier approaches, we are going to refer to this special kind of LTL as *Dynamic LTL* (*DLTL*).

3.1 Syntax of *DLTL*

DLTL is a linear temporal logic over AspectJ pointcuts featuring dynamic binding of free variables. AspectJ pointcuts are, as explained in section 2.3.1.2, predicates over joinpoints, those being intervals in the dynamic control flow of a running application.

Temporal logic usually does not reason about *intervals*. The atomic unit of most temporal logics, including LTL, is a *state*. Hence we would like to be able to identify points in the dynamic control flow and interpret those as states. Fortunately, AspectJ gives us the opportunity to execute code both *before* and *after* each joinpoint (cf. section 2.3.1.4).

Making use of this feature, we do not define joinpoints as atoms of our logic but rather the entry and exit events of joinpoints. Distinguishing, as AspectJ does, between a *normal return* and a *return by exception*, this leads to the following syntax for propositions:

$$\begin{array}{rcl}\langle\text{proposition}\rangle & \longrightarrow & \texttt{entry(}\,\langle\text{pointcut}\rangle\,\texttt{)} \\ & | & \texttt{exit(}\,\langle\text{pointcut}\rangle\,\texttt{)} \\ & | & \texttt{exit(}\,\langle\text{pointcut}\rangle\,\texttt{)}\ \texttt{returning}\,\langle\text{identifier}\rangle \\ & | & \texttt{exit(}\,\langle\text{pointcut}\rangle\,\texttt{)}\ \texttt{throwing}\,\langle\text{identifier}\rangle\end{array}$$

The term constructors for formulae are defined as in usual LTL. Normally for LTL the operators U,X,$\neg$,$\vee$, and $\wedge$ suffice for full expressiveness. For convenience we allow the full set of LTL operators plus the operators $\rightarrow$ (*implies*) and $\leftrightarrow$ (*equivalent*).

$$\begin{array}{rcl}\langle\text{arg}\rangle & \longrightarrow & \langle\text{proposition}\rangle \mid \langle\text{formula body}\rangle \\ \langle\text{formula body}\rangle & \longrightarrow & \texttt{F(}\,\langle\text{arg}\rangle\,\texttt{)} \mid \texttt{G(}\,\langle\text{arg}\rangle\,\texttt{)} \\ & | & \texttt{X(}\,\langle\text{arg}\rangle\,\texttt{)} \mid \texttt{!(}\,\langle\text{arg}\rangle\,\texttt{)} \\ & | & \texttt{(}\,\langle\text{arg}\rangle\,\texttt{U}\,\langle\text{arg}\rangle\,\texttt{)} \mid \texttt{(}\,\langle\text{arg}\rangle\,\texttt{R}\,\langle\text{arg}\rangle\,\texttt{)} \\ & | & \texttt{(}\,\langle\text{arg}\rangle\,\texttt{||}\,\langle\text{arg}\rangle\,\texttt{)} \mid \texttt{(}\,\langle\text{arg}\rangle\,\texttt{\&\&}\,\langle\text{arg}\rangle\,\texttt{)} \\ & | & \texttt{(}\,\langle\text{arg}\rangle\,\texttt{->}\,\langle\text{arg}\rangle\,\texttt{)} \mid \texttt{(}\,\langle\text{arg}\rangle\,\texttt{<->}\,\langle\text{arg}\rangle\,\texttt{)}\end{array}$$

Here "!" means negation, "||" the nonexclusive *or*, and "&&" means *and*.

In *J-LO* propositions may define and access free variables that bind objects at runtime. Those variables have to be typed in order to allow for static type checking and to adhere to the fully typed AspectJ semantics. Thus, free variables need to be declared with a type in front of the formula:

$$\langle\text{formula}\rangle \quad \longrightarrow \quad [\langle\text{formal parameter list}\rangle :]\ \langle\text{formula body}\rangle$$

```
 1 Stack s:
 2 G(
 3  (
 4   exit( call(Stack.new(..)) ) returning s
 5  ) −> (
 6   X(
 7    F(
 8     entry( call(∗ Stack.push(Object)) && target(s) )
 9    )
10   )
11  )
12 )
```

Table 3.1: Stack example in *DLTL*

Table 3.1 gives a short example of what such a formula could look like.

Line 4 specifies a proposition which holds at the exit event of a constructor call. Further, it binds the free variable s (declared in line 1). Line 8 specifies the entry event of a call to `push` with call target s.

The formula states that *globally* whenever a new Stack s is constructed, then *finally* `push` is invoked on s.

Having this first initial picture of how properties can be specified with *J-LO*, let us now move to the semantics of such a specification and let us define what it actually means for an LTL formula to *hold* on a *finite* path. The basic idea is that all *safety* requirements ("something bad never happens") are restricted to the given finite path and all eventualities (lifeness conditions - "something good eventually happens") have to be fulfilled before the path ends.

3.2 Towards a declarative semantics

3.2.1 Notation

For propositions in general we write $p, q, \dots \in \mathcal{P}$.

Such propositions may bind free variables. For instance a proposition `exit(* call(A.foo()) && target(x))` binds the variable `x` — we say it *defines* a value for `x`. We write $p(\vec{x}), q(\vec{y}), \dots$ for propositions p and q defining variables $\vec{x} := \{x_1, \dots, x_n\}$ respectively $\vec{y} := \{y_1, \dots, y_m\}$.

Also, proposition may refer to variables, which have been defined earlier on a path. For instance a proposition `exit(* call(A.foo()) && target(x) &&`

`if(x!=y))` refers to a variable `y`, which is not at the same time defined by the proposition. We say that the proposition *uses* `y`. We underline used variables and so write $p(\vec{x}, \underline{\vec{x'}})$ for a proposition p defining variables $\vec{x}$ and using $\underline{\vec{x'}}$.[1]

A *state* shall in our semantics be identified with the propositions that hold at this state. Hence, we define the state set S by $S := 2^{\mathcal{P}}$.

3.2.2 General finite path semantics

Let $\mathcal{P}$ be a set of atomic propositions and $\pi = \pi[0]...\pi[n-1] \in S^n$ a finite path with $n \geq 0$. For each path position $\pi[i]$ $(0 \leq i < n)$ and proposition $p \in \mathcal{P}$ and formulae φ and ψ:

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

Note that formulae $\varphi\ \mathbf{R}\ \psi$ hold on the empty path with $n = 0$ whereas **U**-formulae do not. Here, still the following equations hold:

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

Also, still for each LTL formula with finite path semantics there exists an equivalent formula in negation normal form with solely $\neg, \mathbf{X}, \mathbf{U}, \mathbf{R}$ operators and negation only in front of propositions, just as over infinite paths (cf. section 2.1).

Remark 3.2.1 (General assumption)
In the following, we want to assume that φ is in negation normal form. This enables us to restrict definitions to the above operators. It is straightforward

[1] It shall be noted that this distinction between variable definitions and uses is only necessary for the declarative and operational semantics. Opposed to implementations as e.g. *HAWK* [dH05, HBS03a, HBS03b, HBS04] (cf. section 6.2.2), *J-LO* automatically infers whether a variable is used or defined by a given proposition. This is done by the means of a static analysis explained in section 3.4.

to extend any of the definitions to further operators as **F** and **G** or the *weak until* operator **W**, which is frequently used in related literature and is defined as $\varphi\ \mathbf{W}\ \psi := (\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\ \varphi$.

The above definition of finite path semantics has been used in various publications in the field of Runtime Verification during the past years [HR01c, HBS03a, HR04]. Initial experiments during the development of *J-LO* successfully used the very same semantics without any changes. However, when introducing the possibility of free variables in propositions, we found that those semantics lead to problems. The question to be answered turns out to be the following:

For a proposition $p(\vec{x})$ with free variables $\vec{x}$,
when does $p(\vec{x})$ hold at a state $\pi[i] \in 2^{\mathcal{P}}$?

The next section explains this problem in detail.

3.2.3 Why usual quantification semantics are insufficient

Usually, when dealing with free variables in mathematical logics of any kind, the idea is to bind those variables by implicit or explicit quantifications so that they are not free any more:

Given a variable x with possible valuations over a *finite* domain DOM and a formula $\varphi(x)$ where x occurs *free* in the usual meaning. Then the semantics of $\varphi(x)$ can simply be defined through quantification.

For universal quantification: $[\![\varphi(x)]\!] := [\![\forall x.\varphi(x)]\!] = \bigwedge_{a \in DOM} [\![\varphi(a)]\!]$

For existential quantification: $[\![\varphi(x)]\!] := [\![\exists x.\varphi(x)]\!] = \bigvee_{a \in DOM} [\![\varphi(a)]\!]$

The reader should note the following:

1. $\varphi(a)$ is a formula without any free variables. Thus its semantics are clear.
2. For the above approach, it is essential that $|DOM| < \infty$, because otherwise the equations do not hold.

The second point is the essential reason for why this approach is not feasible in the case where propositions contain free variables. The question is: What is DOM?

A first idea would be to define DOM as the set of all *objects on the heap* of a Java application. This however imposes several problems: First of all, this set can be of arbitrary size. Theoretically, there is no limit on how many objects can be instantiated. This size only depends on the available amount of memory.

Of course one could still argue that the available amount of memory is always limited and thus we yet have a finite domain. However, even then it would not be possible for any Java application to conduct any proof over that domain, since no such program has direct access to the heap[2] and hence the domain cannot be enumerated.

A second approach, which *is* feasible and indeed was taken in an early Haskell prototype [SH04], would be to define DOM as the set of all valuations of free variables which occur during the execution of the path, and then quantify over this domain.

For example given the formula $p(x) \vee q(y, \underline{x})$ and the path $\pi = \{q(1, \underline{x})\}$ we could say that possible valuations are $(x, y) \in dom(x) \times dom(y) = \emptyset \times \{1\}$.

The Haskell prototype uses universal quantification, so we get:

$$\begin{array}{lll} \pi & \models & p(x) \vee q(y, \underline{x}) \iff \\ \pi & \models & \bigwedge_{x' \in dom(x)} \bigwedge_{y' \in dom(y)} p(x) \vee q(y, \underline{x}) \iff \\ \pi & \models & \bigwedge_{x' \in \emptyset} \bigwedge_{y' \in \{1\}} p(x) \vee q(y, \underline{x}) \iff \\ & \mathbf{true} & \end{array}$$

Given that without taking variables into account, certainly **false** should be issued, this result is somewhat undesired. Note that in the one and only state $q(1, \underline{x})$ holds. Actually it would be open to debate, if a proposition $q(y, \underline{x})$ holds at this state or not, due to the use of the undefined value of x. Is this constraint fulfilled or not? Quantification takes away this complexity: Since in the example the domain of x is empty, there is nothing to check.

In Haskell this behaviour is natural because formulae are modeled by lambda functions, which are evaluated lazily: A formula $\varphi(x, y)$ is actually a function $\lambda x \lambda y.\varphi(x, y)$. Such a function cannot be evaluated before x *and* y have a defined value.

The approach however, suffers from two problems in practice:

1. We are bound to universal quantification. This could naturally be solved by making quantification explicit, penalizing simplicity of the syntax.

[2]There is a chance of getting a handle to all objects by instrumenting the constructor execution of `java.lang.Object`. However, this means instrumenting the Java Runtime Library, which is not always desirable, nor would it be compliant with the Sun License (see `http://java.sun.com/j2se/1.5.0/`). Also it would doubtfully be efficient to do so.

2. Since the domain of any variable is determined by the valuations of this variable over the whole path, this whole path must be known in order to fully evaluate a formula. This is a major restriction, given that one of the design goals of *J-LO* is early fault detection.

The second problem was the reason for defining the semantics of *J-LO* in a dynamic way. The semantics of a *DLTL* formula are *not* defined over a finite domain, which is to be known in advance. Rather, the domain establishes itself as one walks along the path, binding valuations as we go. In order to be able to do so, we show that each LTL formula φ can be split into two parts, $now(\varphi)$ and $next(\varphi)$, with respect to a state, so that at this state, φ holds iff both $now(\varphi)$ and $next(\varphi)$ hold.

However, this approach forbids formulae as the one above, where propositions *use* variables which are still unbound at the time they are to be evaluated. Hence, in chapter 3.4 we present a static analysis that is able to detect such formulae.

3.2.4 Transformation to *now* and *next*

The declarative semantics of *DLTL* are based on the idea that valuations are collected over time and in this way form the domain over which we check.

Essential to the idea are the notions of *the current state* and *future states*. Each LTL formula can be thought of a partition of subformulae, each talking about either the current state or the rest of the path.

3.2.4.1 The notion of *now* and *next*

An important observation is that any LTL formula φ can be partitioned, with respect to the current state π into two formulae $now(\varphi)$ and $next(\varphi)$ in such a way that $\pi \models \varphi$ iff $\pi \models now(\varphi)$ and $\pi \models next(\varphi)$.

In the following we assume a path $\pi = \pi[0], \ldots, \pi[n-1]$ with $n > 0$.

Definition 3.2.2 (Function *now*)
The function $now : LTL \rightarrow LTL$ is recursively defined as:

$$\begin{aligned}
now(p) &:= p \\
now(\neg p) &:= \neg now(p) \\
now(\mathbf{X}\ \varphi) &:= \mathbf{true} \\
now(\varphi \wedge \psi) &:= now(\varphi) \wedge now(\psi) \\
now(\varphi \vee \psi) &:= now(\varphi) \vee now(\psi) \\
now(\varphi\ \mathbf{U}\ \psi) &:= now(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
&= now(\psi) \vee now(\varphi) \\
now(\varphi\ \mathbf{R}\ \psi) &:= now(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
&= (\ now(\psi) \wedge now(\varphi)\ ) \vee now(\psi) \\
&= now(\psi)
\end{aligned}$$

Note that for any φ the result of $now(p)$ is a Boolean combination of propositions. The function $now(\varphi)$ reflects that part of φ that can be fully evaluated in state $\pi[i]$, under the assumption that φ holds on the subsequent path.

Definition 3.2.3 (Function *next*)
For $0 \leq i < n$, the function $next : LTL \rightarrow LTL$ is recursively defined by $next(\varphi) := \mathbf{X}\ next'(\varphi)$ with $next'(\varphi)$ defined as:
If $i < n$ then:

$$\begin{aligned}
next'(p) &:= \begin{cases} \mathbf{true} & \text{if } \pi[i] \models p, \\ \mathbf{false} & \text{otherwise} \end{cases} \\
next'(\neg p) &:= \neg\ next'(p) \\
next'(\mathbf{X}\ \varphi) &:= \varphi \\
next'(\varphi \wedge \psi) &:= next'(\varphi) \wedge next'(\psi) \\
next'(\varphi \vee \psi) &:= next'(\varphi) \vee next'(\psi) \\
next'(\varphi\ \mathbf{U}\ \psi) &:= next'(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
next'(\varphi\ \mathbf{R}\ \psi) &:= next'(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi)))
\end{aligned}$$

Else ($i = n$):

$$next'(\varphi) := \mathbf{false}$$

Note that the definition of $next$ depends on the state $\pi[0]$. Also note that whenever $next(p) \in \{\mathbf{true}, \mathbf{false}\}$, then we have the opportunity to apply early fault detection: The formula is fully determined. One can report satisfaction respectively failure at once.

Example 3.2.4 (Functions *now* and *next*)
Given the formula $\varphi = p \mathbf{\ U\ } q$ and the path $\pi = \{p\}\{q\}$, s.th. $\pi \models \varphi$ holds.

The calculation of *now* leads to:

$now(\varphi) = now(p \mathbf{\ U\ } q) = q \vee p$

The calculation of *next* leads to:

$next(\varphi)$
$= \mathbf{X}\ next'(p \mathbf{\ U\ } q)$
$= \ldots$
$= \mathbf{X}(\ \mathbf{false} \vee (\ next'(p) \wedge next'(\mathbf{X}(p \mathbf{\ U\ } q))\ )\ )$
$= \mathbf{X}(\ \mathbf{false} \vee (\ \mathbf{true} \wedge (p \mathbf{\ U\ } q)\ )\ )$
$= \mathbf{X}(\ p \mathbf{\ U\ } q\ )$

Now it holds that $\pi = \{p\}\{q\} \models now(\varphi) = q \vee p$ and $\pi = \{p\}\{q\} \models next(\varphi) = \mathbf{X}(\ p \mathbf{\ U\ } q\ )$.

Theorem 3.2.5 (Correctness of *now* and *next*)
For all $\varphi \in LTL$ and all $\pi \in S^+$ it holds that:

$$\pi \models \varphi \iff \pi \models now(\varphi) \wedge next(\varphi)$$

Proof 3.2.6 (Correctness of *now* and *next*)
Completeness ($\Rightarrow$):

Assume a formula $\varphi \in DLTL$ and a path $\pi \in S^+$ and assume that $\pi \models \varphi$.

Then in particular it holds that $\pi[0] \models \varphi$. Also we know that all parts of φ which are (implicitly[3] or explicitly) guarded by an $\mathbf{X}$ operator do *not contribute* to the truth value of $\pi[0] \models \varphi$. Hence it directly follows that $\pi[0] \models now(\varphi)$. Since $now(\varphi)$ does not contain any temporal operators, it follows that $\pi \models now(\varphi)$.

According to the general finite path semantics of *LTL* (cf. section 3.2.2) we can easily see that starting from the evaluation of $\pi[1]$ the formulae φ and $next(\varphi)$ are equivalent, because the definition of *next* follows the semantics in every case but the handling of the $\mathbf{X}$ operator: The function *next* pushes the $\mathbf{X}$ operator to the outermost level. Since $\mathbf{X}$ is distributive over all LTL operators, this is an equivalent transformation. Hence $\pi^1 \models next(\varphi)$. Since $next(\varphi)$ is of the form $\mathbf{X}\ next'(\varphi)$, it also holds that $\pi \models next(\varphi)$.

Soundness ($\Leftarrow$):

The proof of soundness is similar and is left as an exercise to the reader.

$\blacksquare$

In order to be able to proceed it is important to note the following.

[3]By *implicitly* we mean cases where $\mathbf{X}$ appears in the semantic evaluation of $\mathbf{U}$ or $\mathbf{R}$.

Observation 3.2.7
The evaluation of $now(\varphi)$ depends only on those propositions in $cl(\varphi)$, which are not (implicitly or explicitly) guarded by an $\mathbf{X}$ operator. Evaluation of the latter may be *deferred* at least until the next state. With respect to free variable bindings this means that we only need to make sure that all variables in the portion $now(\varphi)$ of φ are bound when we want to evaluate φ in the current state.

Using *now* and *next*, we are now able to define our declarative semantics.

3.3 Declarative semantics of *DLTL*

In order to do so, we define in an introductory subsection a formal notion of joinpoints, states, pointcuts, and propositions with free variables. In particular a pointcut must be able to provide a valuation at a given joinpoint so that we can use this valuation to bind objects within the formula.

The section is then concluded by the definition of the declarative semantics for a *DLTL* formula holding such propositions.

For this section we assume the following notation.

1. $\mathcal{V}$ is a finite set of variable names. O is some possibly infinite object domain.

2. We sometimes write functions in λ-notation: A term $\lambda x \lambda y \,.\, x < y$ represents an anonymous function which takes two arguments x and y and returns the Boolean value of $x < y$.

3. For some function we use a set based notation: $\{x \mapsto 1\}$ stands for the partial function which maps x to 1. (In all other cases the function is undefined.)

4. Some functions f are defined over propositions, pointcuts or bindings. Sometimes we apply those functions to whole formulae φ. In this context we mean that the function is applied to all propositions/pointcuts/bindings in $cl(\varphi)$ and the resulting formula is returned. Also such functions may be overloaded for sets of propositions, which mean that the function is applied to all elements and the appropriate set is returned. In any case, the function $\tilde{f}$ shall denote the *appropriately* overloaded version of f for its context.

In order to proceed, we still need the following assumption.

3.3.1 Filtered paths

The declarative semantics are based on the general assumption that we only observe states where at least one proposition holds. For instance if we specify that each call to a method `f()` is followed by a call to `g()`, then the actual execution trace

$$\{h\}\{f,h\}\{h\}\{g\}$$

is *filtered*. The reduced path which is used for further evaluation is

$$\{f\}\{g\}$$

because h is not contained in $cl(\varphi)$. We proceed by introducing general terms our semantics are based on.

3.3.2 Basic definitions

Definition 3.3.1 (Joinpoint)
Let O be a (possibly infinite) set of objects. A *joinpoint* in *DLTL* is a tuple $\iota = (\mathbf{this}_\iota, \mathbf{target}_\iota, \mathbf{args}_\iota, \mathbf{ret}_\iota, \mathbf{ex}_\iota)$ with:

$\mathbf{this}_\iota \in O \cup \{\oslash\}$ the currently executing object at ι,

$\mathbf{target}_\iota \in O \cup \{\oslash\}$ the call target object at ι,

$\mathbf{args}_\iota \in O^k \cup \{\oslash\}$ the argument vector of a method call or execution at ι,

$\mathbf{ret}_\iota \in O \cup \{\oslash\}$ the object returned by a method call or execution at ι,

$\mathbf{ex}_\iota \in O \cup \{\oslash\}$ the exception thrown at a method call or execution at ι.

Any of this/target/args/ret/ex may be undefined (for instance when executing in a static context, see figure 2.10). This is reflected by a value of $\oslash$.
We denote the set of all joinpoints by JP.
Further we define the set of all objects provided by a joinpoint ι, O_ι as:

$$O_\iota := (\ \{\mathbf{this}_\iota, \mathbf{target}_\iota, \mathbf{ret}_\iota, \mathbf{ex}_\iota\} \cup \{\bigcup_{i=1}^{k}\{o_i\} \mid \mathbf{args}_\iota = \{o_1, \ldots, o_k\}\}\ ) \setminus \{\oslash\}.$$

Definition 3.3.2 (Control flow)
Joinpoints can be cascaded at runtime: Since joinpoints are *intervals* in the control flow, one joinpoint can occur within another.
Hence for a joinpoint $\iota \in JP$ we define its control flow $cflow(\iota)$ as the sequence $cflow(\iota) := \iota_0, \ldots, \iota_{n-1} \in JP^n$ where $\iota_{n-1} = \iota$ and for all $0 \leq i < n-1$ it holds that ι_{i+1} occurs within ι_{i+1}.
We define the set of all *available objects in the control flow of* ι as:

$$O_\iota^{cflow} := \bigcup_{\iota' \in cflow(\iota)} O_{\iota'}$$

Definition 3.3.3 (Entry/exit kinds)
We define the set K of all *entry/exit kinds* of propositions as

$$K := \{\textbf{entry}, \textbf{exit}, \textbf{exit returning}, \textbf{exit throwing}\}.$$

Also we define a partial order $\trianglelefteq \subset K \times K$ as:

$$\trianglelefteq := \{\ (k,k) \mid k \in K\ \} \cup \{\ (\textbf{exit}, \textbf{exit returning}),\ (\textbf{exit}, \textbf{exit throwing})\ \}.$$

This shall reflect the fact that **exit** is not only matched by **exit** but also by **exit returning** and **exit throwing** (cf. definition of *advice* in section 2.3.1.4).

Definition 3.3.4 (State)
A state s is a tuple $s = (\iota_s, k_s) \in JP \times (K \backslash \{\textbf{exit}\})$. We denote the set of all states as $S := JP \times (K \backslash \{\textbf{exit}\})$. **exit** is here excluded because any exit event is either an exit by throwing an exception (**exit throwing**) or an exit by returning some value (**exit returning**).

Definition 3.3.5 (Pointcut)
A *Poincut* (or *Crosscut*, *X-Cut*) χ is a tuple $\chi = (\mu_\chi, \vec{v}_\chi, \sigma_\chi)$ with:

$\mu_\chi\ : JP \to \mathbb{B}$ the *matching function* of χ,

$\vec{v}_\chi\ = \{l_1, \ldots, l_n\} \in 2^{\mathcal{V}}$ the set of *variables defined* by χ,

$\sigma_\chi\ : JP \to (\vec{v}_\chi \to O)$ the *valuation function* of χ.[4]

We denote the set of all pointcuts by PC.

