TKK Reports in Information and Computer Science

Espoo 2008 TKK-ICS-R4

INTERFACE SPECIFICATION METHODS FOR SOFTWARE COMPONENTS

Jani Lampinen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

TKK Reports in Information and Computer Science
Espoo 2008 TKK-ICS-R4

INTERFACE SPECIFICATION METHODS FOR SOFTWARE COMPONENTS

Jani Lampinen

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi

ISBN 978-951-22-9453-4 (Print)
ISBN 978-951-22-9454-1 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://www.otalib.fi/tkk/edoc/

TKK ICS
Espoo 2008

Abstract: This report presents an interface specification language developed as a part of the LIME-project (LightweIght formal Methods for distributed component-based Embedded systems) and a tool implementation to support it. The intention is to provide a methodology that is lightweight and complementary to the existing means of quality assurance in a software process.

The specification language provides a mechanism for specifying both external usage of a software component, as well as the internal behavior of a one. The former is referred to as *interface specification*, and the latter to as *library specification*. Should the interface specification be breached between two interacting components, the calling component is incorrect. Likewise, if the called component does not obey library specification, it will be the one to take the blame. Both types of specification can be written using either propositional linear temporal logic (PLTL) or by regular expressions, and may contain claims about the component's state or the currently executing method.

Java has been used as the implementation language on the approach because of the preexisting metadata mechanism (Java annotations) and good tool support. The tool implementation relies on *aspect-oriented programming* developed by Gregor Kiczales and his team at Xerox PARC in the late 90s. It employs annotation-guided generation of temporal safety aspects to synthesize the defined properties as behavioral invariants to the runtime execution of the program. The aspects simulate finite state automata which keep track of the state of the interaction and signal an exception in case of an error in it is observed.

Keywords: Lightweight methods, Interface specification, Java, PLTL, Aspect-oriented programming

CONTENTS

1 INTRODUCTION

As software has become an irreplaceable part of our daily lives, the quality and correctness of it is crucial to our society. Not only do we literally trust systems such as cars, nuclear power plants, health care appliances and airplanes with our lives, but we also put a huge financial trust to our banking systems and data-communication satellites. Of course a failure in a software system does not always have catastrophic consequences but it may lose customers or tarnish the public image of its producer or of the company which operates it.

The traditional approach to quality assurance and assessment in software processes has been testing. Although testing is applied with a planned and disciplined manner in many of the software organizations, it is still somewhat ad-hoc by nature. Formal methods such as model checking and model-based testing (see, e.g., [11, 18]) have been suggested as methodologies for achieving better quality in software. These methods, however, impose considerable requirements for the software process in which they are applied, and for the expertise of the people applying them. They are also typically based on a complete model of the system which might be difficult produce, especially for legacy systems that have already been implemented.

The lightweight formal method presented here is based on formal interface specifications and monitoring them during runtime execution. This approach has been taken in order to gain benefits of both full-blown formal methods and traditional testing while avoiding some of the pitfalls [5]. Runtime monitoring can only, however, detect errors in execution but can not provide correctness guarantees like model checking does.

One thing that combines the various formal specification languages and their corresponding tool implementations is that they are in some way based on mathematical formalism. The degree of complexity varies greatly from simple propositional assertions to Turing equivalent languages (see, e.g., [29]) with more than enough expressive power for any specification. In the lower end of the spectrum when it comes to expressive power are the stateless *design by contract (DBC)* languages based on *the Hoare triple* [17]. One of the modern implementations of this discipline is *Java Modeling Language (JML)*, see, e.g., [4]. The IEEE standard *Property Specification Language (PSL)* [19] is one of the more expressive languages with ω-regular expressiveness [6] designed for specifying properties of hardware systems. Graphical representations of formalisms, such as *Live Sequence Charts (LSC)* and *UML state machines* have also been used in specifications (see, e.g., [23, 32][8, 36]).

The understanding of how the underlying mathematics of specification languages work is not the whole story – it is at least equally important to understand what to specify. In [9, 10] Dwyer et al. present a pattern system for finite state specifications analogous to that presented by Gamma et al. for object-oriented design in [13]. Similar practical guidance for specifying properties of concurrent systems have been given by Manna and Pnueli in [30]. Specifications are artifacts that can also themselves be verified, *Requirements Analysis Tool (RAT)* has been presented for analysis of functional requirements (specifications) in [34].

There are many runtime verification frameworks implemented for Java programming language which are of particular interest here. For example, *Java PathExplorer (JPAX)* can be used to analyze programs for concurrency errors and for monitoring user provided specification with either past time (ptLTL) or future time linear temporal logic (LTL) [16]. In JPAX the observer instrumentation is done into byte code [16]. The *Java Logic Observer (JLO)* uses AspectJ [20] aspects to implement observation of formulas specified with LTL over join points [38]. Kiviluoma et al. present a CASE tool that generates AspectJ aspects that simulate state automatons from behavioral requirements given in UML Sequence Diagrams [22]. Similar approach has been taken in [23, 32] but in these the specifications are given as more expressive Live Sequence Charts.

Component-based design and verification of distributed embedded systems is a very hard task to accomplish with traditional development methods. The working hypothesis in the LIME project and in this report is that the required methodology should be more rigorous than the traditional approaches which leads to the concept of lightweight formal methods. In the presented approach, the focus is on extending the interface specifications methods of components.

In traditional strongly typed programming languages the interpretation for correct interaction of two components is limited to the agreement on number, order and type of the parameters between the caller and the called component [7]. This correctness requirement in the interaction can be extended to the cover stateful protocol behavior related to it. The called component, e.g., a library, may require a certain order for the function calls through its interface or make requirements for not only types but also values of its parameters. Similarly, there may be requirements for the component to fulfill as well, for example, some explicitly stated relation between received arguments and returned values which the caller can rely on.

It is important to detect faulty interaction between components, but it is equally necessary to point out which one of the components is to blame for it. The basis for the model of interaction is presented in Fig. 1.1. In this setting there is an application that is using a library through an interface. The communication is always initiated by the application component. The model of communication is divided into two parts – to an *interface specification* (IF in Fig. 1.1) which specifies how a component should be used, and to a *library specification* (LS in Fig. 1.1) which specifies how the component should respond. Should the interface specification be breached, the calling component is incorrect, and if the library does not obey its specification, it will be the one to take the blame.

The specification language combines two complementary ways for expressing the proper behavior of software objects – regular expressions and propositional linear temporal logic (PLTL, see, e.g., [3]). The properties that are desirable to be described here include but are not limited to correct orderings of function calls and the relation between arguments and return values of functions. The former is understood to depict the

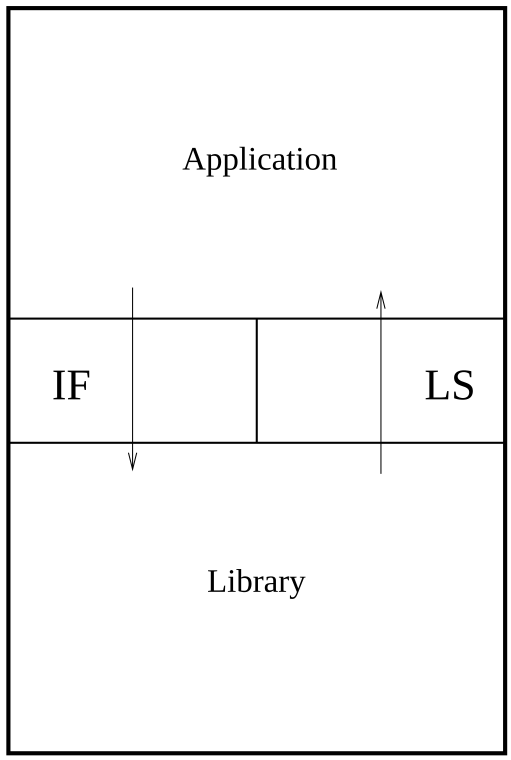

Figure 1.1: The interaction model

protocol aspect (call ordering) of interfaces, where as the latter describes how the library should behave.

The interpretation of PLTL [3] for runtime monitoring can be seen as a continuation of the work in [14] and [15]. The language employed is extended in this work to also contain future time operators using the SCheck tool [28].

This report presents an interface specification language developed as part of the LIME project (LightweIght formal Methods for distributed component-based Embedded systems), and a tool implementation to support its runtime monitoring. Automatic test generation is also a part of the LIME project but it is not considered here. This report is structured as follows. Chapter 2 describes the theoretical foundation for the specification language. Chapter 3 introduces the specification language itself. Chapter 4 explains shortly aspect-oriented programming which is used to implement the runtime monitoring. Chapter 5 describes the programming interface for using the specification language and the tool implementation to support it. Chapter 6 contains experimental runtime observers generate with the tool. Finally, Chap. 7 concludes the report and discusses the future of the specification language.

2 THEORETICAL BACKGROUND

The purpose of this chapter is to present the theoretical background needed for understanding the rest of this work.

As the protocol behavior of a software interface or component must be unambiguously specified, well-studied formalisms are used to express it. This forms a rigorous theoretical foundation for the language. Syntax and semantics of the used formalisms, regular expressions and PLTL, are presented in the following subsections, and their interpretation in software is discussed in the subsequent chapters. The definition of PLTL semantics is adapted from [3].

2.1 PROPOSITIONAL FORMULAS

Propositional formulas form a basis for both regular expressions and PLTL formulas. Let AP be a finite non-empty set of atomic propositions. Intuitively, atomic propositions are statements that are either true or false in a state of the system. The propositional connectives are defined with their usual semantics, and their shorthand connectives adopted for convenience of notation. We define propositional formulas over the set of atomic propositions AP.

Definition 1 *Proposition formulas over the set of atomic propositions AP are inductively defined as:*

- *Each atomic proposition ($p \in AP$) is a propositional formula.*
- *Let p, p_1 and p_2 be propositional formulas, then*
  - *$\neg p$ (negation),*
  - *$p_1 \wedge p_2$ (conjunction), and*
  - *$p_2 \vee p_2$ (disjunction) are also propositional formulas.*
- *There are no other propositional formulas.*

Shorthand notations for propositional formulas are defined for convenience of notation as follows:

Definition 2 *Let p, p_1 and p_2 be propositional formulas. Then the following equivalences hold:*

- *$\top \equiv p \vee \neg p$ (true literal) for some $p \in AP$.*
- *$\bot \equiv \neg\top$ (false literal).*
- *$p_1 \Rightarrow p_2 \equiv \neg p_1 \vee p_2$ (implication).*
- *$p_1 \Leftrightarrow p_2 \equiv (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$ (equivalence).*

Def. 3 defines the semantics of propositional formulas in a truth assignment $a \in 2^{AP}$. A truth assignment a is said to *model* an atomic proposition p iff $p \in a$, this is denoted by $a \models p$. Def. 3 gives an inductive definition for semantics of propositional logic.

Definition 3 *Semantics of propositional logic formulas are inductively defined as follows:*

- $a \models p$ *iff* $p \in a$*, for* $p \in AP$.
- $a \models \neg p$ *iff* $p \not\models a$.
- $a \models p_1 \wedge p_2$ *iff* $a \models p_1$ *and* $a \models p_2$.
- $a \models p_1 \vee p_2$ *iff* $a \models p_1$ *or* $a \models p_2$.

2.2 REGULAR EXPRESSIONS

Regular expressions are an intuitive and familiar convention for pattern recognition widely used in the field of programming. They can also be used as a specification language. Here, the execution of a program ($w \in \Sigma^*$) is viewed as a string of consecutive sets of atomic propositions that hold in it [5], i.e., $\Sigma = 2^{AP}$.

Definition 4 *Regular expressions over propositional formulas are inductively defined as follows:*

- *Each propositional formula (see Sect. 2.1) is a regular expression.*
- *Let* r*,* r_1 *and* r_2 *be regular expressions, then their*
  - r^* *(closure or Kleene star) are also regular expressions.*
  - $r_1 \circ r_2$ *(concatenation),*
  - $r_1 \mid r_2$ *(union), and*
- *There are no other regular expressions.*

Definition 5 *The follow shorthand notation for regular expression* r *is defined to hold:*

- $r^+ \equiv r \circ r^*$ *(iteration).*

The syntax defined in Def. 4 can be extended to cover complement and intersection of regular expressions. In this report these constructs are referred to as *extended regular expressions.*

Definition 6 *Extended regular expressions over propositional formulas are inductively defined as follows:*

- *Each propositional formula is an extended regular expression.*
- *Let* er*,* er_1*, and* er_2 *be extended regular expressions, then their*

- $er_1 \circ er_2$ *(concatenation),*
- $er_1 \mid er_2$ *(union),*
- er^* *(closure or Kleene star),*
- $er_1 \& er_2$ *(intersection), and*
- $\overline{er}$ *(complement) are also extended regular expressions.*

- *There are no other extended regular expressions.*

Unfortunately the algorithms required for runtime monitoring with extended regular expressions are too time and memory consuming to do at runtime, see, e.g., [35]. Therefore the extended regular expressions are not considered a prime candidate for a practical interface specification language in the LIME project and left out from the scope of this report. In this report regular expressions are supported only as defined in Def. 4.

