TKK Reports in Information and Computer Science
Espoo 2009 TKK-ICS-R25

INTERFACE SPECIFICATION METHODS FOR SOFTWARE COMPONENTS

Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, and Keijo Heljanko

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

TKK Reports in Information and Computer Science
Espoo 2009 TKK-ICS-R25

INTERFACE SPECIFICATION METHODS FOR SOFTWARE COMPONENTS

Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, and Keijo Heljanko

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 9 470 01
Fax +358 9 470 23369
E-mail: series@ics.tkk.fi

ISBN 978-952-248-278-5 (Print)
ISBN 978-952-248-279-2 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2009/isbn9789522482792.pdf

TKK ICS
Espoo 2009

Abstract: This work presents an interface specification language developed as a part of the LIME-project (LightweIght formal methods for distributed component-based Embedded systems). The intention is to provide a mechanism for specifying both external usage of a software component, as well as the internal behavior of a one. The described methodology is considered lightweight because there is no assumption of a complete model of a software component or its interface. The presented approach is an incremental description of properties that are at least expected to hold. The described approach can also be applied to a component which is already (completely or partially) implemented.

Keywords: Interface specification, lightweight methods

Acknowledgement: Work financially supported by Tekes - Finnish Funding Agency for Technology and Innovation, Conformiq Software, Elektrobit, Nokia, Space Systems Finland, Academy of Finland (projects 112016,126860,128050), and Technology Industries of Finland Centennial Foundation.

Contents

1 GOAL

Component-based design and verification of distributed embedded systems is a very hard task to accomplish with traditional development methods. The work hypothesis in the LIME project 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. The reader is requested to consult [10] for a more concise presentation of the LIME interface specification language and the motivation behind its development. The automated Java testing tool also developed in the LIME project is documented in [9].

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 [5]. This correctness requirement in the interaction can be extended to cover 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 communication 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. The model of communication is divided into two parts – to an *call specification* (CS in Fig. 1) which specifies how a component should be used, and to a *return specification* (RS in Fig. 1) which specifies how the component should respond. Should the call specification be breached, the calling component is incorrect, and if the called component does not obey its specification, it will be the one to take the blame.

The specification language combines three complementary ways for expressing the proper behavior of software objects – regular expressions, nondeterministic finite automata (NFA) and Linear Temporal Logic with Past (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 protocol aspect (call ordering) of interfaces, where as the latter describes how the called component should behave.

The interpretation of PLTL [3] for runtime monitoring can be seen as a continuation of the work in [7] and [8]. The language employed is extended in this work to also contain future time operators using the SCheck tool [11].

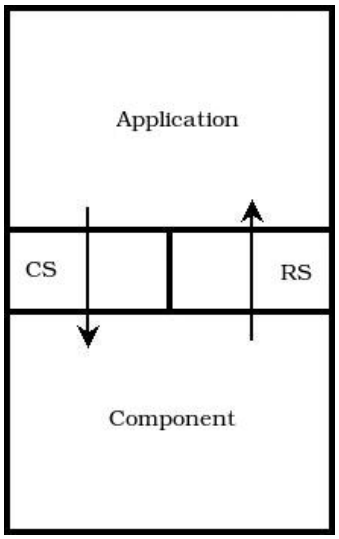

Figure 1: The interaction model

2 THEORETICAL BACKGROUND

The purpose of this section 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, NFAs and PLTL, are presented in the following subsections, and their interpretation in software is discussed in the subsequent sections. 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 [4], 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., [14]. 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 [12].

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 $w \in \Sigma^*$ [12].*

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 [7].*

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

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 $L : S \to \Sigma$ with $\Sigma = 2^{AP}$.*

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

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

Example 2 - Semantics of PLTL formulas in a finite path

Fig. 2 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 $0 \leq n \leq 3$.
■

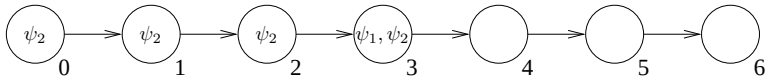

Figure 2: Semantics of $\psi_1 \mathbf{R} \psi_2$ in a finite path

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

Nondeterministic finite automata (NFAs) are equivalent to regular expressions in their power to express all regular languages. The main motivation for including also NFAs in the LIME interface specification language is to ease the integration of other tools generating interface specifications from other more high-level specification languages without the need to go through regular expressions incurring a potential blow-up.

Definition 15 A (nondeterministic and finite) automaton A *is a tuple* $(\Sigma, S, S^0, \Delta, F)$*, where*

- Σ *is a finite* alphabet*,*
- S *is a finite set of* states*,*
- $S^0 \subseteq S$ *is the set of* initial states*,*
- $\Delta \subseteq S \times \Sigma \times S$ *is the* transition relation *(no ϵ-transitions allowed), and*
- $F \subseteq S$ *is the set of* accepting states*.*

A finite automaton A accepts a set of words $L(A) \subseteq \Sigma^*$ called the *language* accepted by A, defined as follows:

- A *run* r of A on a finite word $a_0, \ldots, a_{n-1} \in \Sigma^*$ is a sequence $s_0, \ldots, s_n$ of $(n+1)$ states in S, such that $s_0 \in S^0$, and $(s_i, a_i, s_{i+1}) \in \Delta$ for all $0 \leq i < n$.
- The run r is *accepting* iff $s_n \in F$. A word $w \in \Sigma^*$ is accepted by A iff A has an accepting run on w.

