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 Mathematik, Physik und Informatik der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im **XXX** 2005

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

Dankwort bla bla

Widmung bla bla

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 06.08.2005 Eric Bodden

Zusammenfassung

Abstract

In this work we present the tool *Java Logical Observer*, *J-Lo* for short, which enables checking of temporal assertions during runtime of an arbitrary Java application. Such temporal assertions are defined using a linear temporal logic (LTL). They can be deployed using standard Java 5 annotations and thus become part of the public interface of this class. Formulae in this temporal logic are parsed by an extended version of the *abc* compiler. From the formula representation as abstract syntax tree, we produce an alternating automaton with semantics which are equivalent to the semantics of the original formula. This alternating automaton is then rendered into an aspect in the aspect-oriented language *AspectJ*. Such an aspect implements the runtime checking of a formula as separate compilation unit. It is then woven into the original application, resulting in an instumented version of the original application. All transformations are performed purely on Java bytecode and thus allow also instrumentation of third party libraries.

As an extension we explain how alternating automata can be modified to bind state while the execution trace is monitored. This state can then be reasoned about leading to enhanced expressiveness. To our best knowledge, J-Lo is the first verification tool to offer such a feature.

Contents

1 Motivation **5**

1.1 Semantic Interfaces and Temporal Interdependencies 5

1.2 Motivating Examples 6

1.2.1 A simple stack 6

1.3 An overview of our solution 8

2 Background **9**

2.1 Formal verification 9

2.1.1 Static verification - Model checking 9

2.1.1.1 Transition systems and Kripke structures 10

2.1.1.2 CTL*, CTL and LTL 10

2.1.1.3 LTL model checking 13

2.1.1.4 From LTL to alternating automata 14

2.1.1.5 From alternating automata to Büchi automata . 17

2.1.1.6 LTL Model Checking in Spin 17

2.1.1.7 From Model Checking to Runtime Verification . 20

2.1.2 Runtime Verification 20

2.2 Java 5 Metadata 21

2.3 Aspect-oriented programming 23

2.3.1 The anatomy of an aspect 24

2.3.1.1 Joinpoints 24

2.3.1.2 Pointcuts 24

2.3.1.3 Example 26

2.3.1.4 Advice 29

2.3.1.5 Advice precedence 31

2.4 AspectJ and metadata 31

2.4.1 Supplying metadata 32

2.4.2 Consuming metadata 32

3 The syntax and semantics of *DLTL* **33**

3.1 Syntax of *DLTL* . . . 34
3.2 Towards a declarative semantics . . . 35
3.2.1 Notation . . . 35
3.2.2 General finite path semantics . . . 36
3.2.3 Why usual quantification semantics are insufficient . . . 37
3.2.4 Transformation to *now* and *next* . . . 39
3.2.4.1 The notion of *now* and *next* . . . 39
3.3 Declarative semantics of *DLTL* . . . 42
3.3.1 Basic definitions . . . 43
3.3.2 Bindings by example . . . 46
3.3.3 Possible valuations . . . 48
3.3.4 Redefinition of *next* for *DLTL* . . . 49
3.3.5 Declarative semantics of a *DLTL* formula . . . 50
3.4 Static analysis . . . 51
3.5 Operational semantics of *DLTL* . . . 57
3.5.1 General assumptions . . . 57
3.5.1.1 Alternating automata . . . 57
3.5.1.2 Valid formulae . . . 58
3.5.1.3 Valid `if` pointcuts . . . 58
3.5.1.4 No garbage collection . . . 58
3.5.1.5 No side effects . . . 58
3.5.2 Basic Definitions . . . 58
3.5.2.1 Definition of δ . . . 62
3.5.3 Operational semantics of a *DLTL* formula . . . 63
3.5.4 Proof of equivalence of declarative and operational semantics . . . 64
3.5.4.1 Correctness of alternating automata . . . 64
3.5.4.2 Correctness on the propositional level . . . 64
3.5.4.3 Correctness in general . . . 65

4 Implementation **67**
4.1 Annotation extraction . . . 67
4.2 The AspectBench compiler . . . 68
4.2.1 Structure of *abc* . . . 68
4.2.2 Polyglot . . . 69
4.2.3 Soot . . . 70
4.3 Code generation . . . 72
4.3.1 Propositions . . . 72
4.3.1.1 If-closures . . . 72

4.3.2 Initial formula . . . 74
4.3.3 Initialization/bootstrapping code . . . 75
4.3.4 Mechanism for collecting propositions . . . 75
4.3.5 Triggering a transition . . . 78
4.3.6 Multithreading issues . . . 78
4.4 Dealing with exceptional runtime behaviour . . . 79
4.4.1 Notification of shutdown . . . 79
4.4.2 Behaviour under presence of garbage collection . . . 80
4.4.3 Observering configuration changes . . . 81
4.4.3.1 User caused exceptions . . . 81
4.4.3.2 Custom observers . . . 83

5 Metrics and performance 84
5.1 Correctness of the implementation . . . 84
5.2 Performance . . . 85
5.2.1 Theoretical performance . . . 85
5.3 Performance of the implementation . . . 86

6 Related Work 88
6.1 Design by contract . . . 88
6.1.0.1 JML . . . 90
6.2 Runtime Verification . . . 90
6.2.0.2 Java PathFinder and Java PathExplorer . . . 90
6.2.0.3 ptrace . . . 90
6.2.0.4 Temporal rover . . . 90
6.2.0.5 Orchids . . . 90

7 Conclusion 91
7.1 Future work . . . 91
7.2 Pitfalls we came across . . . 91
7.3 Own related publications . . . 91

8 APPENDIX 92
8.1 AspectJ pointcuts . . . 92

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 builtin. 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 11 implements the precondition `child != null`, which is informally stated in the documentation of the application interface (API) in line 6. If the application is started with the command line parameter `-enableassertions` and 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. According to Sun[1], disabled assertions impose no runtime overhead at all.

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 localized 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

[1] http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html

```
1  public class InnerNode implements TreeNode {
2
3    private TreeNode child;
4
5    /** Constructs a new inner node with child <code>child</code>.
6     * @param child The child node. May not be <code>null</code>. */
7    public InnerNode(TreeNode child) {
8      assert child != null;
9      this.child = child
10   }
11
12   ...
13 }
```

Table 1.1: Java assertion checking for non-nullness (available since Java 1.4)

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 [**?**], which is an excellent AlphaWorks 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 [**?**] 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 a running example for this thesis. However, inspecting the specification in detail, we may find the following shortcomings, which we are seeking to address in our implementation.

1. All free variables are untyped. It is unclear what type `n`, `x` or `y` should have.
   In *J-LO*, all variables are typed. This is achieved by a list of formal variable declarations which precedes each formula.

2. The statements are not bound to an object. It is unclear, whether e.g. `push(x)` refers to a call of `push(x)` on a particular stack or on any stack in the system.
   In *J-LO*, all names provide sufficient qualification to disambiguate such situations. This is automatically provided by the use of AspectJ pointcuts, which may use identifiers.

3. All formulae implicitly quantify over free variables. However, it is unclear, how such quantification should be implemented. This implies that it is also unclear, when exactly a variable is bound and when it is matched against.
   We provide full declarational and operational semantics for our logic.

4. It is unclear whether an implicit condition `x`$\neq$`y` exists or not.
   In *J-LO*, one can explicitly state if this condition should hold.

5. It is unclear whether `until` means a weak or strong Until. [2]
   J-LO provides the LTL-like strong Until operator. A weak Until operator can easily be simulated.

6. Nested events cannot easily be identified using the formalism employed above. A possible assertion could be that `size()` is never called from within the `push(..)` method. This would require an expression which can refer to the points on the time line where the execution of `push(..)` is entered and left.
   The formalism of *J-LO* is rich enough to provide a way of specifying such behaviour.

[2] For a strong `until` it holds that φ `until` ψ is `false` if φ always holds but ψ never holds. A weak `until` would be satisfied on such a path.

Please note that all of the above specification 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: 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.

In the following 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.

Chapter 2

Background

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 [**?**] the term *model checking* as

> *Model checking* [is a method] by which a desired behavioral 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 behaviors 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 behavioral property, which shall here be given in the form of a finite set of temporal formulae $\Phi = \{\varphi_1, \ldots, \varphi_n\}$.

The output of a model checking process is an answer `true` or `false` to the question *Does M satisfy Φ ?* or for short:

$$M \stackrel{?}{\models} \Phi$$

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) [**?**] in detail, since LTL is a formalism we employ for *J-LO* as well.

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 $P = \{p_1, \ldots, p_n\}$ 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
- $L : S \to 2^P$ 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 builds the model which is to be verified by a model checking process. In the following when referring to the term *Kripke structure* we implicitly mean that this structure is pointed.

2.1.1.2 CTL*, CTL and LTL

Linear temporal logic [**?**] or *LTL* for short is a fragment of the richer *generalized computational tree logic CTL**. Thus we first define CTL* and then explain the borders of LTL and 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 [**?**].