Here for all $\chi \in PC, \iota \in JP$, $\sigma_\chi(\iota)$ is defined if and only if $\mu_\chi(\iota) = true$. This is due to the fact that a pointcut which does not match a joinpoint cannot expose any values at this joinpoint. Hence, for cases where $\mu_\chi(\iota) = false$, we write $\sigma_\chi(\iota) = \oslash$. Further, the range of $\sigma_\chi(\iota)$ shall be restricted to values of O_ι^{cflow}, because according to the AspectJ semantics only objects in the control flow can be accessed.

Note that for the sake of an easy notation we denote matching functions by the appropriate pointcut expressions with their natural semantics as defined by AspectJ.

Example 3.3.6 (Pointcut)
Assume the following AspectJ pointcut definition:
`pointcut pc(Stack s):  call(Object Stack.pop()) && target(s);`
In our notation this would define a pointcut $\chi = (\mu_\chi, \vec{v}_\chi, \sigma_\chi)$ with:

$\mu_\chi\ =$ `call(Object Stack.pop()) && target(s)`,

[4]Note that here we use the set $\vec{v}_\chi$ as type. This shall denote that the mapping functions returned by σ_χ are partial functions over $2^{\mathcal{V}}$ but fully defined nonpartial functions over $\vec{v}_\chi$.

$\vec{v}_\chi = \{ \texttt{s} \}$,

$\sigma_\chi = \lambda\iota.\{ \texttt{s} \mapsto \textbf{target}_\iota \}$.

Definition 3.3.7 (Binding)
A *binding* is a partial function $\beta : \mathcal{V} \dashrightarrow (O \cup \{\oslash\})$. We denote the set of all possible bindings as $B := \{\beta \mid \beta : \mathcal{V} \dashrightarrow (O \cup \{\oslash\})\}$.

A binding function β may define certain variables as *unbound*[5]. We denote the fact that a variable x is unbound by $\{ \texttt{x} \mapsto \oslash \}$.

For each $\mathcal{V}' \subseteq \mathcal{V}$, we define the set $B|_{\mathcal{V}'}$ of all bindings over $\mathcal{V}'$ as $B|_{\mathcal{V}'} := \{\beta : \mathcal{V}' \dashrightarrow (O \cup \{\oslash\})\}$.

Definition 3.3.8 (Proposition)
Let L be a finite set of labels. Then a proposition is a tuple $p = (l_p, \chi_p, k_p, \beta_p) \in L \times PC \times K \times B$.

We call l_p the *label* of p, χ_p is the *pointcut associated* with p. k_p denotes the *entry/exit kind* of p while we call β_p the *current binding* of p.

The binding function β_p is dynamic over time. It is initialized as:

$$\beta_p := \{ x \mapsto \oslash \mid x \text{ is a variable in } \chi_p \}.$$

This includes also *used variables*, variables contained in χ_p but not in $\vec{v}_{\chi_p}$.

An example for such a proposition will be given on page 49. We denote the set of all propositions by $\mathcal{P}$. Further we write $\mathcal{P}_\varphi$ for the set of all propositions of a formula φ:

$$\mathcal{P}_\varphi := cl(\varphi) \cap \mathcal{P}$$

In the following, we define what it means for a proposition p to hold at a given state (we say, that p *matches* the state). This definition is based on the current binding of p which, as we will see in later sections, is dynamic over time. The way in which those bindings propagate is the key point of the semantics of *J-LO* and also the point where we will use the splitting into *now* and *next*.

Definition 3.3.9 (Matching)
For a state $s = (\iota_s, k_s) \in S$ and a proposition $p = (l, \chi, k, \beta) \in \mathcal{P}$, we say that *p matches s* or *holds* in s, $s \models p$ for short, if with $\mu'_\chi := (\tilde{\sigma_\chi} \circ \tilde{\beta})\ (\mu_\chi)$, the following conditions hold:

1. $\mu'_\chi(\iota_s) = \textbf{true}$,

[5]One could raise the questions why one does not just drop *unbound* mappings from the function definition. The reason for this design decision is that our operational semantics uses such unbound mappings to replace them by appropriate bindings to objects.

2. $k \trianglelefteq k_s$.

The first requirement states that the pointcut of p must match the given joinpoint, where in the matching function free variables have been replaced by first bindings and then the valuations of the contained pointcut. The second one requires that the entry/exit kinds are compatible. If a proposition p matches a state under a certain binding β, we say that β is a *satisfying binding* for p.

In order to be able to evaluate $\mu'_\chi(\iota_s)$, one must make sure that μ'_χ does not contain any free variables. Variables can be bound by either the binding function β or the valuation function σ_χ of the current joinpoint. `if` pointcuts, may refer to bound values not exposed by the current joinpoint. Here one must make sure that the binding function β is rich enough to bind all free variables. The static analysis we introduce in section 3.4 allows to ensure this. Let us discuss this problem by an example.

Example 3.3.10 (Proposition)
Assume the following proposition:

exit(**call**(Object Stack.pop()) && **if**(o1!=o2)) **returning** o1

In our semantics this yields a proposition $p = (l_p, \chi_p, k_p, \beta_p)$ with:

- $l_p =$ `"exit( call(Object Stack.pop()) && if(o1!=o2) ) returning o1"`
- $\chi_p = (\mu_{\chi_p}, \vec{v}_{\chi_p}, \sigma_{\chi_p})$ with
  - $\mu_{\chi_p} =$ `call(Object Stack.pop()) && if(o1!=o2)`
  - $\vec{v}_{\chi_p} = \{$ `o1` $\}$
  - $\sigma_{\chi_p} = \lambda\iota.\ \{$ `o1` $\mapsto \mathbf{ret}_\iota \}$
- $k_p =$ **exit returning**
- $\beta_p = \{$ `o1` $\mapsto \emptyset,$ `o2` $\mapsto \emptyset\}$

Note that in particular, the matching function `call(Object Stack.pop()) && if(o1!=o2)` uses variables `o1` and `o2`. The valuation function of the associated pointcut, σ_{χ_p}, is however only rich enough to define a value for `o1`. As a consequence, in order to evaluate $\mu'_\chi := (\tilde{\sigma}_{\chi_p} \circ \tilde{\beta}_p)\ (\mu_\chi)$, one must make sure that the binding β_p provides a value $\neq \emptyset$ for `o2`, when this proposition is to be evaluated.

While the static analysis we define in section 3.4 will ensure this, for now assume the following:

For any formula φ, at any given state $\pi[i]$, all propositions contained in $now(\varphi)$ are sufficiently bound, so that any variable used in $now(\varphi)$ has a defined value $\neq \oslash$.

As we mentioned earlier, the key point of the dynamic semantics of *DLTL* is to understand how, when, and where free variables should be bound and most importantly *why* one should do it the way we define it.

In the following subsection we hence want to motivate this mechanism by another example.

In this example as well as all the following sections of this chapter we want to assume that the functions *now* and *next* as well as the satisfaction relation $\models$ are equally defined for *DLTL* formulae as they are for *LTL* formulae. Indeed the semantics are fully equivalent except the different semantics of $s \models p$ for a state s and a proposition p, because here we need to take bindings into account.

3.3.3 Bindings by example

In this section we use the following notations for propositions with bindings:

The term $p(x)$ stands for a proposition p with $\vec{v}_{\chi_p} = \{x\}$ and $\beta_p = \{x \mapsto \oslash\}$ (x is a variable in p, which is currently unbound).

A term $p(1)$ should informally denote the proposition where x has been bound to $1 \in O$, so that now $\beta_p = \{x \mapsto 1\}$.

Example 3.3.11 (Propagation of bindings)
Let $\varphi(x) := \mathbf{G}(p(x) \rightarrow \mathbf{F}\ q(x))$ and $\pi = \{p(1), p(2)\}\{q(1)\}$.

We want the semantics of this formula to imply that for *each possible* valuation x' of x on the occurrence of $p(x)$, we *finally* see the proposition $q(x')$ on the path. In other words, the quantification over free variables is reduced to quantification over states. The given path π would violate φ, because $p(2)$ gives a valuation $x = 2$ so that there is *no* matching $q(2)$ to follow.

In order to see how the desired effect can be achieved let's have a look at *now* and *next*. At $\pi[0]$ It holds that:

$$now(\varphi(x)) = \mathbf{true} \quad \text{and} \quad next(\varphi(x)) = \mathbf{X}(\ \varphi(x) \wedge \mathbf{F}\ q(x)\ )$$

Of particular interest here is the result of *next*. This formula imposes the obligation on the subsequent path that has to be fulfilled in order to satisfy φ. Here this obligation says that finally $q(x)$ has to hold as well as again $\varphi(x)$ (this is due to the $\mathbf{G}$ operator). What we would actually like is that on the subsequent path $q(1)$ and $q(2)$ should hold at some point. Also, $\varphi(x)$ should hold for *all* possible valuations that are yet unknown.

Hence, the *desired* obligation would be $\mathbf{X}(\ \varphi(x) \wedge \mathbf{F}\ q(1) \wedge \mathbf{F}\ q(2)\ )$. In the final state $\pi[1]$ this would evaluate to $\varphi(x) \wedge \mathbf{F}\ q(2)$, expressing that the requirement $\mathbf{F}\ q(2)$ was not yet fulfilled. (Note that $\varphi(x) \wedge \mathbf{F}\ q(2)$ is a nonfinal state in an AFA, while $\varphi(x)$ is final.)

Informally the semantics are as follows.

Observation 3.3.12 (Declarative semantics, informally)
For a formula $\varphi(\vec{x})$ and a state $\pi[i]$ it holds that $\pi[i] \models \varphi(\vec{x})$ if and only if for all possible valuations $\vec{x'}$ at $\pi[i]$ both, $now(\varphi(\vec{x'}))$ and $next(\varphi(\vec{x'}))$ hold, where $next$ leaves variables in the original $\varphi(\vec{x})$ unbound.

The last property might seem as an unusual exception at a first glance. However, when looking at the equivalent AFA, it becomes clear that the original formula occurs as subformula of $next(\varphi(\vec{x}))$ in exactly those cases where further evaluation of the formula is deferred to the next state. Taking into account that the AFA is partially ordered (see page 2.1.1.4), one can say that variables are bound if and only if one moves further down in this order, so if one moves from a state φ_1 to φ_2 with $\varphi_1 \succeq \varphi_2$.

This observation is already a good start however there is still one uncertainty that needs disambiguation: What are "all possible valuations $\vec{x'}$ at $\pi[i]$" ?

This shall be clarified by the next subsection.

3.3.4 Possible valuations

Again, we want to approach this problem by an example.

Example 3.3.13 (Possible valuations)
Assume the formula $\varphi(x, y) := p(x, y) \rightarrow \mathbf{XG}\ p(y, x)$ and the path $\pi := \{p(1, 2)\}\{p(2, 1)\}$. Intuitively, the path should clearly satisfy φ, since $p(x, y)$ matches $\pi[0] = \{p(1, 2)\}$ with $x = 1, y = 2$ and the formula states that in this case, $p(y, x)$, i.e. $p(2, 1)$ under those bindings, should hold on the subsequent path. This is obviously satisfied by the only subsequent state $\pi[1] = \{p(2, 1)\}$.

Important to this example is that $p(x, y)$ and $p(y, x)$ are essentially *the same propositions*, not taking bindings into account. Being unbound, they share the same matching semantics . This means that each state s on π is matched by $p(x, y)$ if and only if it is matched by $p(y, x)$ or, in other words, $p(x, y)$ holds if and only if $p(y, x)$ holds.

Hence, in state $\pi[0]$ we can identify two matching propositions: $p(x, y)$ with $x = 1, y = 2$ and $p(y, x)$ with $y = 1, x = 2$, leading to possible valuations as $x \in \{1, 2\}, y \in \{1, 2\}$ as a first try.

If we took this as defined "set of possible valuations" as referred to in observation 3.3.12, this would mean, that in state $\pi[1]$ one would have the following obligations to fulfill: $\mathbf{G}\ p(2,1)$ and $\mathbf{G}\ p(1,2)$. This is violated by $\pi[1] = \{p(2,1)\}$, since $p(1,2)$ does not hold.

So apparently the valuation $y = 1, x = 2$ is not a "possible valuation" in the above sense, but why not? The solution becomes clear through inspection of the given formula. Here, we can see that $now(\varphi) = \neg p(x,y)$ does not contain $p(y,x)$. Hence, the binding $y = 1, x = 2$ should not contribute to the truth value of $now(\varphi)$, nor should it be a binding to persist for the evaluation of future states.

This observation yields the following definitions.

Definition 3.3.14 (Active propositions)
Let $\varphi \in DLTL$. Then $\mathcal{P}_\varphi^{act} \subseteq \mathcal{P}$, the set of *active propositions* of φ, is defined as the set of propositions contained in $now(\varphi)$: $\mathcal{P}_\varphi^{act} := cl(now(\varphi)) \cap \mathcal{P}$.

Definition 3.3.15 (Active propositions at a state)
$\mathcal{P}_\varphi^{act}(s) \subseteq \mathcal{P}$, the set of active propositions in φ matching s is defined as:

$$\mathcal{P}_\varphi^{act}(s) := \{\ p \in \mathcal{P}_\varphi^{act} \mid \mu_{\chi_p}(\iota_s) \wedge k_s \trianglelefteq k_p\}.$$

Definition 3.3.16 (Active Bindings)
In order to build the set of active bindings, first we list all possible mappings and then extract all possible *mapping functions* from this set. Let $s = (\iota_s, k_s) \in S$ and $\varphi \in DLTL$.

Define $all_\varphi(s)$ as:

$$all_\varphi(s) := \{(x,o) \in \mathcal{V} \times O \mid \exists p \in \mathcal{P}_\varphi^{act}(s) : \{x \mapsto o\} \in \sigma_{\chi_p}(\iota_s)\}$$

Then $B_\varphi^{act}(s) \in 2^B$, the set of *active bindings* at s under φ is defined as:

$$\begin{gathered} B_\varphi^{act}(s) := \{\beta \in B \mid \forall x \text{ with } (x,o) \in all_\varphi(s) : \\ \exists_1 \{x \mapsto o'\} \in \beta \text{ s.th. } (x,o') \in all_\varphi(s)\} \end{gathered}$$

Here $\exists_1$ means: "exists exactly one".

Those *active bindings* are the "possible valuations" we referred to in observation 3.3.12.

In this observation we also postulated that *next* should not bind any free variables "in the original formula". Hence, in the following subsection we redefine *next* accordingly: The function remains unchanged for all subformulae φ' of a formula φ except in cases where φ' has the form $\mathbf{X}\varphi''$: Here we only bind values in the case where $\varphi'' \neq \varphi$, because only in those cases where we move "further down" in the alternating automata, we make a "real step" according to the partial order $\preceq$.

3.3.5 Redefinition of *next* for *DLTL*

Definition 3.3.17 (Function *next* for *DLTL*)
Let $\pi \in S^n$. Let $\varphi(\vec{x}) \in DLTL$.

We define $next(\varphi(\vec{x})) := next_{\varphi(\vec{x})}(\varphi(\vec{x}))$ with $next_{\varphi(\vec{x})} : DLTL \to DLTL$ being recursively defined as:

If $\varphi' \in cl(\varphi), \vec{x'} \subseteq \vec{x}$ and $\varphi'(\vec{x'}) = \mathbf{X}\ \varphi''(\vec{x'})$ then:

$$next_{\varphi(\vec{x})}(\varphi'(\vec{x'})) = next_{\varphi(\vec{x})}(\mathbf{X}\ \varphi''(\vec{x'})) := \begin{cases} \varphi''(\vec{x}) & \text{if } \varphi'' = \varphi, \text{ (stay unbound)} \\ \varphi''(\vec{x'}) & \text{otherwise, } \qquad\qquad \text{(bind)} \end{cases}$$

Else:

$$next_{\varphi(\vec{x})}(\varphi'(\vec{x'})) := next(\varphi'(\vec{x'})) \qquad \text{(as before)}$$

Now we are ready to define the general declarative semantics. Here we postulate a function $valid : DLTL \to \mathbb{B}$ with $valid(\varphi) = \textbf{true}$ if and only if in φ any variable is defined before it is used. In section 3.4 we are going to explain this function in detail and introduce a static analysis which decides if $valid(\varphi)$ holds for a given formula φ.

3.3.6 Declarative semantics of a *DLTL* formula

Definition 3.3.18 (Declarative semantics of a *DLTL* formula)
Let $\varphi(\vec{x}) \in DLTL$ with $valid(\varphi) = \textbf{true}$ and $\pi \in S^+$.

$$\begin{gathered} \pi \models \varphi(\vec{x}) \\ :\Longleftrightarrow \\ \forall \beta \in B^{act}_{\varphi}(\pi[0]) : \\ \pi \models now(\tilde{\beta}(\varphi(\vec{x}))) \land \pi \models next_{\varphi(\vec{x})}(\tilde{\beta}(\varphi(\vec{x}))) \end{gathered}$$

For the case of the empty path where $|\pi| = 0$, we define:

$$\begin{gathered} \pi \models \varphi \\ :\Longleftrightarrow \\ \varphi = (\varphi'\ \mathbf{R}\ \psi') \text{ for some } \varphi', \psi' \in cl(\varphi) \end{gathered}$$

Example 3.3.19 (Declarative semantics of a *DLTL* formula)
Let $\varphi(x, y) := \mathbf{G}(\ p(x) \to \mathbf{XF}\ q(y, \underline{x})\ )$ and $\pi := \{p(1)\}\{q(2, \underline{x})\}$.

Further assume that for μ_{χ_q}, the matching function of q has such a structure that $q(2, 1)$ is a satisfying binding (cf. definition 3.3.9).

Then we have:

$$\pi \models \varphi(x,y)$$

$$\Longleftrightarrow$$

$$\forall \beta \in B^{act}_{\varphi}(\pi[0]) :$$
$$\pi \models now(\tilde{\beta}(\varphi(x,y))) \;\wedge\; \pi \models next_{\varphi(x,y)}(\tilde{\beta}(\varphi(x,y)))$$

$$\Longleftrightarrow$$

$$\pi \models now(\underbrace{(\lambda x.x \mapsto 1)\ (\varphi(x,y))}_{\varphi(1,y)}) \;\wedge\; \pi \models next_{\varphi(x,y)}(\underbrace{(\lambda x.x \mapsto 1)\ (\varphi(x,y))}_{\varphi(1,y)})$$

$$\Longleftrightarrow$$

$$\pi \models \underbrace{now(\varphi(1,y))}_{\textbf{true}} \;\wedge\; \pi \models \underbrace{next_{\varphi(x,y)}(\varphi(1,y))}_{\mathbf{X}(\ \varphi(x,y)\ \wedge\ \mathbf{F}\ q(y,1)\ )}$$

$$\Longleftrightarrow$$

$$\underbrace{\pi \models \textbf{true}}_{\textbf{true}} \;\wedge\; \pi \models \mathbf{X}(\ \varphi(x,y) \;\wedge\; \mathbf{F}\ q(y,1)\ )$$

$$\Longleftrightarrow$$

$$\pi^1 \models \underbrace{\varphi(x,y) \;\wedge\; \mathbf{F}\ q(y,1)}_{=:\varphi'(x,y)}$$

$$\Longleftrightarrow$$

$$\forall \beta \in B^{act}_{\varphi'(x,y)}(\pi[1]) :$$
$$\pi^1 \models now(\tilde{\beta}(\varphi'(x,y))) \wedge \pi^1 \models next_{\varphi'(x,y)}(\tilde{\beta}(\varphi'(x,y)))$$

$$\Longleftrightarrow$$

$$\pi^1 \models now((\lambda y.y \mapsto 2)\ (\varphi'(x,y))) \;\wedge\; \pi^1 \models next_{\varphi'(x,y)}((\lambda y.y \mapsto 2)\ (\varphi'(x,y)))$$

$$\Longleftrightarrow$$

$$\pi^1 \models \underbrace{now(\varphi(x,2) \;\wedge\; \mathbf{F}\ q(2,1))}_{\textbf{true}} \;\wedge\; \pi^1 \models \underbrace{next_{\varphi'(x,y)}(\varphi(x,2) \wedge \mathbf{F}\ q(2,1))}_{\mathbf{X}\ \varphi(x,y)}$$

$$\Longleftrightarrow$$

$$\pi^1 \models \textbf{true} \wedge \pi^1 \models \mathbf{X}\ \varphi(x,y)$$

$$\Longleftrightarrow$$

$$\pi^1 \models \textbf{true} \wedge \underbrace{\pi^2 \models \varphi(x,y)}_{\textbf{true}}$$

$$\Longleftrightarrow$$

$$\textbf{true}$$

Note that in the last step $|\pi^2| = 0$ and $\varphi(x,y)$ is a *Release* formula.

This example shall conclude our definition of the declarative semantics of *DLTL*. We now proceed with the definition and correctness proof of the static analysis which detects invalid formulae.

3.4 Static analysis

The analysis is based on the idea of use-definition chains (UD chains) as they are known from compiler construction theory. Opposed to usual UD chains, which calculate the set of definitions which *potentially* reach a variable, our analysis in conservative, meaning that for a used variable we calculate the set of definitions which *certainly* reach this variable[6]. If this is empty, we report the formula as invalid.

Again, we want to derive the details of this analysis by looking at an example.

Example 3.4.1 (Static Analysis - propagation over "$\wedge$")
Take for example the following formula:

$$\varphi(x,y) := p(x) \wedge \mathbf{XF}\ q(y,\underline{x})$$

This formula has an obvious splitting to the subformulae $now(\varphi(x,y)) = p(x)$, and $next(\varphi(x,y))$, which depends on the current state s.

Here, two cases can occur:

1. $s \models p(x)$, say with a binding $x = 1$. This binding is available for the rest of the path and in particular for the evaluation of $next_{\varphi(x,y)}(\varphi(1,y)) = \mathbf{F}\ q(y,\underline{1})$ on subsequent states.

2. $p(x) \not\models p(x)$. In this case, we have no binding for x at the current state. However, this binding would not be needed anyway, since the formula $now(\varphi(x,y))$ evaluates to **false** already.

So informally one can say that *bindings defined by propositions progagate over the $\wedge$-operator*: A binding that is defined by a proposition in one branch of an $\wedge$-term is also available in the other branch.

A case which is a bit harder to identify is the following.

Example 3.4.2 (Static Analysis - propagation over "$\vee$")

$$\varphi(x,y) := p(x) \rightarrow \mathbf{X}\ \mathbf{F}\ q(y,\underline{x})$$

which is in NNF:

$$\neg p(x) \vee \mathbf{X}\ (\ \mathbf{true}\ \mathbf{U}\ q(y,\underline{x})\ )$$

[6]Note that it is no error, if a variable has more than one such definition. In this case, the variable is bound by the first occurring definition. The latter definitions are automatically turned into uses of this variable.

At a first glance it seems unclear how a binding should be available for the evaluation of $\mathbf{F}\ q(y, \underline{x})$, given that $p(x)$ occurs in negated form.

However, again it helps to look at the possible cases:

1. $p(x)$ does not hold at the current state s. In this case, we have no binding for x at the current state. However, again this does not hurt, since both formulae $now(\varphi(x,y)) = \neg p(x)$ and $next_{\varphi(x,y)}(\varphi(x,y))$ evaluate to **true**.

2. $p(x)$ holds at the current state s, say with a binding $x = 1$. Again, this binding is available for the rest of the path and in particular for the evaluation of $next_{\varphi(x,y)}(\varphi(1,y))s = \mathbf{F}\ q(y, \underline{1})$ on subsequent states.