3 THE SPECIFICATION LANGUAGE

The purpose of this section is to introduce the reader to how call and return specifications can be written and to define the mechanisms and policies used in the language. In Sect. 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 Sect. 3.2 the correctness requirements that the specifications impose are defined, and the policies and mechanisms of the language are discussed in detail. Note that in the LIME interface specification language all specification forms: regular expressions, PLTL formulas, and also NFAs specify desired properties of the interface in question instead of specifying the undesired behaviors of the interface.

3.1 Specifying interfaces and components – Running examples

Example 4 demonstrates how regular expressions can be used for defining interface behavior.

Example 4 - 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 specification to enforce its expected use. The specification is monitored at runtime to keep 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 1 for the rest of the annotations). It is necessary to tie the specification into events (method calls) in the interface. This is done by annotating the desired methods to observe the corresponding specification either when body of the method is entered, or when it is exited depending of its type. Call specifications will be observed on entry and return specifications on exit of the method.

```
@CallSpecifications(
  callPropositions = {
    "open ::= open()",
    "close ::= close()",
    "read ::= read()",
    "write ::= write()"
  },
  regexp = {
    "FileUsage ::= (open ; (read | write)* ; close)*"
  }
)
public interface LogFile {
  @Observe(specs = {"FileUsage"})
  public void open();
  @Observe(specs = {"FileUsage"})
  public void close();
  @Observe(specs = {"FileUsage"})
  public String read();
  @Observe(specs = {"FileUsage"})
  public void write(String s);
  public int length();
}
```

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

Example 5 introduces default policies that conform to the natural interpretation of the specifications making their declaration less verbose.

Example 5 - Less verbose specifications with default observers

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 specification declaration means a call proposition, which true iff `open()` is being executed (see Def. 19). This happens

when a specification is observed on entry or on exit depending on the specification type of the `open()` procedure. Secondly, if a specification contains a call proposition the specification is automatically observed 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:

```
@CallSpecifications(
  regexp = {
   "FileUsage ::= (open() ; (read() | write())* ; close())*"
  }
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String s);
  public int length();
}
```

■

Example 6 introduces PLTL as a specification formalism. PLTL specifications are sometimes more succinct and natural way of describing the desired behavior than regular expressions.

Example 6 - Extending call 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 specification. Now, the property is expressed in a PLTL specification (see annotation details in Table 2) which states "always when write is called, it receives a proper String". With the adopted default enforcement policy, the specification is automatically observed on entry of the write procedure. In the example `s` 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).

```
@CallSpecifications(
  regexp = {
    "FileUsage ::= (open() ; (read() | write())* ; close())*"
  },
  valuePropositions = {
    "properData ::= (#s != null)"
  },
  pltl = {
    "ProperData ::= G (write() -> properData)"
  }
```

```
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String s);
  public int length();
}
```

■

Example 7 combines call specifications with return specifications. Data handling is introduced as a new feature in the return specification.

Example 7 - A return 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 `s` argument.

```
@CallSpecifications(
  regexp = {
    "FileUsage ::= (open() ; (read() | write())* ; close())*"
  },
  valuePropositions = { "properData ::= (#s != null)" },
  pltl = { "ProperData ::= G (write() -> properData)" }
)
@ReturnSpecifications(
  valuePropositions = {
    "okLength ::= "+
      "#this.length() == #pre(#this.length() + #s.length())"
  },
  pltl = { "ProperWrites ::= G (write() -> okLength)" }
)
public interface LogFile {
  public void open();
  public void close();
  public String read();
  public void write(String s);
  public int 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)`.

■

Example 8 shows how behavior can be specified as a finite automaton.

Example 8 - Specifying legal behavior as a finite automaton

The syntax of automaton specifications is illustrated in this example. For additional information on the syntax, see Appendix A. The syntax consists of two parts. In the first part all atomic propositions have to be declared and given names. In the second part the automaton specifications are listed. Each NFA specification is declared using the keyword `always_nfa` to underline the fact that the NFA specifies the good behavior of the interface in question. The NFA itself is given in a syntax closely following the "neverclaim" syntax of the Spin model checker. The only change is that all states are by default accepting, and you have to add a state label prefix `reject_` in order to declare a non-accepting state. The runtime monitor will report a violation if the execution observed so far is not in the language of the specified NFA. To do this, the tool internally determinizes and complements the automaton specification provided by the user.