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 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:

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

This definition identifies R as the dual operator to U: it always holds that

$$(M,\pi) \models \varphi\ \mathbf{U}\ \psi \text{ iff } (M,\pi) \not\models \neg\varphi\ \mathbf{R}\ \neg\psi.$$

Example

bla

Figure 2.1: Example Kripke structure

Given be the Kripke structure of figure 2.1. We evaluate the following formulae at the state s_0:

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

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 $AFGp_i$ is a valid formula in CTL* but not in CTL. As in CTL*, a CTL formulae is evaluated at the starting state of a Kripke structure.

LTL is the *linear temporal logic*, a fragment of CTL* gained by removing the path quantifiers E and A. Thus, a LTL formula always reasons about the structure of a single path. A Kripke structure (M, s) satisfies a LTL formula φ if φ holds on *all paths* trough M starting at s. We define the set of all LTL formulae over a set L of propositional labels simply as LTL_L.

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 a LTL formula φ, both over propositions $\{p_1, \ldots, p_n\}$, 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 violating path through M.

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 2.1.2 (Büchi automaton)
A nondeterministic Büchi automaton is a quintupel $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ 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 $\pi = (\pi[0], \pi[1], \ldots) \in S^\omega$ of infinite length is an infinite sequence $\rho = (\rho_0, \rho_1, \ldots) \in Q^\omega$ 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 word w if there exists a run ρ of $\mathcal{A}$ on w that visiting states in F infinitely often.

Generally for any automaton $\mathcal{A}$, we define $\mathcal{L}_\mathcal{A}$ as:

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

Based on those Büchi automata, the model checking process then can be defined as follows:

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

Step 1 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. Thus we want to explain this step in detail. For the subsequent two conversions we point the interested reader to [**?**].

2.1.1.4 From LTL to alternating automata

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

Definition 2.1.3 (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 $\{\{q1, q2\}, \{q3\}\}$ represents the Boolean combination $(q1 \wedge q2) \vee q3$. So a transition leading from q_0 to $\{\{q1, q2\}, \{q3\}\}$ would mean a nondeterministic choice between moving simultaneously to $q1$ and $q2$ on the one hand or just to $q3$ 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 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. $\mathcal{A}$ accepts an input word $w \in \Sigma^\omega$ if there exists a run on $\mathcal{P}$, 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 *negational normal form* (NNF). In this form, negations only occur in front of propositions. We define the function NNF as follows:

$$
\begin{aligned}
NNF : LTL_L &\rightarrow LTL_L^{NNF} \\
\neg\mathbf{tt} &\mapsto \mathbf{ff} \\
\neg\mathbf{ff} &\mapsto \mathbf{tt} \\
\neg\neg\varphi &\mapsto NNF(\varphi) \\
\neg(\varphi \wedge \psi) &\mapsto \neg NNF(\varphi) \vee \neg NNF(\psi) \\
\neg(\varphi \vee \psi) &\mapsto \neg NNF(\varphi) \wedge \neg NNF(\psi) \\
\neg\mathbf{X}\ \varphi &\mapsto \mathbf{X}\ \neg NNF(\varphi) \\
\neg(\varphi\ \mathbf{R}\ \psi) &\mapsto (\neg NNF(\varphi)\ \mathbf{U}\ \neg NNF(\psi)) \\
\neg(\varphi\ \mathbf{U}\ \psi) &\mapsto (\neg NNF(\varphi)\ \mathbf{R}\ \neg NNF(\psi)) \\
else\ \varphi &\mapsto \varphi
\end{aligned}
$$

Here we assume that the operators **F** and **G** have already been reduced according to their semantics:

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

Given this negational normal form, we can now proceed with the specialization of our automaton model.

Definition 2.1.4 (AFA for a 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 $\varphi \in LTL_L^{NNF}$. Let the *closure of a formula*, $cl(\varphi)$, be the set of all sub-formulae of φ. Then

- $Q := cl(\varphi) \subseteq LTL_L^{NNF}$
- $\Sigma := 2^L$

- $q_0 := \varphi$
- $F := \{q \in Q \mid q = (\varphi \mathbf{R} \psi) \text{ for some } \varphi, \psi \in LTL_L^{NNF}\} \cup \{\mathbf{tt}\}$.

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 $\mathcal{P} \subseteq L, p \in L, \varphi, \psi \in LTL_L^{NNF}$. Then

- $\delta(p, \mathcal{P}) = \delta(\mathbf{tt}, \mathcal{P})$ resp. $\delta(\mathbf{ff}, \mathcal{P})$ if $p \in \mathcal{P}$ resp. $p \notin \mathcal{P}$
- $\delta(\neg p, \mathcal{P}) = \delta(\mathbf{tt}, \mathcal{P})$ resp. $\delta(\mathbf{ff}, \mathcal{P})$ if $p \notin \mathcal{P}$ resp. $p \in \mathcal{P}$ [1]
- $\delta(\mathbf{tt}, \mathcal{P}) = \{\{\}\}$ and $\delta(\mathbf{ff}, \mathcal{P}) = \emptyset$,
- $\delta(\varphi \wedge \psi, \mathcal{P}) = \delta(\varphi, \mathcal{P}) \bigotimes \delta(\psi, \mathcal{P})$
- $\delta(\varphi \vee \psi, \mathcal{P}) = \delta(\varphi, \mathcal{P}) \bigcup \delta(\psi, \mathcal{P})$
- $\delta(\mathbf{X}\ \varphi, \mathcal{P}) = \{\{\varphi\}\}$
- $\delta(\varphi\ \mathbf{U}\ \psi, \mathcal{P}) = \delta(\psi, \mathcal{P}) \vee (\delta(\varphi, \mathcal{P}) \wedge (\varphi\ \mathbf{U}\ \psi))$
- $\delta(\varphi\ \mathbf{R}\ \psi, \mathcal{P}) = \delta(\psi, \mathcal{P}) \wedge (\delta(\varphi, \mathcal{P}) \vee (\varphi\ \mathbf{R}\ \psi))$

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 [**?**].

A weak a 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.

[1] Note that negations occur only in front of propositions, since formulae are in NNF.

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 [**?**].

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 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 excurse 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 [**?**] is probably the most widely used LTL model checker today. Spin uses a process oriented modelling language, the *Process Meta Language*. The Spin website [2] states:

> *PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language.*

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

Table 2.1 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 process a 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.

[2] http://spinroot.com/spin/whatispin.html

```
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        od
12 }
13
14 proctype B() {
15        do
16        :: ch!b
17        od
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          od
27 }
28
29 init { atomic { run A(); run B(); run C() }
30 }
```

Table 2.1: A simple Spin example program

For example calling Spin with `spin -f '[]<> seen_a' model.prm` [3] specifies that proposition *seen_a* should **not**[4] hold infinitely often, which is equal to the fact that a is not received infinitely often.

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

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

[4] Note that Spin already takes the negated formula $\neg\varphi$ as input.

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 2.2.

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 tells, 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.

```
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 2.2: Never claim for *GF seen_a*

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*, which is 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.

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 [**?**, **?**], 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 often either model very 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.

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 this (finite) path.

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 quality of service. As a consequence, Runtime Verification usually colaborates well with unit testing [**?**], 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 talks about one single path, which is the current execution path in our case. The path is always finite 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[5] are evaluated over the whole path and their final state is reported during application shutdown.

Specification of those formulae takes place using *metadata annotations*. Those 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 [**?**]. Its specification took place in the Java Community Process (JCP) with the Java Specification Request 175 [**?**]. 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 2.3.

[5] such as *something happens again and again*

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

Table 2.3: 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 accorings 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 `PARAMETER`, `LOCAL_VARIABLE`, `ANNOTATION_TYPE` and `PACKAGE`, all of which seem not to be very suitable locations for the purpose of specifying formulae. We will further elaborate on our choice of allowed Element Types in section **??**. It should be noted that annotations on local variables can only have `SOURCE` retention, since Java bytecode is stack based and thus there are no local variables to annotate.

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 2.4.

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 adhere to the LTL syntax we define. `value` is actually a special label for a annotation parameter: If an annotation has only one parameter and its name is `value`, then this name not needs to be given 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* then extracts those annotations and applies the appropriate verification semantics to the application.

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

Table 2.4: Example LTL annotation in *J-LO*

In order to do so, *J-LO* needs to employ some intrumentation 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 [**?**, **?**]. Figure 2.2 [**?**] shows how the implementation of logging is originally implemented in the *Apache Tomcat* [**?**] servlet container. The 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 2.2: Logging as croscutting 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, imparative or object-oriented programming languages.

Functional languages like LISP and SCHEME [**?**] tend to have support for AOP virtually built-in [**?**], and indeed the idea of AOP, as one of the pioneers of AOP, Gregor Kiczales, mentions [**?**], 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[6] in the late 90's. further 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[7], 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.

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*.

2.3.1 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 a region 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 bindings pointcuts
  Those are used to expose context (objects) to the aspect for internal use.
- Kinded pointcuts
  They are the primitive pointcuts. They pick out join points of a certain kind (e.g. method calls, field accesses).
- Lexical pointcuts
  When conjoined with other pointcuts, those pointcuts can restrict the set of matched joinpoints by lexical scopes.

[6] Palo Alto Research Center
[7] http://www.eclipse.org/

- Control flow-based pointcuts
  Those restrict matching to joinpoints inside a certain control flow.

- Expression-based pointcuts
  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 depicting its kind, such as `call`, `execution` etc., and a set of brackets holding a body. This body can be of different kinds, depending on the kind of the pointcut:

- *TypePattern* This stands for a pattern over an arbitrary Java type signatures. Such patterns can contain the wildcart '`*`', 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 wildcarts 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 methods 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 arbitraty length.)