Again, informally one can conclude that *bindings defined by negated propositions progagate over the* $\vee$*-operator.*

This should tell us that the following formula should be considered as *invalid.*

Example 3.4.3 (Static Analysis - invalid formula)
Recall again the example from section 3.2.3, which the Haskell version evaluated to **true** by quantifying over an empty domain.

$$p(x) \vee q(y, \underline{x})$$

Here, $p(x)$ is not negated and is a direct subformula of a $\vee$-formula.

We can distinguish the following cases:

1. $p(x)$ holds at the current state, thus providing a binding for x. Then *now* and *next* both evaluate to **true**.

2. $p(x)$ does not hold at the current state, hence we have no binding for x available. This is interesting with respect to the evaluation of q:

   (a) $q(y, \underline{x})$ does not hold. This is the easy case. Since neither of the propositions hold, we should evaluate to **false**.

   (b) $q(y, \underline{x})$ holds, providing a binding for y. However, $q(y, \underline{x})$ uses $\underline{x}$, whose value is *undefined.* Those are exactly the cases we want to exclude.

In order to check if a given formula $\varphi \in DLTL$ is valid informally, we proceed as follows:

1. For each possible point in time (i.e. joinpoint) produce a set $def(\varphi)$ of variables which are defined at this time.

2. For each such point in time also check for each variable if this variable is defined on this or one of the previous points in time.

Such possible points in time can be distinguished by explicit or implicit occurrences of the **X** operator. The set $\mathit{def} \subseteq 2^{\mathcal{V}}$ of defined variables is then calculated according to the above observations:

Definition 3.4.4 (Function *def*)
Let DLTL^{nnf} the set of all *DLTL* formulae in negation normal form (cf. section 2.1) and $\varphi \in \mathit{DLTL}^{nnf}$. Let $p \in \mathcal{P}$. Then we define $\mathit{def} : \mathit{DLTL}^{nnf} \rightarrow 2^{\mathcal{V}}$ as:

$$\begin{array}{rcl}
\mathit{def}(\varphi) & := & \mathit{def}_+(\varphi) \ \cup \ \mathit{def}_-(\varphi) \\
 & \text{where} & \\
\mathit{def}_+(p) & := & \vec{v}_{\chi_p} \\
\mathit{def}_+(\neg p) & := & \emptyset \\
\mathit{def}_+(\mathbf{X}\ \varphi) & := & \emptyset \\
\mathit{def}_+(\varphi \wedge \psi) & := & \mathit{def}_+(\varphi) \ \cup \ \mathit{def}_+(\psi) \\
\mathit{def}_+(\varphi \vee \psi) & := & \mathit{def}_+(\varphi) \ \cap \ \mathit{def}_+(\psi) \\
\mathit{def}_+(\varphi\ \mathbf{U}\ \psi) & := & \mathit{def}_+(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
 & = & \mathit{def}_+(\varphi) \ \cap \ \mathit{def}_+(\psi) \\
\mathit{def}_+(\varphi\ \mathbf{R}\ \psi) & := & \mathit{def}_+(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
 & = & \mathit{def}_+(\psi) \\
 & \text{and} & \\
\mathit{def}_-(p) & := & \emptyset \\
\mathit{def}_-(\neg p) & := & \vec{v}_{\chi_p} \\
\mathit{def}_-(\mathbf{X}\ \varphi) & := & \emptyset \\
\mathit{def}_-(\varphi \wedge \psi) & := & \mathit{def}_-(\varphi) \ \cap \ \mathit{def}_-(\psi) \\
\mathit{def}_-(\varphi \vee \psi) & := & \mathit{def}_-(\varphi) \ \cup \ \mathit{def}_-(\psi) \\
\mathit{def}_-(\varphi\ \mathbf{U}\ \psi) & := & \mathit{def}_-(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
 & = & \mathit{def}_-(\psi) \\
\mathit{def}_-(\varphi\ \mathbf{R}\ \psi) & := & \mathit{def}_-(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
 & = & \mathit{def}_-(\varphi) \ \cap \ \mathit{def}_-(\psi)
\end{array}$$

Here $\mathit{def}_+(\varphi)$ provides the variables which are bound by propositions contained in *nonnegated* form at the current point in time, while $\mathit{def}_-(\varphi)$ provides those for propositions which occur *under negation.*

Next, we define the logical counterpart, the function *use* which represents the variables of $\mathcal{P}_\varphi$ which are *used* by any of the propositions at some point in time but not *defined* by the same proposition.

Definition 3.4.5 (Function *use*)
Assume that $\varphi \in \mathit{DLTL}^{nnf}$ is in negation normal form. Let $p \in \mathcal{P}$. Let $\odot_2 \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\}$. Then we define $\mathit{use} : \mathit{DLTL}^{nnf} \to 2^{\mathcal{V}}$ as:

$$\begin{aligned}
\mathit{use}(p) &:= \{x \in \mathcal{V} \mid (x, o) \in \beta_p \wedge x \notin \mathit{def}(p)\} \\
&= \{x \in \mathcal{V} \mid (x, o) \in \beta_p \wedge x \notin \vec{v}_{\chi_p}\} \\
\mathit{use}(\neg p) &:= \mathit{use}(p) \\
\mathit{use}(\mathbf{X}\ \varphi) &:= \emptyset \\
\mathit{use}(\varphi \odot_2 \psi) &:= \mathit{use}(\varphi) \cup \mathit{use}(\psi)
\end{aligned}$$

With those definitions it is now straightforward to define the function $\mathit{valid}(\varphi)$, which is **true** if and only if φ defines any free variable before it is used.

Definition 3.4.6 (Function *valid*)
Assume that $\varphi \in \mathit{DLTL}^{nnf}$ is in negation normal form. Let $p \in \mathcal{P}$. Let $\odot_1 \in \{\neg, \mathbf{X}\}, \odot_2 \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\}$. Then we define $\mathit{valid} : \mathit{DLTL}^{nnf} \to \mathbb{B}$ as:

$$\begin{aligned}
\mathit{valid}(\varphi) &:= \mathit{valid}_{\mathit{def}(\varphi)}(\varphi) \\
\text{where} \quad &\text{for} \quad \mathcal{D} \subseteq \mathcal{V} : \\
\mathit{valid}_{\mathcal{D}}(p) &:= \mathit{use}(p) \subseteq \mathcal{D} \\
\mathit{valid}_{\mathcal{D}}(\odot_1 \varphi) &:= \mathit{valid}_{\mathcal{D} \cup \mathit{def}(\odot_1 \varphi)}(\varphi) \\
\mathit{valid}_{\mathcal{D}}(\varphi \odot_2 \psi) &:= \mathit{valid}_{\mathcal{D} \cup \mathit{def}(\varphi \odot_2 \psi)}(\varphi) \ \wedge \ \mathit{valid}_{\mathcal{D} \cup \mathit{def}(\varphi \odot_2 \psi)}(\psi)
\end{aligned}$$

For an implementation one might want to inline the definitions and so derive reductions as the following:

$$\begin{aligned}
\mathit{valid}_{\mathcal{D}}(\mathbf{X}\ \varphi) &:= \mathit{valid}_{\mathcal{D} \cup \mathit{def}(\mathbf{X}\ \varphi)}(\varphi) \\
&= \mathit{valid}_{\mathcal{D}} \\
\mathit{valid}_{\mathcal{D}}(\neg p) &:= \mathit{valid}_{\mathcal{D} \cup \mathit{def}(\neg p)}(\varphi) \\
&= \mathit{valid}_{\mathcal{D} \cup \{\vec{v}_{\chi_p}\}}
\end{aligned}$$

. . . and so forth.

Example 3.4.7 (Static analysis - invalid formula (formally))
Let us assume again the following formula, where $\underline{x}$ means that $\underline{x}$ is not defined by q, i.e. $\underline{x} \notin \vec{v}_{\chi_q}$:

$$\varphi := p(x) \vee q(y, \underline{x})$$

Then we have:

$$\begin{aligned}
\mathit{valid}(\varphi) &= \mathit{valid}_{\mathit{def}(\varphi)}(\varphi) = \mathit{valid}_{\emptyset}(\varphi) \\
&= \mathit{valid}_{\emptyset}(p(x) \vee q(y, \underline{x})) \\
&= \mathit{use}(\varphi) \subseteq \emptyset \ \wedge \ \mathit{valid}_{\emptyset \cup \mathit{def}(\varphi)}(p(x)) \ \wedge \ \mathit{valid}_{\emptyset \cup \mathit{def}(\varphi)}(q(y, \underline{x})) \\
&= \{\underline{x}\} \subseteq \emptyset \ \wedge \ \mathit{valid}_{\emptyset}(p(x)) \ \wedge \ \mathit{valid}_{\emptyset}(q(y, \underline{x})) \\
&= \mathbf{false}
\end{aligned}$$

Theorem 3.4.8 (Correctness of function *valid*)
For any formula $\varphi \in \mathit{DLTL}^{nnf}$ it holds that:

$\mathit{valid}(\varphi) \iff$ any variable in φ is defined before it is used

Proof 3.4.9 (Correctness of function *valid*)
Soundness ($\Rightarrow$):

Let $\varphi \in \mathit{DLTL}^{nnf}$. Assume $\mathit{valid}(\varphi) = \mathit{valid}_{\mathcal{D}}(\varphi) = \mathbf{true}$ with $\mathcal{D} = \mathit{def}(\varphi)$. We perform a proof by structural induction and distinguish the following cases:

1. $\varphi = p$ for some $p \in \mathcal{P}$. Since $\mathit{valid}(p) = \mathbf{true}$, we know that $\mathit{use}(p) := \{x \in \mathcal{V} \mid (x, o) \in \beta_p \wedge \neg x \in \mathit{def}(p)\} \subseteq \mathcal{D}$. Also we know by the definition of *def* that $\mathcal{D}$ contains only variables which are defined on this or previous temporal layers, because later temporal layers (which are explicitly or implicitly guarded by $\mathbf{X}$) do not contribute to the function *def*. Hence, any variable in p is defined before it is used.

2. $\varphi = \odot_1 \varphi'$ for some $\varphi' \in \mathit{DLTL}^{nnf}$. Since $\mathit{valid}(\varphi) = \mathbf{true}$, it must also hold that $\mathit{valid}_{\mathcal{D} \cup \mathit{def}(\varphi)}(\varphi') = \mathbf{true}$. So by induction hypothesis φ' defines all variables before they are used. Since the move from φ' to φ introduces no new variables, the same holds for φ.

3. $\varphi = \varphi' \odot_2 \psi'$ for some $\varphi', \psi' \in \mathit{DLTL}^{nnf}$. This case can be handled as above.

Completeness ($\Leftarrow$):

Let $\varphi \in \mathit{DLTL}^{nnf}$. Assume any variable in φ is defined before it is used. We distinguish the following cases:

$\varphi = p$ for some $p \in \mathcal{P}$. Since in p any variable which is used is defined by the context, we have that $\forall x : ((x, o) \in \beta_p \wedge \neg x \in \mathit{def}(p)) \rightarrow x \in \mathcal{D}$. Hence, $\mathit{use}(p) \subseteq \mathcal{D}$ and so $\mathit{valid}(p) = \mathbf{true}$.

$\varphi = \odot_1 \varphi'$ for some $\varphi' \in \mathit{DLTL}^{nnf}$. Assume, that in φ all variables are defined before they are used. Then by induction hypothesis, $\mathit{valid}(\varphi') = \mathbf{true}$. Hence also $\mathit{valid}(\varphi) = \mathbf{true}$.

$\varphi = \varphi' \odot_2 \psi'$ for some $\varphi', \psi' \in DLTL^{nnf}$. Again, this case can be shown as above.

■

It should be noted that this analysis is conservative: It assures that if a formula φ is valid, then there is *no path* π so that there is a variable x in φ which could be used before it is defined when evaluating φ over π. Obviously it could be that those cases do not occur for a given formula and some given set of possible runtime paths. Hence one *could* argue that invalid formulae should be treated with a warning rather than a *fast-fail* error message. However, when bypassing the static analysis in that way, the soundness of the matching semantics for a proposition cannot be guaranteed any more: At the moment where we *know* that a formula is valid and we come to decide if a proposition p over variables $\vec{x}$ matches, then we *know* that it is sound to define that p does *not* match if any of the values of those variables are undefined. When bypassing the analysis, this guarantee can no longer be given: Hence, one would have no means of determining an invalid path/formula combination at runtime, which would give up soundness.

Example 3.4.10 (Unsound evaluation without static analysis)
Let us assume the following formula, where again $\underline{x}$ is used but not defined in the proposition q:

$$\varphi(x, y) := p(x) \wedge q(y, \underline{x})$$

In the case where we do *not* perform a static analysis, we obviously do *not* know if φ is valid. In the case where $p(x)$ does not hold at state but $q(y, \underline{x})$ does, say with a value of $y = 1$ so that $q(1, \underline{x})$ holds, we have to decide *locally* if $q(y, \underline{x})$ matches this current state.

Under the assumption that φ is valid, we *know* that it is safe to evaluate $\delta(q(y, \underline{x}))$ to **false** in cases where $\underline{x}$ is unbound, because those are *only exactly those cases* where $p(x)$ does *not* hold and hence φ evaluates to **false** anyway. In particular, when we evaluate $\delta(r)$ for a proposition r, this transition is always well-defined.

If we do *not* know if φ is valid, we cannot make the assumption that φ is valid. Hence, two cases can lead to the evaluation of a proposition r with insufficient binding:

1. The well-defined cases as above.

2. Cases where a binding is indeed missing, meaning that a variable $\underline{x}$ is used before it is defined.

In the second case, there is no value given for $\underline{x}$. Hence, the one has to decide if $\delta(r(\underline{x}))$ should evaluate to **true** or **false** without actually taking x into account, which cannot lead to sound results in general.

3.5 Operational semantics of *DLTL*

Our operational semantics follows the declarative semantics very closely. The two major differences lie in a necessary special treatment of `if` pointcuts in order to evaluate them in the right context.

The operational semantics are based on the following ideas.

1. We use an automaton based approach to propagate formulae over time. We employ alternating automata (cf. section 2.1.1.4) like they are used in Model Checking.

2. For each *DLTL* formula we generate an aspect which at startup registers the initial configuration of an automaton with an evaluation engine. This initial configuration is equivalent to the given formula. Then, as the application runs, the aspect reports at each joinpoint of interest the currently active set of propositions to the engine, which then calculates the successor states under those propositions.

3. In most cases, the evaluation of a matching function μ_{χ_p} for a given proposition can entirely be handled by the AspectJ backend. Difficulties only arise when pointcuts use `if` pointcuts, because in those cases the semantics of *DLTL* and AspectJ differ: While `if` pointcuts in AspectJ can only access values exposed at the *current* joinpoint, *DLTL* allows to also access values which were defined by the formula on previous points in time. The solution is to extract `if` pointcuts (say `if(`*expr*`)`) from a proposition, replacing them by `if(`**`true`**`)` within the matching function while at the same time putting a *constraint* "*expr*" on the proposition[7].

3.5.1 General assumptions

For easier reading, we make some general assumptions which shall be assumed to hold throughout this chapter. In chapter 4 we will show how *J-LO* makes sure that those assumptions are met.

3.5.1.1 Alternating automata

In the following, for each $\varphi \in DLTL$ we denote by $\mathcal{A}_\varphi$ the *alternating automaton* for φ as it was first mentioned in definition 2.1.5. The construction can naturally be extended for formulae over *DLTL*.

[7]Note that it does not suffice to simply eliminate `if` pointcuts from the pointcut expression: For a pointcut `if(expr()) || pc()`, this would lead to a residual pointcut `pc()`, matching potentially less joinpoints than the original pointcut, while our technique generates `if(true) || pc()`, which matches *any* joinpoint, so that no joinpoint is lost.

3.5.1.2 Valid formulae

In this section we assume that each given $\varphi \in DLTL$ is *valid*, i.e. it holds that $valid(\varphi) = \mathbf{true}$.

3.5.1.3 Valid `if` pointcuts

For each pointcut `if(`*expr*`)` contained in any proposition of any given *DLTL* formula φ, we assume that *expr* is *valid* in the sense that for any possible valuation it does not throw an exception and so provides a Boolean value as result. This allows us to interpret *expr* as a function over bindings into $\mathbb{B}$. Note that because all formulae are valid, whenever *expr* is evaluated one can assume that either all variables in *expr* are bound or *expr* can safely be evaluated to **false**, as φ is assumed to be valid.

3.5.1.4 No garbage collection

For each object o being bound within a *DLTL* formula φ, we assume for this section that by binding o, this object is prevented from being garbage collected. In particular, o is *available* for subsequent matching and evaluation of `if` pointcuts. This is important because the declarative semantics assume that at the point in time where a proposition p is evaluated, its matching function μ_{χ_p} can be evaluated, meaning that it does not contain any free variables any more. If one of the objects referenced by μ_{χ_p} was garbage collected, this could lead to `null` being tried to dereferenced during the evaluation of μ_{χ_p} and hence to a runtime exception. We handle garbage collection by means of weak references as described in section 4.4.2.

3.5.1.5 No side effects

We assume that for a given specification $\Phi \subseteq DLTL$ it holds that the evaluation of any $\varphi_1 \in \Phi$ is side effect free, meaning that it has no impact whatsoever on the evaluation of any $\varphi_2 \in \Phi$ $(\varphi_2 \neq \varphi_1)$. Also we assume that the evaluation of any such $\varphi \in \Phi$ has no impact on the behaviour of the underlying application and hence, the verified path π is independent on the specification Φ. The implementation of *J-LO* makes sure that the application can be *oblivious* of the inserted instrumentation. From a theoretical point of view this might seem straightforward to achieve but in settings where AOP and multi-threading are involved, some issues can arise which easily give up this propery. For instance one has to make sure that accesses to shared variables are properly synchronized. Section 4.3.6 explains how we deal with such issues.

3.5.2 Basic Definitions

As done for the declarative semantics, we would first like to introduce some basic definitions. Most of them are necessary to provide a framework for the special treatment of `if` pointcuts.

Definition 3.5.1 (Constraints)
For each $\mathcal{V}' \subseteq \mathcal{V}$, we define the set $C_{\mathcal{V}'}$ of all *constraints* over variables from $\mathcal{V}'$ as:

$$C_{\mathcal{V}'} = \{c \mid c : B|_{\mathcal{V}'} \to \mathbb{B}\ \}$$

Those constraints are used to represent `if` pointcuts in the right context, where variables refer to values *previously defined on the timeline* rather than *at the current joinpoint* (which is the AspectJ semantics).

Definition 3.5.2 (Operational proposition)
Let $p \in \mathcal{P}$ with $\mathcal{P}$ as it was defined in definition 3.3.8 with $p = (l_p, \chi_p, k_p, \beta_p) \in L \times PC \times K \times B$.

We define the *operational proposition* $\hat{p}$ as: $\hat{p} := (l_p, \chi'_p, k_p, \beta_p, \vec{\gamma_p}) \in L \times PC \times K \times B \times 2^{C_{dom(\beta_p)}}$ where:

$$\begin{array}{rcl}
\chi'_p & := & (\mu'_{\chi_p}, \vec{v}_{\chi_p}, \sigma_{\chi_p}) \text{ with} \\
\mu'_{\chi_p} & := & \mu_{\chi_p}\text{where all } \texttt{if} \text{ pointcuts in } \mu_{\chi_p} \text{ have been replaced by } \texttt{if(true)}, \\
\vec{\gamma_p} & := & \{expr \mid \texttt{if(}expr\texttt{)}\text{is an } \texttt{if} \text{ pointcut in } \mu_{\chi_p}\} \\
 & & \text{the } \textit{set of constraints} \text{ of } \hat{p}.
\end{array}$$

We define the set $\hat{\mathcal{P}}$ of all such operational propositions as: $\hat{\mathcal{P}} := \{\hat{p} \mid p \in \mathcal{P}\}$.

Example 3.5.3 (Operational proposition)
As in example 3.3.10 we assume the following proposition p, where `o2` is used but not defined by p:

exit(**call**(Object Stack.pop()) && **if**(o1!=o2))
  returning o1

Then $\hat{p}$ is defined as $\hat{p} := (l_p, \chi'_p, k_p, \beta_p, \vec{\gamma_p})$ with:

$$\begin{array}{rcl}
\mu'_{\chi_p} & = & \texttt{call(Object Stack.pop()) \&\& if(true)} \\
 & = & \texttt{call(Object Stack.pop())} \\
\vec{\gamma_p} & = & \{\ \lambda\beta\ .\ (\ \beta(o_1) \neq \beta(o_2)\ )\ \forall\beta \text{ s.th. } \{\texttt{o1} \mapsto o_1,\ \texttt{o2} \mapsto o_2\} \subseteq \beta\ \}
\end{array}$$

and l_p, k_p, β_p as before.

Definition 3.5.4 (Matching of operational propositions)
For a state $s = (\iota_s, k_s) \in S$ and an operational proposition $\hat{p} = (l_p, \chi_p, k_p, \beta_p, \vec{\gamma_p})$ $\in \hat{\mathcal{P}}$ we say that $\hat{p}$ *matches* s or *holds* in s, $s \models \hat{p}$ for short, if the following holds:

1. $\mu_{\chi'_p}(\iota_s) = \mathbf{true}$,
2. $k_s \trianglelefteq k_p$,
3. $\forall \gamma \in \vec{\gamma_p} : \gamma(\beta_p) = \mathbf{true}$.

So the definition is essentially equivalent to definition 3.3.9, however additionally all constraints have to be fulfilled. Please note that the static analysis (cf. section 3.4) makes sure that each formula is valid, i.e. each binding function is rich enough to allow evaluation of all constraints.

Remark 3.5.5 (Implementation of matching)
The AspectJ based implementation automatically makes sure that the first two facts are always fulfilled: At each state, only those propositions hold, whose matching function matches the associated joinpoint and whose entry/exit kind is compatible with the one of the state. So the only condition that needs to be checked by our backend is that all constraints are satisfied under the current binding.

Definition 3.5.6 (Operational formula)
For each formula $\varphi \in DLTL$ we denote by $\hat{\varphi}$ the copy of φ where each proposition $p \in \mathcal{P}_\varphi$ is being replaced by $\hat{p}$.

Definition 3.5.7 (Alternating automaton for a formula $\varphi \in DLTL$)
We define the AFA $\mathcal{A}_\varphi$ essentially as it was done for the purpose of Model Checking (cf. section 2.1.5).

The only small problem that arises with this definition is the fact that the transition function δ is a function from Q into 2^{2^Q}. In an implementation we want to iterate the application of δ and hence we need to apply it to its own result. However, it is quite easy to overload δ in such a way that it is defined for elements of 2^{2^Q} as well. We are going to do so in section 3.5.2.1.

We call elements of 2^{2^Q} a *clause set* or a *configuration* of $\mathcal{A}_\varphi$. By overloading δ in the aforementioned way, one can start with an initial configuration of $\{\{\hat{\varphi}\}\}$ and then simply change to the successor configuration by applying δ.

We call a configuration q *accepting* if the following holds:

$$\exists c \in q \; \forall \psi \in c : \; (\; \psi = \mathbf{tt} \vee \exists \; \varphi_1, \psi_1 : \psi = (\varphi_1 \; \mathbf{R} \; \psi_1) \;)$$

Consequently we overload the set F of accepting states to a set F of accepting configurations in that way.

We define the language of $\mathcal{A}_\varphi$ as:

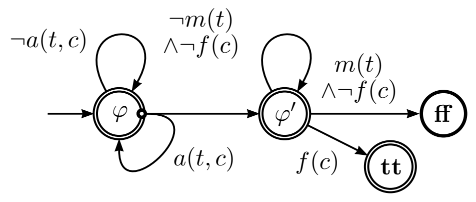
$$\mathcal{L}(\mathcal{A}_\varphi) := \{\pi \in S^+ \mid \text{after reading } \pi,\ \mathcal{A}_\varphi \text{ is in an accepting configuration } \}$$

The behaviour of such an AFA shall be reflected by the following example taken from [BS06].