Semantics of regular expressions in Def. 7 and Def. 8 has been adopted from [29].

Definition 7 *Let ε be the empty word. Kleene star, concatenation, and union of a language $L \subseteq \Sigma^*$ are defined as follows:*

- $L^* = \{ w \in \Sigma^* \mid w = \varepsilon$ *or* $w = w_1 \circ \cdots \circ w_k$ *for some* $k \geq 1$ *and some* $w_1, \ldots, w_k \in L\}$.
- $L_1 \circ L_2 = \{ w_1 w_2 \in \Sigma^* \mid w_1 \in L_1$ *and* $w_2 \in L_2 \}$.
- $L_1 \mid L_2 = \{ w \in \Sigma^* \mid$ *at least one of the following holds: (i)* $w \in L_1$*, or (ii)* $w \in L_2 \}$.

Definition 8 *Let $\oslash$ be the empty regular expression, $\emptyset$ be the empty set and $\mathcal{L}(r)$ be the language represented by regular expression r. The semantics of regular expressions are as follows:*

- $\mathcal{L}(\oslash) = \emptyset$*, and* $\mathcal{L}(p) = \{a \in \Sigma \mid a \models p\}$ *where p is a propositional formula.*
- *Let r, r_1 and r_2 be regular expressions, then*
  - $\mathcal{L}(r^*) = \mathcal{L}(r)^*$.
  - $\mathcal{L}(r_1 \circ r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2)$.
  - $\mathcal{L}(r_1 \mid r_2) = \mathcal{L}(r_1) \mid \mathcal{L}(r_2)$.

In the monitoring context focus is on the execution trace observed so far. Intuitively, it corresponds to *a prefix* which is formally defined in Def. 9.

Definition 9 *If $w = vy$ for some $v, y \in \Sigma^*$ then v is a prefix of $w \in \Sigma^*$ [29].*

If a correct execution trace of a program is observed then all prefixes of that trace must also have been correct. This property is formalized in Def. 10 as *prefix closed language*.

Definition 10 *Let $w \in L \subseteq \Sigma^*$ then the following holds. The set of prefixes of w are $pref_L(w) = \{v \in \Sigma^* \mid w = vy \text{ for some } y \in \Sigma^*\}$ The* prefix closure of L *is* $pref(L) = \bigcup_{w \in L} pref_L(w)$. *The language L is* prefix closed *iff $pref(L) = L$.*

Example 1 - A prefix closed language

Let $L \subseteq \Sigma^*$ be a prefix closed language and $abcd \in L$, then $\varepsilon \in L$, $a \in L$, $ab \in L$, and $abc \in L$.
■

2.3 PLTL

Propositional linear temporal logic (PLTL) is a commonly used specification logic with both past and future temporal operators. The sublogic consisting of only the future temporal operators is referred to as *LTL* and the sublogic consisting of only the past temporal operator is referred as *ptLTL*. The semantics of a PLTL formula is in this work defined along finite paths $\pi = s_0 s_1 \dots s_{k-1}$ of states. Each state s_i is labelled with the atomic propositions that hold in that state by a labelling function L such that $L(s_i) \in 2^{AP}$, where AP is a set of atomic propositions.

The temporal operators are divided to two groups: future time and past time operators. The future time operators are $\mathbf{X}\,\psi$ ('next'), $\psi_1 \,\mathbf{U}\, \psi_2$ ('until') and $\psi_1 \,\mathbf{R}\, \psi_2$ ('release'). The past time operators are $\mathbf{Y}\,\psi$ ('yesterday'), $\mathbf{Z}\,\psi$ ('weak yesterday'), $\psi_1 \,\mathbf{S}\, \psi_2$ ('since') and $\psi_1 \,\mathbf{T}\, \psi_2$ ('trigger'). The syntactically legal PLTL formulas are given in Def. 11 and their semantics in Def. 14.

Definition 11 *PLTL formulas for the set of atomic propositions AP are inductively defined as follows:*

- *If $p \in AP$, then p is a PLTL formula.*
- *Let ψ, ψ_1 and ψ_2 be PLTL formulas then*
  - $\neg\psi_1$, $\psi_1 \vee \psi_2$, *and* $\psi_1 \wedge \psi_2$,
  - $\mathbf{X}\,\psi_1$, $\psi_1 \,\mathbf{U}\, \psi_2$, *and* $\psi_1 \,\mathbf{R}\, \psi_2$; *and*
  - $\mathbf{Y}\,\psi$, $\mathbf{Z}\,\psi$, $\psi_1 \,\mathbf{S}\, \psi_2$, *and* $\psi_1 \,\mathbf{T}\, \psi_2$ *are PLTL formulas.*
- *There are no other PLTL formulas.*

The following operators are defined as syntactic shorthands for future time temporal operators: $\mathbf{F}\,\psi$ ('finally'), $\mathbf{G}\,\psi$ ('globally'), $\psi_1 \,\mathbf{U_w}\, \psi_2$ ('weak until') and $\psi_1 \,\mathbf{S_w}\, \psi_2$ ('weak since'). Similarly, the following temporal operators are defined as shorthands for past-time operators: $\mathbf{H}\,\psi$ ('historically'), $\mathbf{O}\,\psi$ ('once'), $\uparrow \psi$ ('start'), $\downarrow \psi$ ('end'), $[\psi_1, \psi_2)_s$ ('interval') and $[\psi_1, \psi_2)_w$ ('weak interval').

Definition 12 *The here presented derived propositional and temporal operators are adopted as abbreviations. The monitoring operators ($\uparrow \psi$, $\downarrow \psi$, $[\psi_1, \psi_2)_w$ and $[\psi_1, \psi_2)_s$) have been presented in [14].*

$$
\begin{array}{lcl}
\top & \equiv & p \vee \neg p \ \textit{for some } p \in AP \\
\bot & \equiv & \neg \top \\
\psi_1 \Rightarrow \psi_2 & \equiv & \neg \psi_1 \vee \psi_2 \\
\psi_1 \Leftrightarrow \psi_2 & \equiv & (\psi_1 \Rightarrow \psi_2) \wedge (\psi_2 \Rightarrow \psi_1) \\
\mathbf{F}\,\psi & \equiv & \top\,\mathbf{U}\,\psi \\
\mathbf{G}\,\psi & \equiv & \neg \mathbf{F}\,\neg \psi \\
\mathbf{O}\,\psi & \equiv & \top\,\mathbf{S}\,\psi \\
\mathbf{H}\,\psi & \equiv & \neg \mathbf{O}\,\neg \psi \\
\psi_1\,\mathbf{U_w}\,\psi_2 & \equiv & \mathbf{G}\,\psi_1 \vee \psi_1\,\mathbf{U}\,\psi_2 \\
\psi_1\,\mathbf{S_w}\,\psi_2 & \equiv & \mathbf{H}\,\psi_1 \vee \psi_1\,\mathbf{S}\,\psi_2 \\
\uparrow\psi & \equiv & \psi \wedge \mathbf{Y}\,\neg \psi \\
\downarrow\psi & \equiv & \neg \psi \wedge \mathbf{Y}\,\psi \\
[\psi_1, \psi_2)_s & \equiv & \neg \psi_2 \wedge ((\mathbf{Y}\,\neg \psi_2)\,\mathbf{S}\,\psi_1) \\
[\psi_1, \psi_2)_w & \equiv & (\mathbf{H}\,\neg \psi_2) \vee [\psi_1, \psi_2)_s
\end{array}
$$

Def. 13 defines *valuation* as a function that maps a state into a truth assignment of atomic propositions that hold in the state.

Definition 13 *Let S be the set of states and $s \in S$. Valuation $L(s)$ is a function $L : S \rightarrow \Sigma$ with $\Sigma = 2^{AP}$.*

Definition 14 *Let π^i denote the path $\pi = s_0 s_1 \dots s_{k-1}$ with current state indexed i. The semantics of PLTL formulas in a finite path of length k is defined as follows [3]:*

$$
\begin{array}{lcl}
\pi^i \models_k \psi & \Leftrightarrow & \psi \in L(s_i),\ \textit{for } \psi \in AP. \\
\pi^i \models_k \neg \psi & \Leftrightarrow & \psi \notin L(s_i),\ \textit{for } \psi \in AP. \\
\pi^i \models_k \psi_1 \vee \psi_2 & \Leftrightarrow & \pi^i \models_k \psi_1 \ \textit{or } \pi^i \models_k \psi_1. \\
\pi^i \models_k \psi_1 \wedge \psi_2 & \Leftrightarrow & \pi^i \models_k \psi_1 \ \textit{and } \pi^i \models_k \psi_1. \\
\pi^i \models_k \mathbf{X}\,\psi & \Leftrightarrow & i < k \ \textit{and } \pi^{i+1} \models_k \psi. \\
\pi^i \models_k \psi_1\,\mathbf{U}\,\psi_2 & \Leftrightarrow & \exists i \le j \le k \ \textit{such that } \pi^j \models_k \psi_2 \ \textit{and } \pi^n \models_k \psi_1 \\
 & & \quad \textit{for all } i \le n < j. \\
\pi^i \models_k \psi_1\,\mathbf{R}\,\psi_2 & \Leftrightarrow & \exists i \le j \le k \ \textit{such that } \pi^j \models_k \psi_1 \ \textit{and } \pi^n \models_k \psi_2 \\
 & & \quad \textit{for all } i \le n \le j. \\
\pi^i \models_k \mathbf{Y}\,\psi & \Leftrightarrow & i > 0 \ \textit{and } \pi^{i-1} \models_k \psi. \\
\pi^i \models_k \mathbf{Z}\,\psi & \Leftrightarrow & i = 0 \ \textit{or } \pi^{i-1} \models_k \psi. \\
\pi^i \models_k \psi_1\,\mathbf{S}\,\psi_2 & \Leftrightarrow & \exists 0 \le j \le i \ \textit{such that } \pi^j \models_k \psi_2 \ \textit{and } \pi^n \models_k \psi_1 \\
 & & \quad \textit{for all } j < n \le i. \\
\pi^i \models_k \psi_1\,\mathbf{T}\,\psi_2 & \Leftrightarrow & \textit{for all } 0 \le j \le i : \pi^j \models_k \psi_2 \ \textit{or } \pi^n \models_k \psi_1 \\
 & & \quad \textit{for some } j < n \le i.
\end{array}
$$

Example 2 - Semantics of PLTL formulas in a finite path

Fig. 2.1 presents a finite path of consecutive states. The states 0-6 are labeled with formulas ψ_1 and ψ_2 iff they hold in the corresponding state. It can be seen for example that $\pi^0 \models_5 \psi_1\,\mathbf{R}\,\psi_2$ since $\pi^3 \models_5 \psi_1$ and $\pi^n \models_5 \psi_2$ for all $0 \le n \le 3$.
■

It is always possible to rewrite any formula to *positive normal form*, where all negations appear only in front of atomic propositions. Note that

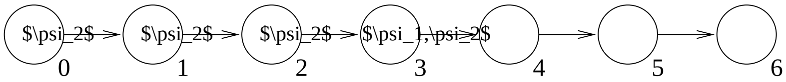

Figure 2.1: Semantics of $\psi_1 \, \mathbf{R} \, \psi_2$ in a finite path

this is actually required to evaluate formulas with negations that are not directly before atomic propositions. It can be accomplished by using the dualities $\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2$, $\neg(\mathbf{X}\,\psi) \equiv \mathbf{X}\,\neg\psi$, $\neg\,(\psi_1 \,\mathbf{U}\, \psi_2) \equiv \neg\psi_1 \,\mathbf{R}\, \neg\psi_2$ $\neg(\mathbf{Y}\,\psi) \equiv \mathbf{Z}\,\neg\psi$, and $\neg(\psi_1 \,\mathbf{S}\, \psi_2) \equiv \neg\psi_1 \,\mathbf{T}\, \neg\psi_2$, see, e.g., [3].