```
1  class Stack extends ArrayList {
2
3      final static int INITIALSIZE = 1;
4
5      static int initialSize() {
6          return INITIALSIZE;
7      }
8
9      Stack(int initialSize) {
10         super(initialSize>=0?initialSize:initialSize());
11     }
12
13     Object pop() {
14         return this.remove(this.size()-1);
15     }
16
17         ...
18 }
19
20 class Main {
21
22     public static void main(String[] args) {
23         Stack s = new Stack(-1);
24         try {
25             s.pop();
26         } catch(Exception ex) {
27             System.err.println(ex);
28         }
29     }
30 }
```

Table 2.5: Example implementation of a stack

The full list of available pointcuts in AspectJ is given in the appendix on page 92. 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 2.5.

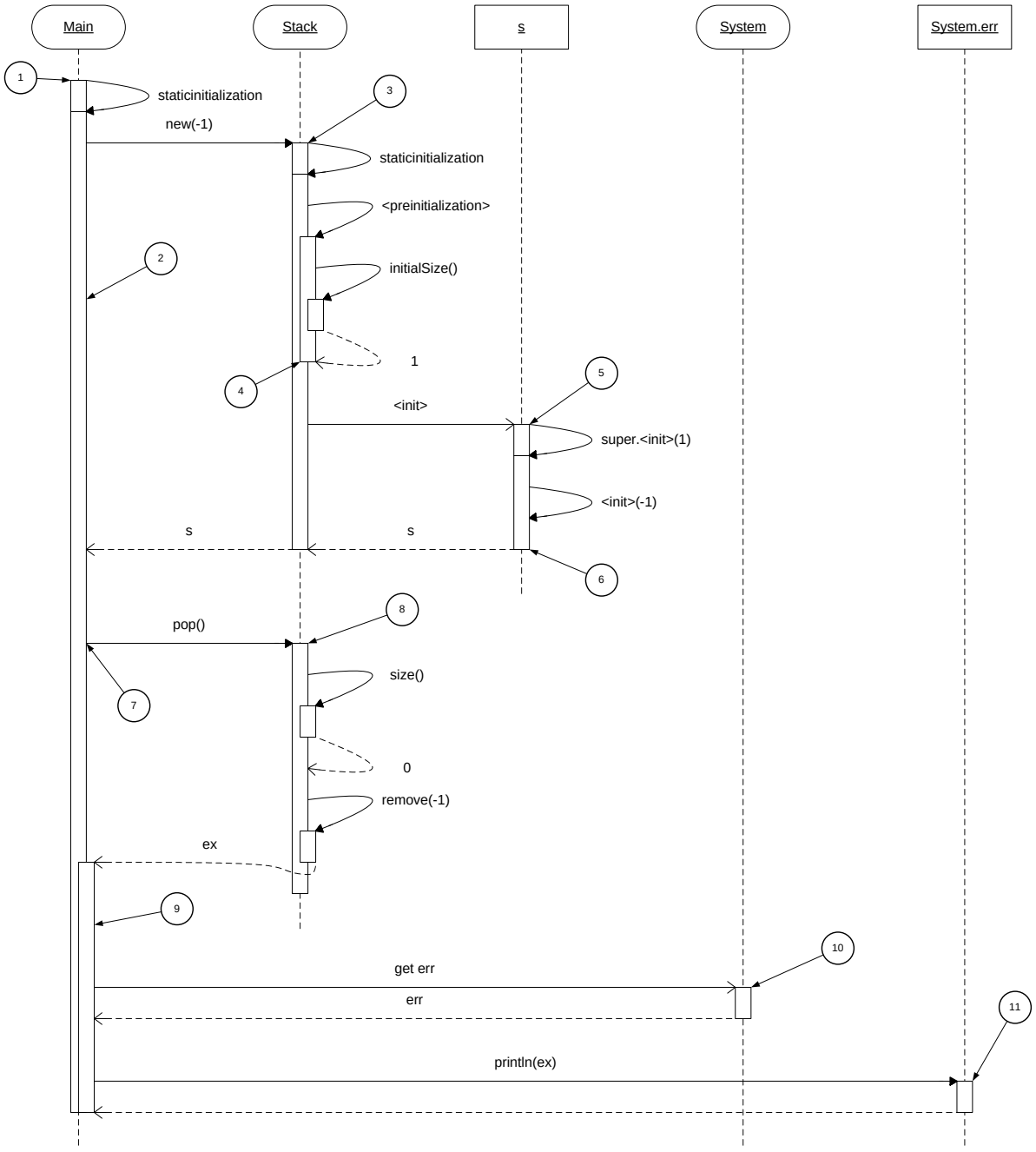
In line 23, the `main` method constructs a new Stack with an initial capacity of `-1`, which is immediately overriden by the value of `INITIALSIZE` inside the

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`.

Figure 2.3 shows a sequence diagram of the full execution of the `main` method. Classes (static context) are drawn as curved boxes while normal objects have sharp edges. The numbers are used to label points or regions in the control flow. The next paragraph describes them in detail.

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

- A pointcut `staticinitialization(*)` would match 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)` would bind `a` to the value `1`, while at region 6, it would be 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)` would bind `t` to the object `s`.
- Point number 7 is e.g. matched by `call(Object Stack.pop())`. Here `target(t)` would bind `t` to the Stack instance `s`. `this(m)` would not match, 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)` would bind `e` to the thrown exception `ex`.
- The pointcut `get(PrintStream *)` would match region 10.


Main
Stack
s
System
System.err
1
staticinitialization
3
new(-1)
staticinitialization
<preinitialization>
2
initialSize()
1
4
5
<init>
super.<init>(1)
<init>(-1)
s
s
6
8
pop()
size()
7
0
remove(-1)
ex
9
10
get err
err
11
println(ex)

Boolean combinations and invalid 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 would match all calls to methods `foo` on types which are an instance of `Foo`. The call target would be 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 would not be valid and an error would 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, an *advice* 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 actual 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.

```
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 2.6: Advanced exception handling by the means of an around 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 executed.

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 2.6 shows how a semantically 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)`, where `s` is bound to the call target.

In line 4, the advice invokes `proceed(..)`. This either calls the next advice matching the same joinpoint or the joinpoint itself if there is no further advice, which is the case 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, if an `ArrayIndexOutOfBoundsException` is thrown, this 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. The solution to this problem is a defined *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 25) as arguments.

The following statement shows a `declare precedence` statement that would give 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 [**?**].

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 [**?**] 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 only consume metadata: The pointcutlanguage 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 * *.*(..));
```

Futher details can be found in Laddad's DeveloperWorks article at [**?**]. This article was published within the series *AOP@Work* which is surely 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. Based on this background 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 [**?**] 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 φ_{now} and φ_{next}, 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 full 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 semantics we employ and proove 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 by *Dynamic LTL* (*DLTL*).

3.1 Syntax of *DLTL*

DLTL is a linear temporal logic over AspectJ pointcuts featuring dynamic bindings. AspectJ pointcuts are, as explained in section 2.3.1.2, predicates over joinpoints, those being regions in the dynamic control flow of a running application (cf. section 2.3.1.1).

Temporal logic usually does not reason about *regions*. 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).

Therefore, 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:

⟨proposition⟩ ⟶ `entry(` ⟨pointcut⟩ `)` |
`exit(` ⟨pointcut⟩ `)` |
`exit(` ⟨pointcut⟩ `) returning` ⟨identifier⟩ |
`exit(` ⟨pointcut⟩ `) throwing` ⟨identifier⟩

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*).

⟨argument⟩ ⟶ ⟨proposition⟩ | ⟨formula body⟩
⟨formula body⟩ ⟶ `F(` ⟨argument⟩ `)` | `G(` ⟨argument⟩ `)` |
`X(` ⟨argument⟩ `)` | `!(` ⟨argument⟩ `)` |
`(` ⟨argument⟩ `U` ⟨argument⟩ `)` | `(` ⟨argument⟩ `R` ⟨argument⟩ |
`(` ⟨argument⟩ `||` ⟨argument⟩ `)` | `(` ⟨argument⟩ `&&` ⟨argument⟩ |
`(` ⟨argument⟩ `->` ⟨argument⟩ `)` | `(` ⟨argument⟩ `<->` ⟨argument⟩

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:

⟨formula⟩ ⟶ [⟨formal parameter list⟩ :] ⟨formula body⟩

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

```
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*

Line 4 specifies a proposition which holds at the exit event of a constructor call. Further, it binds[1] 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, which certainly is a sensible assumption.