Example 3.5.8 (Automaton for a *DLTL* formula)
Let us consider a tree-like data structure that is built up by recursively adding child nodes to a root. A child c can be added to a tree t, yielding the proposition $add(t, c)$. The proposition $modify(t)$ holds whenever t is modified and $finish(c)$ holds at the point in time where the creation of the whole subtree at c is finished. Now it is a requirement that t should not be modified until the creation of a subtree has finished. This can be modelled by:

$$\varphi := \mathbf{G}(\ add(t,c) \rightarrow (finish(c)\ \mathbf{R}\ \neg modify(t))\ )$$

Figure 3.5.8 shows the AFA $\mathcal{A}$ for φ and the corresponding run of its binding automaton for the following example. The initial formula, together with an empty binding, is the initial state of $\mathcal{B}_{\mathcal{A},\mathcal{V}}$. When reading $add(t_1, c_1)$, the automaton moves to $\varphi' := finish(c)\ \mathbf{R}\ \neg modify(t)$ with the actual binding of the current joinpoint and at the same time stays at φ with the original (empty) binding because of the self-loop that is part of the outer $\mathbf{G}$. Note that both states are within one and the same clause, which means that they are conjoined. (The corresponding edge is drawn with a bulleted origin.)



$$\{\ \{(\varphi, [t \mapsto \emptyset, c \mapsto \emptyset])\}\ \}$$
$$\downarrow add(t_1, c_1) \downarrow$$
$$\{\ \{(\varphi, [t \mapsto \emptyset, c \mapsto \emptyset]), (\varphi', [t \mapsto t_1, c \mapsto c_1])\}\ \}$$
$$\downarrow finish(c_1) \downarrow$$
$$\{\ \{(\varphi, [t \mapsto \emptyset, c \mapsto \emptyset])\}\ \}$$
$$\downarrow add(t_1, c_2) \downarrow$$
$$\{\ \{\varphi, [t \mapsto \emptyset, c \mapsto \emptyset]), (\varphi', [t \mapsto t_1, c \mapsto c_2])\}\ \}$$
$$\downarrow modify(t_1) \downarrow$$
$$\underbrace{\{\ \{(\varphi, [t \mapsto \emptyset, c \mapsto \emptyset]), \underbrace{(\mathbf{ff}, [t \mapsto t_1, c \mapsto c_2])}_{nonfinal}\}\ \}}_{nonfinal}$$

Figure 3.2: Alternating automaton for a *DLTL* formula

Remark 3.5.9 (Transition function)
The transition function δ is the essence of our implementation. Hence we wish to describe it in detail.

When taking a transition from one configuration to another, this transforms one set of clauses to another. Each clause holds subformulae of $\hat{\varphi}$. All the transition function has to do is to make sure that for each $\pi[i]$ the appropriate successor states are generated for all current valuations at $\pi[i]$, as it was described in section 3.3.13.

Example 3.5.10 (Operational semantics by example)
Take for example the following formula, where we write $q(y)_{y \neq x}$ for the proposition $q(y, \underline{x})$ where x is used in a constraint $y \neq x$:

$$\varphi(x, y) := \mathbf{G}(p(x) \rightarrow \mathbf{XF}\ q(y)_{y \neq x})$$

and the trace $\pi := \{p(1), p(2)\}\{q(2)\}\{q(3)\}$, which satisfies φ.

The alternating automaton would start in configuration $q_0 := \{\{\varphi(x, y)\}\} \in F$. At the first state $\pi[0] = \{p(1), p(2)\}$ we would get possible valuations of $x \in \{1, 2\}$. It holds that $\pi[0] \models \mathit{now}(\varphi(x, y)) = \mathbf{true}$. For valuations $x \in \{1, 2\}$ we get $\mathit{next}_{\varphi(x,y)}(\varphi(1, y))\{p(1), p(2)\} = \varphi(x, y) \wedge \mathbf{F}\ q(y)_{y \neq 1}$ respectively $\mathit{next}_{\varphi(x,y)}(\varphi(2, y))\{p(1), p(2)\} = \varphi(x, y) \wedge \mathbf{F}\ q(y)_{y \neq 2}$. In the terms of alternating automata operating on clause sets, this yields a successor configuration of:

$$q_1 := \{\ \{\varphi(x, y), \mathbf{F}\ q(y)_{y \neq 1}\},\ \{\varphi(x, y), \mathbf{F}\ q(y)_{y \neq 2}\}\ \}$$

Note that $q_1 \notin F$.

This is the next *current* configuration for the evaluation of $\pi[1]$.

At $\pi[1] = \{q(2)\}$ we only have one valuation: $y \in \{2\}$. Under this valuation, the subformulae $\varphi(x, y)$ and $\mathbf{F}\ q(y)_{y \neq 2}$ remain unchanged, because the former only "reacts" on p and in case of the latter the constraint $y \neq 2$ evaluates to **false** under the valuation $y = 2$. Hence, $\pi[1] \not\models q(y)_{y \neq 2}$ for $y = 2$. In the case of the subformula $\mathbf{F}\ q(y)_{y \neq 1}$ it holds that $\pi[1] \models q(y)_{y \neq 1}$ for $y = 2$ and so $\mathbf{F}\ q(y)_{y \neq 1}$ evaluates to **true**, which is $\{\ \emptyset\ \}$ in the terms of alternating automata. This makes the subformula simply disappear and yields the successor state:

$$q_2 := \{\ \{\varphi(x, y)\},\ \{\varphi(x, y), \mathbf{F}\ q(y)_{y \neq 2}\}\ \}$$

In the last state $\pi[2] = \{q(3)\}$ it holds that $\pi[2] \models_{y=3} q(y)_{y \neq 2}$ and so we obtain:

$$q_3 := \{\ \{\varphi(x, y)\}\ \} = q_0$$

Note that $q_3 \in F$, since q_3 is a *Release* formula. Hence, $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$.

When looking at state q_2 we can make the following observation.

Observation 3.5.11 (Minimal specification)
The configuration q_2 specifies that the formula $\varphi(x,y) \vee (\varphi(x,y) \wedge \mathbf{F}\ q(y)_{y\neq 2})$ has to hold on the subsequent path. It holds that $\varphi(x,y) \vee (\varphi(x,y) \wedge \mathbf{F}\ q(y)_{y\neq 2}) = \varphi(x,y) \wedge \mathbf{F}\ q(y)_{y\neq 2}$. This yields the state $q_2' := \{\ \{\varphi(x,y), \mathbf{F}\ q(y)_{y\neq 2}\}\ \}$, which is equivalent to q_2. We call q_2' the *minimal specification* for q_2. The observation leads to what we call the *subsumption reduction*.

Definition 3.5.12 (Subsumption reduction)
Let $\varphi \in DLTL$ and $q = \{c_1, \ldots, c_n\} \in 2^{2^{cl(\varphi)}}$ where the c_i are clauses, sets of states. Then we define the *subsumption reduction* of q, $ssr(q)$ as:

$$ssr(q) := \begin{cases} q & \text{if } |q| < 2 \\ \bigcup\limits_{i\neq j} \{c_i \cap c_j\} & \text{otherwise} \end{cases}$$

Theorem 3.5.13 (Correctness of subsumption reduction)
Let $\varphi \in DLTL$, and $q \in 2^{2^{cl(\varphi)}}$. Let $s \in S$. Let $\mathcal{A}_{\varphi,q}$ the copy of $\mathcal{A}_\varphi$ with an initial configuration of q. Then it holds that:

$$s \in \mathcal{L}(\mathcal{A}_{\varphi,q}) \iff s \in \mathcal{L}(\mathcal{A}_{\varphi,ssr(q)})$$

This theorem is known as Lee's Theorem [Lee67]. The easy proof is left as an exercise to the interested reader. It can be looked up in [Lee67].

3.5.2.1 Definition of δ

Let $s \in S$ and $\varphi \in DLTL$. Further let $\mathcal{P}' := \mathcal{P}^{act}_\varphi(s) \subseteq \mathcal{P}$ and $B' := B^{act}_\varphi(s)$.

Then, according to definition 3.5.7, the AFA $\mathcal{A}_\varphi$ equivalent to φ is defined as: $\mathcal{A}_\varphi := (Q, \Sigma, q_0, \delta, F)$, where we define $\delta : Q \times \Sigma \to Q$ by:

$$\begin{aligned} \delta(q,s) &:= \delta(q, \underbrace{\mathcal{P}^{act}_\varphi(s)}_{=:\mathcal{P}'}, \underbrace{B^{act}_\varphi(s)}_{=:B'}) && (3.1)\\ \text{where} \quad \delta(\{c_1,\ldots,c_n\}, \mathcal{P}', B') &:= \bigcup_{1\le i\le n} \delta(\{c_{i_1},\ldots,c_{i_{n_i}}\}, \mathcal{P}', B'), && (3.2)\\ \text{where} \quad \delta(\{\varphi_1,\ldots,\varphi_m\}, \mathcal{P}', B') &:= \bigotimes_{1\le i\le m} \delta(\varphi_i, \mathcal{P}', B'), && (3.3)\\ \text{where} \quad \delta(\varphi, \mathcal{P}', B') &:= \bigotimes_{\beta\in B'} \delta(\varphi, \mathcal{P}', \beta), && (3.4)\\ \text{where} \quad \delta(\varphi, \mathcal{P}', \beta) &:= \delta(\varphi, \mathcal{P}', \beta, def(\varphi)). && (3.5) \end{aligned}$$

Here equation 3.1 reduces δ of a configuration and a state to δ of a configuration, the active propositions and bindings at this state. Equation 3.2 then reduces δ

of a configuration to the join of δ of all clauses. Then equation 3.3 reduces δ of a clause to the clause product of the results for all single formulae. Equation 3.4 reduces δ of a formula and a set of valuations to the clause product of all results for a single valuation and this formula. This reflects the fact that a formula should hold *for all* possible valuations at a given state. Eventually 3.5 defines the start of a recursive descent, initializing a context $\mathcal{D}$ to the set of variables defined by φ, $\mathit{def}(\varphi)$. The recursion then continues as follows.

Definition 3.5.14 (Filtered bindings)
For any set of defined variables $\mathcal{D} \subset \mathcal{V}$ we define the filtered binding $\beta|_{\mathcal{D}}$ as: $\beta|_{\mathcal{D}} := \{x \mapsto o \in \beta \mid x \in \mathcal{D}\}$

Remark 3.5.15 (Specialization of bindings)
The declarative semantics include a *specialization* step, where *unbound* bindings are replaced by bindings to objects. In order to reflect this on the operational side, for any $\beta \in \mathcal{V} \to O$, we overload β with $\hat{\beta} : \mathcal{P} \to \mathcal{P}$ such that for each $p = (l_p, \chi_p, k_p, \beta_p) \in \mathcal{P}$:

$$\hat{\beta}(p) := (l_p, \chi_p, k_p, \beta'_p)$$

where

$$\beta'_p := \{\{x \mapsto o\} \in \beta_p \mid o \neq \oslash\} \;\cup\; \{\{x \mapsto o\} \in \beta \mid \beta_p(x) = \oslash\}.$$

For whole sets of propositions $\mathcal{P}' \subseteq \mathcal{P}$ we define respectively $\hat{\beta} : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ as:

$$\hat{\beta}(\mathcal{P}') := \{\hat{\beta}(p) \mid p \in \mathcal{P}'\}$$

Using those definitions, we can define the recursive descent of δ over φ as follows:

$$\begin{array}{rcl}
\delta(p, \mathcal{P}', \beta, \mathcal{D}) & := & \begin{cases} \{\ \emptyset\ \} & \text{if } \forall \gamma \in \vec{\gamma_p} : \gamma(\beta) = \mathbf{true} \wedge \hat{\beta}|_{\mathcal{D}}(p) \in \hat{\beta}|_{\mathcal{D}}(\mathcal{P}') \\ \emptyset & \text{otherwise} \end{cases} \\
\delta(\neg p, \mathcal{P}', \beta, \mathcal{D}) & := & \begin{cases} \{\ \emptyset\ \} & \text{if } \delta(p, \mathcal{P}', \beta, \mathcal{D}) = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\delta(\mathbf{X}\ \varphi, \mathcal{P}', \beta, \mathcal{D}) & := & \hat{\beta}|_{\mathcal{D}}(\varphi) \\
\delta(\varphi \wedge \psi, \mathcal{P}', \beta, \mathcal{D}) & := & \delta(\varphi, \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi \wedge \psi)\ ) \otimes \\
 & & \delta(\psi, \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi \wedge \psi)\ ) \\
\delta(\varphi \vee \psi, \mathcal{P}', \beta, \mathcal{D}) & := & \delta(\varphi, \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi \vee \psi)\ ) \cup \\
 & & \delta(\psi, \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi \vee \psi)\ ) \\
\delta(\varphi\ \mathbf{U}\ \psi, \mathcal{P}', \beta, \mathcal{D}) & := & \delta(\ \psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))\ , \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi\ \mathbf{U}\ \psi)\ ) \\
\delta(\varphi\ \mathbf{R}\ \psi, \mathcal{P}', \beta, \mathcal{D}) & := & \delta(\ \psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))\ , \mathcal{P}', \beta,\ \mathcal{D} \cup \mathit{def}(\varphi\ \mathbf{R}\ \psi)\ )
\end{array}$$

Here the only interesting equation is the first one specifying the successor for a propositional state p. Here, according to remark 3.5.5 we make sure that all constraints of p are fulfilled under the given binding β and then check of the specialized version of p under this binding matches one of the propositions that hold at the current state.

Based on this definition on the AFA $\mathcal{A}_\varphi$, we now define the operational semantics.

3.5.3 Operational semantics of a *DLTL* formula

Definition 3.5.16 (Operational semantics of a *DLTL* formula)
Let $\varphi \in \mathit{DLTL}$, $\pi \in S^+$. We say that π is a valid path for φ, if and only if:

$$\pi \in \mathcal{L}(\mathcal{A}_\varphi).$$

In the following we want to prove that the declarative and operational semantics coincide.

Theorem 3.5.17 (Equivalence of declarative and operational semantics)
Let $\varphi \in \mathit{DLTL}$, $\pi \in S^*$. Then:

$$\pi \models \varphi \iff \pi \in \mathcal{L}(\mathcal{A}_\varphi)$$

3.5.4 Proof of equivalence of declarative and operational semantics

The proof is structured as follows:

First we briefly give reasons for why an approach based on alternating automata [MSS88] is correct in general. A formal proof was conducted by Vardi [Var96] who introduced such automata for the purpose of LTL Model Checking.

What follows is the proof of equivalence on the level of a single valuation and a single proposition. Based on the assumption that the semantics coincide on this level, it is then easy to prove equivalence of the whole semantics.

3.5.4.1 Correctness of alternating automata

Vardi gave a formal correctness proof for alternating automata over infinite paths in chapter 3 of [Var96]. The idea is a simple induction. Based on the definition of δ it is obvious that the successor configuration of a configuration c equivalent to a formula φ holds exactly the Boolean combination of states, which represent $\mathit{next}(\varphi)$. The definition of the acceptance set F in our case is consistent with the fact that each configuration represents a disjunct of conjuncts. A run of an AFA is accepting in this model, if there exists at least one clause in the final configuration such that all states in this clause are *Release* formulae. This reflects that obligations (eventualities) which are put on a path are represented by *Until* formulae. So when there exists a clause without such obligations, this means that there exists a run such that all obligations have been fulfilled. In the other case where no such clause exists, this means that in all clauses at least one *Until* formula exists and hence on each run there is at least one obligation not fulfilled.

3.5.4.2 Correctness on the propositional level

Let $s = (\iota_s, k_s) \in S$ and $p = (l, \chi, k, \beta) \in \mathcal{P}_\varphi$ for some formula φ. We need to show that:

$$s \models p \iff \delta(p, \mathcal{P}_\varphi^{act}(s), \beta) = \{ \emptyset \}$$

Completeness ($\Rightarrow$):

Assume $s \models p$. Then according to section 3.3.9 the following holds:

- $\mu'_\chi(\iota_s) = \mathbf{true}$,
- $k \trianglelefteq k_s$

for $\mu'_\chi := (\tilde{\sigma_\mu} \circ \tilde{\beta})\ (\mu_\chi)$.

On the operational side it holds for any context $\mathcal{D}$ that:

$$\delta(p, \mathcal{P}_\varphi^{act}(s), \beta) = \delta(p, \mathcal{P}_\varphi^{act}(s), \beta, \mathcal{D}) := \begin{cases} \{ \emptyset \} & \text{if } \beta|_\mathcal{D}(p) \ \in \ \beta|_\mathcal{D}(\mathcal{P}_\varphi^{act}(s)) \\ \emptyset & \text{otherwise} \end{cases}$$

So we need to show that:

$$s \models p \Rightarrow \beta|_\mathcal{D}(p) \ \in \ \beta|_\mathcal{D}(\mathcal{P}_\varphi^{act}(s)).$$

Due to our static analysis we know that $\beta|_\mathcal{D}$ is rich enough so that there are no free variables in $\beta|_\mathcal{D}(p)$. Since $s \models p$ we know that there is a $p' \in \mathcal{P}_\varphi^{act}(s)$ such that $\mu_{\chi'_{\hat{p'}}}(\iota_s) = true$ and $k_s = k_{p'}$. (This is assured by the AspectJ implementation; cf. remark 3.5.5.) Since $\beta|_\mathcal{D}$ is also rich enough to bind all free variables in p', and obviously $k_s = k$, it holds that $p' = p$. Hence $\beta|_\mathcal{D}(p) \ \in \ \beta|_\mathcal{D}(\mathcal{P}_\varphi^{act}(s))$ holds as well.

Soundness ($\Leftarrow$):

The argument here is the very same. Since $\delta(p, \mathcal{P}(s), \beta) = \{ \emptyset \}$ holds, we know that $\beta|_\mathcal{D}(p) \ \in \ \beta|_\mathcal{D}(\mathcal{P}')$. Hence it also holds that $\mu'_\chi(\iota_s) = \mathbf{true}$ and thus also $s \models p$.

3.5.4.3 Correctness in general

For proving the general correctness, we distinguish the two cases of the declarative semantics where the current state $\pi[0]$ either satisfies $now(\tilde{\beta}(\varphi(\vec{x})))$ or not.

Case $\pi[0] \not\models now(\tilde{\beta}(\varphi(\vec{x})))$:

What we need to show is that in this case $\delta(\varphi, \pi[0], \beta) = \emptyset$. Since $\pi[0] \not\models now(\tilde{\beta}(\varphi(\vec{x})))$ holds, we know that even under the assumption that the subsequent path fulfills all obligations guarded by the $\mathbf{X}$ operator, the path violates φ. Hence, it suffices to show that $\delta(now(\varphi), \pi[0], \beta) = \emptyset$. We want to prove the claim by contradiction:

Assume that $\delta(now(\varphi), \pi[0], \beta) = \{c_1, \ldots, c_n\}$ with $n > 0$ and $\exists i(1 \leq i \leq n) : |c_i| > 0$. Let $\varphi' \in c_i$ for one such i. By definition, $now(\varphi)$ is a Boolean combination of propositions. Hence, the calculation of $\delta(now(\varphi), \pi[0], \beta)$ is fully reduced to δ of Boolean combinations of **ff** and **tt** (resp. their equivalents $\emptyset$ and $\{\ \emptyset\ \}$). Hence it suffices to concentrate on the Boolean connectives. For the clause product $\otimes$ and any clause sets $s1, s2$ it holds that $|s1 \otimes s2| > 0 \iff |s1| > 0 \wedge |s2| > 0$. For the join operation $\cup$ and any clause sets $s1, s2$ it holds that $|s1 \cup s2| > 0 \iff |s1| > 0 \vee |s2| > 0$. So due to our assumption, it has to hold that $now(\varphi)$ consists of a Boolean combination such that for each *join* term the evaluation of at least one of the two branches results in $\{\ \emptyset\ \}$ and for each $\otimes$ term, the evaluation of both branches results in $\{\ \emptyset\ \}$. If this were the case, however, this would also mean that $\pi[0] \models now(\varphi(\beta(\vec{x})))$ which violates the assumption.

Case $\pi[0] \models now(\varphi(\beta(\vec{x})))$:

In this case we need to show that if $\delta(\varphi, \pi[0]) = c$ it holds that $\delta^*(c, \pi^1) \in F \iff \pi^1 \models \varphi$ (where δ^* is the transitive hull of δ).

When looking at δ, one can easily see that the only case where formulae are "produced" for the successor configuration is the one of $\varphi = \mathbf{X}\ \varphi'$: Here the inner formula φ' is added - as a copy with specialized bindings. So all formulae contained in c are subformulae of φ. Also, only those φ' with $\varphi' \neq \varphi$ are specialized, according to the definition of *next* for *DLTL*. Hence it should be clear that c is equivalent to $next_{\varphi(\vec{x})}(\varphi(\beta(\vec{x})))$. Since the subsumption reduction is sound and complete, it follows that $\delta^*(c, \pi^1) \in F \iff \pi^1 \models \varphi$.

■

Chapter 4

Implementation

The implementation of the runtime environment follows almost completely the operational semantics. As the instrumented application runs, this runtime environment is triggered by *aspects* which are generated from formulae. Hence, in a first step we extract formulae from the annotations of Java bytecode. This is explained in section 4.1. For each formula $\varphi \in DLTL$ we generate an aspect in the AspectJ language. This code generation is performed using an extended version of the *AspectBench Compiler* (*abc*) [ACH$^+$05]. This compiler is introduced in section 4.2. After code generation the generated aspects are instantly woven into the original application by the *abc* backend. Figure 4.1 recalls the workflow of *J-LO*. Further details about how we extended *abc* in order to accomplish this code generation can be found in our seminar paper [Bod05c]. In this work we only want to give an overview of the employed tools. The details of the generated code are given in section 4.3. Here we describe the generated aspects and their members and explain how they relate to the operational semantics. Section 4.4 explains the treatment of special runtime behaviour such as exceptions, garbage collection and application shutdown. In particular our implementation does not prevent any objects of the underlying application from being garbage collected. In order to do so, we use special hash maps with weak references (as they were explained in section 2.5) as values.

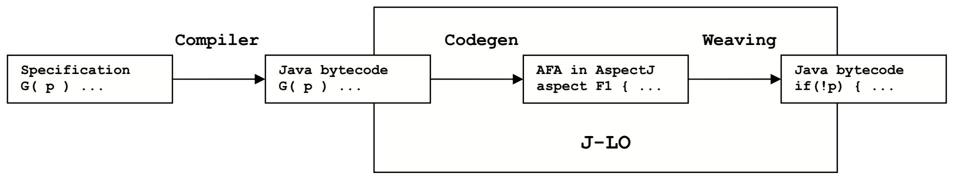


Figure 4.1: Workflow of *J-LO* usage

Figure 4.1 is repeated to remind the reader of workflow of *J-LO*.

4.1 Annotation extraction

For annotation extraction in *J-LO*, we used the bytecode engineering toolkit *Bat2* [Eic05] which is being developed at the Darmstadt Software Technology Group and is an offspring of the *Magellan* [EMOS04] framework for cross-artefact information retrieval. Specifically, we used *BAT2XML*, an extension of BAT2, which allows for transformation of classes in the Java bytecode format into an XML representation. *J-LO* uses BAT2XML to generate an XML representation for each given class. This XML representation is then parsed in order to extract the LTL formula annotations in String format, using standard techniques.

BAT2XML allows to preserve line number information contained in the Java bytecode as well as debug information such as the `.java` source file which generated the corresponding bytecode file. Though not yet implemented, future versions of *J-LO* could make use of this information in order to point the user to the location[1] where a formula was specified for the purpose of debugging.

As a result of this process, for a given set of class files, *J-LO* holds a list of LTL formulae specified in those classes. The formulae are available in String format, which means that they have to be parsed to be processed any further. This parsing is accomplished with an extended version of the AspectBench compiler and also performed at compile time.