```
@CallSpecifications(
  callPropositions = {
    "start ::= start()",
    "ignite ::= ignite()"
  },
  nfa = {
    "ProperStartsA ::= " +
    "  always_nfa {" +
    "    state1_init: " +
    "      if " +
    "        :: (start) -> goto reject_state2; " +
    "        :: (ignite) -> goto state3; " +
    "      fi; " +
    "    reject_state2: " +
    "      if " +
    "        :: (1) -> goto reject_state2; " +
    "      fi; " +
    "    state3: " +
    "      if " +
    "        :: (1) -> goto state3; " +
    "      fi; " +
    "  }"
  }
)
public interface Car {
  @Observe(
    specs = {"ProperStartsA"})
  public void start();

  @Observe(
    specs = {"ProperStartsA"})
  public void ignite();
}
```

■

Notice the slightly cumbersome way of dealing with multiline strings in Java. Another thing to observe is that the NFA specified by the user should specify a prefix closed language but this is easily achieved by, e.g., not using any reject states at all.

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 three 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.
   - `exceptionPropositions` – Claims about thrown exceptions (e.g., RuntimeException has been thrown by a method). Specifically, they are propositions available in return specifications that are true if and only if the observed method threw a specific exception. Exception propositions are not supported in call specifications.
2. A mechanism to combine propositions to describe expected properties of a software components. These are referred to as specifications and subdivided to classes according to the underlying formalism.
   - `regexp` – specifications expressed with regular expressions.
   - `pltl` – specifications expressed with PLTL.
   - `nfa` – specifications expressed with NFAs.
3. A mechanism to tie specification to the program flow. This is referred to as observing a specification. A specification can be observed 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. 16.

Definition 16 *Let $\pi = s_0 s_1 \ldots s_{k-1}$ denote the program trace observed so far. The 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 11. Let $pref(L)$ denote prefix closure of L, i.e., the union of prefixes of all the words in L. Def. 17 formalizes the breach of regular expression specification.

Definition 17 *Let $\pi = s_0 s_1 \ldots s_{k-1}$ denote the program trace observed so far, r be a regular expression specified in a regular expression checker, and $L = \mathcal{L}(r)$ be the language it accepts. The regular expression specification r is broken iff $\pi \notin pref(L)$.*

Definition 18 *Let $\pi = s_0 s_1 \ldots s_{k-1}$ denote the program trace observed by the checker so far, A be the NFA specified in an nfa specification. The nfa specification A is broken iff $\pi \notin L(A)$.*

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

Policies and notation

The considered interaction model (see Fig. 1 on page 8) suggests that there are two kinds, call and return, specifications to consider. From the specification language standpoint, the two are very similar, yet not the same. In the call specifications the return values are not under consideration, but rather the call orderings and argument values. In the return specifications, however, the typical specification does make claims about return values. The default observing policies (Def. 19) reflect this.

Definition 19 *If a method is mentioned in a specification through a call proposition, the specification is implicitly enforced in the corresponding method. The specification is observed on entry if the specification is an call specification, and on exit if the specification is a return specification.*

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 specification definitions.

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

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

It is possible to write a specification that contains a value proposition which is not defined in all methods in which it is observed. This may happen, for example, when one of the methods takes an argument when others do not. One could observe a specification that states `G (write()`

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

Table 1: Regular expressions and their corresponding annotations

`-> <{ #s != null }>`) (where `#s` is the argument of `write(String s)`) in `read()` method which takes no arguments and therefore does not know the value of `#s`. 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 specification contains a value proposition which is not defined in some method it is observed in, the undefined propositions are defined to be* false. *If the value proposition is not defined in any method it is observed in, this is considered to be an error.*

Propositions, named or inlined, are combined into PLTL formulas, NFAs or regular expressions in the specification. Regular expressions, as defined in Def. 4 can be expressed with annotations given in Table 1. Note that propositional formulas can appear in the regular expressions and their corresponding annotations can be found in Table 2. Similarly, corresponding annotations for PLTL are presented in Table 2. When writing PLTL specifications precedence rules presented in Table 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 write the specifications 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)`.

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`	$\bot$	`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 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: Precedence of logic operators

Observing specifications

As the incremental approach for interface specification suggests, all the created specifications 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 specifications 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 observed. This has a 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 specification is observed in a particular object instance.

Fig. 3 illustrates how time is perceived to pass by `FileUsage` (a call specification) and `ProperWrites` (a return specification) specifications of the earlier examples (see Sect. 3.1) over a sequence of method invocations through the interface. The runtime observer of `FileUsage` is ran *before* executing the bodies of `open()`, `read()`, `write()` or `close()` where as `ProperWrites` observer executes *after* the body of `write()`.

Data handling

Return specifications, and the observers 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. 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 Sect. 3.1. There we specified a proposition `okLength` to be equivalent to expression `#this.length()`

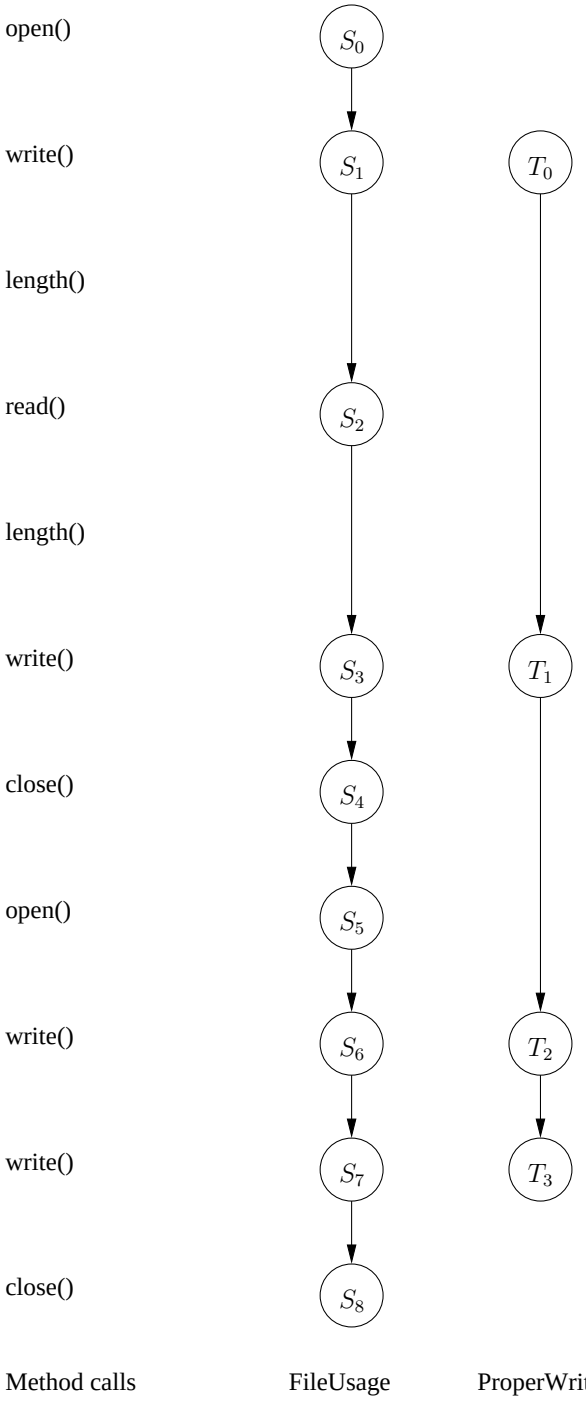

Figure 3: Time, as observed by two different specifications

== #pre(#this.length() + #s.length()). This proposition was used in return specification ProperWrites which stated G (write() -> ok-Length). 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, the storing of prevalues to the history variables does not induce a new state. Thus the previous time step for ProperWrites specification 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 specification observer.

Exceptions

Exception propositions can (and to be useful, need to) be used in PLTL, regexp or automaton specifications. For example, the following PLTL specification asserts that no RuntimeException is thrown:

```
@ReturnSpecifications(
    exceptionPropositions = {
        "exc ::= java.lang.RuntimeException"
    },
    pltl = {
        "NoException ::= G(!exc)"
    }
)
```

Now this can be used to check a method in a class:

```
@ReturnSpecifications(
    exceptionPropositions = {
         "exc ::= java.lang.RuntimeException"
    },
    pltl = {
        "ProperWrites ::= " +
        "G(<{ #pre(#s.length() + #this.length()) == " +
            "#this.length() }>)",
        "NoException ::= G(!exc)"
    }
)
public interface LogFile {
    @Observe( specs = { "ProperWrites", "NoException" } )
    public void write(String s);
    ...
}
```

When an exception is thrown, there is no return value. Thus we define any native language proposition which refers to #result to be false in the case an exception is thrown.

The kind of exception thrown when a violation of a call or return specification is detected can be specified. This may help in automatically running test code.

By default, CallSpecification violations cause an CallSpecificationException to be thrown and ReturnSpecification violations cause a ReturnSpecificationException to be thrown. These both are subclasses of SpecificationException in aspectmonitor package.

For specifying the kinds of exceptions thrown, two new fields in @Observe are defined: callException and returnException. This example defines that on a violation of the call property "ProperStartsA", a RuntimeException is thrown instead of the default CallSpecificationException:

```
public class Car {
    @Observe(
        specs = {"ProperStartsA"},
        callException = RuntimeException.class)
    public void start() {}

    @Observe(
        specs = {"ProperStartsA"},
        callException = RuntimeException.class)
    public void ignite() {}
}
```

3.3 The specification language with C

Most of the above applies as well to the C version of the specification language. The most notable differences are the C's lack of Java-like interfaces the the workaround to that, the lack of Exceptions and thus Exception propositions, and the addition of observer creation and deletion macros.

From interfaces to headers

Maybe the most notable difference to the Java version of the specification language from the users point of view is that there are no Interfaces to write specifications to. To work around this feature of C, we have decided the most natural place to write the interface specifications would be header files (.h files). This was chosen because header files usually already hold the "structure" of the program in the form of function prototypes, and thus is the natural place to add additional information about how these functions should be used.

Also the syntax of the annotations has been changed a bit to make it work with C. Instead of using just:

```
@CallSpecifications(
    pltl = { ... }
)
```

```
@ReturnSpecifications(
    regexp = { ... }
)
```

the user should put all the interface specifications inside special comment tags: `/*@` and `@*/`, like in the following example:

```
/*@ CallSpecifications(
    pltl = { ... }
) @*/

/*@ ReturnSpecifications(
    regexp = { ... }
) @*/
```

Also, the problem with Java multiline literals (mostly apparent on nfa specifications) doesn't exist with C, so:

```
nfa = {
    "ProperStartsA ::= " +
    "  always_nfa {" +
    "    state1_init: " +
    "      if " +
    "        :: (start) -> goto reject_state2; " +
    "        :: (ignite) -> goto state3; " +
    "      fi; " +
    "    reject_state2: " +
    "      if " +
    "        :: (1) -> goto reject_state2; " +
    "      fi; " +
    "    state3: " +
    "      if " +
    "        :: (1) -> goto state3; " +
    "      fi; " +
    "  }"
}
```

can be written simply as:

```
nfa = {
    "ProperStartsA ::=
      always_nfa {
        state1_init:
          if
            :: (start) -> goto reject_state2;
            :: (ignite) -> goto state3;
          fi;
        reject_state2:
          if
```

```
            :: (1) -> goto reject_state2;
          fi;
        state3:
          if
            :: (1) -> goto state3;
          fi;
      }"
}
```

One additional thing to note is that the interface specifications should be written at the relative beginning of the header file for implementation related reasons. This is because the tool used to read the specifications (Doxygen in C's case) associates them to the following variable or function prototype, and if there isn't one, the specification is lost in the process. So in short, the header file where the interface specifications reside should always end with something else than an interface specification.

Changed language features

As mentioned above, the specification language for C is pretty close to the Java version, with only a couple of changes.

The first of these changes is the removal of Exception propositions. This is only natural, since C doesn't really have exceptions in the first place, so there shouldn't really be a need to specify anything about them.

The second change isn't quite as straightforward. Sometimes the user might need to have several copies of the same observer. For example, when specifying the usage of a lock interface, it seems only natural to have a separate observer instance for each of the locks under observation. However, the lack of objects in C prevents us from doing this automatically, and the user has to provide additional specifications when multiple observer instances are needed. This can be done via a new annotation: `/*@ Instance( instance = { "any_c_expression"} ) @*/`

This annotation should be written in the header file immediately before the function what needs to be aware of multiple observer instances. The `any_c_expression` should somehow refer to the arguments of the annotated function, and should return the same value (when casted to unsigned long) as the identifier given when creating each observer instance (explained below).

An example header file which uses the Instance annotation with a simple lock implementation follows:

```
typedef struct lock {
  int locked;
} lock_t;

/*@ CallSpecifications(
    callPropositions = {
        "lock ::= lock()",
        "unlock ::= unlock()"
    },
    regexp = {"alternation ::= (lock ; unlock)*"}
```

```
) @*/

/*@ Instance(instance = {"foo"}) @*/
void lock(lock_t *foo);
/*@ Instance(instance = {"bar"}) @*/
void unlock(lock_t *bar);
```

Observer creation and deletion

Another thing to note when using the C version of the specification language is that observers can't automatically be created and destroyed along with the object under observation, since as mentioned above, C isn't an object oriented language. Also the creation and usage of several instances of the same observer, like in the example above, isn't really possible. To work around these issues, couple of macros are provided for the user to create observers when they are needed, and to free them later.

These macros are as follows:

```
_LIME_create_obs(char *observer)
_LIME_create_obs_inst(char *observer, unsigned long identifier)
_LIME_free_obs(char *observer)
_LIME_free_obs_inst(char *observer, unsigned long identifier)
```

The usage of the macros is pretty straightforward, the user adds the `_LIME_create_obs()` macro to the source code whenever some specification needs to be observed. The name of the specification should be provided via the `observer` argument. If the user feels the observation of the specification is no longer necessary, the observer can be freed with the corresponding `_LIME_free_obs()` macro.

The remaining two macros are used to create multiple instances of the same observer in situations like the lock example above. They are used similarily to observer creation macros, except for the fact that user should provide an unique `identifier` for each observer instance, which are then used to identify them later. These indentifiers are internally casted to unsigned long, so for example the memory location of a struct is a valid candidate for identifier.

These macros are defined in a provided lime.h header file, which the user should include whenever they need to be used. The macros do not generate any code unless the LIME specifications are requested by the user, to allow the compilation of the program without unnecessary errors, when the observation of the LIME specifications isn't a priority, without additional changes.

The following example of the main method for the lock header above should hopefully clarify a bit how the macros should be used. When the Instance annotations aren't used, there is no need to create observer instances, only a single observer creation macro will suffice.

```
int main() {
    _LIME_create_obs(alternation);

    lock_t *lock1 = malloc(sizeof(lock_t));
```

```
    _LIME_create_obs_inst(alternation, lock1);
    lock_t *lock2 = malloc(sizeof(lock_t));
    _LIME_create_obs_inst(alternation, lock2);

    lock(lock1);
    unlock(lock1);
    lock(lock2);
    unlock(lock1);

    _LIME_free_obs_inst(alternation, lock1);
    free(lock1);
    _LIME_free_obs_inst(alternation, lock2);
    free(lock2);