Having this first initial picture of what properties could 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 a 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 $p, q, \ldots$

[1] We will explain in the semantics section what this exactly means.

Such propositions may define valuations for free variables. For instance a proposition `exit(* call(A.foo()) && target(x))` *defines* a value for a variable `x`. We write $p(\vec{x}), q(\vec{y}), \ldots$ for propositions p and q defining variables $\vec{x} := \{x_1, \ldots, x_n\}$ respectively $\vec{y} := \{y_1, \ldots, y_m\}$.
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 $p(\vec{x}, \underline{\vec{x'}}), q(\vec{y}, \underline{\vec{y'}}), \ldots$ for propositions p and q defining variables $\vec{x} := \{x_1, \ldots, x_n\}$ and using $\underline{\vec{x'}} := \{\underline{x'_1}, \ldots, \underline{x'_n}\}$ respectively defining variables $\vec{y} := \{y_1, \ldots, y_n\}$ and using $\underline{\vec{y'}} := \{\underline{y'_1}, \ldots, \underline{y'_n}\}$.[2]
A *state* shall in our semantics be identified with the propositions that hold at this state. Hence, we define $S := 2^{\mathcal{P}}$.

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 $n > 0$. For each path position $\pi[i]$ $(0 \leq i < n)$ and proposition $p \in \{p_1, ..., p_m\}$ and formulae φ and ψ:

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

Here, still the following equations hold:

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

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

[2] It shall be noted that this disctinction between variable definitions and uses is only necessary for the declarative and operational semantics. Opposed to implementations as [**?**], *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.

Remark 3.2.1 (General assumption)
In the following, we want to assume that φ is in negational normal form. This enables us to restrict definitions to the above operators. It is straightforward to extend any of the definitions to further operators a **F** and **G** or the *weak until* operator **W**, which is defined as $\varphi \mathbf{W} \psi := (\varphi \mathbf{U} \psi) \vee \mathbf{G} \varphi$.

We leave this as an exercise to the reader.

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

For a proposition $p(\vec{x})$ with free variables $\vec{x}$, what does it mean to 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?

One 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 of 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 Java program has direct access to the heap [3] and hence it cannot enumerate the domain.

A second approach, which *is* feasible and indeed was taken in our Haskell prototype [**?**], 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 $p(x) \vee \mathbf{X}\ (q(y, \underline{x}))$ 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:

$$
\begin{array}{rcl}
\pi & \models & p(x) \vee \mathbf{X}(q(y, \underline{x})) \iff \\
\pi & \models & \bigwedge_{x' \in dom(x)} \bigwedge_{y' \in dom(y)} p(x) \vee \mathbf{X}(q(y, \underline{x})) \iff \\
\pi & \models & \bigwedge_{x' \in \emptyset} \bigwedge_{y' \in \{1\}} p(x) \vee \mathbf{X}(q(y, \underline{x})) \iff \\
\multicolumn{3}{l}{true}
\end{array}
$$

Notice that in the one and only state $q(1, \underline{x})$ holds. Actually it would be open to debate, if a proposition $q(y, \underline{x})$ matches $q(1, \underline{x})$ 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 behavior is natural because formula are modeled by lambda functions, which are lazily evaluated: 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.

[3] 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 [**?**]. 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 error detection.

The second problem was the reason for defining the semantics of *J-LO* in a very special, 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, φ_{now} and φ_{next}, with respect to a state, so that at this state, φ holds iff both φ_{now} and φ_{next} 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 is the notion of *time slices*. Each LTL formula can be thought of a partition of subformulae, each talking about either the current state or the rest of the path, starting with the next state.

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 $\pi[i]$ into two formulae $now(\varphi)$ and $next(\varphi)$ in such a way that $\pi^i \models \varphi$ iff $\pi^i \models now(\varphi)$ and $\pi^i \models next(\varphi)$.

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

Definition 3.2.2 (Function *now*)
The function $now : LTL \to LTL$ 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 and $\pi[i]$ always $now(p)$ is a Boolean combination of propositions. The function $now(\varphi)$ reflects that part of φ that can be fully evaluated in state $\pi[i]$, assuming that $next(\varphi)$ holds on the subsequent path.

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

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

Else $(i = n)$:

$$next'(\varphi)\pi[n] := \mathbf{false}$$

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 $\pi = \{p\}\{q\}$. Obviously it holds that $\pi \models \varphi$.

The calculation of *now* leads to:
$now(\varphi)$
$= now(p \mathbf{U} q)$
$= now(q \vee (p \wedge \mathbf{X}(p \mathbf{U} q))$
$= now(q) \vee now(p \wedge \mathbf{X}(p \mathbf{U} q))$
$= q \vee (now(p) \wedge now(\mathbf{X}(p \mathbf{U} q)))$
$= q \vee (p \wedge \mathbf{true})$
$= q \vee p$

The calculation of *next* leads to:
$next(\varphi)$
$= \mathbf{X}\ next'(\varphi)$
$= \ldots$
$= \mathbf{X}(\ \mathbf{false} \vee (next(p)\{p\} \wedge next(\mathbf{X}(p \mathbf{U} q))\{p\})\ )$
$= \mathbf{X}(\ \mathbf{false} \vee (\mathbf{true} \wedge (p \mathbf{U} q))\ )$
$= \mathbf{X}(\ p \mathbf{U} q\ )$
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:

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

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[4] 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 $now(phi)$ 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 $next(phi)$ 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

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

to the outermost level. Since **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.

■

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

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 **X** operator. Evaluation of the latter may be deferred at least until the next 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 introductry 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.

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 returns 1 if x and y are the same variables. (In all other cases the function is undefined.) For such a function φ we define the update operation $\varphi[x \mapsto 2]$ as $\varphi \setminus \{x \mapsto o \in \varphi\} \cup \{x \mapsto 2\}$. We also allow concurrent updates such as $\varphi[x \mapsto 2,\ y \mapsto 3]$ and so forth (for cases where x is not the same variable as y).

4. Some functions φ are defined over propositions, pointcuts or bindings. Sometimes we apply those functions to whole formulae ψ. In this context we mean that the function is applied to all propositions/pointcuts/bindings in $cl(\psi)$ 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 $\tilde{\varphi}$ shall denote the *appropriately* overloaded version of φ for its context.

3.3.1 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:

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

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

$\mathbf{args}_\iota \in 2^O \cup \{\bot\}$ the argument vector of a method call or execution at ι,

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

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

Any of this/target/args/ret/ex may be undefined (e.g. when executing in a static context, see figure 2.3). This is reflected by a value of $\bot$.

We denote the set of all joinpoints by JP.

Further we define the set of all objects provided by ι, O_ι as:
$O_\iota := (\ \{\mathbf{this}_\iota, \mathbf{target}_\iota, \mathbf{ret}_\iota, \mathbf{ex}_\iota\} \cup \mathbf{args}_\iota\ ) \setminus \{\varnothing\}$.

Definition 3.3.2 (Control flow)
Joinpoints can be cascaded at runtime: Since joinpoints are *regions* in the control flow, one joinpoint can occur within another.

Hence for a joinpoint $\iota \in JP$ we define its control flow 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.

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

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

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

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

This shall reflect the fact that **exit** is not only matched by **exit** but also by **exit returning** and **exit throwing**.

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 $S := JP \times (K \backslash \{\textbf{exit}\})$.

Definition 3.3.5 (Pointcut)
A *Poincut* (or *Crosscutt*, *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.[5]

We denote the set of all poincuts 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 poincut which does not match a joinpoint cannot expose any values at this joinpoint. Hence, for cases where $\mu_\chi(\iota) = false$, we write $\sigma_\chi(\iota) = \bot$. Further, the range of $\sigma_\chi(\iota)$ shall be restricted to values of O_ι^{cflow}.

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

Example 3.3.6 (Pointcut)
Assume the following AspectJ poincut 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)`,

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

$\sigma_\chi = \lambda\iota.\{$ `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 $B := \{\beta : \mathcal{V} \dashrightarrow (O \cup \{\oslash\})\}$.

A binding function β may define certain variables as *unbound*[6]. We denote the fact that a variable `x` is unbound by { `x` $\mapsto \oslash$ }.

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

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

[6] 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.

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. Finally 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 $\vec{v}_{\chi_p}$.

We denote the set of all propositions by $\mathcal{P}$.

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 if 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 $\mu'_\chi := (\tilde{\sigma_\mu} \circ \tilde{\beta})\ (\mu_\chi)$, the following conditions hold:

1. $\mu'_\chi(\iota_s) = \textbf{true}$,
2. $k \trianglelefteq k_s$ and

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. For most free variables this is no problem, they are also contained in the valuation function of the contained pointcut. `if` pointcuts, however, 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.

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` $\mapsto \oslash$, `o2` $\mapsto \oslash$}

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, one must make sure that the binding β_p provides a value $\neq \oslash$ 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.

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 an example.
In this example as well as all the following sections of this chapter we want to assume that the functions *next* and *now* 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.