In *J-LO* we install *one* formula for each annotation at the startup of the application. Yet, the runtime environment is is open for extensions, so that one could also install formulae dynamically, if necessary.

4.2 The AspectBench compiler

Unfortunately in the past, many proposed AspectJ language extensions have gone into different builds of various compilers - mostly into the *ajc* [HH04] compiler (the original implementation by PARC) but also into others like JAsCo [SVJ03], AspectWerkz [Bon04] or in the form of hand coded preprocessors. The AspectBench Compiler (*abc*) [ACH$^+$05] which was developed in cooperation of the Brics research center, the McGill University, and Oxford University, now facilitates such extensions by providing an extensible, optimizing compiler for the AspectJ programming language. This enables researchers to implement and/or port such extensions into one common framework and so reuse their implementations, as we described in [Bod05a].

[1]Unfortunately this information may not be 100% exact since annotations themselves have no line numbers attached in the bytecode - only executable code has. Hence one would have to approximate the location of the annotation e.g. by assigning the line number of the next executable line of code.

File **X** $S ::= a$ $\quad\mid b$ $\quad\mid c$	File **Y** *include* **X** *extend* $S ::= d$ $\quad\mid e$
File **Z** *include* **Y** *drop* $S ::= b$ $\quad\mid d$	**Result** $S ::= a$ $\quad\mid c$ $\quad\mid e$

Figure 4.2: Grammar extension mechanism

4.2.1 Structure of *abc*

All major Java-based compilers for AOP languages today, are so-called *weaving compilers*: They have two major passes, one compilation pass, where the aspects are translated into Java bytecode using a special compiler for that language, and one weaving pass, where calls to the appropriate pieces of advice are woven into the actual core application at all the places where pointcuts apply. Runtime checks are inserted at all the necessary places.

As such a compiler, *abc* is based on two major frameworks: The compiler front-end is the *Polyglot* [NCM03] compiler toolkit. Polyglot is a compiler framework built as a front-end to PPG, an extensible LALR parser generator based on the CUP LALR parser generator for Java. In PPG, existing grammars can optionally be extended by *extending* or *dropping* productions of a base grammar. The example in Figure 4.2 (using simplified non-PPG syntax) demonstrates the basic principles. An existing grammar can be imported with the *include* keyword. New production rules can then be specified, and one can change existing rules using the keywords *extend* and *drop* to add and remove parts of the rule. More advanced changes, such as modifying the precedence of operators, are also possible. For further details on the specification of grammar, see [BM03]. Also, Polyglot uses object association in favour over class inheritance, employing a sophisticated delegation model. This allows extenders to add or replace functionality piece by piece to distinct node types of the abstract syntax tree (AST) which do not need to share common super types.

As the weaving backend, the bytecode analysis and optimization framework *Soot* is employed. Soot is able to load Polyglot ASTs and/or Java bytecode and transform those into an internal three address code representation called *Jimple*. This representation is stackless and as such allows for relatively easy code transformations and analyses. The weaving process which implements the translation from AspectJ into plain Java, makes use of this representation. Since Soot is also an optimization framework, many intra- and interprocedural analyses are already built-in and can easily be extended. They can be applied to the readily woven code at once, thus generating more efficient code than *ajc* does, in certain situations. With respect to compile time performance however, *abc* tends to be slower than ajc due to it's heavily object-oriented structure. Whereas ajc is optimized for compile time performance, *abc* is optimized for extensibility and run time performance of the resulting bytecode.

Figure 4.3 gives an overview of the design of *abc*.

4.2.2 Polyglot

Polyglot as the *abc* compiler frontend facilitates easy extensibility not only by inheritance but also in other directions, by the means of object composition. This is an enormous benefit over earlier approaches in compiler technologies, which usually only allowed extensibility by class inheritance, and as such is truly one-dimensional: Each AST node inherits functionality from its parent nodes and from nowhere else. During the last years however, many authors like Gamma et al. [GHJV95] have suggested to use object composition in favour over class inheritance, because it tends to lead to a more flexible system design. Polyglot makes consequent use of the delegation pattern, that allows for such object composition:

Each AST node, whenever visited, dispatches this message first to its delegate object, which by default is the visited object itself. Figure 4.4 shows how this delegation model behaves during type checking.

Using this mechanism, one can easily *replace* or extend functionality that is spread over various node types, which do not need to share a common super type.

In addition to delegates, nodes also support a chain of extension objects. An extension is meant to *add members* to a set of node types.

Polyglot also supports type checking and other semantic passes for the Java language. However since we are performing a source to source transformation, we are not fully exploiting those facilities. We only make use of them implicitly through the final transformation processes to Java bytecode.

4.2.3 Soot

Soot [VRHS$^+$99] is a bytecode analysis and optimization framework, which provides common templates for inter- and intraprocedural analyses. Several such analyses are already built-in. They comprise even complex *points-to* and

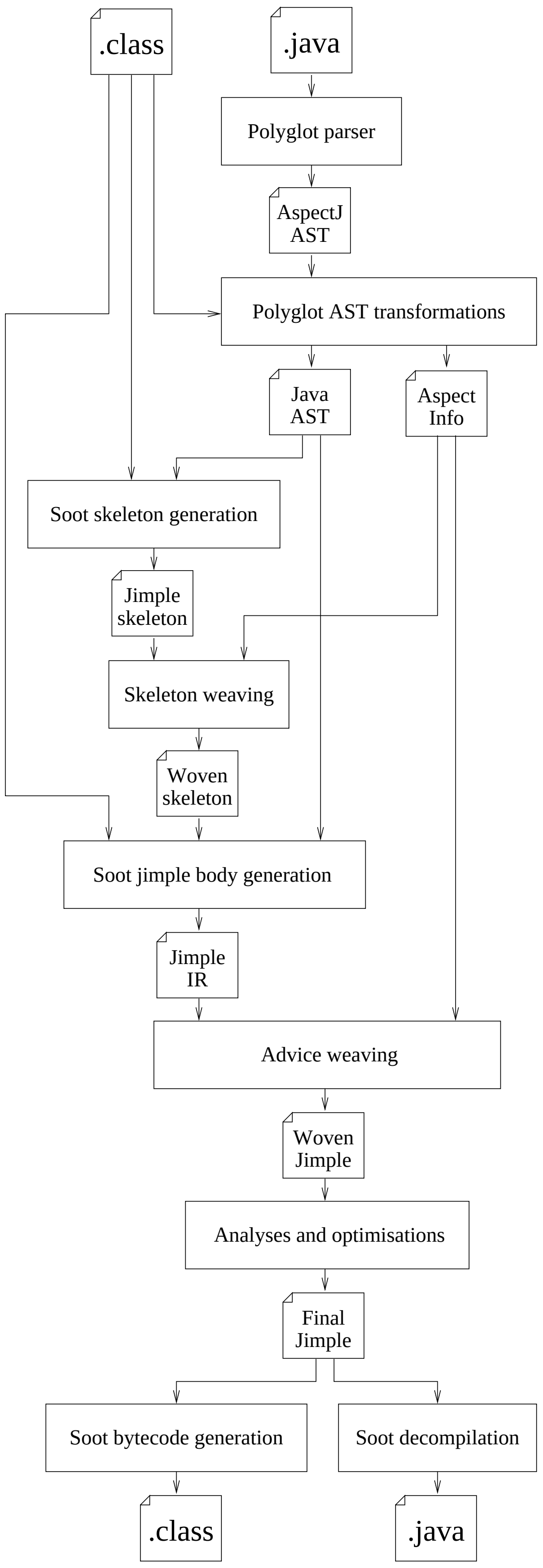


Figure 4.3: Design of the AspectBench Compiler

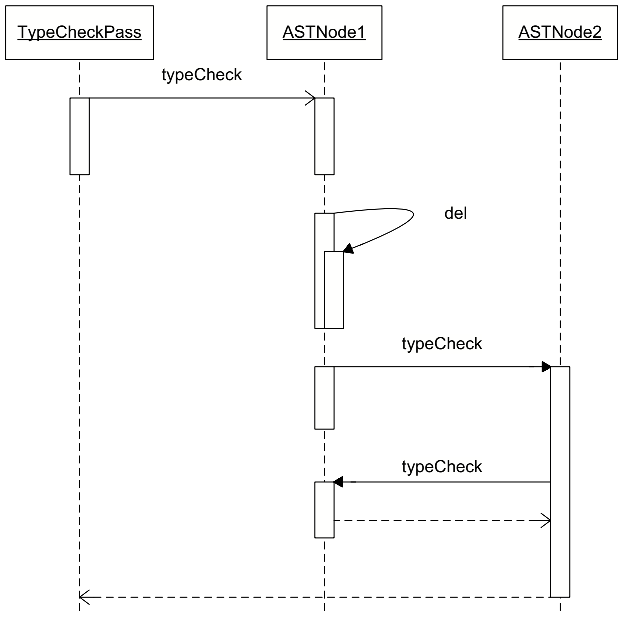


Figure 4.4: Polyglot delegation model (with call back to original receiver)

flow analyses, which can be used to reason about control flow, possible method dispatches at runtime and so forth. Obviously, by making use of information produced by such static analyses, an AspectJ compiler can generate much more efficient code under certain circumstances. For instance, the evaluation of *cflow* could be dramatically accelerated by replacing stacks with counters, which is possible in most common situations [DGH$^+$04].

Nevertheless, Soot is in the first place used within *abc* because of the *Jimple* representation it provides. A Jimple program consists of a stackless, three-address code (executing object, arguments and result) representation of Java bytecode. In Jimple, all implicit method invocations (e.g. String concatenation) and implicit references to the currently executing object (`this`) have been resolved. As a result, all objects that contribute to the implementation of a method body are explicitly available in a local variable and each statement consists only of at most one method call and one assignment. This makes Jimple easy to process and an ideal base for modifications as they have to be performed by the advice weaving process.

Figure 4.5 gives an example of this representation: Figure 4.5(a) defines a class in normal Java syntax while figure 4.5(b) shows the corresponding Jimple code.

In *abc*, weaving is implemented by generating a so-called *AspectInfo* data structure, which describes transformations on the level of Jimple code. This code can then, using Soot, be transformed to bytecode or source code again. The latter is particularly useful for educational purposes, since one can see at once, how advice weaving affects given classes.

```
public class Foo {
    int a;

    public int f(int x,int y , int z) {
        return a+x*y+z;
    }
}
```

(a) Java code

```
public class Foo extends java.lang.Object {

    int a;

    public int f(int, int, int) {

        Foo this;
        int x, y, z, $i0, $i1, $i2, $i3;

        this := @this: Foo;
        x := @parameter0: int;
        y := @parameter1: int;
        z := @parameter2: int;
        $i0 = this.<Foo: int a>;
        $i1 = x * y;
        $i2 = $i0 + $i1;
        $i3 = $i2 + z;
        return $i3;
    }

    //Implicit constructor omitted
}
```

(b) Jimple code

Figure 4.5: Java class and corresponding Jimple code

This shall conclude our brief overview of the *abc* framework. Further details about how we extend it to achieve the desired code generation can be found in [Bod05c]. The next section will explain what the generated code looks like and why it fulfills the necessary requirements for the operational semantics.

4.3 Code generation

Generally, for each specified formula, we generate a single aspect. Each such aspect defines implicitly a *singleton* object (see [GHJV95] for a description of the *Singleton* Design Pattern). This object is automatically instantiated at the first time a piece of advice of this aspect is to be executed. All fields we declare on such an aspect are of *private* scope and hence only visible to the declaring aspect. This ensures the desired property that evaluation of a single formula should not interfere with the evaluation of other formulae - at least for the case of single-threaded applications. For the case of multi-threaded applications we need to handle some synchronization issues. This is described in section 4.3.6.

The following subsections explain the components which are generated for each formula/aspect. We assume that a formula $\varphi \in DLTL$ is given.

4.3.1 Propositions

For each proposition in $\mathcal{P}_\varphi$, we generate a constant of type `IProposition` using the following factory method of the class `IFormulaFactory`:

```
IProposition Proposition(
    String propLabel,
    String []  boundFormals,
    IIfClosure []  ifClosures
)
```

The parameters have the following semantics:

- `propLabel` - The textual representation of this proposition, corresponding to l_p.
- `boundFormals` - An array holding all names of variables which are bound by this proposition, corresponding to $\vec{v}_{\chi_p}$.
- `ifClosures` - An array of *if-closures* (constraints) that have to be fulfilled by this proposition, corresponding to γ_p.

```
public interface IIfClosure {

    public boolean satisfiedUnderBindings(
        WeakValuesMap<String, Object> currentBinding
    ) throws UserCausedException;

    public String[] variableNames();

    public String toString();

}
```

Table 4.6: Interface for if-closures

4.3.1.1 If-closures

An *if-closure* is a simple closure that encapsulates the evaluation of a constraint. Each such closure adheres to the interface shown in table 4.6.

The method `satisfiedUnderBindings` returns for a given binding if the expression represented by this closure is satisfied under the given bindings. The parameter `currentBinding` represents the function β, which maps variable names (type `String`) to objects (type `Object`). In cases where the evaluation of the represented expression leads to an exception, this exception is wrapped in a `UserCausedException`. This mechanism allows *J-LO* to gracefully report such exceptions to the user instead of just shutting down.

The method `variableNames` returns an array of names of all variables that are used in the expression this closure represents. This is used within the proposition to enable binding of those variables.

The method `toString` returns a String representation of the associated expression for debugging purposes.

Example 4.3.1 (If-closure)
For a pointcut `if(s!=t)` we generate the closure shown in table 4.7.

In lines 5-8, the values for `s` and `t` are retrieved from the map. In the case where those values have wrong types, a `ClassCastException` is thrown and **false** is returned.

Line 10 then evaluates the actual expression. In the case where `s` or `t` are `null`, a `NullPointerException` is thrown and **false** is returned[2].

If everything goes fine, the Boolean value of the expression is returned. If the evaluation of line 10 causes an exception this is known to be due to invalid

[2]Actually the runtime library ensures type safety and non-nullness of `s` and `t` so this check is really pedantic.

```
1  new IIfClosure() {
2      public boolean satisfiedUnderBindings(
3              WeakValuesMap currentBindings) {
4          try {
5              final Singleton s =
6                  (Singleton) currentBindings.get(”s”);
7              final Singleton t =
8                  (Singleton) currentBindings.get(”t”);
9              try {
10                 return s != t;
11             } catch (java.lang.Exception ex) {
12                 throw new IIfClosure.UserCausedException(
13                         ex,
14                         ”s != t”
15                 );
16             }
17         } catch (java.lang.NullPointerException ex) {
18             return false;
19         } catch (java.lang.ClassCastException ex) {
20             return false;
21         }
22     }
23
24     public java.lang.String[] variableNames() {
25         return new java.lang.String[] { ”s”, ”t” };
26     }
27
28     public java.lang.String toString() {
29         return ”s != t”;
30     }
31 }
```

Table 4.7: Example if-closure

input. Hence, we wrap the exception to be recognized as *caused by the user* (lines 12-15).

This concludes our examination of if-closures and propositions.

Each proposition is assigned to a private variable `prop<i>` where `<i>` is a natural number ≥ 0.

Those propositions are then combined with temporal operators to form the actual formula.

4.3.2 Initial formula

A private final field `formula` is generated and initialized with a term representation of the formula, using the previously defined propositions as atoms. For instance the formula of example 3.5.10, $\mathbf{G}(\ p(x) \rightarrow \mathbf{XF}\ q(y)_{y \neq x}\ )$, would induce the representation shown in table 4.8.

```
private final IFormula formula =
    factory.G(
        factory.Impl(
            prop0,
            factory.X(
                factory.F(prop1)
            )
        )
    );
```

Table 4.8: Example formula instantiation

We can see that although Java has no native support for algebraic data types, one can easily get equivalent functionality by chaining objects together to trees. *J-LO* instantiates such formulae at startup of the application. As one can see, it would be no problem though, to install further formulae as the application runs. This could be a desirable feature for future work.

4.3.3 Initialization/bootstrapping code

Initialization is performed within the constructor of the aspect. The constructor registers the initial formula (see section 4.3.2) with the `VerificationRuntime`. This induces a small problem: Aspects are instantiated lazily. The pointcuts generated for an aspect are defined by the propositions contained in the formula. An aspect is instantiated immediately before the first time, a piece of advice of this aspect is to be executed. Liveness conditions such as $\mathbf{F}\ p$ would imply that whenever p does *not* occur on a path, the aspect would not even be instantiated, because μ_{χ_p} never matches and hence no advice is executed. This would mean that the formula would never be installed and hence not be verified. Therefore we generate an additional empty advice in each such aspect, which just initializes the aspect at startup[3] :

```
before():
execution (public static void *.main(String[])) {}
```

[3]We are aware of the fact that Java applications can be run without actually having a `main` method by bootstrapping the application within a `static` block. However we believe that this is not actually made use of in any real Java application.

Apart from those members which define and register a formula, a verification aspect also contains a mechanism for collecting propositions at joinpoints of interests and for triggering transitions of the associated AFA.

4.3.4 Mechanism for collecting propositions

Propositions are collected using the set `currentProps`, which is private to each aspect. Each time when the matching function pointcut μ_{χ_p} matches a joinpoint ι, an appropriate proposition is instantiated and added to `currentProps`. In the case where multiple such propositions match the same joinpoint, all those propositions are added in the same way. Here it is important to make use of a well defined *advice precedence* (cf. section 2.3.1.5). We recall that if all `before` advice precede `after` advice, all matching pieces of advice are executed in the order in which they are written down in the source code. So at the end of each aspect we generate a *transition advice*, which reports the set `currentProps` to the `VerificationRuntime` and so demands a transition of the related AFA under those propositions.

Table 4.10 shows the piece of advice that are generated for the formula of table 4.9, specifying the semantics of the *Singleton* design pattern (cf. [GHJV95]): There should only be one single instance of any subclass of `Singleton`.

Note that in a general AspectJ setting it is usually not necessarily true, that each call to a constructor returns a *new* object every time, since an aspect can intercept the call and return a *cached* object instead. The formula assures that *never* a new one is returned after the first call.

Lines 1-7 of the excerpt in table 4.10 define the advice responsible for collecting the proposition $p :=$ `exit( call(Singleton+.new(..)) ) returning s`. Line 1 defines that the advice is to be executed `after` the joinpoint. This is due to the fact that we have an `exit` proposition. Also, line 1 binds the variable `s` to the return value. Line 2 is a generic pointcut that constraints this advice to *not match* any joinpoints which lie within the control flow of joinpoints within our Runtime Verification package. This assures that the instrumentation does not monitor itself. Line 3 holds the pointcut μ'_{χ_p}. Line 4 generates a new hash map with weak values (this is explained further in section 4.4.2) representing the binding function β_p. Line 5 builds up this binding by associating the variable name `"s"` with the object `s`. Lines 6-8 then add a copy of `prop0` to `currentProps`, which is *specialized* under β_p. This is the proposition $\hat{p}$, which we referred to earlier.

Lines 11-19 behave equally with respect to the second contained proposition $q :=$ `exit( call(Singleton+.new(..)) && if(s!=t)) returning t`. The only notable difference is that this proposition contains an `if` pointcut, namely `if(s!=t)`. As explained in the operational semantics, this leads on the one hand to an if-closure generated on `prop1` (cf. table 4.7) and on the other hand

```
Singleton s, Singleton t:
G(
   (
      exit( call(Singleton+.new(..)) ) returning s
   ) -> (
      X(
        !(
          F(
             exit( call(Singleton+.new(..)) && if(s!=t))
             returning t
          )
        )
      )
   )
)
```

Table 4.9: Formula for the *Singleton* Design Pattern

to a substitution of `if(s!=t)` by `if(true)` in μ'_{χ_q}, as one can see in line 13. As a result, the associated piece of advice will *execute also if* `s!=t` *does* not *hold*. However, in this case the if-closure will evaluate to **false** and hence the proposition as a whole will *not* hold in this case.

Note that there is no way to directly decide `s!=t` from within the piece of advice (lines 11-19) because one has no access to `s`. (`s` is *used* by q but *not defined* by q.)

4.3.5 Triggering a transition

Lines 21-32 of table 4.10 show the *transition advice*. As mentioned before, this advice is executed at each matching joinpoint *after* all other pieces of advice matching the same joinpoint. To assure as well as possible that the transition advice is only triggered when necessary, lines 23-26 hold a disjunct of all matching functions of all contained propositions. Hence, the advice is executed whenever at least one proposition holds at the current joinpoint. Lines 27-30 trigger the transition of the associated AFA under the given propositions. Line 31 eventually clears the set `currentProps` for later reuse.

4.3.6 Multithreading issues

Nowadays many Java applications tend to be multi-threaded and hence this was an issue we needed to address.

```
after() returning(Singleton s):
  !cflow(within(rwth.i2.ltlrv..*)) &&
  call((Singleton+).new(..)) {
    final WeakValuesMap bindings = new WeakValuesHashMap();
    bindings.put("s", s);
    currentProps.add(
            prop0.specializeBindings(bindings)
    );
}

after() returning(Singleton t):
  !cflow(within(rwth.i2.ltlrv..*)) &&
  (call((Singleton+).new(..)) && if(true)) {
    final WeakValuesMap bindings = new WeakValuesHashMap();
    bindings.put("t", t);
    currentProps.add(
            prop1.specializeBindings(bindings)
    );
}

after():
  !cflow(within(rwth.i2.ltlrv..*)) &&
  (
          call((Singleton+).new(..)) ||
          call((Singleton+).new(..)) && if(true)
  ) {
    VerificationRuntime.getInstance().updateFormula(
            "Formula1",
            currentProps
    );
    currentProps.clear();
}
```

Table 4.10: Generated pieces of advice

All functions implementing the transition relation δ are performing nondestructive updates. Also all those functions except some inside the class `Proposition` are stateless. Hence, making them thread safe was not a difficult task.

The only problem we came across when testing our concurrent lock order reversal example (see section 5.2.3.3), was about "collecting the set of propositions holding at a state": As noted above, propositions for each formula φ are collected by a unique aspect instance associated with φ. In a multi-threaded environment it may happen that multiple joinpoints on multiple threads occur at the same time. Without precaution, propositions of both threads could be merged in the set `currentProps` of this aspect before finally the transition advice is executed by one of the threads.

Theoretically there are at least two ways to solve this problem. The first is to lock the aspect whenever the first advice executes and unlock it after a transition is taken. This would be safe, however could very much slow down the system. Also is would forbid concurrent calculation of δ for multiple threads.

The other option is to make the field `currentProps` a `ThreadLocal`. This means that any thread in the virtual machine gets its own copy of the field. Hence the sets cannot be accidently be mixed. *J-LO* implements this behaviour.

This concludes our summary of the code generation part of *J-LO*. The next section gives some details about special cases of exceptional runtime behaviour such garbage collection, shutdown and exceptions caused by invalid input.

4.4 Dealing with exceptional runtime behaviour

4.4.1 Notification of shutdown

One crucial point of the *DLTL* semantics is that they are defined over paths of finite length. As a consequence, *J-LO* needs to be notified somehow about the end of the execution path in order to report about the final configuration of each AFA.

This is accomplished by adding the additional aspect `ShutdownHook` as shown in table 4.11.

Line 3 declares that this aspect should have precedence over all others. With other words no other aspect can intercept the execution of `ShutdownHook`.

The empty advice at lines 5-8 causes this aspect to be instantiated the first time when a class is instantiated which resides not within the runtime verification package.