    _LIME_free_obs(alternation);
    return 0;
}
```

4 PARTIALLY IMPLEMENTED SYSTEMS

The purpose of this section is to present how the LIME interface specifications can be used with systems that have been only partially implemented. Only systems written in Java are considered in this section. The approach taken here is to close the system by using automatically generated stub code. In this section two different cases of partially implemented systems are considered. In the first case an implementation of an interface is available, but the part that uses the interface has not yet been implemented. We will refer to this case as closing the system from top. In the second case the interface implementation is missing but the part that calls the interface has been implemented. We will call this case as closing the system from bottom. These two cases will be discussed in the following subsections.

4.1 Closing the system from top

The main idea in closing a system from top is to generate a stub code implementation of the part of the system that calls a given interface. This can be done by an automated stub code generator that creates an implementation of an environment that nondeterministically calls one of the interface methods with random valued arguments. In this work, only primitive data types and strings are supported in this fashion. Other types of input objects are simply created with their respective default constructors and passed to the called method.

A stub implemention that nondeterministically calls the interface methods is likely to violate a call specification written with the LIME specification language. In order to the generated stub code to be useful in testing, the stub code will override the exceptions thrown by the observers in case of call specifications and throw a special exception instead. These

exceptions are used to indicate that the specification was violated because the stub code implementation was too coarse (i.e., it does not satify the specifications). The test runs that cause these overriden exception to be thrown are reported as inconclusive. This is natural as when the system is closed from the top, we are interesting in testing the component that implements the interface that the stub code calls and not the actual stub code. Naturally, if the return specification is violated, we know that the stub code called the interface according to the specifications but the component implementation failed to respond to the method call as specified. In these cases the specification violations are reported as normal.

As an example of closing a system from the top, let us consider the LogFile interface given bellow. Let us also assume that the interface has been implemented in FileImpl class.

```
public interface LogFile {
   public void write(String s);
   public String read();
   public int length();
   public void open(int number);
   public void close();
}
```

The stub code that will be automatically generated from this interface is shown bellow. Notice that the implementation of the interface is called nondeterministically to some user defined test depth. In this example the testing consists of five calls to the interface.

```
public class GeneratedTop {
    public static void main( String[] args ) {
        Random r = new Random();
        int testDepth = 0;
        FileImpl obj = new FileImpl();

        java.lang.String javalangString1;
        int int1;

        ExceptionOverride.setCallException(obj,
    InconclusiveException.class);

        while (testDepth < 5) {
           testDepth++;
           int i = r.nextInt(5);
           switch (i) {
                case 0: obj.length(); break;
                case 1: javalangString1 = RandomString.getString(r);
                        obj.write(javalangString1); break;
                case 2: obj.read(); break;
                case 3: obj.close(); break;
                case 4: int1 = r.nextInt(); obj.open(int1); break;
           }
        }
    }
}
```

The presented approach to close a system from the top is limited in the sense that with many interfaces and their respective call specifications calling the interface randomly leads to many inconclusive test runs. Therefore this approach is intented to be used together with a suitable test generation tool that can avoid creating inconlusive tests. Such a tool is future work and it is also being developed within the LIME project.

4.2 Closing the system from bottom

The basic idea in closing a system from bottom is the same as in the previous case. A stub code generator is used to generate an implementation of a given interface in such a way that all the implemented methods return only random values. Randomization of only primitive data types and String objects are supported.

As before, these random values can lead to violating the given return specifications and in these cases the test runs should be classified as inconclusive. To achieve this the stub code overrides the exceptions thrown by the return specification observers.

As an example, a stub code implementation of the LogFile interface discussed in the previous subsection is shown bellow.

```
public class GeneratedBottom implements LogFile {
  private Random r;

  GeneratedBottom() {
    r = new Random();
  }

  int length() {
    ExceptionOverride.setReturnException(obj,
      InconclusiveException.class);
    return r.nextInt();
  }

  void write(java.lang.String arg1) {
    ExceptionOverride.setReturnException(obj,
      InconclusiveException.class);
  }

  java.lang.String read() {
    ExceptionOverride.setReturnException(obj,
      InconclusiveException.class);
    return new java.lang.String();
  }

  void close() {
    ExceptionOverride.setReturnException(obj,
      InconclusiveException.class);
  }

  void open(int arg1) {
```

```
    ExceptionOverride.setReturnException(obj,
      InconclusiveException.class);
  }
}
```

5 THE TOOL IMPLEMENTATION

In this section 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 section (Sect. 5.1) describes the tool for its potential users, whereas the second part (Sect. 5.3) 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 specifications, and the description of annotations used.

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 Sect. 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 Sect. 3.2.

5.2 Specifications

Both call and return specifications are declared by respective annotations, `@CallSpecifications` (Fig. 4) and `@ReturnSpecifications` (Fig. 5).

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 specifications 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 specifications are 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 specifications is given in Table 1 on page 21. The syntactic rules for declaring PLTL specifications are given in Table 2 on page 21 and the precedence rules for operators in Table 3 on page 22.

```
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 CallSpecifications {
  String[] valuePropositions() default {};
  String[] callPropositions() default {};
  String[] pltl() default {};
  String[] regexp() default {};
  String[] nfa() default {};
}
```

Figure 4: Annotation for declaring call specifications

Observing specifications

A property can be enforced by observing its corresponding specification in a method. The default enforcement policy defined in Def. 19 on page 20 is employed, i.e., if a specification contains a call proposition it is automatically observed in the corresponding method. However, it is possible to explicitly observe a specification in a method. This is done with an annotation `@Observe` (see Fig. 6). Note that this annotation can be used for changing the exception thrown when a specification observer notices an error (default exception is fi.hut.ics.aspectmonitor.SpecificationException). The annotation consists the following fields:

- `specs` – the list of specifications to be run in the annotated method. Call specifications will be run before and return specifications after the method invocation.