Example 3 - Positive normal form of a PLTL formula

The formula $\neg[\psi_1, \psi_2)_w$ can be turned into positive normal form with the following procedure. Def. 12 defines the interval operators as syntactic shorthands for other PLTL operators and they can thus be replaced with their PLTL counterparts.

$$\begin{array}{lcl} [\psi_1, \psi_2)_w & \equiv & (\mathbf{H}\,\neg\psi_2) \vee [\psi_1, \psi_2)_s \\ [\psi_1, \psi_2)_s & \equiv & \neg\psi_2 \wedge ((\mathbf{Y}\,\neg\psi_2)\,\mathbf{S}\,\psi_1) \\ \hline [\psi_1, \psi_2)_w & \equiv & (\mathbf{H}\,\neg\psi_2) \vee (\neg\psi_2 \wedge ((\mathbf{Y}\,\neg\psi_2)\,\mathbf{S}\,\psi_1)) \end{array}$$

After this conversion the positive normal form can be derived as follows.

$$\begin{array}{ll} \neg((\mathbf{H}\,\neg\psi_2) \vee (\neg\psi_2 \wedge ((\mathbf{Y}\,\neg\psi_2)\,\mathbf{S}\,\psi_1))) & \equiv \\ \neg(\mathbf{H}\,\neg\psi_2) \wedge \neg(\neg\psi_2 \wedge ((\mathbf{Y}\,\neg\psi_2)\,\mathbf{S}\,\psi_1)) & \equiv \\ (\mathbf{O}\,\psi_2) \wedge (\psi_2 \vee \neg((\mathbf{Y}\,\neg\psi_2)\,\mathbf{S}\,\psi_1)) & \equiv \\ (\mathbf{O}\,\psi_2) \wedge (\psi_2 \vee (\neg(\mathbf{Y}\,\neg\psi_2)\,\mathbf{T}\,\neg\psi_1)) & \equiv \\ (\mathbf{O}\,\psi_2) \wedge (\psi_2 \vee ((\mathbf{Z}\,\psi_2)\,\mathbf{T}\,\neg\psi_1)) & \end{array}$$

Hence the positive normal form of $\neg[\psi_1, \psi_2)_w$ is $\mathbf{O}\,\psi_2 \wedge (\psi_2 \vee ((\mathbf{Z}\,\psi_2)\,\mathbf{T}\,\neg\psi_1))$.
■

2.4 SAFETY PROPERTIES

When monitoring a system in execution, it is required that (*i*) an error will occur after a finite execution of the system, and (*ii*) the possible error can be observed. In this section the formal background for these two requirements is introduced.

Informally, *safety properties* are the class of properties that intuitively state "something bad never happens" [26]. In contrast, *liveness properties* are identified as the class of properties state intuitively that "something good must eventually happen" [26] but they are not considered here. Every violation of a safety property occurs after a finite execution of the system and monitoring can be used to detect these failures [28]. The finite execution of a system that leads into an error is formalized in Def. 15 as a *bad prefix*.

Definition 15 *Let $L \subseteq \Sigma^\omega$ be a language of infinite words over the alphabet Σ. A finite word $x \in \Sigma^*$ is a bad prefix for language L, if for every $y \in \Sigma^\omega : xy \notin L$ [28].*

Definition 16 *Given a language L, if all $w \in \Sigma^\omega \backslash L$ have a bad prefix we call L a* safety language *[28].*

A bad prefix is *informative* if it can demonstrate completely why a formula has failed [28]. Safety properties are classified into three subclasses according to the informativeness of their bad prefixes in [24].

1. Intentionally safe properties – all bad prefixes are informative.
2. Accidentally safe properties – for every computation that violates the property, there is an informative bad prefix.
3. Pathologically safe properties – there is a computation that violates the property with no informative bad prefix.

The Def. 14 for PLTL semantics actually matches the definition of informative prefixes of [24] extended to all of PLTL. In [24], safety is considered with the PLTL interpreted over infinite words $\pi \in \Sigma^\omega$, for the semantics of PLTL in this setting, see [24, 3].

The method described in [28] produces counterexamples for both intentionally and accidentally safe properties. We currently don't know of any implementation that would produce counterexamples for a pathologically safe formula. The approach presented in [28] is suited for finding a counterexamples for all non-pathological safety languages. Pathological safety formulas are not practically important as they do not add any expressiveness to the specification language, i.e., for every pathological formula there is a non-pathological counterpart [24]. They can, however, be detected using methodology described in [24].

Example 4 - Syntactic subsets that are non-pathologically safe

- A formula $\mathbf{G}\,\varphi$, where φ contains only past modalities, is a safety formula. Any safety property expressible with LTL is expressible in this way [31].
- Every propositional formula is a safety formula, and if ψ and φ are safety formulas then so are $\psi \vee \varphi$, $\psi \wedge \varphi$, $\mathbf{X}\,\psi$, $\mathbf{G}\,\psi$ and $\psi\,\mathbf{R}\,\varphi$ [37].

■

For deeper discussion about safety properties and their theory see, e.g., [27].

3 THE SPECIFICATION LANGUAGE

The purpose of this chapter is to introduce the reader to how interface and library specifications can be done, and to define the mechanisms and policies used in the language. In Sec. 3.1 the specifications are explained through running examples, and although they are about the specification language, they are written in Java form to put them in context of a real programming language. In Sec. 3.2 the correctness requirements that the specifications impose are defined, and the policies and mechanisms of the language are discussed in detail.

3.1 SPECIFYING INTERFACES AND COMPONENTS – RUNNING EXAMPLES

Example 5 demonstrates how regular expressions can be used for defining interface behavior.

Example 5 - Regular expression in interface specification

Consider a log file interface that expects the client first to `open` the file, then use (`read` or `write`) it and finally `close` it. For describing this behavior, claims about method calls are needed. The call proposition `open ::= open()` declares a proposition open that is true iff the body of method `open()` declared in the annotated interface is currently executing. Notice that argument overloading is not yet considered, and that the write proposition therefore refers to all write methods regardless of their argument types.

The interface defines a checker to enforce its expected use. The checker keeps track of the call orderings through the interface and in case the protocol is violated it signals an exception. In this example a regular expression expresses the previously described call order. You may notice that concatenation($\circ$) is denoted by `;` and Kleene star(*) is expressed with `*` (see Table 3.1 for the rest of the annotations). It is necessary to tie the checker into events (method calls) in the interface. This is done by annotating the desired methods to trigger the corresponding checker either when body of the method is entered, or when it is exited depending of its type. Interface checkers will trigger on entry and library checkers on exit of the method.

```
@InterfaceCheckers(
  callPropositions = {
    "open ::= open()",
    "close ::= close()",
    "read ::= read()",
    "write ::= write()"
  },
  regexpCheckers = {
    "FileUsage ::= (open ; (read | write)* ; close)*"
  }
)
```

```
public interface LogFile {
  @TriggeredCheckers(checkers = {"FileUsage"})
  public void open();
  @TriggeredCheckers(checkers = {"FileUsage"})
  public void close();
  @TriggeredCheckers(checkers = {"FileUsage"})
  public String read();
  @TriggeredCheckers(checkers = {"FileUsage"})
  public void write(String entry);
  public long length();
}
```

It is noteworthy that in the given example calls to `length()` do not violate the `FileUsage` specification. The corresponding checker is not triggered when it is called, hence the checker is perfectly oblivious of the method's existence and any calls made to it.

■

Example 6 introduces default policies that conform to the natural interpretation the specification making their declaration less verbose.

Example 6 - Less verbose specifications with default triggering

The specification seems too verbose to describe such a simple behavior and therefore default policies are adopted to make the language more succinct. Firstly, `open()` in a checker declaration means a call proposition, which true iff `open()` is being executed (see Def. 19). This happens when a checker is triggered on entry or on exit depending on the checker type of the `open()` procedure. Secondly, if a checker contains a call proposition the checker is automatically triggered in the corresponding procedure (see Def. 20). These policies are enforced for the remainder of this work.

After adopting these policies the interface specification can be expressed in the following form:

```
@InterfaceCheckers(
  regexpCheckers = {
   "FileUsage ::= (open() ; (read() | write())* ; close())*"
  }
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String entry);
  public long length();
}
```

■

Example 7 introduces PLTL as a specification formalism. PLTL specification are sometimes more succinct and natural way of describing the desired behavior than regular expressions.

Example 7 - Extending interface specification with a PLTL formula

The file interface may also expect that the write method is never called with a null argument. This can be described as another checker. Now, the property is expressed in a PLTL checker (see annotation details in Table 3.2) which states “always when write is called, it receives a proper String”. With the adopted default enforcement policy, the checker is automatically trigger on entry of the write procedure. In the example `entry` has a special '`#`' sign in front of it. This is the convention for referring to argument values in the specification language. This is purely something made necessary to simplify the implementation and could be avoided if the Java expression could be parsed properly (requiring a full fledged Java parser).

```
@InterfaceCheckers(
  regexpCheckers = {
    "FileUsage ::= (open() ; (read() | write())* ; close())*"
  },
  valuePropositions = {
    "properData ::= (#entry != null)"
  },
  pltlCheckers = {
    "ProperData ::= G (write() -> properData)"
  }
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String entry);
  public long length();
}
```

■

Example 8 combines interface specifications with library specifications. Data handling is introduced as a new feature in the library specification.

Example 8 - A library specification with data handling

Now that interface has established boundaries in which it operates, it may give guarantees as well. Let us assume that the `write` operation is specified to append the file with String in the `entry` argument.

```
@InterfaceCheckers(
  regexpCheckers = {
    "FileUsage ::= (open() ; (read() | write())* ; close())*"
  },
  valuePropositions = { "properData ::= (#entry != null)" },
  pltlCheckers = { "ProperData ::= G (write() -> properData)" }
)
@LibraryCheckers(
```

```
  valuePropositions = {
    "okLength ::= "+
      "#this.length() == #pre(#this.length() + #entry.length())"
  }
  pltlCheckers = { "ProperWrites ::= G (write() -> okLength)" }
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String entry);
  public long length();
}
```

These kinds of specifications require data handling from the specification language. In this approach, primitive values can be stored on method call entry to be used in evaluation at the method exit using a special `#pre` expression. The user must supply a type for this value if it is not an integer which is considered to be the default type. This is done by adding the type to the expression as follows: `#pre.boolean(#this.length() > 0)`.
■

3.2 SPECIFICATION LANGUAGE

The core idea of the specification language is to provide a declarative mechanism for defining component interactions in a manner that their correctness can be verified. The verification is done at runtime by observing the defined specifications (call orderings and relation between arguments and return values in function calls, for example). In a nutshell, the specification language consists of:

1. A mechanism to make claims about program execution or state. These claims are referred to as *atomic propositions* and subdivided to two classes:
    - `valuePropositions` – Claims about program state or values of arguments (e.g., `#this.x == 0`). A value proposition is true if and only if the native language expression evaluates true.
    - `callPropositions` – Claims about function execution (e.g., the body of `open()` is executing). A call proposition is true if and only if the named method is executing.
2. A mechanism to combine propositions to describe expected properties of a software components. These are referred to as checkers and subdivided to classes according to the underlying formalism.
    - `regexpCheckers` – Checkers expressed with regular expressions.

- `pltlCheckers` – Checkers expressed with PLTL.

3. A mechanism to tie checkers to the program flow. This is referred to as triggering a checker. A checker can be triggered by the default enforcement policy presented in Def. 19 or by an explicit annotation.

One of the benefits of using formal specification methods is that there is a well defined basis for deciding is a particular property holds or not. The concept of when a program does not obey its PLTL specification is formalized in Def. 17.

Definition 17 *Let $\pi = s_0 s_1 \dots s_{k-1}$ denote the program trace observed by the checker so far. A PLTL specification φ is broken after π iff $\pi^0 \models_k \neg\varphi$.*

Recall the definition of a prefix closure of a language L in Def. 10 on the page 7. Let $pref(L)$ denote prefix closure of L, i.e., the union of prefixes of all the words in L. Def. 18 formalizes the breach of regular expression specification.

Definition 18 *Let $\pi = s_0 s_1 \dots s_{k-1}$ denote the program trace observed by the checker so far, r be a regular expression specified in a regular expression checker, and $L = \mathcal{L}(r)$ be the language it accepts. A regular expression specification is broken iff holds: $\pi \notin pref(L)$.*

In the subsequent subsections the specification language, and its policies and mechanisms are examined in detail.

3.2.1 Policies and notation

The considered interaction model (see Fig. 1.1 on page 3) suggests that there are two kinds, interface and library, specifications to consider. From the specification language standpoint, the two are very similar, yet not the same. In the interface specifications the return values are not under consideration, but rather the call orderings and argument values. In the library specifications, however, the typical specification does make claims about return values. The default triggering policies (Def. 19) reflect this observation.

Definition 19 *If a method is mentioned in a checker through a call proposition, the checker is implicitly enforced in the corresponding method. The checker is triggered on entry, if the checker is an interface checker, and on exit, if the checker is a library checker.*

While the specification could require each proposition to be explicitly declared, it would lead to a cumbersome and verbose notation. Therefore, the language allows special forms to represent both call and value propositions directly in checker definitions.