3.3.2 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, 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. 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*. 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 **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 $\varphi(x) \wedge \mathbf{F}\ q(1) \wedge \mathbf{F}\ q(2)$. 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[7].

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 need disambiguation: What are "all possible valuations $\vec{x'}$ at $\pi[i]$" ?

This shall be clarified by the next subsection.

[7]Note that $\varphi(x) \wedge \mathbf{F}\ q(2)$ is a nonfinal state in an AFA, while $\varphi(x)$ is final.

3.3.3 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, $p(y,x)$, which is $p(2,1)$ under those bindings, should hold on the subsequent path, which 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 matches $p(x,y)$ if and only if it matches $p(y,x)$ 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 variables)
Let $\varphi \in DLTL$ with $\mathcal{P}_\varphi^{act} = \{p_1(\vec{x_1}), \ldots, p_n(\vec{x_n})\}$. Then $\mathcal{V}_\varphi^{act}$, the set of *active variables* of φ is defined as the set of all unbound variables within active propositions of φ:

$$\mathcal{V}_\varphi^{act} := \mathcal{V}_{\mathcal{P}_\varphi^{act}}^{act}$$

where

$$\mathcal{V}^{act}_{\{p_1(\vec{x_1}),\ldots,p_n(\vec{x_n})\}} := \bigcup_{1\leq i\leq n} \mathcal{V}^{act}_{p_i(\vec{x_i})}$$

where

$$\mathcal{V}^{act}_{p(x_1,\ldots,x_n)} := \{x \in \{x_1,\ldots,x_n\} \mid \beta_p(x) = \oslash\}$$

Definition 3.3.16 (Active propositions at a state)
$\mathcal{P}^{act}_{\varphi}(s) \subseteq \mathcal{P}$, the set of active propositions in φ matching s is defined as:

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

Definition 3.3.17 (Active Bindings)
Let $s = (\iota_s, k_s) \in S$ and $\varphi \in DLTL$.
Define $all_{\varphi}(s)$ as:

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

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

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

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 cases where φ' has the form $\mathbf{X}\varphi''$: Here we we only bind values in the case where $\varphi'' \neq \varphi$, because only in those cases we move "further down" in the alternating automata, we make a "real step" according to the partial order $\preceq$.

3.3.4 Redefinition of *next* for *DLTL*

Definition 3.3.18 (Function *next* for *DLTL*)
Let $\pi \in S^n$. Let $\varphi(\vec{x}) \in DLTL$.
For $\varphi' \in cl(\varphi)$, 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 $\vec{x''} \subseteq \vec{x'} \subseteq \vec{x}$ and $\varphi'(\vec{x'}) = \mathbf{X}\ \varphi''(\vec{x''})$ then:

$$next_{\varphi(\vec{x})}(\varphi'(\vec{x'})) = next_{\varphi(\vec{x})}(\mathbf{X}\ \varphi''(\vec{x''})) := \begin{cases} \varphi''(\vec{x}) & \text{if } \varphi'' = \varphi, \text{ (remain unbound)} \\ \varphi''(\vec{x''}) & \text{otherwise, (take bound version)} \end{cases}$$

Else:
$next_{\varphi(\vec{x})}(\varphi'(\vec{x'})) := next(\varphi'(\vec{x'}))$

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 will explain this function in detail and introduce a static analysis which decides if $valid(\varphi)$ holds for a given formula φ.

3.3.5 Declarative semantics of a *DLTL* formula

Definition 3.3.19 (Declarative semantics of a *DLTL* formula)
Let $\varphi(\vec{x}) \in DLTL$ with $valid(\varphi) = \textbf{true}$ and $\pi \in S^+$.

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

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 3.3.20 (Declarative semantics of a *DLTL* formula)
Let $\varphi(x, y) := \mathbf{G}(\ p(x) \rightarrow \mathbf{XF}\ q(y, \underline{x})\ )$ 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:

$$\begin{gathered}\pi \models \varphi(x, y) \\ \Longleftrightarrow \\ \forall \beta \in B_\varphi^{act}(\pi[0]) : \\ \pi \models now(\tilde{\beta}(\varphi(x, y))) \ \wedge \ \pi \models next_{\varphi(x,y)}(\tilde{\beta}(\varphi(x, y))) \\ \Longleftrightarrow \\ \pi \models now(\underbrace{(\lambda x.x \mapsto 1)\ (\varphi(x, y))}_{\varphi(1,y)}) \ \wedge \ \pi \models next_{\varphi(x,y)}(\underbrace{(\lambda x.x \mapsto 1)\ (\varphi(x, y))}_{\varphi(1,y)}) \\ \Longleftrightarrow \\ \pi \models \underbrace{now(\varphi(1, y))}_{\textbf{true}} \ \wedge \ \pi \models \underbrace{next_{\varphi(x,y)}(\varphi(1, y))}_{\mathbf{X}(\ \varphi(x,y)\ \wedge\ \mathbf{F}\ q(y,1)\ )} \\ \Longleftrightarrow\end{gathered}$$

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

$$\Longleftrightarrow$$

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

$$\Longleftrightarrow$$

$$\forall \beta \in B^{act}_{\varphi^1(x,y)}(\pi[1]) :$$

$$\pi^1 \models now(\tilde{\beta}(\varphi^1(x,y))) \wedge \pi^1 \models next_{\varphi^1(x,y)}(\tilde{\beta}(\varphi^1(x,y)))$$

$$\Longleftrightarrow$$

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

$$\Longleftrightarrow$$

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

$$\Longleftrightarrow$$

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

$$\Longleftrightarrow$$

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

$$\mathbf{true}$$

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[8]. 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.

[8]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 occuring definition. The latter definitions are automatically turned into uses of this variable.

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

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

This formula has an obvious splitting in the subformulae $now(\varphi(x, y)) = p(x)$, and $next_{\varphi(x,y)}(s)$, which depends on the 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 $next_{\varphi(x,y)}(\varphi(1, y))s = \mathbf{F}\ q(y, \underline{1})$ 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 ∧-operator.*

A case which is harder to solve is the following.

Example 3.4.2 (Static Analysis - propagation over "∨")

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

which is in NNF:

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

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 $next_{\varphi(x,y)}(\varphi(x, y))s$ 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 ∨-operator.*

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

Example 3.4.3 (Static Analysis - invalid formula)

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

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

Here the possible cases are:

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 do the follwing.

1. For each time slice defined by φ, produce a set $def(\varphi)$ of variables which are defined on this time slice.
2. For each time slice defined by φ check for each variable if this variable is defined on this or one of the previous (outer) time slices.

The set $def \subseteq 2^{\mathcal{V}}$ of defined variables is calculated according to the above observations:

Definition 3.4.4 (Function *def*)
Assume that $\varphi \in DLTL^{NNF}$ is in negation normal form (cf. section 2.1). Let $p \in \mathcal{P}$. Then we define $def : DLTL^{NNF} \to 2^{\mathcal{V}}$ as:

$$
\begin{array}{rcl}
def(\varphi) & := & def_+(\varphi) \cup def_-(\varphi) \\
\multicolumn{3}{c}{\text{where}} \\
def_+(p) & := & \vec{v}_{\chi_p} \\
def_+(\neg p) & := & \emptyset \\
def_+(\mathbf{X}\ \varphi) & := & \emptyset \\
def_+(\varphi \wedge \psi) & := & def_+(\varphi) \cup def_+(\psi) \\
def_+(\varphi \vee \psi) & := & def_+(\varphi) \cap def_+(\psi) \\
def_+(\varphi\ \mathbf{U}\ \psi) & := & def_+(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
 & = & def_+(\varphi) \cap def_+(\psi) \\
def_+(\varphi\ \mathbf{R}\ \psi) & := & def_+(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
 & = & def_+(\psi) \\
\multicolumn{3}{c}{\text{and}} \\
def_-(p) & := & \emptyset \\
def_-(\neg p) & := & \vec{v}_{\chi_p} \\
def_-(\mathbf{X}\ \varphi) & := & \emptyset \\
def_-(\varphi \wedge \psi) & := & def_-(\varphi) \cap def_-(\psi) \\
def_-(\varphi \vee \psi) & := & def_-(\varphi) \cup def_-(\psi) \\
def_-(\varphi\ \mathbf{U}\ \psi) & := & def_-(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
 & = & def_-(\psi) \\
def_-(\varphi\ \mathbf{R}\ \psi) & := & def_-(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
 & = & def_-(\varphi) \cap def_-(\psi)
\end{array}
$$

Here $def_+(\varphi)$ provides the variables which are bound by propositions contained in nonnegated form on the current time slice, while $def_-(\varphi)$ provides those for propositions which occur under negation.

Next, we define the logical counterpart, the function *use* which represents the used variables of the current timeslice.

Definition 3.4.5 (Function *use*)
Assume that $\varphi \in DLTL^{NNF}$ is in negation normal form. Let $p \in \mathcal{P}$. Let

$\odot_2 \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\}$. Then we define $use : DLTL^{NNF} \to 2^{\mathcal{V}}$ as:

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

It is remarkable that according to this definition for any $p \in Prop$ it holds that $def(p) = def(\neg p)$ and $use(p) = use(\neg p)$.