```
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 ReturnSpecifications {
  String[] valuePropositions() default {};
  String[] callPropositions() default {};
  String[] pltl() default {};
  String[] regexp() default {};
  String[] nfa() default {};
}
```

Figure 5: Annotation for declaring return specifications

- `exception` – the class of the exception thrown when a property is violated (name of the specification is given as parameter to the exception or in case of an anonymoys specification, the number of that specification is used).

5.3 Tool architecture

The implementation is built using the Spoon framework [13] 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. 7 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.lime.common) and Aspect Monitor (fi.hut.ics.lime.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.

```
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.SpecificationException;

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

Figure 6: Annotation for observing specifications

- **SCheck** is used for converting temporal logic formulas into finite state automata.
  - GPL license.

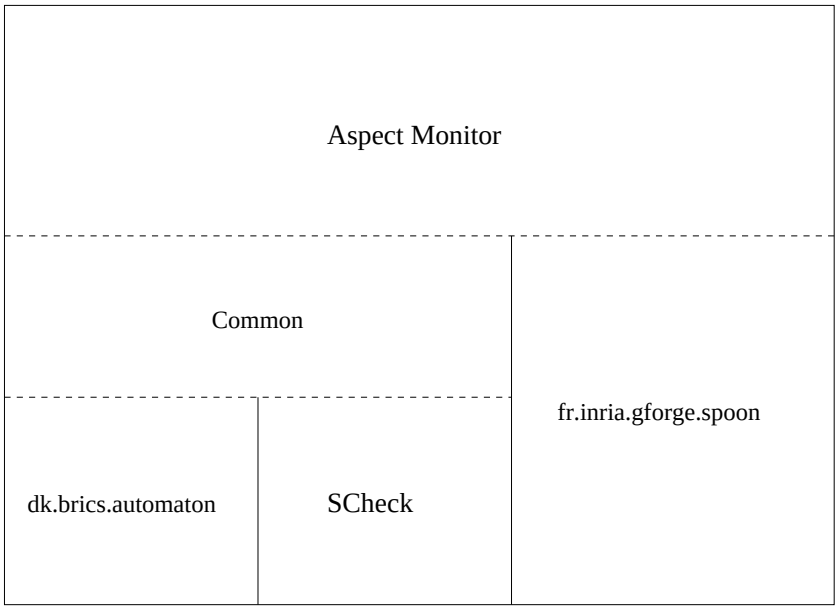

Figure 7: Basic modular decomposition

Fig. 8 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 specification observer. The `AbstractSpecification` class is subclassed into `PltlSpecification`

and `RegExpSpecification` classes to accommodate their structural differences.

If a type has two specification 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 `@Observe` (see Fig. 6). 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 observe the specification.

In the synthesis pass the specification objects are turned into aspects. Code generation is discussed in more detail in Sect. 5.3.

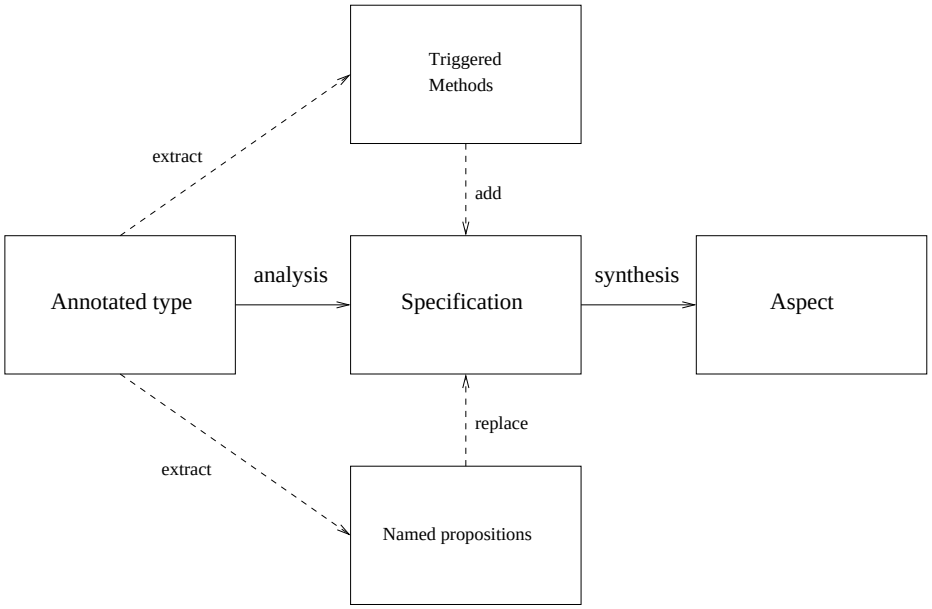

Figure 8: Generating temporal safety aspect from an annotated class

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

The logic package (fi.ics.hut.lime.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. 9. 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 [6] 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.lime.common.regexp) serves as 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.lime.common.pltl) serves as an adapter package for the SCheck module. The main function for this package is