Definition 20 *The call propositions can be inlined to a checker definition by referring to a function or a procedure by name in a checker and adding () to denote it is an inlined call proposition.*

expression	annotation	expression	annotation
$r \circ s$	`r ; s`	$r \mid s$	`r | s`
r^*	`r*`	r^+	`r+`

Table 3.1: Regular expressions and their corresponding annotations

Definition 21 *The value propositions can be inlined, i.e., host language boolean expressions can be used in a checker by using <{ boolean expression }> notation (for example* `<{ #this.x > 0 }>`*).*

It is possible to specify a checker that contains a value proposition which is not defined in all methods in which it is triggered. This may happen, for example, when one of the methods takes an argument when others do not. One could trigger checker that states `G (write() -> <{ #entry != null }>)` (where `#entry` it the argument of `write(String entry)`) in `read()` method which takes no arguments and therefore does not know the value of `#entry`. This would make sense since the left hand side of the implication (call proposition `write()` would be always false when `read()` method is executing thus making it true regardless of what is on the right hand side. This observation leads to the following definition:

Definition 22 *If a checker contains a value proposition which is not defined in some method it is triggered in, the undefined propositions are defined to be* false*. If the value proposition is not defined in any method it is triggered in, this is considered to be an error.*

Propositions, named or inlined, are combined into PLTL formulas or regular expressions in the checkers. Regular expressions, as defined in Def. 4 can be expressed with annotations given in Table 3.1. Note that propositional formulas can appear in the regular expressions and their corresponding annotations can be found in Table 3.2. Similarly, corresponding annotations for PLTL are presented in Table 3.2. When specifying PLTL checkers precedence rules presented in Table 3.3 apply. Therefore, for example, `p -> q || r` is parsed `p -> (q || r)` and `p <-> q S r` is parsed `(p <-> q) S r`. It is not advised to specify the checkers in manner that leaves their interpretation open regardless of the precedence rules, e.g., `p S t T u` but rather use parentheses to make the specification explicit, e.g., `p S (t T u)`.

3.2.2 Triggering a checker

As the incremental approach for interface specification suggests, all the created checkers are independent from each other. Thus, adding a new rule which limits the behavior of a software component will in no way interfere with the previously declared rules.

The independence of checkers implies also one important feature about them: they can perceive time or advance in their input string of consecutive program states only when they themselves are triggered. This has a

Past time		Future time		Propositional	
formula	annotation	formula	annotation	formula	annotation
$\mathbf{Y}\, p$	`Y p`	$\mathbf{X}\, p$	`X p`	$p \Leftrightarrow q$	`p <-> q`
$\mathbf{Z}\, p$	`Z p`			$p \Rightarrow q$	`p -> q`
$\mathbf{O}\, p$	`O p`	$\mathbf{F}\, p$	`F p`	$p \wedge q$	`p && q`
$\mathbf{H}\, p$	`H p`	$\mathbf{G}\, p$	`G p`	$\neg p$	`! p`
$p\, \mathbf{S}\, q$	`p S q`	$p\, \mathbf{U}\, q$	`p U q`	$p \vee q$	`p || q`
$p\, \mathbf{S_w}\, q$	`p Sw q`	$p\, \mathbf{U_w}\, q$	`p Uw q`	$\perp$	`FALSE`
$p\, \mathbf{R}\, q$	`p R q`	$p\, \mathbf{T}\, q$	`p T q`	$\top$	`TRUE`
$[p, q)_s$	`[p, q)s`			p	`p`
$[p, q)_w$	`[p, q)w`				
$\uparrow p$	`Start(p)`				
$\downarrow p$	`End(p)`				

Table 3.2: PLTL formulas and their corresponding annotations

1. $[\psi_1, \psi_2)_s$, $[\psi_1, \psi_2)_w$
2. $\psi_1\, \mathbf{S}\, \psi_2$, $\psi_1\, \mathbf{S_w}\, \psi_2$, $\psi_1\, \mathbf{T}\, \psi_2$, $\psi_1\, \mathbf{U}\, \psi_2$, $\psi_1\, \mathbf{U_w}\, \psi_2$, $\psi_1\, \mathbf{R}\, \psi_2$
3. $\psi_1 \Leftrightarrow \psi_2$
4. $\psi_1 \Rightarrow \psi_2$
5. $\psi_1 \wedge \psi_2$
6. $\psi_1 \vee \psi_2$
7. $\neg\psi$, $\mathbf{Y}\, \psi$, $\mathbf{Z}\, \psi$, $\mathbf{H}\, \psi$, $\mathbf{O}\, \psi$, $\mathbf{X}\, \psi$, $\mathbf{G}\, \psi$, $\mathbf{F}\, \psi$, $\uparrow \psi$, $\downarrow \psi$

Table 3.3: Precedence of logic operators

concrete interpretation when it comes to, e.g., the semantics of the temporal operators next ($\mathbf{X}\, \psi$) and yesterday ($\mathbf{Y}\, \psi$). The previous (or the next) moment in time is understood to be the previous (or next) time when a particular checker is triggered in a particular object instance.

Fig. 3.1 illustrates how time is perceived to pass by `FileUsage` (an interface checker) and `ProperWrites` (a library checker) checkers of the earlier examples (see Chap. 3.1) over a sequence of method invocations through the interface. `FileUsage` checker is ran *before* executing the bodies of `open()`, `read()`, `write()` or `close()` where as `ProperWrites` checker executes *after* the body of `write()`.

3.2.3 Data handling

Library specifications, and the checkers that enforce them, are included into the specification language to establish the correct responses for the method invocations. The responses are not known until the method has been executed, hence the enforcement must happen at the method exit.

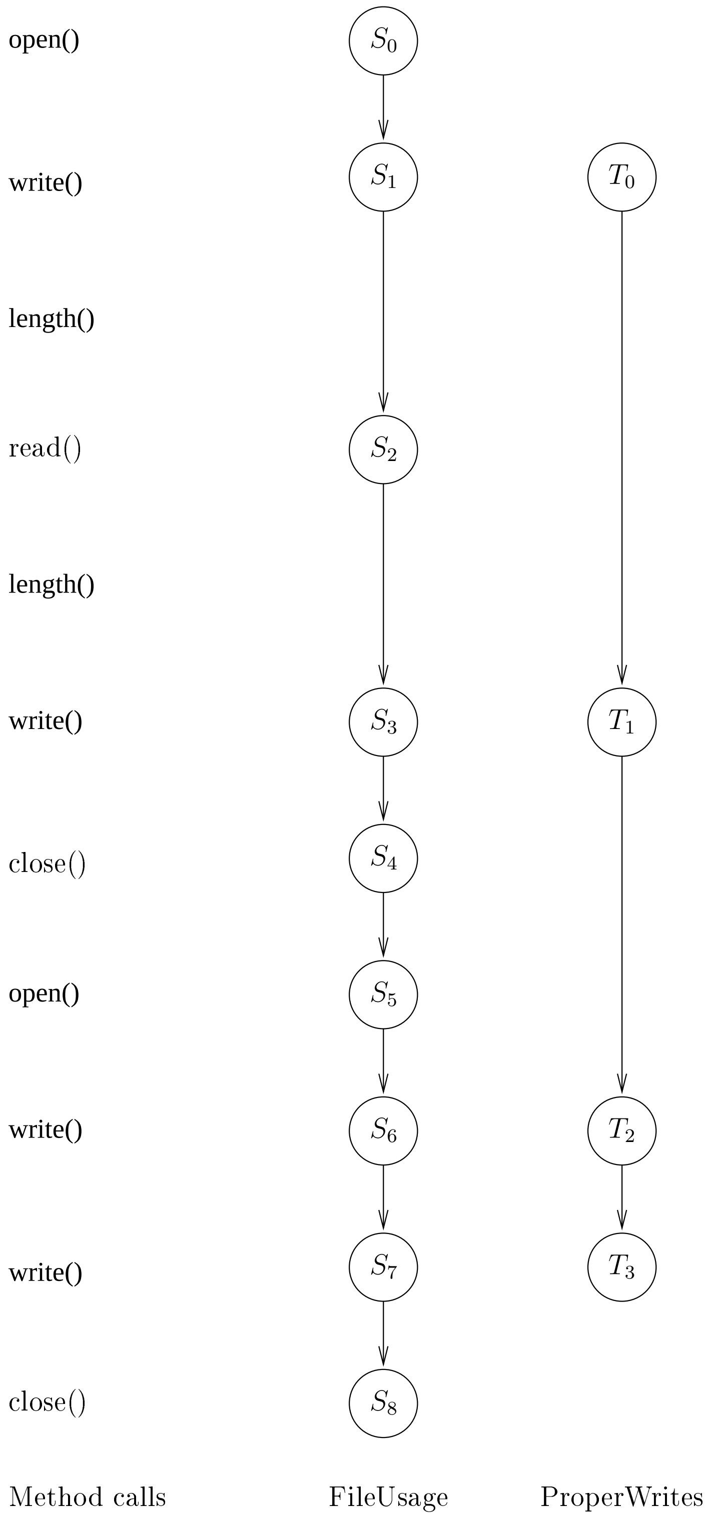

Figure 3.1: Time, as observed by two different checkers

The relation between input parameters and return values is not trivial to establish as the method body may have altered the arguments given to it.

We employ *a history variable mechanism* to store values as they were when execution of a method started to enable the comparison of these pre values to the post values at exit. This was used already in the `LogFile` interface example presented in Chap. 3.1. There we specified a proposition `okLength` to be equivalent to expression `#this.length() == #pre(#this.length() + #entry.length())`. This proposition was used in library checker `ProperWrites` which stated `G (write() -> okLength)`. One could read this “for writing to a log file to be successful, its length should be incremented by length of the new log entry”. To make this comparison, the sum of lengths for the appended file and the new entry are stored to a new history variable which is guaranteed to remain unchanged during the execution of the method. Note that as can be seen from Fig. 3.1, the storing of prevalues to the history variables does not induce a new state. Thus the previous time step for `ProperWrites` checker is always the preceding invocation of `write()`.

The specification language supports only primitive values to be stored this way since, in the general case, it is not possible to store an object in this way. Storing an object reference would not prevent the body of the method from altering it. Furthermore, the type of the stored value should be announced in the `#pre` statement, e.g., `#pre.char(#c)` or `#pre.integer(#this.length())`. If type is not announced it is expected to be integer. Note that storing the #pre history value does not induce a time step into the checker.

4 TECHNICAL BACKGROUND

The purpose of this chapter is to introduce enough technical aspects for the reader to understand the implementation of the tool described in Chap. 5.

4.1 ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming (AOP) is a programming paradigm originally developed by Gregor Kiczales et al. in Xerox Palo Alto Research Center. It was developed to capture and modularize concerns that crosscut the traditional programming constructs such as objects. Such crosscutting concerns tend to be important parts of the program behavior that are not directly included into its functionality, such as failure handling strategy, communication strategy and behavioral invariants. [21]

Robert E. Filman and Daniel P. Friedman define the structural essence of aspect-oriented programming in [12]:

> "AOP can be understood as the desire to make quantified statements about the behavior of programs and to have these quantifications hold over programs that have no explicit reference to the possibility of additional behavior."

They further classify AOP to *black-box* and *clear-box AOP* according to its quantification mechanism. In black-box AOP the quantification happens over public interfaces of components, where as in clear-box AOP the quantification can be done over parsed structure of components. They also identify the concept of *incomplete obliviousness* which used to describe the sometimes relevant need of communication between the application programmer and the aspect. [12]

The quantifications can be made over the public interfaces of components but the specification language also allows accessing private data of software components. The implementation relies on predescribed (See Chap. 3) annotations for communication between aspects and the application programmer. In terms of the forementioned definitions, the approach taken in this work is establishment of behavioral invariants through clear-box AOP with incomplete obliviousness.

4.2 ASPECTJ

AspectJ is a general purpose AOP extension to Java which has a join point model does allow clear-box AOP [12]. It is distributed by the Eclipse Foundation under Eclipse Public License (EPL). AspectJ allows two kinds of crosscutting: *dynamic* and *static*. The former allows aspects to define additional implementations to run at certain well-defined points of program execution. The latter allows aspects to define new operations to already existing types. [20]

```
aspect LockAspect {
  private boolean opened = false;
  before(): call (public void Lock+.open()) {
    if(opened)
      throw new RuntimeException("bad open");
    opened = true;
  }
  before(): call (public void Lock+.close()) {
    if(!opened)
      throw new RuntimeException("bad close");
    opened = false;
  }
}
```

Figure 4.1: Lock safety aspect

In this work and in the LIME project, the purpose is to generate dynamically cross-cutting aspects such as presented in Fig. 4.1.