When this happens, this causes the constructor defined by the lines 8-16 to execute. The constructor then installs a *Shutdown Hook*[4], a nonactive thread,

[4]see `http://java.sun.com/j2se/1.5.0/docs/guide/lang/hook-design.html`

```
1  public aspect ShutdownHook {
2
3     declare precedence: ShutdownHook,*;
4
5     before():
6         staticinitialization(*) &&
7         !within(rwth.i2.ltlrv ..*) {
8     }
9
10    public ShutdownHook(){
11        Runtime.getRuntime().addShutdownHook(
12            new Thread() {
13                public void run() {
14                    VerificationRuntime
15                        .getInstance().tearDown();
16                }
17            }
18        );
19    }
20
21 }
```

Table 4.11: Shutdown hook aspect in *J-LO*

with the virtual machine. When shutting down, all installed shutdown hooks are executed concurrently. The shutdown hook invokes `tearDown()` on the verification runtime (line 15). This causes that the current configuration of all attached AFAs is reported to all registered observers (see section 4.4.3). A configuration can then be queried if it is final. Any AFA which is in a nonfinal configuration at this state directly relates to a formula which was violated by that path.

Note that shutdown hooks may not be executed in the case where the virtual machine simply dies (e.g. when invoking `kill -9` under Linux).

4.4.2 Behaviour under presence of garbage collection

As already mentioned in section 2.5, in *J-LO* objects should not be prevented from being garbage collected because this could lead into scalability problems. On the other hand we needed to ensure sound semantics for the case of GC. Although we could not make any assumptions about the actual implementation of GC it turned out that there was no need for this because the following invariant always holds:

A proposition referencing an object which is unreachable from the rest of the program can never match again.

The reason for this invariant is that the propositions on an execution path can only expose objects which are *available* on the control flow of the current joinpoint (cf. definition 3.3.9), O_ι^{cflow}. If an object o is unreachable, it can never occur in this set again. Hence, no state on the subsequent path can define a variable with value o any more. As a result no proposition using a variable which is bound to o can match a state on this rest of the path.

For this reason, the implementation of *J-LO* uses a hash map with *weak values* as representation of the binding β_p for any proposition p. Whenever such a weak values map is accessed, all entries pointing to objects which are not accessible any more are pruned from the map. (This can easily be implemented by using a `ReferenceQueue`.)

Additionally, *J-LO* stores for each proposition p the initial size b of β_p. Whenever a proposition is tried to match against a state, we first check if $|\beta_p| < b$. If this is the case, p changes its internal state in such a way that it will never match again. Then it is semantically equivalent to **ff**.

It should be noted that during calculation of the successor of a configuration using δ, those references are *temporarily* made strong in order to avoid cases where objects are available during the evaluation of one branch of a formulae but not on another.

4.4.3 Observing configuration changes

After having explained how we assure that the *J-LO* implementation complies with the general assumptions of the operational semantics, we now explain how changes of the configuration of an AFA can be intercepted.

The key component here is the interface `VerificationRuntime.Listener` as shown in table 4.12.

Lines 3-7 define the method `notifyRegistered`, which notifies the observer that a new formula was registered with the verification runtime. It propagates a unique formula ID, the thread which registered the formula and a `Configuration` object which represents the initial configuration of the formula. This configuration can be rendered into String format and can also be queried if it is (non)final or (non)accepting.

Lines 9-13 define the method `notifyUpdate` which is called whenever a transition was taken for the given formula. The parameters are equal to the ones of `notifyRegistered`. The `configuration` is here the new configuration resulting from the transition.

The method `notifyTearDown` is defined by the lines 15-18. It is called when the virtual machine shuts down and hence the end of the execution path is reached.

```
public interface Listener {

    public void notifyRegistered(
            String formulaId,
            Thread associatedThread,
            Configuration initialConfig
    );

    public void notifyUpdate(
            String formulaId,
            Thread associatedThread,
            Configuration newConfig
    );

    public void notifyTearDown(
            String formulaId,
            Configuration config
    );

    public void notifyOnUserCauseException(
            String formulaId,
            Thread associatedThread,
            String ifExpression,
            Throwable exception,
            Configuration config
    );

}
```

Table 4.12: Listener interface in *J-LO*

When this happens, this method is called for any currently installed formula with the formula ID and the final configuration as a parameter. By inspecting if this configuration is final it can easily be determined whether the associated formula is satisfied on the observed path.

Important note for users: Note that `notifyTearDown` is executed within the control flow of a shutdown hook (cf. section 4.4.1). No other shutdown hooks may be installed from within such a context. While usually one would never even try to do so we found that apparently some methods of the Sun Abstract Windowing Toolkit (AWT) implicitly do so, in particular the methods of `java.awt.Toolkit`.

4.4.3.1 User caused exceptions

Lines 20-26 define the method `notifyOnUserCausedException`. This method is called whenever the evaluation of an if-closure (cf. section 4.3.1.1) leads to an exception which is caused by an invalid expression. A typical case would be the `if` pointcut `if(1/0<2)` which would raise an exception due to the division by zero. When this happens, the formula is *removed* from further verification. In particular no calls to `notifyUpdate` and `notifyTearDown` will be performed any more for this formula.

Also `notifyOnUserCausedException` is called with the following arguments: The ID of the formula, the thread which triggered the evaluation (This might be important to debugging because the expression could access the thread.), the expression that caused the exception in String format, the exception that was thrown and the last configuration before the transition was tried to be taken. Given that the configuration holds all current bindings it should be straightforward to determine the cause of the exception.

4.4.3.2 Custom observers

J-LO provides a default implementation of this interface, which just dumps all the available information to the console. Of course more intelligent observers are possible. For instance an observer could have a special treatment for certain formulae to implement fast fail semantics ("abort application immediately") or issue notifications over some communication channel etc. This however is out of the scope of this work and will be addressed in the future.

This shall conclude our summary of the implementation details of *J-LO*. Further information about how *J-LO* is used along with further examples may be found on `http://www-i2.informatik.rwth-aachen.de/JLO/`.

The next chapter will elaborate on the correctness and performance of the *J-LO* implementation.

Chapter 5

Metrics and performance

In this section we want to report on how we tested *J-LO* with respect to correctness and performance.

5.1 Correctness of the implementation

Though we did not formally prove the implementation of *J-LO* correct, we are reasonably confident that it correctly implements the operational semantics given in section 3.5 for the following reasons:

First of all most of the functions defined by the operational semantics are reasonably small and straightforward to implement. We have used the Eclipse metrics plug-in[1] to derive the following data:

- An average method of the *J-LO* runtime library has about 7 lines of code, the whole library has about 2000 method lines of code (MLOC[2]).
- The only methods that were reported as "out of bounds" because they were unreasonably long or had unreasonably many possible paths were *generated* methods implementing `equals` and `hashCode` for some objects. Those methods are not thought to be read or altered by human beings anyway.

Hence we believe that the implementation is easy to follow and should hence be easy to prove equivalent to the operational semantics if needed.

Additionally we employed the tool FindBugs[3] in order to find potential sources of bugs in the implementation of the runtime library. Two minor issues were found and immediately resolved.

[1]available at `http://metrics.sourceforge.net/`

[2]non-blank and non-comment lines within method bodies

[3]available at `http://findbugs.sourceforge.net/`

With respect to the code generation part of *J-LO* we would have to prove *abc* correct as well as our implementation of additional compiler passes, which would certainly be a hard task. We did not conduct such a proof and rely on the brought user base of *abc* and its principal components Soot and Polyglot. Indeed *abc* has proven to be very stable in the past and bugs do not seem to be reported very frequently.

5.2 Performance

This section is split into two parts. First we want to give some arguments for the general theoretical performance of the employed algorithms. Then in the second subsection we elaborate on the performance of the specific implementation of *J-LO*.

5.2.1 Theoretical performance

As noted above, the evaluation of each formula can be performed separately. Thus the overall cost of Runtime Verification is *linear in the number of formulae.*

Each formula first needs to be brought into negation normal form and then installed with the runtime engine. This can be seen as constant cost over the running time of the application. The cost of the evaluation of installed formulae then heavily depends on the following factors:

1. The size of the formula.
2. The kind of pointcuts defined by the propositions of the formula.
3. The number of different bindings available at a joinpoint.

The first point is general to all algorithms employing LTL: For a given formula φ it is known (e.g. [VW86]) that the calculation of a successor of φ has an exponential worst case complexity in $|\varphi| := |cl(\varphi)|$.

Since in the case of *J-LO* we generate successor states on-the-fly, we *know* the set of propositions to calculate the (unique) successor for, so here the cost is constant with respect to the number of propositions.

To some amount δ may be statically precalculated, which theoretically should yield a constant cost for taking transitions at runtime. However, the use of dynamic bindings leads into problems here. We comment on this further in the section about related work, specifically section 6.3.2.3.

The second point is specific to AspectJ: *J-LO* allows arbitrary AspectJ pointcuts to occur within a proposition. Recalling our definition of a filtered path (cf. section 3.3.1), this of course means that the more joinpoints are matched by the pointcuts contained in a formula, the more frequently the AFA for this formula is updated.

The worst case scenario is here an unguarded `if` pointcut: A pointcut as `if(A.field==true)` leads, as described in section 4.3.1.1, to a piece of advice with associated pointcut `if(true)`. This means that this piece of advice and hence the evaluation of δ is triggered *at every possible joinpoint*, which is certainly very expensive. Hence we advise users to guard such pointcuts e.g. by writing instead: `set(A.field) && args(bool) && if(bool==true)`. This would only capture events where the field is *set*, which should be sufficient in most cases and would capture much less joinpoints, hence being more efficient.

The third point depends on the way how proposition specifications overlap within the definition of a formula. Usually the number of possible valuations at a joinpoint is 1. However in cases where multiple pointcuts holding an overlapping set s of variables match an overlapping set of joinpoints, this leads to multiple valuations for all variables in s. According to the definition of δ, we have to evaluate formulae for all such valuations, which makes the calculation of δ more expensive.

5.2.2 Performance of the implementation

We did some performance measurements using the commercial tool JProbe [JPr]. JProbe allows to take detailed profiles down to the level of single Java statements.

Interestingly, the analysis showed that the largest part of the overhead was caused by the fact that we use a set based implementation and not actually by the intrinsic complexity of the algorithms. Specifically, the implementation uses hash sets over hash sets of formulae. Whenever a formula is added to such a set, its hash code needs to be calculated. Over 70% of the time were spent in this calculation of hash codes. Calculation is done by recursively calculating hash codes for all propositions over the whole term structure of the formula. For the calculation of the hash codes of propositions it is crucial that bindings are taken into account, because a proposition $p(x)$ has different semantics than $p(1)$ where x has been bound to 1. This again makes it necessary to calculate a hash code for a weak values hash map (see section 4.4.2) and this is where performance is lost: Since weak values maps hold a volatile set of mappings, in order to calculate the hash code one needs to generate a *current snapshot* of the contents of the map and generate a hash code from this snapshot. This is rather expensive, especially when done so frequently.

We implemented some different caching techniques to counterbalance this behaviour with notable success. Yet it seems to be a good idea to employ a different implementation technique in future versions.

In addition to profiling, we tested the running time of some small example applications in contrast to their counterpart which had been instrumented with *J-LO*.

Here it showed up that it is not only important, what formulae are specified in the system but also how expensive the execution of the *uninstrumented code* (we call this the "*shadow*" of the formula) is.

Specifically we attached a formula to an ArrayList based stack, stating that after each `push` operation `top` returns the pushed element until another `push` or `pop` is invoked. Then we pushed 1000 times the same object in the stack. Naturally, the calculation of δ for this formula was rather constant. Though, this constant overhead proved quite expensive compared to the usual `push` operation. In fact it slowed down the operation by about a factor 1000.

When reasoning about operations on a higher level however, those operations tend to be more expensive themselves so that the constant cost of the evaluation of a specific formula is rather small compared to the execution of the shadow.

Altogether one can say that if joinpoints of interest occur reasonably seldom and show a reasonably small shadow then the instrumentation overhead showed negligible.

Static precalculation (see section 6.3.2.3 and our publication [Bod05b]) should help to mitigate this overhead to the feasible minimum.

5.2.3 Benchmarks

We performed the following further benchmarks to see how well *J-LO* behaves when applied to different problem classes.

- *Iterator* - This is an example specification taken from the Java API, which checks for the correct use of iterators. This benchmark puts particular stress on the application because the access to an iterator is usually quite fast compared to what has to be done on the verification side, so the overhead of the instrumentation could show a significant slowdown.
- *HashSets* - This benchmarks is similar to the one before but it implements a specification, which *cannot* be implemented using normal object oriented techniques in a convenient way.
- *LockOrderReversal* - This benchmarks looks for locks (semaphores) being taken in a wrong order and is hence important to concurrency. This example is particularly interesting because it uses virtually all features of *J-LO*, in particular `if` pointcuts and concurrency and uses a relatively *large formula*. Also it shows a property which approximates erroneous behaviour and hence should issue a warning rather than an error.

In the following we report on all those benchmarks in detail.

```
Collection c, Iterator i:
G(
  (
    exit(call(* Collection+.iterator()) && target(c))
      returning i
  ) -> (
    X(
      G(
        (
          entry(
            (   call(* Collection+.add*(..))
              || call(* Collection+.remove*(..))
              || call(* Collection+.clear())
            ) && target(c)
          )
        ) -> (
          G(
            !(
              entry(call(* Iterator.next()) && target(i))
            )
          )
        )
      )
    )
  )
)
```

Figure 5.1: Safe iterator formula

5.2.3.1 Safe iterators

The `safe iterator`-pattern mentioned in the introduction 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.

It is enforced in the Java 5 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.

In our formalism the requirement can be specified *in a modular way* through the formula in Figure 5.1.

Line 1 declares the free variables `c` and `i` that *each* collection and iterator in question will be bound to. The actual formula is stated in lines 2-25, 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 remainder of the execution that after a call to `add`, `remove` or `clear` no call to `i.next()` must occur.

We have successfully validated this formula in practice: [AAS+04] 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.

With respect to runtime performance, the formula showed the expected behaviour: When used to instrument collections/iterators in the standard JDK, the overhead was significant, i.e. we could observer a slowdown of several orders of magnitude. That is because the specialized checks in the JDK involve only one single integer comparison while in the case of *J-LO* we have to evaluate a whole formula. The instrumentation in JHotDraw performed much better, since the iterators were used less frequently so that in this case, the additional overhead was relatively low. Still, we could observe *some* overhead, which suggests that there is room for optimization.

5.2.3.2 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. We captured this behaviour 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 *J-LO*, specifying this property is easily done by a straight forward translation into linear temporal logic (see Figure 5.2). In contrast to the safe iterators, this is an application-specific formula that requires understanding and analysis of the application. Consequently, false use of hash sets which would be detected by this formula would go unnoticed with the standard Java development toolkit.

A comprehensive survey of existing verification patterns and how to express them in various specification formalism including LTL can be found in [DAC99].

```
Collection c, HashSet s:
G(
  (
    exit(call(* HashSet+.add(..))
      && target(s) && args(c))
  ) -> (
    X(
      G(
        (
          entry(
            ( call(* Collection+.add*(..))
              || call(* Collection+.remove*(..))
              || call(* Collection+.clear())
            ) && target(c)
          )
        ) -> (
          (
            entry(call(* HashSet+.remove(..))
              && target(s) && args(c))
          ) R (
            !(
              entry(call(* HashSet+.contains(..))
                && target(s) && args(c))
            )
          )
        )
      )
    )
  )
)
```

Figure 5.2: HashSet formula

It can serve as a starting-point into specifying properties. The `HashSet` requirement can for example 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 `modify` and R the `remove`-action.

5.2.3.3 Lock order reveral

To avoid the problem of lock-order reversal (cf. [Hav00], [SH04]), 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 these locks are also being used 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 interested in mere warnings, as a violation of the formula might not 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):

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

Notice that the formula has four parameters: two locks and two threads. 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).

```
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) )))
```

Table 5.3: Lock order reveral formula

Translating the above formula into a new variant of the LTL-syntax allowing predefined named pointcuts results in the expression shown in table 5.3. (We distinguish between logical operators in LTL and AspectJ for clarity.)

We have successfully used this formula to identify the pattern of lock order reversal in the Java 5 development kit, which introduced a new concurrency API.

Step-by-step instructions along with all related code are available on our website: `http://www-i2.informatik.rwth-aachen.de/JLO`

Note that part of this section was published in [SB05, BS06] and hence was co-authored by Volker Stolz.

Chapter 6

Related Work

In this section we present related and previous work both, in the field of Runtime Verification and the field of aspect-oriented programming.

6.1 Design by contract

Runtime Verification can be seen as an extension of the well-known *Design By Contract* (DBC) principle, which became popular through the work of Bertrand Meyer [Mey92a] and his reference implementation in the programming language Eiffel [Mey92b].

6.1.1 DBC in Eiffel

In DBC, the programmer is able to annotate a method with *preconditions*, *postconditions* and *invariants*, which are checked during runtime before, after respectively during the execution of the annotated method.

Table 6.1 (adopted from the Eiffel manual pages[1]) gives an example of such conditions. The annotated method shall put the element `x` into a map, so that it is retrievable by `key`.

Lines 2 to 5 state required preconditions: The capacity should not be exceeded and the key may not be empty. Lines 7 to 11 state postconditions which shall hold after the method body has been executed: `x` shall be contained in the map; `item` should return `x` for this `key` and the value `count` should have increased by 1.

There are other languages with native support for DBC, namely D[2], Lisaac[3], and the ADA [Led83] based SPARK[4], which aims at high-integrity software development.

[1]`http://archive.eiffel.com/doc/manuals/technology/contract/page.html`

[2]`http://www.digitalmars.com/d/index.html`

[3]`http://isaacos.loria.fr/`

[4]`http://www.praxis-his.com/sparkada/`

```
put (x: ELEMENT; key: STRING) is
    require
        count < capacity
        not key.empty
    do
         ... Some insertion algorithm ...
    ensure
        has (x)
        item (key) = x
        count = old count + 1
    end
```

Table 6.1: Example for pre and postconditions in Eiffel

Several DBC implementation for Java exist. We are aware of JML[BCC+05], Contract4J, Jose, Barter, iContract, JMSAssert, JContract, Jass, conaj and OCL4Java (see table 6.2). They all work with source code based specification of pre and postconditions. The one that comes closest to the technology of *J-LO* is Contract4J [con] version 1. (Contract4J version 2 which was released in October 2005 is an entirely different tool with its own architecture.)

Tool	URL
Barter	`http://barter.sourceforge.net/`
Cona	`http://www.ccs.neu.edu/home/lorenz/papers/`
Contract4J	`http://www.contract4j.org/`
iContract	`http://www.javaworld.com/javaworld /jw-02-2001/jw-0216-cooltools.html`
Jass	`http://csd.informatik.uni-oldenburg.de/~jass/`
JContract	`http://www.parasoft.com/jsp /products/home.jsp?product=Jcontract`
JML	`http://www.cs.iastate.edu/~leavens/JML/`
JMSAssert	`http://www.mmsindia.com/JMSAssert.html`
Jose	`http://www.faculty.idc.ac.il/yishai/jose/`
OCL4Java	`http://www.ocl4java.org/`

Table 6.2: DBC tools for Java

6.1.2 Contract4J

Similar to *J-LO*, Contract4J is also based on Java 5 annotations and the generation of AspectJ code. However, Contract4J uses the Java Annotation Processing

Tool (APT)[5], which comes with the Java Development Kit. This allows annotation extraction from source code only. Contract4J uses APT to produce an XML structure holding all annotation information. Then it uses XSLT [Cla99] transformations to produce aspects implementing runtime checks for the given conditions. In that way, the internal workflow is quite similar to the one of *J-LO* except *J-LO* does not employ XSLT transformations but rather a real compiler for code transformations, providing static type checking and more.

We believe that apart from the automaton-based backend, that *J-LO* provides, Contract4J and *J-LO* have much in common and so we met with the developer of Contract4J, Dean Wampler at the AOSD '05 conference and talked about a possible bundling of efforts.

Equal to all other aforementioned tools, Contract4J allows simple DBC, while the logic provided by *J-LO* is much richer. Although those pre and postconditions are very valuable and already much more expressive and convenient to use than the aforementioned assertions, they still do not provide any temporal notion: Pre and postconditions as well as invariants only reason about each single method invocation. There is no way of specifying temporal interdependencies as they can be expressed with LTL. On the other hand, any pre and postcondition and invariant can easily be expressed in our formalism. Hence, Runtime Verification provides a superset of expressiveness compared to DBC.

In July 2005 an expert group was founded with the goal to bring DBC into the standard Java Development Kit under a Java Specification Request. The group consisted of team members from development groups of most of the aforementioned tools. The author of this work was representing *J-LO*. The group discussed the different tradeoffs that were taken in the various interpretations of DBC.

For instance the specification language in question can be implemented by a language extension to Java. This however requires new compilers. Another standard technique would be to use annotations as *J-LO* does. A drawback here was that some Strings need to be escaped. Especially line feeds need to be treated by concatenation of single line Strings, which looks not very nice. Another option would be to use the specification language within comments, which does not require such escaping. This however leads to semantical difficulties because usually comments should have no effect on the program semantics whatsoever.

Unfortunately after discussing those and other issues, in August the group eventually fall apart when Sun decided to not support the development of DBC any further. Hence it is very questionable that support for any Runtime Verification beyond the `assert` statement which is contained today, will make it into native Java. Microsoft on the other hand seems to have recognized the usefulness of

[5]`http://java.sun.com/j2se/1.5.0/docs/guide/apt/`

such approaches by providing Spec# [BLS04], a DBC language extension for the C# programming language.

6.2 Runtime Verification

With respect to Runtime Verification, few tools have been released so far.

6.2.1 Java PathExplorer

The probably best known project is the Java PathExplorer (JPaX) [HR04] developed by Havelund and Roşu at NASA AMES, not to be mistaken for the Java PathFinder (JPF) [WKGS00] which is a model checker for Java. While JPF was made open source in 2005, JPaX has unfortunately been under closed source development by now.

Despite this fact, there have been many publications about the design and successful application of the tool [HR01a, HR01b, HR04] which give reasonable input for comparison with *J-LO*.

The JPaX system consists of two distinct components, which are loosely coupled, the first one being an *instrumentation engine*, which adds hooks to the application to be monitored. Such hooks consist of the specification of a place where the hook should be applied to (e.g. the name of a field) and a labeled Boolean proposition which can access values that lie within the scope of the instrumentation point. Those propositions can then be used in a specification, which can for instance be pinned down using LTL. Such specifications can then be verified at runtime using the second JPaX component: The hooks in the instrumented application issue messages to a *backend*, stating which propositions hold at the current time. The backend then evaluates the specification over this explicit trace.

Table 6.3 gives an example specification taken from [HR04]. Line 2 states that the field `C.x` is to be monitored. Line 3 then defines a proposition `A` that holds iff `C.x` is greater than 0. Line 5 specifies the formula `F1`, which demands that A holds on the entire path. (In JPaX, **G** is written as `[]` and **F** as `<>`, just as in Spin, cf. section 2.1.1.6.)

6.2.1.1 Trace model

As the example suggests, propositions in JPaX are *atomic*, meaning that at each point in time a propositions either holds or not, irrespective of any other condition. In particular, propositions *cannot contain any free variables.* This makes it easy to separate the instrumented application from the evaluating backend: All the instrumentation has to do is issue a message "*proposition* p_i

```
// instrumentation:
monitor C.x;
proposition A is C.x > 0;
// verification:
formula F1 is []A
```

Table 6.3: Example specification in JPaX

changed" whenever the truth value of a proposition p_i changes from **true** to **false** or vice versa. Such messages can easily be sent over arbitrary streams, hence allowing to execute the evaluation backend in another process or even on another machine. In the case of *J-LO*, this is not possible in the same way, since propositions have to expose objects to the backend. In particular, if-closures need to be able to access bound objects in order to have their truth value determined. This could however be solved by evaluating if-closures within the application and sending the calculated truth values to the backend along with the current set of propositions. *J-LO* is currently designed to be executed in the same process as the instrumented application. Future versions might however address this issue as explained above.

6.2.1.2 Evaluation of specifications

In JPaX, specifications over propositions can be written in different formalism as future time LTL (the kind of LTL *J-LO* uses as well), past time LTL [HR04] and others. The mechanism is generally extensible, because JPaX uses the rewriting engine Maude [CDE$^+$99, CDE$^+$03] for evaluation. In Maude recursive definitions as the one of LTL operators can easily be specified by just a few lines. According to the authors, Maude is very efficient and is usable for formulae of a larger size.