Using those definition sets it is now straightforward to define the function $valid(\varphi)$, which is **true** if and only if φ defines any free variable before it is used.

Definition 3.4.6 (Function *valid*)
Assume that $\varphi \in DLTL^{NNF}$ 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 $valid : DLTL^{NNF} \to \mathbb{B}$ as:

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

Example 3.4.7 (Static analysis - invalid formula (formally))
Assume again the following formula:

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

Then we have:

$$valid(p(x) \vee q(y, \underline{x})) = valid_{def(p(x) \vee q(y, \underline{x}))}(p(x) \vee q(y, \underline{x})) = valid_{\emptyset}(p(x) \vee q(y, \underline{x})) = use(p(x) \vee q(y, \underline{x})) \subseteq \emptyset \wedge valid_{\emptyset \cup def}$$

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

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

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

Let $\varphi \in DLTL^{NNF}$. Assume $valid(\varphi) = \mathbf{true}$. We distinguish the following cases:

$\varphi = p$ for some $p \in \mathcal{P}$. Since $valid(p) = \mathbf{true}$, we know that $use(p) := \{x \in \mathcal{V} \mid (x, o) \in \beta_p \wedge \neg x \in def(p)\} \subseteq \mathcal{D}$. Also we know that $\mathcal{D}$ holds only variables which are defined on this or previous temporal layers, because later temporal layers guarded (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.

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

$\varphi = \varphi' \odot_2 \psi'$ for some $\varphi', \psi' \in DLTL^{NNF}$. As above.

Completeness ($\Leftarrow$):

Let $\varphi \in DLTL^{NNF}$. 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 $\neg \forall x : ((x, o) \in \beta_p \wedge \neg x \in def(p)) \rightarrow x \in \mathcal{D}$. Hence, $use(p) \subseteq \mathcal{D}$ and so $valid(p) = \mathbf{true}$.

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

$\varphi = \varphi' \odot_2 \psi'$ for some $\varphi', \psi' \in DLTL^{NNF}$. 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 formula should be treated with a warning rather than a fast-failing error message. However, when bypassing the static analysis in that way, we cannot guarantee any more the soundness of the matching semantics for a proposition: 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 in undefined. When bypassing the analysis, this guarantee cannot longer be given: Hence, one would have no means of determining an invalid path/formula combination at runtime, which would give away soundness.

3.5 Operational semantics of *DLTL*

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

So the operational semantics are based on the following ideas.

1. We use a automaton based approach to propagate formulae over time. We employ alternating automata (cf. section 2.1.1.4) as 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, equivalent to this formula, with an evaluation engine. 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 state under those propositions.
3. In most cases, the evaluation of a matching function p_{μ_χ} 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 timeslices. 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.

For easier reading, we make some general assumptions which should hold throughout this chapter.

3.5.1 General assumptions

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 defintion 2.1.4. The construction can naturally be extended for formulae over *DLTL*.

3.5.1.2 Valid formulae

In this section we assume that for each given $\varphi \in DLTL$ it holds that $valid(\varphi) =$ **true**.

3.5.1.3 Valid `if` pointcuts

For each `if` 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 [9] and so provides a Boolean value as result. This allows us to interpret `expr()` as a function over bindings into $\mathbb{B}$. Please note that 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, that this prevents o from being garbage collected [10] . In particular, o is *available* for subsequent matching and evaluation of `if` pointcuts.

3.5.1.5 No side effects

We assume that for a given specification $\Phi \subseteq DLTL$ it holds that the evaluation of any $\varphi_1 \in \Phi$ 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.

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 (Propositions of a formula)
Let $\varphi \in DLTL$. We denote the set of propositions in φ with $\mathcal{P}_\varphi := cl(\varphi) \cap \mathcal{P}$.

Definition 3.5.2 (Constraints)
For each $\mathcal{V}' \subseteq \mathcal{V}$, we define the set $C_{\mathcal{V}'}$ of all *constraints* over labels from $\mathcal{V}'$ as:

[9] *J-LO* does properly handle those cases as we explain in chapter 4.
[10] This issue is also gracefully handled by *J-LO* and explained, as well, in chapter 4.

$$C_{\mathcal{V}'} = B|_{\mathcal{V}'} \to \mathbb{B}.$$

Those constraints are used to represent `if` pointcuts in the right context.

Definition 3.5.3 (Operational proposition)
Let $p \in \mathcal{P}$ as $\mathcal{P}$ was defined in definition 3.3.8 with $p = (l_p, \chi_p, k_p, \beta_p) \in \mathcal{V} \times PC \times K \times B$.

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

$\chi'_p := (\mu'_{\chi_p}, \vec{v}_{\chi_p}, \sigma_{\chi_p})$

with:

$\mu'_{\chi_p} := \mu_{\chi_p}$ where all `if` pointcuts in μ_{χ_p} have been replaced by `if(true)`

and:

$\vec{\gamma_p} := \{expr \mid$ `if(`$expr$`)` is an `if` pointcut in $\mu_{\chi_p}\}$

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

Example 3.5.4 (Operational proposition)
As in example 3.3.10 we assume the following proposition 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:

- μ'_{χ_p} = `call(Object Stack.pop()) && if(true)`
- $\vec{\gamma_p} = \{\lambda(\ \{$`o1`$\mapsto o_1,\ $`o2`$\mapsto o_2\}\ ) \,.\, o_1 \neq o_2\}$
- l_p, k_p, β_p as before

Definition 3.5.5 (Matching of operational propositions)
For a state $s = (\iota_s, k_s) \in S$ and operational proposition $\hat{p} \in \hat{\mathcal{P}}$ with $\hat{p} = (l_p, \chi_p, k_p, \beta_p, \vec{\gamma_p})$ we say that $\hat{p}$ *matches* s or *holds* in s, $s \models \hat{p}$ for short, if the following holds:

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

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

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 2.1.4).

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}. However, it is quite easy to overload δ in such a way that it is defined for whole clause sets over 2^{2^Q} as well. We are going to do so in section 3.5.2.1.

We call auch a clause set a *configuration* of $\mathcal{A}_\varphi$. That 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:

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

We define the language of $\mathcal{A}_\varphi$ as:
$\mathcal{L}(\mathcal{A}_\varphi) := \{\pi \in S^+ \mid$ after reading π, $\mathcal{A}_\varphi$ is in an accepting configuration $\}$

Remark 3.5.8 (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 sucessor states are generated for all possible valuations at $\pi[i]$, as it was described in section 3.3.13.

Example 3.5.9 (Operational semantics by example)
Take for example the following formula

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

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

The alternating automaton would start in configuration $q_0 := \{\{\varphi(x,y)\}\}$. At the first state $\pi[0] = \{p(1), p(2)\}$ we would get possible valuations of $x \in \{1,2\}$.
It holds that $\pi[0] \models now(\varphi)$. For valuations $x \in \{1,2\}$ we get $next_{\varphi(x,y)}(\varphi(1,y))\{p(1),p(2)\} = \varphi(x,y) \wedge \mathbf{F}\ q(y)_{y \neq 1}$ respectively $next_{\varphi(x,y)}(\varphi(2,y))\{p(1),p(2)\} = \varphi(x,y) \wedge \mathbf{F}\ q(y)_{y \neq 2}$.
In the terms of alternating automata operating on clause sets, this yields a sucessor 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 now 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, $\pi[1] \not\models_{y=2} q(y)_{y\neq 2}$. In the case of the subformula $\mathbf{F}\ q(y)_{y\neq 1}$ it holds that $\pi[1] \models_{y=2} q(y)_{y\neq 1}$ and so $\mathbf{F}\ q(y)_{y\neq 1}$ evaluates to $\mathbf{tt}$, which is $\{\{\}\}$ 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 gain:

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

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 3.5.10 (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 and leads to what we call the *subset reduction*.

Definition 3.5.11 (Subset reduction)
Let $\varphi \in DLTL$, $\varphi' := NNF(\hat{\varphi})$ and $q = \{c_1, \ldots, c_n\} \in 2^{2^{cl(\varphi')}}$. Then we define the *subset reduction* of q, $ssr(q)$ as:

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

Theorem 3.5.12 (Correctness of subset reduction)
Let $\varphi \in DLTL$, $\varphi' := NNF(\hat{\varphi})$ and $q \in 2^{2^{cl(\varphi')}}$. Let $s \in S$. Let $\mathcal{A}_{\varphi,q}$ the copy of $\mathcal{A}_\varphi$ where q_0 has been replaced by q. Then it holds that:

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

The easy proof is left as an exercise to the interested reader.