4.2.1 Join points and pointcuts

In AOP, *join points* are well-defined points in program execution. The join point model of an AOP language defines interactions of aspect and non-aspect code can have [20]. In this work, the main interest is on joining aspect code to method calls.

A *pointcut* consists of a set of join points and optionally also of values in program execution context [20]. In Fig. 4.1, `call (public void Lock+.open())` is a *pointcut designator*, a predicate over join points which selects the join points of interest. This designator expresses the set of calls to `open()` method in any object which implements the Lock interface or any interface that extends it (this is the case because of the `+` sign). In particular, since the designator is of the form `call(`*signature*`)` the execution is still on the caller side. Should `execution(`*signature*`)` be used instead the callee would have started executing.

Pointcut designators such as `call` and `execution` (formerly `calls` and `executions`) are called *primitive pointcut designators*. *Compound pointcuts* can be formed using and (`&&`), or (`||`) and not (`!`) operators [20]. For example, the compound pointcut `call (public void Lock+.open()) || call (public void Lock+.close())` selects the set of join points matching either of the designators. Furthermore, AspectJ allows users to form *user-defined pointcut designators*. By defining:

```
pointcut lockCalls():
  call (public void Lock+.open()) ||
  call (public void Lock+.close());
```

the user introduces a new pointcut designator, `lockCalls()`, which could be used to instead of the compound designator.

4.2.2 Advices and aspects

With the join point model and pointcuts explained the AspectJ advices and aspects are discussed here.

Advices

An *advice* is the additional implementation to be attached to a join point or join points expressed by a pointcut designator. In Fig. 4.1,

```
before(): call (public void Lock+.open()) {
  if(opened)
    throw new RuntimeException("bad open");
  opened = true;
}
```

defines additional behavior to be run before the program execution may enter a join point in the designated pointcut. This advices simply checks if the lock is already open before it allows the execution of `open()` method to start.

Sometimes it is necessary to run the advice not before or after but around a join point. Consider Example 8 on page 13 and the library checker `ProperWrites` defined in it. The checker states that "after write method is called the log file's length will be its length before the call plus the length of the given argument". This clearly implies that the program state should be observed on two separate times before and after the call to establish the truth value of the claim. The resulting advice could be written as:

```
void around(LogFile f, String e) :
  call(public void LogFile+.write(String entry)) && args(e) &&
    target(f) {
  int preLength = e.length() + f.length();
  proceed(f, e);
  if(!(f.length() == preLength))
    throw new RuntimeException("ProperWrites checker fails");
}
```

Here, the primitive pointcut designators `args` and `target` are used to bind arguments of the call, i.e., the new entry, and the target of the call, i.e., the log file, within the namespace of the advice. This allows reference to them in the advice code. This is called passing context from a join point to the advice [25]. The advice also illustrates how around advices differ from before and after advices. Whereas before and after advices are strictly additional behavior, around advice is run *instead of* the join point. AspectJ offers the special `proceed()` syntax to run the next advice of lower precedence (or ultimately the method itself). The context exposed to the around advice's pointcut, e.g., arguments and call target, are given to `proceed` as parameters. In the presented example, the reader may assume that the actual `write` method is executed during `proceed` instruction.

```
aspect LockAspect pertarget(lock() || close()) {
  private boolean opened = false;
  pointcut open() : call (public void Lock+.open());
  pointcut close() : call (public void Lock+.close());
  before(): open() {
    if(opened)
      throw new RuntimeException("bad open");
    opened = true;
  }
  before(): close() {
    if(!opened)
      throw new RuntimeException("bad close");
    opened = false;
  }
}
```

Figure 4.2: Lock safety aspect with `pertarget` instantion

Aspects

Aspects are class-like components which encapsulate other AOP elements such as pointcuts and advices. Like Java classes, AspectJ aspects can contain also regular object elements like fields and methods, and also form inheritance hierarchies. They can be declared abstract like it is the case with classes or as concrete aspects. They cannot, however, be used interchangeably with classes.

Aspects are instantiated via *per clauses* (in contrast to objects which are instantiated with `new` operator). If an aspect contains no per clause it uses the default `issingleton()` instantiation, i.e. there only will one instance of it. In the presented example, Fig. 4.1, there would only be one aspect for all instances of that implement the `Lock` interface. It is clear that this not something that is wanted but rather there should instead be an aspect instance for each `Lock` instance. In Fig. 4.2, *pertarget* instantiation ensures that there is an aspect instance for every instance of a `Lock` that is used.

Privileged aspects

The expressiveness of AspectJ aspects is not, however, limited to this. Declaring an aspect to be *a privileged aspect* allows it to access private fields and methods in classes, and this feature is employed in this work. It would also be possible to declare new parents or implemented interfaces for classes but this feature is not needed in this work.

Aspect precedence

It is sometimes imperative that advices are executed in a particular order in a given join point. AspectJ provides `declare precedence` construct for this purpose which can be used inside aspect declarations [25].

```
declare precedence : IFChecker*, LSChecker*;
```

The concrete aspects of the form `IFChecker*` are now said to *dominate* the concrete aspects of the form `LSCheckers*`. In case of an around advice the higher-precedence aspect encloses the lower-precedence aspect's advice [25].

Weaving

Aspect code and non-aspect code is combined by a compilation process called *weaving* and the processor doing the job is called a *weaver*. Weaving can be done straight into the source or byte code, or it can be done by loading the compiled aspects into the *Java Virtual Machine (JVM)* and performed in just-in-time style. In all cases, the resulting program is a valid Java program and will run in any conformant virtual machine. [25]

5 THE TOOL IMPLEMENTATION

In this chapter the tool implementation for the specification language is described. Java has been chosen as the host language due to ease of instrumentation and preexisting metadata mechanism (Java annotations). First part of this chapter (Sec. 5.1) describes the tool for its potential users, whereas the second part (Sec. 5.2) discusses the architecture of the implementation.

5.1 PROGRAMMING INTERFACE

Annotations are used for specifying the desired behavior which is then synthesized as monitors for runtime verification. This section describes the tool interface from users' point of view. The focus here is on the language dependent details of the implementation such as the lexical form of propositions and checkers, and the description of annotations used.

5.1.1 Propositions

The propositions have been divided into two categories, to claims about method invocations (or calls), and to claims about object's state. As discussed in Chap. 3 the former is referred to as call propositions, and the latter to as value propositions. Def. 23 defines the proper lexical form for named propositions that the tool expects. Notice that the semicolon (;) is left out of the set of proper characters since it serves as statement terminator in Java. Value propositions that contain side effects should be avoided.

Definition 23 *The lexical form for atomic propositions is defined as (where Σ is the set of proper host language characters):*

- `"[a-z]([a-zA-Z])`* `::=` $(\Sigma - \{$`;`$\})^+$`"`, *where the left side gives a name for the proposition, and the right side is name of the method (followed by parentheses, e.g.,* `open()`*) in case of a call proposition, or a boolean expression in case of a value proposition.*

The named propositions start with a lower case letter because that makes them easily detectable in lexical analysis. The host language independent syntax by which propositions are declared is discussed in Sec. 3.2.

5.1.2 Checkers

Both interface and library checkers are declared by respective annotations, `@InterfaceCheckers` (Fig. 5.1) and `@LibraryCheckers` (Fig. 5.2). These annotations can be targeted (attached) to a Java type (an interface or a class). The annotations have source retention policy, i.e., they will not be present in the compiled byte code.

The current implementation limits the way how checkers can be declared using PLTL. It is not possible to have future time formulas as subformulas for past time operators (this means that, e.g., $\mathbf{O}(\mathbf{G}(p))$ is of illegal form).

Definition 24 *Lexically, regular expression and PLTL checkers is specified as (where Σ is the set of proper host language characters):*

- `"[a-zA-Z]`$^+$ `::=` Σ^+`"`

The host language independent syntax for declaring regular expression checkers is given in Table 3.1 on page 16. The syntactic rules for declaring PLTL checkers are given in Table 3.2 on page 17 and the precedence rules for operators in Table 3.3 on page 17.

```
package fi.hut.ics.aspectmonitor.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(value = RetentionPolicy.SOURCE)
@Target(value = ElementType.TYPE)
public @interface InterfaceCheckers {
  String[] valuePropositions() default {};
  String[] callPropositions() default {};
  String[] pltlCheckers() default {};
  String[] regexpCheckers() default {};
}
```

Figure 5.1: Annotation for declaring interface checkers

```
package fi.hut.ics.aspectmonitor.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(value = RetentionPolicy.SOURCE)
@Target(value = ElementType.TYPE)
public @interface LibraryCheckers {
  String[] valuePropositions() default {};
  String[] callPropositions() default {};
  String[] pltlCheckers() default {};
  String[] regexpCheckers() default {};
}
```

Figure 5.2: Annotation for declaring library checkers

Triggering checkers

A property can be enforced by triggering its corresponding checker in a method. The default enforcement policy defined in Def. 19 on page 15 is employed, i.e., if a checker contains a call proposition it is automatically triggered in the corresponding method. However, it is possible to explicitly trigger a checker in a method. This is done with an annotation `@TriggeredCheckers` (see Fig. 5.3). Note that this annotation can be used for changing the exception thrown when a checkers notices an error (default exception is fi.hut.ics.aspectmonitor.CheckerException). The

```
package fi.hut.ics.aspectmonitor.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import fi.hut.ics.aspectmonitor.CheckerException;

@Retention(value = RetentionPolicy.SOURCE)
@Target(value = { ElementType.METHOD })
public @interface TriggeredCheckers {
  String[] checkers() default {};
  Class<? extends Exception> exception()
    default CheckerException.class;
}
```

Figure 5.3: Annotation for triggering checkers

annotation consists the following fields:

- `checkers` – the list of checkers to be run in the annotated method. Interface checkers will be run before and library checkers after the method invocation.

- `exception` – the class of the exception thrown when a property is violated (name of the checker is given as parameter to the exception).

5.2 TOOL ARCHITECTURE

The implementation is built using the Spoon framework [33] and operates by visiting the *abstract syntax tree (AST)* produced by the Sun Microsystems Java-compiler. The instrumentation program is to be distributed as a *Spoonlet*. Spoonlets contain AST visitors which can be used for program analysis and transformation at compile-time. They are also attractive in the sense that they can be integrated into Maven 2 compiler (`http://maven.apache.org/`) and Eclipse development environment (`http://www.eclipse.org/`) with respective plug-ins.

Fig. 5.4 describes the layered architecture of the implementation. An upper layer module may use lower level module if they have a dashed line between them. The implementation effort here consists of the Common (fi.hut.ics.common) and Aspect Monitor (fi.hut.ics.aspectmonitor) modules. Spoon (fr.inria.gforge.spoon), Automaton (dk.brics.automaton) and SCheck are adopted as third-party software:

- **Spoon** is used for analyzing the program and interfacing to existing tools (Maven 2 and Eclipse).
  - CeCILL-C license - French equivalent to LGPL.
- **dk.brics.automaton** is used for internal representation and manipulation of regular expression checkers.
  - BSD license.
- **SCheck** is used for converting temporal logic formulas into finite state automata.
  - GPL license.

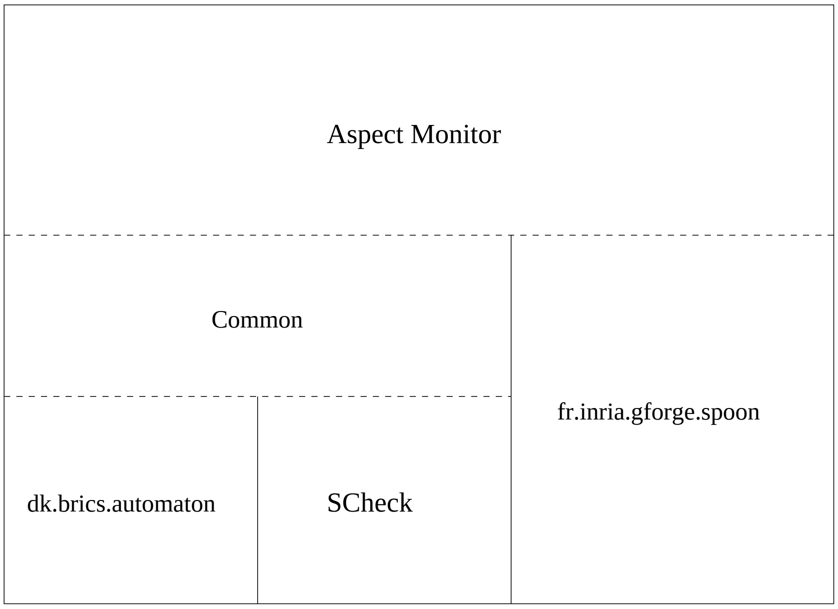

Figure 5.4: Basic modular decomposition

Fig. 5.5 gives an overview of transforming an annotated type into an aspect. The process consists of two main passes – analysis and synthesis. The products of the analysis pass are specification objects that are used as the internal representation of a checker. A specification corresponds to a checker annotated into a type (e.g., an interface). The `AbstractSpecification` class is subclassed into `PltlSpecification` and `RegExpSpecification` classes to accommodate their structural differences.