6.2.1.3 Expressiveness

As mentioned above, tracking objects by binding them to free variables is not possible in JPaX, which is certainly the biggest advantage of the *DLTL* used in *J-LO*. Another advantage which must be taken into account is that in *DLTL* propositions are more expressive than the usual Boolean expressions which JPaX provides. Everything that can be expressed in JPaX can be expressed in *DLTL* as well, using `if` pointcuts. On the other hand, *DLTL* can use various other pointcuts in order to match on method calls, exceptions being thrown and so forth (see appendix C for a full list). Hence we conclude that *J-LO* provides a formalism which is more powerful than the one provided by JPaX but is less extensible due to higher requirements on the implementation side.

This concludes our overview of JPaX. Another verification tool, very recently developed also by Havelund is *HAWK*, an object-oriented extension to the *EAGLE* logic and its supporting tools.

6.2.2 HAWK and EAGLE

HAWK [dH05] is a programming-oriented extension of the rule-based logic EAGLE [HBS03a, HBS03b, HBS04] that is allows for specifications in various temporal logics of different kinds. EAGLE computes the truth values of temporal formulae by calculating a minimal respectively maximal fixpoint to the recursive definitions of temporal operators such as given by $\mathbf{F}\varphi \equiv \varphi \vee \mathbf{X}\ \mathbf{F}\varphi$. As such EAGLE is very generic and can be used for virtually any kind of specification logic and programming language of the base program. From a specification EAGLE generates an observer that implements its semantics and is notifies whenever events of interest occur.

HAWK is a logic and tool for runtime verification of *Java* and is built on top of EAGLE. Specifications written in HAWK are ultimately being translated into EAGLE monitors. Similar to *J-LO*, HAWK allows users to refer to *events* on the execution trace of a running Java application and to expose objects at those objects in order to bind them to free variables. The way quantification takes place also coincides with our strategy in *J-LO*: Quantification over variables is reduced to quantification over events.

While *J-LO* provides all AspectJ pointcuts in order to match such events, HAWK can only match on method execution and return. Also, the HAWK logic is less flexible: It only allows a subset of LTL to be used, namely the following to forms:

```
<event>proposition  [event]proposition
```

Here `event` corresponds to a method call or return that may bind free variables in `proposition`. Unlike in *J-LO*, variables to be exposed have to be annotated with question marks, such as

```
<o?.someMethod(p?)returns r?> P
```

which binds o, p and r in the scope of P. In *J-LO* the static analysis (cf. section 3.4) determines automatically where objects are to be bound and where they are to be matched against.

The semantics of the construct `<e>p` corresponds roughly to the LTL formula $\mathbf{F}(e \wedge p)$: It has conjunctive semantics, while `[e]p` corresponds to $\mathbf{G}(e \rightarrow p)$. That way it provides the two patterns in LTL that we also found are used most frequently, however it does not support the full LTL formalism.

Another big difference is that events are assumed to be non-overlapping, i.e. that at each state at most one proposition holds. As we show in [BS06], this leads to a less expressive formalism: Certain temporal patterns can only be expressed in a cumbersome way, other cannot be expressed at all.

Also, while in *J-LO* the end of a path is implicitly signalled by a shutdown hook (cf. section 4.4.1), in HAWK the user has to specify the shutdown event manually.

Table 6.4 shows a HAWK example specification stating that for each buffer b, whenever o is put into the buffer this implies that when eventually *get* is invoked on this buffer the returned object k is identical to o. The same property could have been specified in *J-LO* in a similar manner.

```
observer BufferObserver {

        classPath = C:/src
        targetPath = C:/src
        terminationMethod = bufferexample.Barrier.end()

        var Buffer b; var Object o;     var Object k;

        mon B =
          Always ( [b?.put(o?)]
            Eventually ( <b.get() returns k?> (o == k) ).

}
```

Table 6.4: HAWK example specification

It should be noted that the implementors of HAWK do not give any informations about the performance of their approach. In particular, the generated EAGLE code is dynamically parsed at startup of the application which rather seems like something that should be done at compile time.

With respect to aspect-orientation d'Amorim and Havelund write [dH05]:

> To some extend AOP can be seen as just a clean solution to the instrumentation of programs. For this purpose, we used extensively the AspectJ AOP tool. We believe, however, that a natural extension of this work is the introduction of temporal advices, which could be integrated in an AOP tool. In contrast to the usual advices, temporal advices can provide a means to define hooks for code to be executed upon validation or violation of a finite-trace requirement.

In that way, *J-LO* can be seen as the implementation of this long term goal. Especially the syntax we proposed in [BS06] seem to reflect exactly what the authors had in mind.

6.2.3 dtrace

dtrace is a tracing module for Sun's operating system *Solaris*. It is particularly designed for the purpose of allowing *instrumentation of running systems in a production environment*. Hence, careful design was necessary for the hooks that are available in the tracing language. Support by the operating system is necessary in order to allow the installation of hooks into a running application. Using specialized techniques, developers are able to instrument running, uninterpreted applications in such a way that there is a *zero runtime overhead* when the instrumentation is disabled again. The purpose of dtrace is solely tracing, which is quite different from the purpose of Runtime Verification. Yet, dtrace supports a powerful trace filtering language which might be used to achieve similar goals.

The major drawback of dtrace is that it is not portable at all to other platforms. Also concluding from the examples the authors give, it might be cumbersome to express properties using variable bindings.

6.2.4 Temporal Rover

The company Time Rover sells the commercial product *Temporal Rover* (TR), which is naturally under closed source development. However, the documentation[6] and technical papers available online reveal that the basic model of TR are what the authors call *state chart assertions*. Such assertions are temporal assertions similar to the ones that can be expressed with *J-LO*, however are written down in a graphical way using a state chart-like notation.

The documentation reveals further that among the languages that can be used to specify temporal assertions there are also textual temporal logics as future and past time LTL, and *MTL*, the *Metric Temporal Logic*, which provides LTL-like operators that can be annotated with real-time contraints such as $\mathbf{G}_{<10}(p \rightarrow \mathbf{X}q)$ which reads as *always within the next 10 cycles, whenever p holds, q has to hold in the next cycle.*

The formalism is very similar to *Timed Propositional Temporal Logic* (TPTL) which was introduced by Alur et al. [AH89], and has recently been shown [BCM05] to be slightly more expressive than MTL.

Generally we believe that *J-LO* could easily be extended to accommodate a similar extension for real time constraints given that all one would have to do is

[6] `http://www.time-rover.com/tl.html`

record time stamps and match against them. In a certain sense, *DLTL* already allows the definition of such constraints by the use of `if` pointcuts accessing an explicit clock object. Of course one must take into account that runtime instrumentation and evaluation might distort the runtime behaviour with respect to timing guarantees. This of course is a general problem of Runtime Verification and not specific to *J-LO*.

Temporal Rover also provides a syntactic extension of LTL, which allows to express counting properties such as that an event happens exactly n times. While in pure LTL such conditions are generally impossible to express for a nonconstant n (e.g.[Tho81]), its syntax and evaluation can easily be extended to bypass this problem, making LTL recognize exactly the regular languages over finite words.

In Temporal Rover, the temporal assertions are embedded in comments and their semantics are implemented by a source-to-source transformation.

6.2.5 Java MaC

Java MaC [KVK+04] is a runtime-assurance tool for Java. The *Meta Event Definition Language* (MEDL) is used to specify safety properties. As the MaC architecture was designed to 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 or in other words: Free variables cannot be bound to objects.

6.2.6 Valgrind

Valgrind [NS03] 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 [SBN+97].

6.3 Trace Languages

Especially in the aspect-oriented software development community, there have been quite some advances towards trace languages within the past years.

6.3.1 Overview

In the following we first want to give an overview of the different approaches that have been published so far.

6.3.1.1 Stateful Aspects

Starting in 2001, Douence, Fradet, Motelet and Südholt published severel articles [DMS01, DFS02, DFS04b, DFS04a] describing *stateful*, history based aspects. They introduce an aspects calculus where advice can be triggered by a sequence of joinpoints. While this calculus has proven very effective in the area of formal methods, it is less oriented towards an implementation than the approach we took with *J-LO* and *DLTL*. At least one implementation which was inspired by their work is implemented in JaSCo [SV03, VS04], an aspect-oriented framework for component based software development.

6.3.1.2 Tracecuts

In their *URD tool* Walker et al. [WV04] implemented so-called *tracecuts*, certain pointcuts that let the user specify context-free expressions over the execution flow, which have been matched in order to make such a tracecut apply to a certain joinpoint (Table 6.5).

```
tracecut isSafe() ::= a() completed()∗ $;
tracecut completed() ::= a() [completed()] b()
 | c() [completed()] d();
tracecut a() ::= entry(safePc());
tracecut b() ::= exit(safePc());
tracecut c() ::= entry(unsafePc());
tracecut d() ::= exit(unsafePc());
pointcut safePc(): execution(∗ ∗.safe());
pointcut unsafePc(): execution(∗ ∗.unsafe(..));
```

Table 6.5: Tracecuts in the URD tool

The tracecut `isSafe()` is here defined by the means of context-free grammars over pointcuts. It matches nested calls to `safePc()`, not followed by `unsafePc()`.

Since tracecuts allow specification of context-free expressions they allow the detection of recursive events. As a result, the implementation however needs to employ push down automata to recognize those languages, which theoretically can lead to an unbounded stack size in general. Walker's work focuses of the elimination of this overhead in his work.

In tracecuts one is however even able to attach arbitrary *action rules* to each match. A keyword `fail` can be used to conditionally ignore the match. As a consequence, the language of recognizable traces is even more than context-free, presumably even Turing complete. As a consequence, tracecuts are not suitable as a formal specification language, because efficient reasoning about the specification language itself would be impossible.

6.3.1.3 Alpha

In [OM04, BMO05], Bockisch, Mezini and Ostermann introduce a very general extension mechanism for pointcuts, by defining pointcuts as predicates in the logic programming language *Prolog*. They provide an implementation called *Alpha*, which exposes runtime information about the control flow and heap through prolog facts which can hence be matched on using user defined predicates. This mechanism is obviously extremely powerful and certainly well suited for rapid prototyping and initial experiments. However, the authors agree that an implementation using pure Prolog is too inefficient at the current time to be used in production systems. Future work will show how far one can go using such open systems.

6.3.2 Tracematches

The model and implementation which come closest to the one used in *J-LO* is certainly the one of *tracematches* [AAS$^+$04] introduced by the *abc* [ACH$^+$05] development group. Hence we want to compare both tools in detail.

6.3.2.1 Introduction

A tracematch tm can be interpreted as a tuple $tm = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{B})$, where $\mathcal{V}$ is some finite set of typed *variables*, Σ is a finite *alphabet* of symbols $\{a_1, ..., a_n\}$, $\mathcal{P}$ is a *regular expression* over Σ and $\mathcal{B}$ is a *body*, containing pure Java code with access to variable bindings defined over $\mathcal{V}$.

As in *J-LO*, symbols in tracematches can bind free variables from $\mathcal{V}$. Hence, for a symbol a_i binding v, we write $a_i(v)$ in the following.

The regular expressions that can be used in tracematches are the ones known from the usual mathematic calculi, containing concatenation, logical disjunction and the Kleene star. (The implementation also provides the notation $a+$ for *at least one occurrence of* a and $a[n]$ for $a \ldots a$ (n times), however this does not alter the expressiveness.)

An example the authors give is the following (we use an excerpt from the original text in [AAS$^+$04]):

The application is to log the actions of the users of a database whenever a user has logged in, we want to report the queries of that user. For simplicity, we consider a system where only one user is logged in at any time.

Variables that are to be bound in the pattern of a tracematch are declared in its header (line 1). Here there are two such variables, namely the user u and a query q. The first symbol we declare is the one that binds u, via a call to the login(..) method (lines 2-4). We also track logout actions, so that we stop logging when the user has finished (lines 5-6). Finally, we declare a symbol for query events (lines 7-9), and intercept the value of the query in variable q. The pattern is then very simple: we just look for queries that follow a login event (line 11). Whenever this matches a suffix of the current trace, we print an appropriate logging message that reports both the user u and the query q (line 13).

```
 1 tracematch (User u, Query q) {
 2   sym login after returning:
 3       call  (∗ LoginManager.login(User,..))
 4    && args(u,..);
 5   sym logout after:
 6       call(∗ LoginManager.logout());
 7   sym query before:
 8       call(∗ Database.query(Query))
 9    && args(q);
10
11   login  query+
12   {
13     System.out.println(u + " made query " + q);
14   }
15 }
```

Here the tracematch can be identified as $tm = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{B})$ with $\mathcal{V} = \{u, q\}$, $\Sigma = \{login(u), logout, query(q)\}$, $\mathcal{P} = [login(u)\ query(q)\ query(q)^*]$ and the shown body. Note in particular that the symbol *logout* is defined but does not occur in the pattern. This means that any trace where this symbol is seen between a login and a query will *not* be matched.

The semantics of tracematches are quite similar to the ones defined for *DLTL* given that the above tracematch matches on *all possible valuations* for the given variables. In particular given the example tracematch, logging would happen after *each* query which follows a login (given that no logout has happened before).

6.3.2.2 Comparison of the implementation

As a consequence, the implementation is quite similar. The aspects structure with one advice for each symbol and one *transition advice* in the end (they call it *some* advice) is even equal. However in the tracematch implementation transitions are calculated right in the aspect while in the case of *J-LO* this is left to the runtime environment and the aspect only *triggers* a transition. This is neither good nor bad and just a design decision.

Another similarity is that they use no explicit representation of a state. A tracematch is evaluated using its natural representation of its pattern as an NFA. However, the states of this NFA are represented by *constraints*: At the beginning, the NFA is in its initial state, meaning that the constraint of q_0 is **true** while the one of all other states is **false**. As the automaton takes transitions into other states, the constraints of those states are simply updated with the valuations which hold at those states. This is similar to the fact that in *J-LO* states are represented by formulae with their appropriate binding.

6.3.2.3 Static precalculation of states

One major difference to *J-LO* is that the entire NFA is precomputed, including the transition structure. Over LTL this is hardly possible because LTL allows explicit negation and conjunct, which must be paid for by a double exponential blowup when calculating all possible transitions: For any LTL formula φ of size $|\varphi| = |cl(\varphi)| =: m$ over n propositions, there can be up to 2^{mn} different successor configurations, because the AFA can for each different subset of propositions take a transition into any subset of states/subformulae (whose number is bounded by m).

We did gave static precalculation a try using a method described by Giannakopoulou and Havelund [GH01]. The AFA is translated into a DFA as follows: Starting with the initial formula as state, calculate all possible successor configurations for all possible 2^n combinations of propositions. Repeat this for all new configurations until all configurations have been generated. Since there can only be up to 2^m such configurations, the algorithms must eventually terminate. Our implementation should be reasonably fast given that we implemented it by indexing propositions as integers using bit shift operations throughout the whole calculation. The authors are happy to provide the implementation on request. However still, for our LOR example formula, this yielded a DFA with 214 states, whose calculation took about 54 seconds on a 3GhZ Pentium 4 with the JDK 1.5.0_04-b05 on Windows which we consider as too slow for practical applications.

In the case of regular expressions, which have no explicit negation nor conjuncts, the tracematch implementation can use the following equation in order to ease the implementation:

$$\delta(q, \{a_{i_1}, \ldots, a_{i_j}\}) = (\delta(q, a_{i_1}), \ldots, \delta(q, a_{i_j}))$$

This means that if the automaton is in state q takes a transition at a state where symbols $\{a_{i_1}, \ldots, a_{i_j}\}$ hold, it can simply take the transitions one after another. Hence, it suffices to calculate the transition structure for *each single symbol*, which yields a transition table of at most the size of $m \times n$ where m is the number of states of the NFA and n the number of symbols.

6.3.2.4 Expressiveness

As a result, in tracematches it is generally impossible to express that certain propositions should be valid *at the same time*, because the above equality implies a disjunction, not a conjunction. Also negation can only be expressed implicitly. In most cases this is not a real problem because the $\wedge$ and $\neg$ operators can be pushed down into the symbol definitions, because for each two symbols s and t there exists symbols $\neg s$ and $s \wedge t$, due to the fact that symbols are in fact pointcuts. However, in pointcuts (and hence symbols) variables cannot be bound under negation, because if a pointcut does not match, there is no binding to expose. Hence, pushing down negations into symbols is not possible when variables are to be bound by this symbol.

Assume for example the *DLTL* formula $\mathbf{F}(a(x,y) \wedge \mathbf{XF}(b(x) \wedge \neg c(y)))$. Here one would need to find a regular expression that makes sure that when $b(x)$ holds, $c(y)$ does *not* hold at the same time. Since negation is not explicitly possible, a first try could be to include a symbol $c(y)$ in the alphabet Σ which is not contained in $\mathcal{P}$. This however would rule out traces as $a(1,2)c(2)b(1)$, which are matched by the formula but would be forbidden by such a regular expression. Also $b(x) \wedge \neg c(y)$ cannot be combined to a single symbol $b\_nc(x,y)$, because that would mean that the $c(y)$ part would bind y under negation, which is not possible at the moment in the implementation of tracematches. Table 6.6 shows an invalid tracematch which visualizes this attempt. In line 3, a compiler error would be issued, because the symbol *b_nc* is matched whenever c does *not* hold. Hence, *b_nc* cannot bind y.

As a result, such formulae cannot currently be expressed in tracematches. Allowing explicit negation and conjunction in patterns would easily close this gap, however would naturally lead to the same complexity problems with static precalculation.

Regular expressions vs. LTL On the other hand there are certain properties which are hard to express in *DLTL*. For instance the fact that an event a is followed by b which is followed by c can in a regular expression be written simply as $a\ b\ c$, while in LTL one would have to specify this as $a \wedge (\ (\neg b \wedge \neg c)\ U\ (b \wedge\ (\neg c)\ U\ c)\ )$. This is due to the fact that regular expressions are

```
tracematch(X x, Y y) {
    sym a after: a(x,y);
    sym b_nc after: b(x) && !c(y);

    a b_nc {
        //do something
    }
}
```

Table 6.6: Invalid tracematch binding y under negation

evaluated from one element of a concatenation to another, while in LTL formulae are evaluated from one context to another, those being divided by **X** operators as described in the semantics section. Hence, in regular expressions it is cumbersome to specify events that may *interleave*: For instance if one want to specify that somewhere on the trace the events a, b and c happen, this can be written simply as $\mathbf{F}a \wedge \mathbf{F}b \wedge \mathbf{F}c$, while in a regular expression that would need to be written as $(a\Sigma^*(b\Sigma^*c)|(c\Sigma^*b)) \mid (b\Sigma^*(a\Sigma^*c)|(c\Sigma^*a)) \mid (c\Sigma^*(b\Sigma^*a)|(a\Sigma^*b))$.

Generally however, regular expressions are slightly more powerful than LTL in a theoretical context, since LTL is known to be unable to describe *counting* languages (see [TRW03]). The term *counting* is here meant as *modulo counting*: For instance *an even number of a's* can easily be defined by the regular expression $(aa)^*$, while it is impossible to define this language in LTL. However, we think that this kind of counting is of questionable use in the setting of Runtime Verification.

6.3.2.5 Synergies

Generally we found that most design decisions during the implementation of both, *J-LO* and tracematches were reasonable with respect to the actual purpose and targeted user base of the tool: While we still believe that LTL is probably the best suited specification language for the purpose of Runtime Verification, there may well be cases where regular expressions are shorter and easier to read as shown above. In the general setting of tracematches, which allow the execution of arbitrary code and hence form a general purpose language for traces matching, targeting a broad user base, regular expressions are certainly the better choice taking into account their broad acceptance.

Generally both implementations have a lot in common and actually only differ in the input language and the automaton model (AFAs vs. NFAs).

Starting on January 1st 2006, the author of this work will be joining the abc group, who is meanwhile optimizing the runtime overhead of the tracematch implementation by the means of static analysis. Hence, it might well be the case that parts of *J-LO* might be replaced by the tracematch codebase or reused there conversely.

We point the interested reader to [BS06], where we introduce *tracechecks*, an approach to unify both, the tracematch implementation and *J-LO*. In this work we show how tracematches can be expressed in the terms of *J-LO* with only slight modifications to the operational semantics.

Chapter 7

Conclusion

We have presented a runtime verification tool based on the aspect-oriented programming language AspectJ and linear temporal logic (LTL). We make use of AspectJ to instrument the application to verify on the one hand and employ AspectJ pointcuts as propositions of our logic on the other hand. As a result, we gain an expressive formalism which is capable of instance based reasoning about the execution trace of any Java application.

The major contributions of this work are a precise declarative and operational semantics of LTL for free variable bindings and a trace model based on predicate logic which is rich enough to allow for the evaluation of such formulae. We presented the reference implementation *J-LO* and reported on important details of this implementation with respect to the operational semantics. When comparing to related work on the field of aspect-oriented programming, we noticed that all previous approaches model a trace as a pure *sequence* of states — a model, which is not rich enough to be able to express interleaving and overlapping events. Hence, within this field, the trace model as such can be seen as a contribution of its own.

Further we discussed how *J-LO* can be used to identify temporal bug patterns such as faulty use of data structures (stacks, iterators, hast sets) as well as the problem of lock order reversal. We commented on the runtime overhead one has to expect when applying *J-LO* to an application and pointed out room for further optimization.

Future work will comprise optimizing the implementation through static checks approximation and applying *J-LO* to some industrial size applications.

Our prototype of *J-LO* is available from
`http://www-i2.informatik.rwth-aachen.de/JLO/`.

Bibliography

[AAS$^+$04] 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.

[ACH$^+$05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondréj Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. An extensible AspectJ compiler. In *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, USA, March 2005. ACM Press.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. pages 164–169, 1989.

[All02] Eric E. Allen. Diagnosing Java code: Using temporal logic with bug patterns. `http://www.ibm.com/developerworks/`, August 2002.

[Asp] AspectJ online documentation. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[BCC$^+$05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[BCM05] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. In R. Ramanujam and Sandeep Sen, editors, *Proceedings of the 25th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, Lecture Notes in Computer Science, Hyderabad, India, December 2005. Springer.

[Bec00] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes of Computer Science*. Springer, 2004.

[BM03] Michael Brukman and Andrew C. Myers. PPG: a parser generator for extensible grammars, 2003. Available at `www.cs.cornell.edu/Projects/polyglot/ppg.html`.

[BMO05] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execcution. *2nd Dynamic Aspects Workshop (DAW05)*, pages 71–75, 2005.

[Bod04] Eric Bodden. A lightweight LTL runtime verification tool for Java. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 306–307. ACM Press, 2004.

[Bod05a] Eric Bodden. Concern specific languages and their implementation with abc. In *SPLAT 2005: Software engineering Properties of Languages for Aspect Technologies at AOSD '05*, March 2005.

[Bod05b] Eric Bodden. Efficient and Expressive Runtime Verification for Java. In *Grand Finals of the ACM Student Research Competition 2005*, 03 2005.

[Bod05c] Eric Bodden. Implementing concern-specific languages with abc. In *In Seminar on Aspect-oriented technologies. Institut für Informationssysteme, Prof. Steimann, Fachgebiet Wissensbasierte Systeme, Hannover University.*, 02 2005.

[Bon04] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In Murphy and Lieberherr [ML04], pages 5–6.

[BS06] Eric Bodden and Volker Stolz. Tracechecks: Combining tracematches and temporal logic. Download: `http://www.bodden.de/publications`, 2006.

[CCBH03] Adrian Colyer, Andy Clement, Ron Bodkin, and Jim Hugunin. Using aspectj for component integration in middleware. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 339–344, New York, NY, USA, 2003. ACM Press.

[CCHW05] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* Pearson Education, 2005.