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: $\mathcal{A}_\varphi := \mathcal{A}_{\varphi'} = (Q, \Sigma, q_0, \delta, F)$, where we define $\delta : Q \times \Sigma \to Q$ by:

$$\delta(q,s) := \delta(q, \underbrace{\mathcal{P}^{act}_{\varphi}(s)}_{=:\mathcal{P}'}, \underbrace{B^{act}_{\varphi}(s)}_{=:B'}) \tag{3.1}$$

$$\text{where} \quad \delta(\{c_1,\ldots,c_n\},\mathcal{P}',B') := \bigcup_{1\le i\le n} \delta(\{c_{i_1},\ldots,c_{i_{n_i}}\},\mathcal{P}',B'), \tag{3.2}$$

$$\text{where} \quad \delta(\{\varphi_1,\ldots,\varphi_m\},\mathcal{P}',B') := \bigotimes_{1\le i\le m} \delta(\varphi_i,\mathcal{P}',B'), \tag{3.3}$$

$$\text{where} \quad \delta(\varphi,\mathcal{P}',B') := \bigotimes_{\beta\in B'} \delta(\varphi,\mathcal{P}',\beta), \tag{3.4}$$

$$\text{where} \quad \delta(\varphi,\mathcal{P}',\beta) := \delta(\varphi,\mathcal{P}',\beta,def(\varphi)). \tag{3.5}$$

Here equation 3.1 reduces δ of a configuration and a state to δ of a configuration, the active propositions holding at this state and the set of active 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 recursion decent, initializing a context $\mathcal{D}$ to set set of variables defined by φ, $def(\varphi)$. The recursion then continues as follows.

Definition 3.5.13 (Filtered bindings)
For any set of defined variables $\mathcal{D} \subset \mathcal{V}$ we define the binding filtered for defined variables as: $\beta|_{\mathcal{D}} := \{x \mapsto o \in \beta \mid x \in \mathcal{D}\}$

Remark 3.5.14 (Specialization of bindings)
The operational semantics include a *specialization* step, where *unbound* bindings are replaced by bindings to objects. Hence for any $\beta \in \mathcal{V} \to O$, we overload β with $\beta : \mathcal{P} \to \mathcal{P}$ such that for each $p = (l_p, \chi_p, k_p, \beta_p) \in \mathcal{P}$:

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

where

$$\begin{aligned}\beta'_p := &\{\{x \mapsto o\} \mid o \neq \oslash \wedge \{x \mapsto o\} \in \beta_p\} \\ \cup &\{\{x \mapsto o\} \mid \{x \mapsto \oslash\} \in \beta_p \wedge \{x \mapsto o\} \in \beta\}\end{aligned}$$

For whole sets of propositions $\mathcal{P}' \subseteq \mathcal{P}$ we define respectively $\beta : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ as:
$\beta(\mathcal{P}') := \{\beta(p) \mid p \in \mathcal{P}'\}$

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

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

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

3.5.3 Operational semantics of a *DLTL* formula

Definition 3.5.15 (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:

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

It is certainly no suprise that in the following we want to proof that the declarative and operational semantics coincide.

Theorem 3.5.16 (Equivalence of declarative and operational semantics)
Let $\varphi \in \mathit{DLTL}$, $\pi \in S^+$. Then:

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

The proof of this theorem is somewhat long so we want to conduct it in the following subsection.

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 [**?**] is correct in general. We do not want to give a full formal proof here, since it was already conducted by Vardi [**?**] 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 proof 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. The idea is a simple induction. Based on the definition of δ it is obvious that the sucessor configuration of a configuration c equivalent to a formula φ holds exactly those states, which represent $next(\varphi)$. 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 makes sense, given 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 $p = (l, \chi, k, \beta) \in \mathcal{P}$. We need to show that:

$$s \models p \iff \delta(p, \mathcal{P}(s), \beta) = \{\{\}\}$$

Completeness ($\Rightarrow$):

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

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

for $\mu'_\chi := (\sigma_\mu \circ \beta)\ (\mu_\chi)$.
On the operational side it holds that:

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

So we need to show that:

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

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 $p' \in \mathcal{P}'$ such that $\mu_{\chi'_{\hat{p'}}}(\iota_s) = true$. 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 $\beta|_\mathcal{D}(p) \ \in \ \beta|_\mathcal{D}(\mathcal{P}')$ holds as well.

Soundness ($\Leftarrow$):

The argument here is actually the very same. Since $\delta(p, \mathcal{P}(s), \beta) = \{\{\}\}$ 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

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

What we need to show is that in this case, $\delta(\varphi, \pi[0], \beta) = \emptyset$. Since $\pi[0] \not\models now(\varphi(\beta(\vec{x})))$ holds, we know that even under the assumption that the subsequent path fulfills all obligation guarded by the **X** operator, the path violates φ. Hence, it suffices to show that $\delta(\varphi_{now}, \pi[0], \beta) = \emptyset$, where φ_{now} is the copy of φ where all subformulae guarded by **X** operators have been replaced by **tt**. We want to prrof the claim indirectly:

Assume that $\delta(\{\{\varphi_{now}\}\}, \pi[0], \beta) = \{c_1, \ldots, c_n\}$ 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, the calculation of $\delta(\{\{\varphi_{now}\}\}, \pi[0], \beta)$ is fully reduced to δ of Boolean combinations of **ff** and **tt** (resp. its equivalent $\{\{\}\}$). 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 assumtion, it has to hold that φ_{now} consists of a Boolean combination such that if each join term the evaluation of at least one of the two branches results in $\{\{\}\}$ and for each $\otimes$ term, the evaluation of both branches results in $\{\{\}\}$. If this was 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 wee need to show that if $\delta(\varphi, \pi[0], \beta) = c$ it holds that $\delta^*(c, \pi^1) \in F \iff \pi^1 \models \varphi$.

When looking at δ, one can easily see that the only case where formulae are "produced" for the sucessor 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. Hence it should be clear that c is equivalent to $next_{\varphi(\vec{x})}(\varphi(\beta(\vec{x})))\pi[0]$. Since the subset reduction is sound and complete, it follows that $\delta^*(c, \pi^1) \in F \iff \pi^1 \models \varphi$.

■

Chapter 4

Implementation

The implementation follows almost completely the operational semantics. First 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*). This compiler is introduced in the first section of this chapter. Further details about how we extended *abc* in order to accomplish this code generation can be found in our seminar paper [**?**]. 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 set them into relation to the operational semantics. Section 4.4 explains the treatment of special runtime behaviour such as exceptions, garbage collection and application shutdown.

4.1 Annotation extraction

On May 15th, 2005, a development version of *Soot* was published, which supports extraction of annotations on the bytecode level. Soot is a bytecode analysis framework and integral part of the *abc* compiler (see section 4.2.3). At the beginning of 2005, when we started development of *J-LO* however, there was no such support available yet. Hence we had to employ other tool support to accomplish this task.

We used the bytecode engineering toolkit *BAT2* [**?**] which is being developed at the Darmstadt Software Technology Group and is an offspring of the *Magellan* [**?**] 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] of a formula 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.

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* [**?**] compiler (the original implementation by PARC) but also into others like JAsCo [**?**], AspectWerkz [**?**] or in the form of hand coded preprocessors. The AspectBench Compiler (*abc*) which was developed by the McGill and Oxford universities now facilitates such extensions by providing an extensible, optimizing compiler for the AspectJ programming language. This will enable researchers henceforth to implement and/or port such extensions into one common framework and so reuse their implementations at once, as we described in [**?**].

Now first we give a brief overview of the structure of *abc*.

4.2.1 Structure of *abc*

The major Java based compilers for AOP languages that are around today, are all 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: As compiler frontend the Polyglot [**?**] compiler toolkit is used. Polyglot is a compiler framework built as 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. Also, Polyglot uses object association in favor over class inheritance employing

[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.

a sophisticated delegation model. This allows extenders to add or replace functionality piece by piece to distinct node types of the abstract syntax tree which do not need to share common super types.

As the weaving backend, the bytecode analysis and optimization framework Soot is being 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, that 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 builtin 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.

4.2.2 Polyglot

Polyglot as the *abc* compiler frontend, facilitates easy extendability in several dimensions. This is an enormous benefit over earlier approaches in compiler technologies, which usually only allowed extendability by the means of class inheritance, which 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. have suggested to use object composition in favor over class inheritance, because it tends to lead to more flexible system designs (see [**?**], pp. 18-20). 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 4.1: Polyglot delegation model (with call back to original receiver)

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

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 doing a source to source transformation, we

are not going to extend those facilities. We only make use of them implicitly through the final transformation processes to Java bytecode.

4.2.3 Soot

Soot is a bytecode analysis and optimization framework, which provides common templates for inter- and intraprocedural analyses. Several such analyses are already builtin. They comprise even complex *points-to* and *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 [**?**].

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[2] representation of Java bytecode. In Jimple, all implicit method invocations (e.g. String concatenation) and implicit references to the current 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.

Table 4.1 gives an example of this representation. Lines 1-7 define a class in normal Java syntax while lines 9-30 show 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.