If a type has two checker declarations there will be two specification objects created to represent them. The two will share namespace in terms of named call and value propositions which are extracted from the annotations. Specifications are enforced in the methods mentioned in them via call propositions or explicitly annotated with `@TriggeredCheckers` (see Fig. 5.3). Before aspects can be synthesized from specifications, named propositions in formulas are replaced with their corresponding call or value propositions. Also, the annotated methods are made to trigger the checker.

In the synthesis pass the specification objects are turned into aspects. Code generation is discussed in more detail in Sect. 5.2.2.

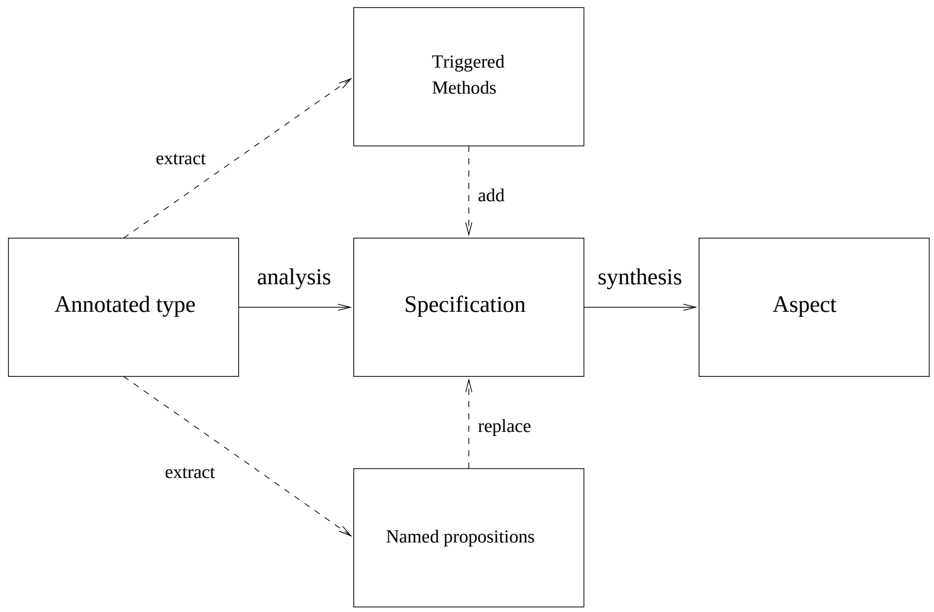

Figure 5.5: Generating temporal safety aspect from an annotated class

5.2.1 Common

The common module (`fi.ics.hut.common`) offers library services for representing, constructing and manipulating regular expressions, finite state automata and temporal logic formulas. It is subdivided into three packages each corresponding to a particular model of representation:

- `fi.ics.hut.common.logic`
- `fi.ics.hut.common.regexp`
- `fi.ics.hut.common.pltl`

The logic package (fi.ics.hut.common.logic) contains the functionality for representing and handling propositional and temporal logic formulas. One of the key services that the package offers is illustrated in Fig. 5.6. It contains *a lexical analyzer (lexer)* which identifies the lexical tokens of a formula from a string representation and transforms it into a list of identified tokens, or rejects a bad input. After lexical analysis, the structure of the formula can be derived from the list of tokens. This is done by *a syntactic analyzer (parser)*.

Syntactic analysis produces an unambiguous tree representation of the formula. In this form visitor pattern [13] can be employed for analyzing and modifying the formula. It is noteworthy that the parser is for full PLTL, i.e., any valid PLTL can be identified by it. However the tool can not translate past-time formulas with future time subformulas into aspects. Therefore a semantic analysis pass is performed to identify and reject these types of formulas. The same parser is also used to propositional formulas in regular expressions. In that context the semantic criteria is that the formulas may not contain any temporal operators. For more about lexical, syntactic and semantic analysis in context of programming languages see, e.g., [1].

Example 9 - On lexical, syntactic and semantic analysis of PLTL formulas

Consider the following examples of strings and their analysis.

- `"Nonsense"` is lexically incorrect (named propositions start with small letters).
- `"!  -> G"` is lexically correct since it can identified into a list of tokens ($\neg$, $\Rightarrow$, $\mathbf{G}$) but syntactically incorrect since it cannot be parsed into a formula.
- `"O (G p)"` is both lexically and syntactically correct (a valid PLTL formula) but it cannot be interpreted by the tool into an aspect and therefore it is considered semantically incorrect.
- `"G (write() -> <{ #data.length() > 0 }>)"` is a syntactically and semantically valid PLTL formula.

■

The regular expression package (fi.ics.hut.common.regexp) serves as an adapter package for dk.brics.automaton module (referred to as the Automaton module from now on). It is responsible for the lexical analysis of regular expressions. Propositional formulas in regular expressions are identified as single tokens at this stage, and their further analysis is done by the logic package as mentioned earlier. The alphabet in the Automaton module regular expressions are characters and not propositional formulas like in the specification language. Therefore the propositional logic formulas must be replaced with characters. The replacement is done by using the union of characters, that each represent a truth assignment of atomic propositions in which the propositional formula is true. The empty language is used if the formula is not true in any truth assignment. The syntactic analysis and automaton operations are done by the Automaton module.

The PLTL package (fi.ics.hut.common.pltl) serves as an adapter package for the SCheck module. The main function for this package is to

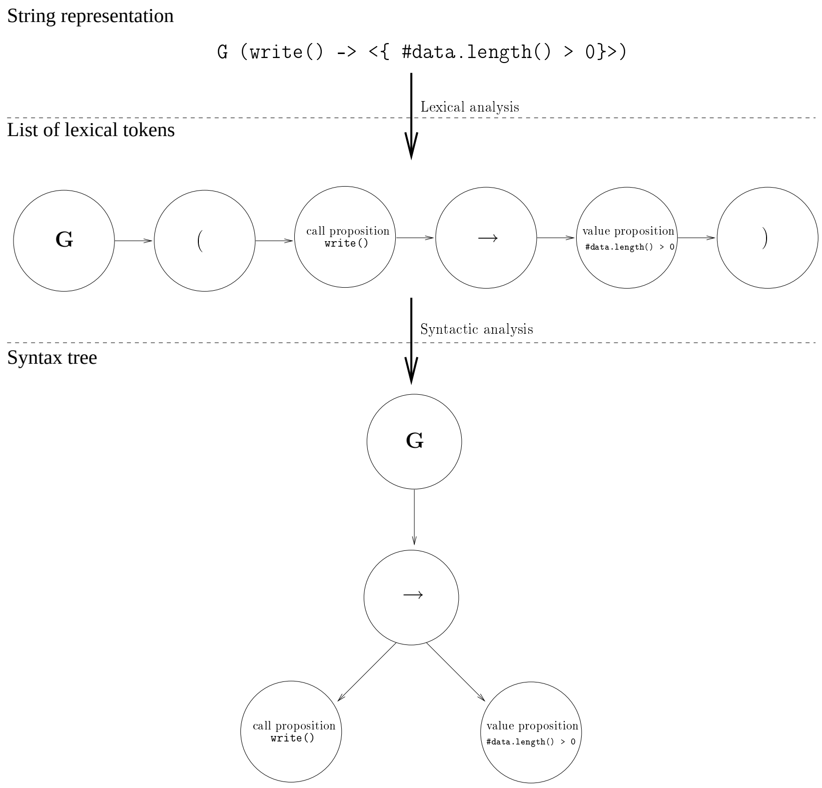

Figure 5.6: Lexical and syntactic analysis of a logic formula

transform a tree representation of the formula produced by the logic package into an automaton. This is done by transforming the tree representation into the format accepted by SCheck. In SCheck the atomic propositions are not handled as they are represented in this tool, therefore a mapping between the two representations must be saved over the conversion process. After SCheck is done with the conversion to an automaton, the propositions are replaced into it. Note that the values of past time subformulas are treated here as propositions, more about this in Sec. 5.2.2.

5.2.2 Aspect monitor

Aspect monitor is the main module of this application which turns annotations in Java types into AspectJ aspects for monitoring their behavior. To accomplish this, the module uses the program analysis services provided by Spoon framework and the formalism services provided by the common module. As presented in Fig. 5.5 this transformation is done in two phases – analysis and synthesis. These phases have corresponding packages in the implementation:

- fi.hut.ics.aspectmonitor.specification
- fi.hut.ics.aspectmonitor.aspect

The specification package implements analysis phase, whereas the aspect package is responsible for synthesis phase.

Combining future and pasts formulas

In [14, 15], Havelund and Roşu describe the synthesizing procedure for formulas of the $\mathbf{G}\,P$, where P is a ptLTL formula. Here this procedure is extended to the subset of PLTL where past-time operators may not have future-time operators in their subformulas.

Consider the PLTL specification $\mathbf{G}\,(\texttt{start()} \rightarrow \mathbf{O}\,\texttt{ignite()})$. It asserts that whenever `start()` is called there has once been a call to `ignite()`. Intuitively, synthesizing this property would require one extra boolean variable which would keep track if `ignite()` has been called. The variable would be initialized to be false and a call to `ignite()` would set it to be true. Then whenever `start()` is called an assertion is made to check if the extra variable is true. The formula is represented on the left in Fig. 5.7, and what was effectively done to it by the introducing a new variable on the right. Now the variable `temp` stands for the evaluation of the ptLTL formula on a given time.

The approach of [14, 15] generalizes this principle for all formulas of the form $\mathbf{G}\,P$, where P is a ptLTL formula. In the approach there is, however, not one but two variables for every past-time operator – one for the current evaluation of the corresponding subformula and one evaluation in the previous state. Let $now_i(\psi)$ be the evaluation of past-time subformula ψ at time i and let $prev_{i-1}(\psi)$ be its evaluation at time $i-1$. Table 5.1 presents the initial values of past-time subformulas and update rules according to their operator.

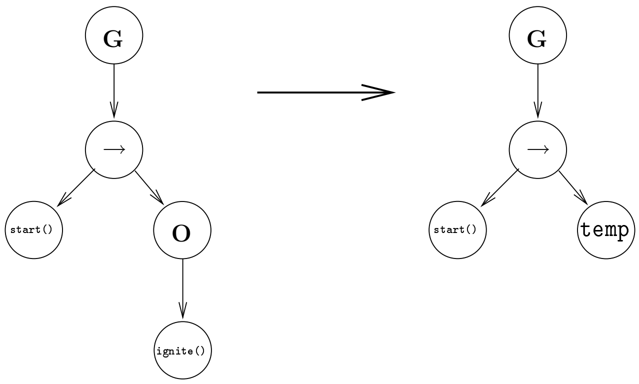

Figure 5.7: Representing past-time subformulas in PLTL

Now coming back to the example, $prev_0(\mathbf{O}\,\texttt{ignite()})$ is initialized to be $\perp$ (see Table 5.1) because the formula is of the form $\mathbf{O}\,p$. As a new state is observed, the new truth values are:

- $now_1(\mathbf{O}\,\texttt{ignite()}) = \texttt{ignite()} \vee prev_0(\mathbf{O}\,\texttt{ignite()}) = \texttt{ignite()} \vee \perp$.
- $prev_1(\mathbf{O}\,\texttt{ignite()}) = now_1(\mathbf{O}\,\texttt{ignite()})$.

Now the upper level formulas may treat the subformula $\mathbf{O}\,\texttt{ignite}$ as just another proposition which evaluates to $now_i(\mathbf{O}\,\texttt{ignite()})$. This true for all past time formulas (see Table 5.1) because their truth values are dependent only from their initial values and update rules.