[CDE$^+$99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* SRI International, 1999.

[CDE$^+$03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[CI91] Inc. Staff CORPORATE IEEE. *IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language.* IEEE Standards Office, 1991.

[Cla99] J. Clark. XSL transformations (xslt. W3C Recommendation, November 1999.

[con] Contract4J, Design-by-Contract extension for Java. `http://www.contract4j.org`.

[Cos03] Pascal Costanza. Dynamically scoped functions as the essence of AOP. *SIGPLAN Notices*, 38(8):29–36, 2003.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st intl. conf. on Software engineering*, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[DFS02] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, 2002.

[DFS04a] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *3rd International Conference on Aspect-oriented Software Development*, pages 141–150, 2004.

[DFS04b] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In *Aspect-oriented Software Development*, pages 141–150. Addison-Wesley, 2004.

[DGH$^+$04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 150–169. ACM Press, 2004.

[dH05] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[DMS01] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2001.

[Eic05] Michael Eichberg. BAT2XML: XML-based java bytecode representation. *Bytecode' 05, Edinburgh, Scotland*, April 2005.

[EMOS04] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In Bob Werner, editor, *Eleventh Working Conference on Reverse Engineering*, pages 182–191, Delft, Netherlands, November 2004. IEEE Computer Society.

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE*, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Third Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *CAV*, 2001.

[Goo01] James Goodwill. *Apache Jakarta-Tomcat*. APress, 2001.

[Hav00] 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.

[HBS03a] K. Havelund H. Barringer, A. Goldberg and K. Sen. EAGLE does space efficient LTL monitoring. Pre-Print CSPP-25, Department of Computer Science, University of Manchester, October 2003.

[HBS03b] K. Havelund H. Barringer, A. Goldberg and K. Sen. Eagle monitors by collecting facts and generating obligations. Pre-Print CSPP-26, Department of Computer Science, University of Manchester, October 2003.

[HBS04] K. Havelund H. Barringer, A. Goldberg and K. Sen. Program monitoring with ltl in eagle. In *18th International Parallel and Distributed Processing Symposium, Parallel and Distributed Systems: Testing and Debugging - PADTAD'04*. IEEE Computer Society Press, April 2004. ISBN 0769521320.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Murphy and Lieberherr [ML04], pages 26–35.

[Hol04] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.

[HR01a] K. Havelund and G. Roşu. Java PathExplorer - A Runtime Verification Tool, June 2001.

[HR01b] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer, 2001. `http://www.elsevier.nl/locate/entcs/volume55.html`.

[HR01c] Klaus Havelund and Grigore Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report RIACS 01.08, Research Institute for Advanced Computer Science, May 2001.

[HR04] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[Jav04] JavaOne conference, 2004. `http://java.sun.com/javaone`.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[JPr] Sitraka Inc. *The JProbe Profiler*. `http://www.jprobe.com`.

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

[KVK$^+$04] 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.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[Lad05] Ramnivas Laddad. AOP and metadata: A perfect match. article on the IBM DeveloperWorks network, 03 2005. `http://www-128.ibm.com/developerworks/java/library/j-aopwork3/`.

[Led83] Henry Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[Lee67] Char-Tung Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms.* PhD thesis, University of California, Berkeley, 1967.

[Lem04] Otávio Augusto Lazzarini Lemos. Is 'advice' adequate?, 08 2004. Thread in the aosd-discuss mailing list (`http://www.aosd.net/`).

[Mey92a] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[Mey92b] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[ML04] Gail C. Murphy and Karl J. Lieberherr, editors. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*. ACM, 2004.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In Yuri Gurevich, editor, *Proceedings of the Third Annual IEEE Symp. on Logic in Computer Science, LICS 1988*. IEEE Computer Society Press, July 1988.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152, 2003.

[NS03] 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.

[Odd] D. Oddoux. LTL2BA. `http://www.liafa.jussieu.fr/~oddoux/ltl2ba/`.

[OM04] Klaus Ostermann and Mira Mezini. Design and implementation of pointcuts over rich program models. Technical report, Darmstadt University of Technology, June 2004.

[Pnu77] A. Pnueli. The Temporal Logics of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[Roh97] Gareth Scott Rohde. *Alternating automata and the temporal logic of ordinals*. PhD thesis, 1997. Adviser-Paul E. Schupp.

[SB05] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*. To be published in ENTCS, Elsevier, 2005.

[SBN$^+$97] 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.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[SFS02] Dharma Shukla, Simon Fell, and Chris Sells. Aspect-oriented programming enables better code encapsulation and reuse. MSDN Magazine, March 2002. `http://msdn.microsoft.com/msdnmag/issues/02/03/aop/`.

[SH04] 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.

[SV03] Davy Suvée and Wim Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In

Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 21–29. ACM Press, March 2003.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD*, pages 21–29, 2003.

[Tei66] W. Teitelman. Pilot: A step towards man-computer symbiosis. Technical report, Cambridge, MA, USA, 1966.

[Tho81] Wolfgang Thomas. A combinatorial approach to the theory of omega-automata. *Information and Control*, 48(3):261–283, 1981.

[Tho03] Wolfgang Thomas. Lecture notes on Model Checking, 2003. `http://www-i7.informatik.rwth-aachen.de/`.

[TRW03] Wolfgang Thomas, Philipp Rohde, and Stefan Wöhrle. Lecture notes on Automata and Reactive Systems, 2003. `http://www-i7.informatik.rwth-aachen.de/`.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[VRHS$^+$99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, 1999.

[VS04] Wim Vanderperren and Davy Suvee. JAsCoAP: Adaptive programming for component-based software engineering. In David H. Lorenz and Yvonne Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, March 2004.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[WKGS00] W.Visser, K.Havelund, G.Brat, and S.Park. Model Checking Programs. In *Proc. of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.

[WV04] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT FSE*, pages 159–169, 2004.

Appendix A

Overview of own related publications

Here we briefly point to own publications which are related to *J-LO*. They are listed in chronological order. All those works can be found at:
`http://bodden.de/publications/`

A.1 A Lightweight LTL Runtime Verification Tool for Java

Eric Bodden. In OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications (Companion), SIGPLAN, Vancouver, Canada, October 2004. Citation code: [Bod04]

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

Background
This was the first work about *J-LO*. Here we first proposed the idea of applying Runtime Verification through specifications on the form of metadata annotations and implemented by the means of AOP. In this early stage we still propose that the *Next* operator should not be used because it seems unclear what the

next joinpoint is in an arbitrary aspect-oriented program. In later versions we saw that there *can* be situations where **X** is useful, so today *J-LO* does allow using this operator. This paper was submitted to the OOPSLA Student Research Competition and won the third prize in the undergraduate category. At the OOPSLA conference we got to know the *AspectBench Compiler*, on which *J-LO* is based on today.

A.2 Concern specific languages and their implementation with abc

Eric Bodden. Proceedings of Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) 2005, Chicago, USA, March 2005. Citation code: [Bod05a]

Abstract

In this work first we introduce the notion of concern specific languages (CSL) which are to a specific crosscutting concern, what domain specific languages are to a specific domain. Implementing such CSLs was a tedious task in the past since no extensible frameworks for implementing crosscutting concerns existed. With the AspectBench Compiler (abc), which was released in October 2004, researchers now have a powerful extensible compiler for the aspect-oriented language AspectJ, enabling easy implementation of language extensions or even whole CSL for a specific crosscutting concern. We first motivate CSLs in general and give examples of such languages which exist already. In the subsequent chapters we introduce one specific CSL and report on our implementation using abc and specifically about how CSLs can interact with and reuse each other. We will see that the use of CSLs in general provides better comprehensibility and analyzability. Finally we show how metadata annotations can facilitate the implementation and deployment of CSLs.

Background

This paper is theoretical in nature and discusses the LTL we use with respect to other domain specific languages. During our research we found that other languages as our LTL exist which are not really specific to an application domain but rather to a common crosscutting concern (e.g. Runtime Verification). This yields a definition of *Concern Specific Languages* (CSL) and identifies the LTL we use as one of those.

A.3 Implementing concern-specific languages with abc

Eric Bodden. In Seminar on Aspect-oriented technologies. Institut für Informationssysteme, Prof. Steimann, Fachgebiet Wissensbasierte Systeme, Hannover

University. February 2005. Citation code: [Bod05c]

Abstract
In this work first we introduce the notion of concern specific languages (CSL) which are to a specific crosscutting concern, what domain specific languages are to a specific domain. Implementing such CSLs was a tedious task in the past since no extensible frameworks for implementing crosscutting concerns existed. Ostermann and Mezini proposed a Prolog based implementation of aspect-oriented programming which would enable easy extension of the programming language. However, it is not yet clear what runtime impact will be involved with such an approach. With the AspectBench Compiler (abc), which was released in October 2004, researchers now have a powerful extensible compiler for the aspect-oriented language AspectJ, enabling easy implementation of language extensions or even whole CSL for a specific crosscutting concern. We first motivate CSLs in general and then introduce the abc framework. In the subsequent chapters we introduce our specific CSL and report on the steps necessary to implementing it using abc. Finally we recapitulate on the ease of use of abc and conclude with a proposal for further development of this compiler framework.

Background
This work gives many implementation details with respect to the frontend, i.e. the *abc* framework. Its purpose is not only to explain how our LTL logic can be implemented with *abc* — since there was only little documentation about *abc* at the time, we also how *abc* can be extended for the implementation of Concern Specific Languages in general.

A.4 Efficient and Expressive Runtime Verification for Java

Eric Bodden. In proceedings of the Grand finals of the ACM Student Research Competition 2005, San Francisco, CA, USA, March 2005. Citation code: [Bod05b]

Abstract
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 over finite traces. 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-scale applications. The instrumentation code can be proven to be side-effect free, meaning, that it is oblivious to the base application.

Background
This work is winner of the Grand Finals of the 2005 ACM Student Research Competition, undergraduate categorie. It is an extended version of the OOPSLA'04 paper containing an extensive comparison to previous work and explaining an approach which directly translates formulae into Büchi automata. Such an approach promises to be highly efficient at runtime because the state to be stored is reduced to a minimum. However, it gives in general no indication of how bindings should be updated during matching. That's why we opted for alternating automata in this thesis. Nevertheless, we provide a discussion of the Büchi automaton attempt in section B.

A.5 Temporal Assertions using AspectJ

Volker Stolz, Eric Bodden. In proceedings of RV'05 - Fifth Workshop on Runtime Verification, Satellite workshop of CAV 2005, Edinburgh, Scotland, UK, July 2005. Citation code: [SB05]

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.

Background
The contribution of this submission is the formal model we employed in *J-LO*. In particular we explain how we make use of alternating automata and how we employ AspectJ in order to retrieve a trace of a running application.

A.6 Tracechecks: Combining tracematches and temporal logic

Volker Stolz, Eric Bodden. Submitted to AOSD'06 - Sixth annual ACM conference on Aspect-Oriented Software Development, Bonn, Germany, March 2006. Citation code: [BS06]

Abstract
Tracechecks are a formalism based on linear temporal logic (LTL) with variable

bindings and AspectJ pointcuts 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 the application. 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.

Tracechecks are closely related to *tracematches*, an AspectJ extension provided in the AspectBench Compiler (abc). Tracematches allow to specify trace conditions as regular expressions.

We provide an extensive comparison of tracechecks and tracematches, specifically showing that, unlike regular expressions, LTL offers quantification, negation and conjunction and yields shorter specifications of interleaving events. Since regular expressions are better suited for some specifications, we show how both approaches can be unified in a single operational semantics. This yields a powerful framework for temporal reasoning, which allows for the specification of temporal assertions in a combined formalism.

Background

In this work we provide an extensive comparison to tracematches [AAS$^+$04] (cf. section 6.3.2) and show that they can be expressed in terms of the alternating automata present in *J-LO* with only slight modifications to the backend. This paper was finished right before finishing this thesis and so represents the latest state of the project.

Appendix B

Pitfalls we came across

In this section we briefly want to comment on some pitfalls we came across during our research, in order to prevent other people making the same mistakes.

B.1 Büchi automata over finite paths

Our initial publication at OOPSLA'04 (cf. section A.1) we were still of the opinion that ordinary Büchi automata can be used to implement finite path semantics of LTL formulae by simple interpreting them as NFA. As Giannakopoulou and Havelund noted [GH01] and as we found out by own test cases, this is not generally possible: The generated Büchi automata have to fulfill certain conditions, which were not always fulfilled when generating Büchi automata using existing tools.

In particular, we applied the program *ltl2ba* [Odd] to several example formulae. ltl2ba takes an LTL formula (without free variables) as input and generates a _automata equivalent to that formula - "equivalent" here meaning equivalence over *infinite* paths. Unfortunately it turned out that the generated automata did not yield the same equivalence over *finite* paths. As an example, consider the automaton in figure B.1.

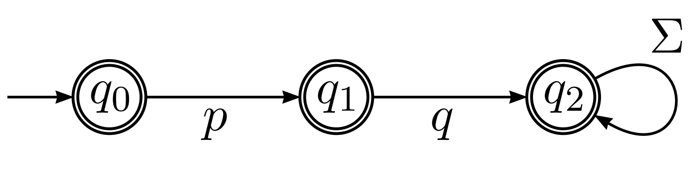


Figure B.1: Büchi automaton for $p \wedge \mathbf{X}q$

It shows the Büchi automaton for the formula $p \wedge \mathbf{X}q$ generated by ltl2ba. Over infinite paths it implements the correct semantics: A final state will be visited infinitely often if and only if p holds at the first state and q at the next one.

Note that in this automaton the state q_1 is *final*. In fact, for the language accepted by this automaton it makes no difference at all whether q_1 is final or not, because any finite prefix of an observed path is ignored in the acceptance condition for Büchi automata.

For any finite path however, the fact if q_1 is final is quite important: Consider the path $\pi = \{p\}$ of length 1. Here p holds at the first state, but q does not hold at the next state because there simply is no next state at all. Of course, we would like to reject such a trace. Hence we cannot simply use the Büchi automaton and interpret it over finite paths, because in this case the trace would be accepted due to the fact that q_1 is final.

So in a nutshell, Büchi generated by ltl2ba are useless for us. Yet there may be other tools around, which generate automata that can be used over finite paths. The webpage `http://www.ti.informatik.uni-kiel.de/~fritz/ABA-Simulation/ltl.cgi` gives an overview of the results of various LTL to Büchi automaton translation tools.

Another reason for why we switched from Büchi automata to alternating automata is that the former give no indication of how bindings should be propagated during evaluation. In the AFA approach, it became clear that free variables have to be bound whenever moving to a new state via the **X** operator and conversely *no* binding should take place when taking a self-loop back to the original state. This behaviour was induced by the partially ordered state set of an AFA. When generating a Büchi automaton from an AFA, one abstracts away all opportunities for such perceptions.

B.2 The trace model

In the beginning of this research we made the same mistake which has been done several times in related work in the field of aspect-oriented programming (cf. section 6.3): We chose the wrong trace model. When thinking of an execution trace one usually thinks of a sequence of events that occur during the execution of a program. However, since we are using pointcuts which are *predicates* over such events, it should become clear that we actually need to employ a predicate logic and an appropriate trace model, mapping each event to the *set* of predicates that match this event. As we show in [BS06], such a trace model is important in order to derive a logic which is closed under negation.

Appendix C

AspectJ pointcuts

The pointcut language of AspectJ 1.2 comprises the following kinds of pointcuts. Informal definitions of id, type, method and constructor patterns are given on page 28.

Context exposure

Those three pointcuts are used to expose context. They all match on actual runtime types.

This

Syntax `this(` *TypePattern* `)`, `this(` *Identifier* `)`

Semantics matches each joinpoint, where the currently executing object is an instance of a type matched by *TypePattern* resp. the declared type of the *Identifier*

binds ***Identifier*** to the currently executing object

Target

Syntax `target(` *TypePattern* `)`, `this(` *Identifier* `)`

Semantics matches each joinpoint, where the called object is an instance of a type matched by *TypePattern* resp. the declared type of the *Identifier*

binds ***Identifier*** to the target object

Args

Syntax `args(` *ArgPattern* `)`

Semantics matches each joinpoint, where the actual types of the arguments are matched by the *ArgPattern*

binds *Identifier* to an array containing the argument objects; primitive are automatically boxed into objects

---

Primitive / kinded pointcuts

Those pick out joinpoints of a certain kind (method call, field access, etc.). They all match the execution of a single statement or a interval of the dynamic control flow. Each pointcut can expose state in combination with `this`, `target`,`args` (see above).

Execution

Syntax `execution(` *MethodPattern* `)`, `execution(` *ConstructorPattern* `)`

Semantics matches the execution of any method/constructor matched by the *MethodPattern*/*ConstructorPattern*

binds `this` to executing object (or `null` if method is static)

binds `target` to executing object (or `null` if method is static)

binds `args` to arguments of method invocation

Call

Syntax `call(` *MethodPattern* `)`, `call(` *ConstructorPattern* `)`

Semantics matches call to any method/constructor matched by the *Method-Pattern*/*ConstructorPattern*

binds `this` to caller object (or `null` if called from static context)

binds `target` to called object (or `null` if method is static)

binds `args` to arguments of method invocation

Get

Syntax `get(` *FieldPattern* `)`

Semantics matches reading access to any field matched by the *FieldPattern*

binds `this` to accessed object

binds `target` to accessed object

binds `args` to empty

Set

Syntax `set(` *FieldPattern* `)`

Semantics matches writing access to any field matched by the *FieldPattern*

binds `this` to accessed object

binds `target` to accessed object

binds `args` to new field value

Static Initialization

Syntax `staticinitilization(` *TypePattern* `)`

Semantics matches initialization of all static members as well as the execution of the `static{...}` block in all types matched by *TypePattern*

binds `this` to `null`

binds `target` to `null`

binds `args` to empty

Pre-Initialization

Syntax `preinitilization(` *ConstructorPattern* `)`

Semantics matches code executed between entry of a constructor matched by *ConstructorPattern* and the first line after the call to `super(...)`

binds `this` to `null`

binds `target` to `null`

binds `args` to arguments of constructor invocation

Initialization

Syntax `initilization(` *ConstructorPattern* `)`

Semantics matches code executed in a constructor matched by *Constructor-Pattern* starting from the first line after the call to `super(...)`

binds `this` to object being initialized

binds `target` to object being initialized

binds `args` to arguments of constructor invocation

Execution Handler

Syntax `handler(` *TypePattern* `)`

Semantics matches code executed inside exception handlers for exceptions matched by *TypePattern*

binds `this` to executing object (or `null` if surrounding method is static)

binds `target` to executing object (or `null` if surrounding method is static)

binds `args` to the exception to handle

Lexical scope pointcuts

Those allow to restrict other pointcuts to certain lexical scopes.

Lexical scoping over types

Syntax `within(` *TypePattern* `)`

Semantics matches code in the lexical scope of a type matched by *TypePattern*

binding of `this`,`target`,`args` `null` resp. empty

Lexical scoping over methods/constructors

Syntax `withincode(` *MethodPattern* `)`, `withincode(` *ConstructorPattern* `)`

Semantics matches code in the lexical scope of a method/constructor matched by *MethodPattern*/*ConstructorPattern*

binding of `this`,`target`,`args` `null` resp. empty

Control flow-based pointcuts

Those allow to restrict matching to the control flow of other pointcuts.

Control flow

Syntax `cflow(` *Pointcut* `)`

Semantics matches any joinpoint occurring in the control flow of the any joinpoint matched by *Pointcut*

binding of `this`,`target`,`args` `null` resp. empty

Control flow (below)

Syntax `cflowbelow(` *Pointcut* `)`

Semantics matches any joinpoint occurring in the control flow of the any joinpoint matched by *Pointcut* which is not matched by *Pointcut* itself

binding of `this`,`target`,`args` `null` resp. empty

Expression-based pointcuts

Those allow the dynamic evaluation of Boolean expressions.

Boolean evaluation

Syntax `if(` *BooleanExpression* `)`

Semantics matches any joinpoint at which the *BooleanExpression* holds; the *BooleanExpression* can only access static members, parameters exposed by the enclosing pointcut or advice, and reflective information

binding of `this`,`target`,`args` `null` resp. empty

Appendix D

Contents of the CD-ROM

Attached to this thesis you find a CD-ROM with the following contents:

- This thesis
- The final presentation
- Copy of the J-LO-Website
- Full source code of the J-LO distribution (version 0.9.1) along with
- Binary distribution of J-LO (version 0.9.1) to run test instances including full API documentation for the runtime library
- Test cases
- Copies of all scientific papers published by Eric Bodden and related to J-LO (see appendix A)

Details are given by the file `index.html` in the root directory of the CD-ROM. It can be viewed using any standard web browser. In order to view all of the documents, you will also need the free *Acrobat Reader*, which can be downloaded at `http://www.adobe.com/products/acrobat/readstep.html`.

Appendix E

Symbols and notations

$\mathcal{A}$	automaton, page 14
$\mathcal{B}$	a Büchi automaton, page 15
F	set of final states, page 14
Q	finite set of states, page 14
q_0	initial state, page 14
ρ	run of an automaton, page 14
Σ	finite alphabet, page 14
Δ	transition relation, page 14
$\mathcal{A}_{(M,s)}$	automaton for a pointed transition system, page 15
$\mathcal{A}_\varphi$	automaton equivalent to φ, page 15
DOM	a global domain, page 40
$dom(x)$	the domain w.r.t. variable x, page 40
$\lambda x.(x > 0)$	the function evaluating $x > 0$ with x free variable, page 41
$\mathcal{L}(\mathcal{A})$	language recognized by $\mathcal{A}$, page 14
$\mathbf{A}\ \varphi$	CTL*/CTL operator *for all paths* φ, page 11
$\mathbf{E}\ \varphi$	CTL*/CTL operator *exists a path s.th.* φ, page 11
$\mathbf{F}\ \varphi$	CTL*/LTL operator *finally* φ, page 11
$\mathbf{G}\ \varphi$	CTL*/LTL operator *globally* φ, page 11
$\mathbf{X}\ \varphi$	CTL*/LTL operator *next* φ, page 11
$\varphi\ \mathbf{R}\ \psi$	CTL*/LTL operator φ *releases* ψ, page 11
$\varphi\ \mathbf{U}\ \psi$	CTL*/LTL operator φ *until* ψ, page 11
$\varphi\ \mathbf{W}\ \psi$	CTL*/LTL operator φ *until* ψ (weak), page 40
LTL^{nnf}	set of LTL formulae in negation normal form, page 16
$\{x \mapsto 1\}$	anonymous function mapping x to 1, page 45
M	a *model* / Kripke structure, page 11
$[\![\varphi]\!]$	semantics of φ, page 40
$\tilde{f}$	appropriately overloaded version of function f, page 45

p	a proposition, page 11
$\mathcal{P}$	a set of propositions, page 11
$p(x)$	a proposition with free variable x, page 38
$p(x, \underline{y})$	a proposition providing a value for x and using y, page 39
S	finite state set of a Kripke structure, page 11
L	labeling function of a Kripke structure, page 11
R	edge relation of a Kripke structure, page 11
$s \models \varphi$	state s models/satisfies formula φ, page 12
$next(\varphi)$	future part of φ, page 42
$now(\varphi)$	current part of φ, page 42
π	a path, page 12
$\pi[i]$	the i-th state of path π, page 12
π^i	the suffix of π starting at $\pi[i]$, page 12
$s \bigotimes t$	clause product of clause sets s and t, page 17
$S_1 \cap S_2$	intersection of S_1 and S_2, page 52
$S_1 \cup S_2$	union of S_1 and S_2, page 17
2^S	set of all subsets of S (powerset), page 11
$\vec{x}$	a set or vector $x = \{x_1, \ldots, x_n\}$, page 38
`<>`	SPIN syntax for *finally*, page 21
`[]`	SPIN syntax for *globally*, page 21
ff	state representing **false**, page 12
tt	state representing **true**, page 12

Index