This shall conclude our brief overview of the *abc* framework. Further details about how we extend it to achive the desired code generation can be found in [**?**]. 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 one single aspect. Each such aspect defines implicitly a *singleton* object (see [**?**] for a description of the

[2] executing object, arguments and result

```
1  public class Foo {
2      int a;
3
4      public int f(int x,int y , int z) {
5          return a+x*y+z;
6      }
7  }
8
9  public class Foo extends java.lang.Object {
10
11     int a;
12
13     public int f(int, int, int) {
14
15         Foo this;
16         int x, y, z, $i0, $i1, $i2, $i3;
17
18         this := @this: Foo;
19         x := @parameter0: int;
20         y := @parameter1: int;
21         z := @parameter2: int;
22         $i0 = this.<Foo: int a>;
23         $i1 = x * y;
24         $i2 = $i0 + $i1;
25         $i3 = $i2 + z;
26         return $i3;
27     }
28
29     //Implicit constructor omitted
30 }
```

Table 4.1: Java class and corresponding Jimple code

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 singlethreaded applications. For the case of multithreaded 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.
- `boundFormals` - An array holding all names of variable which are bound by this proposition.
- `ifClosures` - An array of *if-closures* (constraints) that have to be fulfilled by this proposition.

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 4.2.

```
public interface IIfClosure {

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

    public String[] variableNames();

    public String toString();

}
```

Table 4.2: Interface for if-closures

The method `satisfiedUnderBindings` returns for a given binding if the expression represented by thid closure is satisfied under the given bindings. The parameter `currentBinding` represents the function β, which maps variable names (type `String`) to objects (tyoe `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 4.3.

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 [3].

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

Each such 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 3.5.9 would induce the representation shown in table 4.4.

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 timem, a piece of

[3] 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 4.3: Example if-closure

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 hencenot be verified. Therefore we generate an additional empty advice in each such aspect, which just bootstraps the aspect at startup[4] :

[4]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

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

Table 4.4: Example formula instantiation

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

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} of a pointcut χ_p of a proposition p matches a joinpoint ι, an appropriate proposition is instantiated and added to `currentProps`. In the case where mutiple such propositions match the same joinpoint, all those propositions are added in the same way. Here it is importand 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 lexicographical order. 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 4.5 shows the advice that are generated for the following formula (specifying the semantics of the *Singleton* design pattern).

```
Singleton s, Singleton t:
G(
    (
        exit( call(Singleton+.new(..)) ) returning s
    ) -> (
        X(
```

is not actually made use of in any Java application.

```
        !(
          F(
              exit( call(Singleton+.new(..)) && if(s!=t))
              returning t
          )
        )
      )
    )
)
```

Lines 1-7 of the excerpt in table 4.5 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 control flow of joinpoints within the Runtime Verification package. This assures that the instrumented application is oblivious to the instrumentation as much as possible. 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). This map represents 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 `if(s!=t)`. As explained in the operational semantics, this leads on the one hand to an if-closure generated on `prop1` (c.f. table 4.3) and on the other hand to a substition 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 4.5 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

```
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 4.5: Generated pieces of advice

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 a lot of Java applications tend to be multithreaded and hence this was an issue we needed to address.

All functions implementing the transition relation δ are performing nondestructive updates only stateless. 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 lock order reversal example (see section **??**), 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 multithreaded 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 formulae.

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 again. *J-LO* follows this implementation.

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 lenght. 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 4.6.

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.

```
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 4.6: Shutdown hook aspect in *J-LO*

When this happens, this causes the constructor defined by the lines 8-16 to execute. The costructor then installs a *Shutdown Hook*[5], a nonactive thread, with the virtual machine. When shutting down, all installed shutdown hooks are concurrently executed. 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 really dies (e.g. when invoking `kill -9` under Linux).

4.4.2 Behaviour under presence of garbage collection

An important feature of *managed code* environments such as the Java Runtime Environment is *garbage collection*. Garbage collection (GC) assures that no

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

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 [**?**] 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 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 which references an object which is unreachable from the rest of the program, this proposition 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[6]. 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 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 Observering 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 4.7.

[6] Note that all objects which are on the current call stack *are* reachable per definition.

```
1  public interface Listener {
2
3      public void notifyRegistered(
4              String formulaId,
5              Thread associatedThread,
6              Configuration initialConfig
7      );
8
9      public void notifyUpdate(
10             String formulaId,
11             Thread associatedThread,
12             Configuration newConfig
13     );
14
15     public void notifyTearDown(
16             String formulaId,
17             Configuration config
18     );
19
20     public void notifyOnUserCauseException(
21             String formulaId,
22             Thread associatedThread,
23             String ifExpression,
24             Throwable exception,
25             Configuration config
26     );
27
28 }
```

Table 4.7: Listener interface in *J-LO*

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. 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 (non)final it can easily be determined whether the associated formula is satisfied or falsified on the observed path.

Important: 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, however 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 cause by an invalid expression. (A typical case would be the `if` pointcut `if(1/0<2)`.) 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[7], 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.

[7] This might be important to debugging because the expression could access the thread.

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 nor 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 straightforward to implement and are small. We have have used the Eclipse metrics plugin available at [**?**] 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[1]).

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 being anyway.

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

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

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

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

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 algorithsm. 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*.

For each formula, this formulae 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 runing time of the application. The cost of the evalation of installed formulae then heavily depends on two 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. [**?**]) 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 **??**.

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 **??**), this of course means that the moe 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.

5.3 Performance of the implementation

We did some performance measurements using the commercial tool JProbe [**?**]. 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 intrisic complexity of the algorithms. Specifically, the implementation uses hash sets over hast sets of formulae. So 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, for calculating 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. Though it seems to be a good idea to employ a different implementation technique in future versions.

In addition to profiling, we tested 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 prooved 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 are 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 **??** and our publication [**?**]) should help to mitigate this overhead to the feasible minimum.

TODO: LOR

Chapter 6

Related Work

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

6.1 Design by contract

Runtime Verification can be seen as an extemsion of the well-known *Design By Contract* (DBC) principle, which became popular by the work of Bertrand Meyer [**?**] and his reference implementation in the programming language 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 [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 [**?**] based SPARK [4], which aimes at high-integrity software development.

Several DBC implementation for Java exists [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. They all work with conditions specified in the source code somehow. The one that comes closest to the technology of *J-LO* is Contract4J [**?**]. Similar to *J-LO*

[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

Contract4J is also based on Java 5 annotations and the generation of AspectJ code. However, Contract4J uses the Java Annotation Processing Tool (APT) [**?**], which comes with the JDK [**?**]. This allows annotation extraction from source code only. Contract4J uses APT to produce an XML structure holding all annotation information. Then it uses XSLT [**?**] 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 that *J-LO* does not employ XSLT transformations but rather a real compiler for code transformations, which provides for type safety checks 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 as we will see.

Though 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: All, pre and postconditions as well as invariants only reason about each single method invocation. There is no way of specifying temporal interdependencies such as which can be expressed in LTL. On the other hand, any of this condition can easily be expressed with LTL. Hence, Runtime Verification provides a superset of expressiveness compared to DBC.

6.1.0.1 JML

6.2 Runtime Verification

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

6.2.0.2 Java PathFinder and Java PathExplorer

The probably best known project is the Java PathExplorer (JPaX) [**?**], a successor of Java PathFinder (JPF) [**?**]. The latter one is now open source and can be accessed on the web [**?**], while the former remains under closed source development at NASA AMES [**?**].

The system consists of a virtual machine (VM) which is written in pure Java and thus runs inside the usual Java VM. This double interpretation naturally leads to a rather large runtime overhead compared to other approaches, however provides for powerful and flexible instrumentation. JPF used to support LTL input, as described in [**?**].

6.2.0.3 ptrace

6.2.0.4 Temporal rover

6.2.0.5 Orchids

http://www.lsv.ens-cachan.fr/orchids/

Chapter 7

Conclusion

7.1 Future work

7.2 Pitfalls we came across

7.3 Own related publications

Chapter 8

APPENDIX

8.1 AspectJ pointcuts

The pointcut language of AspectJ version 1.2 comprises the following kinds of pointcuts. Informal definitions of id, type, method and constructor patterns are given on page 25.

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 region 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 *MethodPattern*/*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 *ConstructorPattern* 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.

Lexcial 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

Lexcial scoping over methods/constructors

Syntax `withincode(` *MethodPattern* `)`, `withincode(` *ConstructorPattern* `)`

Semantics matches code in the lexical scope of a method/constrcutor 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 occuring 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 occuring 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

LOR-Formel

$$
\begin{array}{l}
\mathbf{G}(\\
\quad \neg lock(i,x)\mathbf{U}(\\
\qquad lock(i,x) \wedge \mathbf{X}(\\
\qquad\quad \neg unlock(i,x)\mathbf{U}lock(i,y)_{x \neq y} \\
\qquad\qquad \rightarrow \\
\qquad\quad \mathbf{G}\neg(\\
\qquad\qquad \neg lock(j,y)\mathbf{U}(\\
\qquad\qquad\quad lock(i,y)_{i \neq j} \wedge \mathbf{X}(\\
\qquad\qquad\qquad \neg unlock(j,y)\mathbf{U}lock(j,x)_{i \neq j} \\
\qquad\quad) \\
\qquad) \\
\quad) \\
)
\end{array}
$$