ψ	$prev_0(\psi)$	$now_i(\psi)$	$prev_i(\psi)$
$\mathbf{Y}\,p$	$\bot$	$prev_{i-1}(\psi)$	p
$\mathbf{Z}\,p$	$\top$	$prev_{i-1}(\psi)$	p
$\mathbf{O}\,p$	$\bot$	$p \vee prev_{i-1}(\psi)$	$now_{i-1}(\psi)$
$\mathbf{H}\,p$	$\top$	$p \wedge prev_{i-1}(\psi)$	$now_{i-1}(\psi)$
$p_1\,\mathbf{S}\,p_2$	$\bot$	$p_2 \vee (p_1 \wedge prev_{i-1}(\psi))$	$now_{i-1}(\psi)$
$p_1\,\mathbf{T}\,p_2$	$\top$	$p_2 \wedge (p_1 \vee prev_{i-1}(\psi))$	$now_{i-1}(\psi)$

Table 5.1: Initial values and update rules for ptLTL formulas

The *prev* and *now* are implemented as boolean arrays with an element for each past-time operator in the formula. A formula traversal is applied to the tree representation of the formula which assigns each past time formula a larger index than any of its subformulas. The update rules are applied in ascending order of indices so that when evaluating any formula all of its subformulas have already been updated. An aspect conversion for the example formula used in this section is presented in Sec. 6.3 on page 39.

Semantics of value propositions

Call propositions and value propositions are easily identifiable in a formula from the lexical form, see Def. 20 on page 15 and Def. 21 on page 16. This coarse distinction is done by the common module during lexical analysis of the formula. In value propositions there are, however, reserved words that give special semantics for the propositions. In Chap. 3 the following reserved words were introduced:

- `#this` for referencing an instance of the annotated interface.
- `#result` for referencing the return value of the method.
- `#pre[.primitive type](`*`boolean expression`*`)` for referencing an on entry value in library specifications after the actual method has executed.
- `#<argument>` for referencing the arguments given to the annotated method.

The semantics these reserved word are given as follows. The call target can be passed for the advice by `#this` from the join point context. Note that `#this` always refers to the annotated type which is the call target and is not to be confused with the primitive pointcut designator `this()` which refers to the calling component. The value returned by `proceed()` instruction can referenced by using `#result`. If `#result` is not defined, i.e., used in context of a `void` method the corresponding value proposition is defined to be false. If `#result` is not defined in any context a specification is enforced it is interpreted to be an error. Similar policy holds for the use of `#<argument>`. For the use of `#pre` see Sec. 3.2.3 on page 17.

Analyzing a PLTL specification

The main goal for analysis phase is to generate a deterministic automaton which can then be synthesized. For PLTL specifications SCheck is used to produce the automaton. The input for the analysis string representation of the property and a mapping from name to propositions to represented named propositions. The individual steps to accomplish this are as follows:

1. Lex the property into a token list.
2. Parse token list into a tree representation.
3. Perform semantic analysis for the tree representation.
4. Rewrite formula to remove operators which do not have dual operators in the language.
5. Modify the formula to be in positive normal form.
6. Replace named propositions in formula with corresponding call and value propositions.
7. Transform the tree representation into a *directed acyclic graph (DAG)* so that common subtrees appear only once in the representation.
8. Assign indices for possible ptLTL operators.
9. Use SCheck to create an automaton from the formula.

Note that the automatons produced by SCheck (version 1.1 or newer) are deterministic and minimized.

Analyzing a regular expression specification

As with PLTL formulas, the goal with regular expression specifications is to produce a deterministic automaton from a string representation of a property.

1. Lex the property into a token list.
2. Build a regular expression adapter from the token list:
   (a) In lexical analysis, propositional formulas are identified into single tokens which are then parsed with PLTL parser.
   (b) Propositional formulas are replaced with the language consisting of the union of the truth assignments in which the formula in question is true (see Def. 7 on page 6).
   (c) The resulting property is turned into a dk.brics.RegExp.
3. The regular expression is generated into a deterministic automaton.
4. The automaton is prefix closed.
5. The automaton is complemented.
6. All transitions from accepting states are removed.
7. The automaton is minimized.

Although the automaton is minimized, the transitions in the current implementation do contain some redundancy. This is because the way that propositional formulas represented (see 2b). Consider a regular expression contains two atomic propositions $p, q \in AP$ and a transition is enable if p holds. Then the guard for the transition is a disjunction of the truth assignments in which p holds, i.e., $(p \wedge q) \vee (p \wedge \neg q)$. Now both truth assignments are mapped to a different character and it is not possible for the Automaton module to minimized this redundancy. This can be seen in the aspect presented in Fig. 6.2 on page 38.

Synthesizing specifications as aspects

Both types of specification are synthesized following the same baseline. For PLTL formulas initialization and update code for past-time subformulas are generate as well.

1. The current state of the automaton is held in `state` (integer) variable which is initialized to be the initial state.
2. A `refreshState()` method is add to handle the state transitions. The takes the evaluations of atomic propositions as parameters and updates the `state` based on them and the current state.
3. If the specification is a library specification and contains propositions that have prevalues, store the prevalues before the actual method call.
4. For each enforced method an advice is create which passes the evaluations of atomic propositions to `refreshState()` method:
   (a) before the actual method call if the specification is an interface specification, or
   (b) after the actual method call if the specification is a library specification.
5. After `refreshState()` if the current state is an accepting throw the specified exception with the specification's name as argument.

In this section the modular decomposition (Fig. 5.4) and the work flow (Fig. 5.5) of the tool was presented. Also the purpose and roles for third-party software (Spoon, Automaton, and SCheck) was identified. The work flow of transforming an annotated type into an aspect that monitors the specified behavior was divided into two phases – analysis and synthesis, and a basic outline was given for both. This section also showed how past time subformulas can be treated as atomic propositions with truth values that are dependent on the initial values and update rules given in Table 5.1. Also the way of how propositional formulas in regular expressions are handled was discussed.

6 EXPERIMENTS

The purpose of this chapter is to demonstrate generated safety aspects from example specifications. Sect. 6.1 demonstrates a basic interface specification written with a regular expression. A library specification that employs history variable mechanism is considered in Sect. 6.2. Finally Sect. 6.3 demonstrates the technique to capture past-time subformulas in PLTL specifications.

6.1 AN INTERFACE SPECIFICATION FOR A LOCK INTERFACE

Thomas Ball et al. present the proper use of spinlocks as one of the API usage rules for Windows XP kernel in [2]. This simple enough rule assumes that locks are initialized as open, and then it requires a strict alternation of `lock()`s and `unlock()`s to follow. In Fig. 6.1 the automaton on the left corresponds to the complement language of the rule, i.e., should automaton end up in an accepting state during execution the rule is broken.

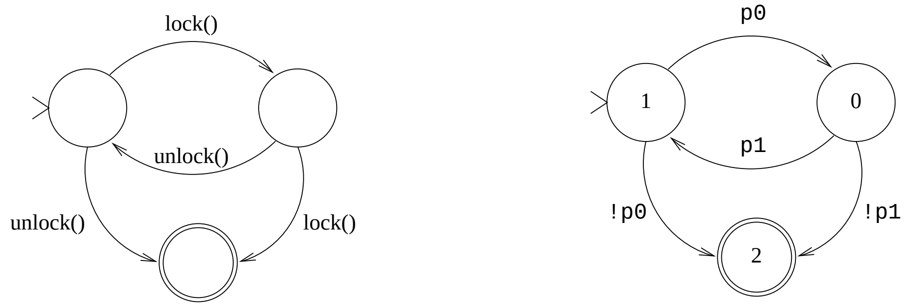

Figure 6.1: Proper use of a lock

This behavior has already been handwritten into an aspect in Fig. 4.2. Now the tool is used to automatically generate an aspect from an interface checker declaration. Clearly in the terminology of the LIME specification language the guards on the transitions translate into call propositions in the checker. Therefore a `Lock` interface is written as follows:

```
package example_lock;

import fi.hut.ics.aspectmonitor.annotation.InterfaceCheckers;

@InterfaceCheckers(
  regexpCheckers = {
    "StrictAlternation ::= (lock() ; unlock())*"
  }
)
public interface Lock {
  public void lock();
  public void unlock();
```

```
}
```

In Fig. 6.1 the automaton on the right illustrates the automaton synthesized in generated safety aspect (Fig. 6.2). In the aspect `p0` corresponds to call to `unlock()` and `p1` corresponds to `lock()`. These values are passed from the advice that capture calls to `refreshState(boolean p0, boolean p1)` method performing the state transition. The guards for transitions are represented as disjunctions of truth assignments in which the transition is enabled, and contain some redundancy.

Now consider the following program contains an erroneous use of the `Lock` interface. Two implementations of the interface are instantiated both having their own instance of `IFCheckerLockStrictAlternation` aspect monitoring method calls to them. On the ninth line of the `main` method there is a second consecutive call to `unlock()` which is by this specification considered an error.

```
package example_lock;

public class Main {
  public static void main(String[] args) {
    Lock lock = new LockImpl();
    Lock lock2 = new LockImpl();
    lock.lock();
    lock.unlock();
    lock2.lock();
    lock.unlock(); // Line 9
  }
}
```

After the safety aspect corresponding to the interface checker has been generated by the tool. It can be woven into the program to monitor its execution. Indeed running the resulting program yields an exception on the line 9 which informs that StrictAlternation property has been violated:

```
Exception in thread "main" fi.hut.ics.aspectmonitor.CheckerException:
StrictAlteration
at example_lock.Main.unlock_aroundBody7$advice(Main.java:120)
at example_lock.Main.main(Main.java:9)
```

6.2 A LIBRARY SPECIFICATION FOR A FILE INTERFACE

The following library specification experiment is based on Example 8 given on page 13 and demonstrates the history variable mechanism in practice. In this scenario, the `LogFile` interface is specified to ensure that each `write` increments its size. The required interface specification, i.e., that `write` is not called with null argument, to ensure as well as the redundant methods to the library specification have been left out from this experiment.

```
package example_lock;
/**Generated from property:
StrictAlteration ::= (lock() ; unlock())*
Which is in parsed form
  (([CALL]:lock) ; ([CALL]:unlock))*
And internally represented as
  (([CALL]:lock) ; ([CALL]:unlock))*
*/
privileged aspect IFCheckerLockStrictAlteration pertarget(
(call(public void example_lock.Lock+.lock(..)) ||
call(public void example_lock.Lock+.unlock(..)))) {
declare precedence: IFChecker*, LSChecker*;
private int state;
public IFCheckerLockStrictAlteration() {
state = 1;
}
void around() : call(public void example_lock.Lock+.unlock(..)) {
boolean p0 = true; // [CALL]: unlock
boolean p1 = false; // [CALL]: lock
refreshState(p0, p1);
if(state == 2)
throw
  new fi.hut.ics.aspectmonitor.CheckerException("StrictAlteration");
proceed();
}
void around() : call(public void example_lock.Lock+.lock(..)) {
boolean p0 = false; // [CALL]: unlock
boolean p1 = true; // [CALL]: lock
refreshState(p0, p1);
if(state == 2)
throw
  new fi.hut.ics.aspectmonitor.CheckerException("StrictAlteration");
proceed();
}
private void refreshState(boolean p0, boolean p1) {
switch(state) {
case 0 :
if((!p0 && !p1) || (!p0 && p1)) state = 2;
if((p0 && !p1) || (p0 && p1)) state = 1;
break;
case 1 :
if((!p0 && !p1) || (p0 && !p1)) state = 2;
if((!p0 && p1) || (p0 && p1)) state = 0;
break;
case 2 :
break;
}
}
}
```

Figure 6.2: A generated interface checker aspect

```
package example_file;

import fi.hut.ics.aspectmonitor.annotation.LibraryCheckers;
import fi.hut.ics.aspectmonitor.annotation.TriggeredCheckers;

@LibraryCheckers(
  pltlCheckers = {
    "ProperWrites ::= "+
      "G(<{ #pre(#s.length() + #this.length()) "+
      "== #this.length() }>)"
  }
)
public interface LogFile {
  @TriggeredCheckers(
    checkers = { "ProperWrites" },
    exception = RuntimeException.class
  )
  public void write(String s);
  public int length();
}
```

Fig. 6.3 shows the safety aspect that has been generated from the specification with the tool. The corresponding state automaton has exactly two states – one for correct behavior and one for an error. The call target, i.e., the LogFile instance and the string `s` given to the `write(String s)` method as parameter are passed to the advice from the join point's context. The history variable `pre0` will hold the sum of argument's length and the length of the `LogFile` instance prior to the method call. Now `pre0` can be referenced when making the assertion after the actual method call.

Recall that the specification language allows only primitive data to be stored in history variables. If mutable objects, that are stored by reference, could be used in this the method body could alter their values and thus make the assertion invalid.

Another observation that can be made from this experiment concerns the default triggering policy (see Def. 19 on page 15). Even though the checker declaration references `length()` method in the `LogFile` interface, the calls to it do not trigger the checker. This is because the reference is done through a value proposition and not through a call proposition as the default triggering policy would require.

6.3 PLTL SPECIFICATION WITH PAST TIME SUBFORMULA

This experiment demonstrates how past time subformulas are handled in PLTL specifications. The interface specification has been done for a `Car` interface which asserts that `ignite()` has been called at least once before `start()` is called. What makes this experiment relevant is the past time operator "once" in the specification.

```
package example_past;
```

```
package example_file;
/**Generated from property:
ProperWrites ::=
  G(<{ #pre(#s.length() + #this.length()) == #this.length() }>)
Which is in parsed form
  (G ([VALUE]:#pre(#s.length() + #this.length())
  == #this.length()))
And internally represented as
  (G ([VALUE]:#pre(#s.length() + #this.length())
  == #this.length()))
*/
privileged aspect LSCheckerLogFileProperWrites pertarget(
call(public void example_file.LogFile+.write(..))) {
declare precedence: IFChecker*, LSChecker*;
private int state;
public LSCheckerLogFileProperWrites() {
state = 0;
}
void around(
java.lang.String s,
example_file.LogFile callTarget) :
((call(public void example_file.LogFile+.write(..)) &&
args(s)) && target(callTarget)) {
int pre0 = (s.length() + callTarget.length());
proceed(s, callTarget);
boolean p0 = pre0 == callTarget.length();
// [VALUE]: #pre(#s.length() + #this.length()) == #this.length()
refreshState(p0);
if(state == 1)
throw new java.lang.RuntimeException("ProperWrites");
}
private void refreshState(boolean p0) {
switch(state) {
case 1:
break;
case 0:
if(p0) state = 0;
if((!(p0))) state = 1;
break;
}
}
}
```

Figure 6.3: A generated library checker aspect

```
import fi.hut.ics.aspectmonitor.annotation.InterfaceCheckers;
import fi.hut.ics.aspectmonitor.annotation.TriggeredCheckers;

@InterfaceCheckers(
  pltlCheckers = {
   "ProperStarts ::= G (start() -> O(ignite()))"
  }
)
public interface Car {
  @TriggeredCheckers(
    checkers = {"ProperStarts"},
    exception = RuntimeException.class
  )
  public void start();
  @TriggeredCheckers(
    checkers = {"ProperStarts"},
    exception = RuntimeException.class
  )
  public void ignite();
}
```

Fig. 6.4 shows the aspect generated from the specification. The boolean variable `p0` corresponds to the call proposition `start()`, and the boolean variable `p1` to the call proposition `ignite()` in the aspect code. This is very much same as in the Lock interface experiment in Sec. 6.1. In this case, however, there is an additional array `nowProperStarts` corresponding the current values of past time subformulas in the property, and `preProperStarts` corresponding to the prior values of those subformulas. Since there there is only one such a subformula `O(ignite())` the arrays hold only one element each.

Recall the initialization and update rules from Table 5.1 on page 33 that allows the past time subformulas be treated as if they were one of the atomic propositions. The initial value for `preProperStarts[0]` is assigned in the aspect constructor according to this technique to be false. The update rules are applied before the guards of the transitions are checked. Finally, the current value of the `O(ignite())`, i.e., `nowProperStarts[0]` appears now as a proposition in the transitions of the automaton generated by SCheck in `refreshState(boolean p0, boolean p1)` method.

```
package example_past;
/**Generated from property:
ProperStarts ::= G (start() -> O(ignite()))
Which is in parsed form
  (G (([CALL]:start) -> (O ([CALL]:ignite))))
And internally represented as
  (G (([CALL]:start) -> (O ([CALL]:ignite))))
*/
privileged aspect IFCheckerCarProperStarts pertarget(
(call(public void example_past.Car+.start(..)) ||
call(public void example_past.Car+.ignite(..)))) {
declare precedence: IFChecker*, LSChecker*;
private boolean[] nowProperStarts;
private boolean[] preProperStarts;
private int state;
public IFCheckerCarProperStarts() {
preProperStarts = new boolean[1];
nowProperStarts = new boolean[1];
state = 0;
preProperStarts[0] = false; // ONCE
}
void around() : call(public void example_past.Car+.start(..)) {
boolean p0 = true; // [CALL]: start
boolean p1 = false; // [CALL]: ignite
refreshState(p0, p1);
if(state == 1)
throw new java.lang.RuntimeException("ProperStarts");
proceed();
}
void around() : call(public void example_past.Car+.ignite(..)) {
boolean p0 = false; // [CALL]: start
boolean p1 = true; // [CALL]: ignite
refreshState(p0, p1);
if(state == 1)
throw new java.lang.RuntimeException("ProperStarts");
proceed();
}
private void refreshState(boolean p0, boolean p1) {
nowProperStarts[0] = p1 || preProperStarts[0]; // ONCE
preProperStarts[0] = nowProperStarts[0];
switch(state) {
case 1:
break;
case 0:
if(((nowProperStarts[0] && p0) || (!(p0)))) state = 0;
if(((!(nowProperStarts[0])) && p0)) state = 1;
break;
}
}
}
```

Figure 6.4: A generated PLTL interface checker with a past subformula

7 CONCLUSIONS AND FUTURE WORK

The purpose of the chapter is to summarize the work what have presented in this report, and discuss the future of the specification language.

7.1 CONCLUSIONS

This report has presented an interface specification language which is based on two complementary formalisms – PLTL and regular expressions. It allows the specification of safety properties for both internal behavior as well as the external usage of a software component. It is possible to specify behavior that is impossible to be fulfilled by any implementation, and therefore specifications themselves should be verified. The specifications may contain claims about the currently executing method and the arguments it has received, and about the internal state of a component. There is currently a limited support for data handling in the language that is called the history variable mechanism. History variables allow propositions to reference input values and relate them to expected results.

The report has shown that the specification language can be used in the context of Java programming language by annotating types with the specifications. A tool implementation has also been presented which can be used to analyze the annotated specifications and then to synthesize them into AspectJ aspects. Aspects can be used for runtime verification of the system, which is a complementary approach for the traditional methods of quality assurance. The tool has been implemented as a Spoonlet and can be integrated into a popular development environment, Eclipse, and into a widely used build tool, Maven 2, with respective plug-ins.

Aspect-oriented programming has proven to be very well suited for establishing behavioral invariants into the runtime execution of the system under test. Generation of aspect source code has also proven to be a good approach compared to, for example, abstract syntax tree instrumentation of the program itself. This approach separates the generated code from the actual implementation in a way that both can be examine separately, for example, in inspection sessions.

The report contributes the use of a combination of future and past time operators in specifications. This has been achieved by combining the approach for handling formulas of the form "always P" where P is a past time formula presented in [14, 15] with the SCheck tool presented in [28].

The complementary formalisms – regular expressions and PLTL formulas have proven to be a good choice. Regular expressions are very good for expressing pattern behavior whereas PLTL allows the use of complement without blow up in the resulting automaton. Tool integration with immediate feedback for the programmer about, for example, syntactically incorrect specifications can also be considered to be an asset for the tool.

7.2 FUTURE WORK

The previous section concluded the current state of the specification language and of the tool that supports it. The purpose in this section is to discuss what has been planned for the future of the language.

One of the areas in which the specification language itself could be improved is data handling. More sophisticated ways than the simple history variable mechanism have been discussed. One approach to do this is to allow values to be stored over method calls in the checker aspects. Then, for example, the return value of one method could be used to specify how another method should used. This would add a level of dynamic behavior into the specifications. In languages such as Java where exceptions are commonly used, propositions about them could be added to the language. By doing this one could specify, for example, that certain kinds of exceptions are allowed in the interaction while others are not.

More elaborate models of interaction could also be considered. The current model of interaction presented in Fig. 1.1 on page 3 assumes a singleton application initiating the interaction with one or more libraries. The crosscutting nature of aspect-oriented programming could allow the state of the protocol behavior be established between pairs of interacting components. The more elaborate model of interaction could also take into account concurrency. This could be approached by giving each thread its own copy of the `state` variable in the aspects (with `java.lang.ThreadLocal<T>` state variable), and taking care of the synchronization issues arising from concurrency.

The tool implementation could be extended to support the full safety subset of PLTL which does not add expressive power to the language but might make some specification easier and more natural to express, and could also be of theoretical interest. The detection of pathologically safe formula presented in [28] could also be integrated into the tool, as well as other aids for error detection in the specification process.

In the immediate future, efforts will be made to integrate the tool presented in this report to the LIME project's automatic test generator. Also logging strategies for test runs will be considered to better capture what exactly went wrong in the interaction. The logging mechanism itself can be probably implemented with an aspect. As LIME is a project for embedded systems the specification language will be adjusted to be better suited for the C programming language, and with the lessons learned from this work a tool to support it will also be made.

It is has also been planned that the interface specification methods are to be extend for systems that are not at all or only partially implemented. This would allow *stub generation*, i.e., mock-up implementations of specified interfaces could created automatically. These generated components that approximate the behavior of a real library could be used as testing aids for applications while the actual library is being implemented.

Similarly mock-up implementations of applications could be created for libraries that would approximate behavior of a real application using the library to test it.

ACKNOWLEDGEMENTS

This report has been done in the Department of Information and Computer Science at the Helsinki University of Technology (TKK) as part of the LIME project (LightweIght formal Methods for distributed component-based Embedded systems).

I would like to thank Tekes, the Finnish Funding Agency for Technology and Innovation, for funding in the LIME project, and Technology Industries of Finland Centennial Foundation for funding prior to LIME.

I would want thank the my supervisor Prof. Ilkka Niemelä for the opportunity to work here at TKK and for his comments and support throughout the project. Also I would wish to express gratitude to my instructor Docent Keijo Heljanko for his time, patience and advises that guided me through this project.

My gratitude also goes for our industrial partners in the LIME project Conformiq Software Oy, Elektrobit Oy, Nokia Oyj, and Space Systems Finland, as well as our research partners at Åbo Akademi and my colleagues at TKK. I wish everyone continued success.

Finally, I dedicate this report for my mom and dad for always being there for me when I have needed it the most.

BIBLIOGRAPHY

[1] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, New York, NY, USA, 2006. ACM.

[3] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5), 2006. (doi: 10.2168/LMCS-2(5:5)2006).

[4] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[5] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.*, 89(2):106–125, 2003. In Proc. of RV'03: the Third International Workshop on Runtime Verification.

[6] Alessandro Cimatti, Marco Roveri, Simone Semprini, and Stefano Tonetta. From PSL to NBA: a modular symbolic encoding. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 125–133, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In Manfred Broy, Johannes Gruenbauer, David Harel, and C.A.R. Hoare, editors, *Engineering Theories of Software-Intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[8] Jori Dubrovin and Tommi Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report B23, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2007.

[9] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[14] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS 2002: Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[15] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.

[16] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[18] Antti Huima. Implementing Conformiq Qtronic. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[19] IEEE-Commission. IEEE standard for property specification language (PSL). Technical report, IEEE, 2005. IEEE Std 1850-2005.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[22] Kimmo Kiviluoma, Johannes Koskinen, and Tommi Mikkonen. Runtime monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 181–190, New York, NY, USA, 2006. ACM.

[23] Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. Temporal logic for scenario-based specifications. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS 2005: Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2005.

[24] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, 2001.

[25] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[26] Leslie Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM.

[27] Timo Latvala. On model checking safety properties. Research Report A76, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2002.

[28] Timo Latvala. Efficient model checking of safety properties. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[29] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[30] Zohar Manna and Amir Pnueli. Tools and rules for the practicing verifier. In *Perspectives on Computer Science, ed. by Anita K. Jones, Academic Press, New York, 1977; CMU Computer Science: A 25th Anniversary Commemorative, Richard F. Rashid (Ed.), ACM Press and Addison-Wesley Publishing Co.* 1991.

[31] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[32] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 219–230, New York, NY, USA, 2006. ACM.

[33] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, INRIA, 2006.

[34] Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 821–826, New York, NY, USA, 2006. ACM.

[35] Grigore Roşu. An effective algorithm for the membership problem for extended regular expressions. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'07)*, volume 4423 of *Lecture Notes in Computer Science*, pages 332–345. Springer, 2007.

[36] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, July 2004.

[37] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[38] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *Proceedings of the Fifth Workshop on Runtime Verification (RV 2005), Satellite workshop of CAV 2005, Edinburgh, Scotland, UK*, volume 144:4 of *Electronic Notes in Theoretical Computer Science*, pages 109–124. "Elsevier", 2006.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R1	Nikolaj Tatti, Hannes Heikinheimo Decomposable Families of Itemsets. May 2008.
TKK-ICS-R2	Ville Viitaniemi, Jorma Laaksonen Evaluation of Techniques for Image Classification, Object Detection and Object Segmentation. June 2008.
TKK-ICS-R3	Jussi Lahtinen Model Checking Timed Safety Instrumented Systems. June 2008.

ISBN 978-951-22-9453-4 (Print)
ISBN 978-951-22-9454-1 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)