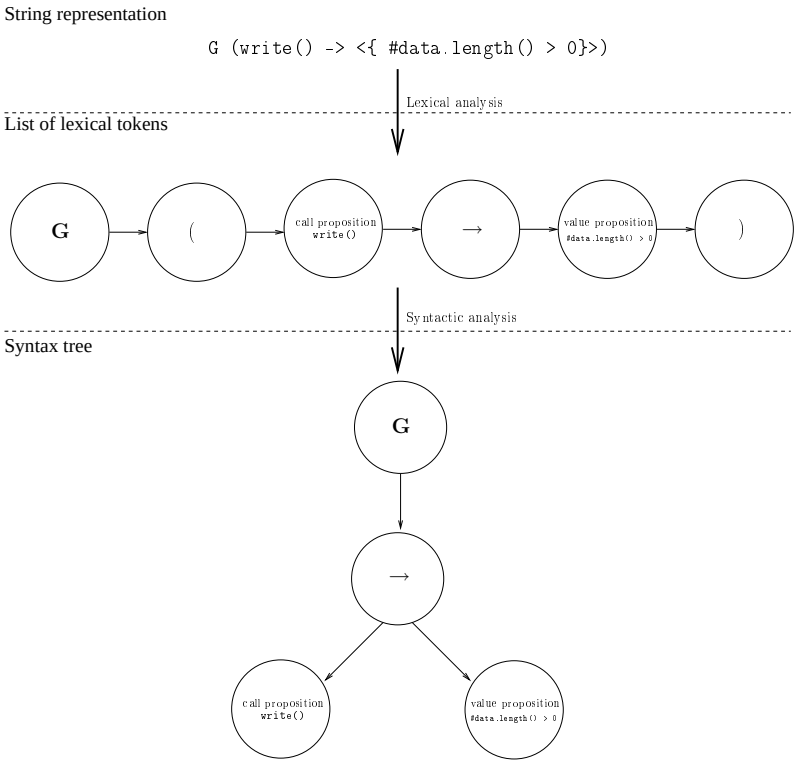

Figure 9: Lexical and syntactic analysis of a logic formula

to 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 Sect. 5.3.

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. 8 this transformation is done in two phases – analysis and synthesis. These phases have corresponding packages in the implementation:

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

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

Semantics of value propositions
Call propositions and value propositions are easily identifiable in a formula from the lexical form, see Def. 20 on page 20 and Def. 21 on page 20. 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 Sect. 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 return 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 Sect. 3.2 on page 22.

5.4 Implementation of the C variant

The C variant of LIME tool (LIME-C tool) does for C essentially what Aspect-Monitor does for Java. It's internal structure is pretty close to that of Aspect-Monitor, and the tools mostly differ in the external tools they rely on. On the C side, Spoon has been replaced by Doxygen (`http://www.stack.nl/~dimitri/doxygen/`) and AspectJ by AspeCt-oriented C (`http://research.msrg.utoronto.ca/ACC/`). Both LIME-C and Aspect-Monitor use the Common package for dealing with the specifications. The following chapters shall briefly go through the major differences in the LIME tools by explaining how the external tools used with LIME-C differ from their Java counterparts, and how these differences have been dealt with to achieve similiar functionality in the tools.

Doxygen
Doxygen is a documentation generator that LIME-C uses as a replacement for Spoon. The differences with these two tools are actually quite significant: as Spoon interprets the bytecode of a Java class and can be used to analyze and alter the program at compile time, Doxygen reads

the source code from a file and outputs it's structure (including special comments attached to functions) to an XML-file. Even though the principle on how Doxygen works compared to spoon is completely different, the same functionality (generation of an aspect from a specification to monitor the execution of a program) can be achieved by exploiting the mechanics of Doxygen cleverly.

LIME-C-tool uses Doxygen to read the interface specifications and the structure of the source code. As the reader might have noticed from Section 3.3, the syntax of the C-version of the specification language is actually really close to the normal comment syntax of the C-language. This is because Doxygen doesn't read traditional comments from the code at all (it only reads comments written in it's own comment syntax), so the existence of interface specifications doesn't conflict with either the normal compilation of the program, or the usage of Doxygen for other purposes. When the LIME-tool is used, the specifications will be internally changed to the Doxygen comment syntax, when Doxygen associates them correctly with the functions they preceed in the source file. So with this change, running Doxygen results in a series of XML-files with all the information the LIME-C-tool needs to create a working aspect. As a difference to the Java version it should be noted that as Aspect-Monitor does its work in compile time (since Spoon allows it), LIME-C-tool is ran separately against one or more source code files and all it does is creating aspects of them. The user must then use other tools to compile these aspects together with the original (a bit altered, more on that in the next chapter) source code files.

AspeCt-oriented C

AspeCt-oriented C (ACC from now on) is an aspect-oriented C compiler used as a replacement for AspectJ. As mentioned before, the user must compile the aspects generated by LIME-C-tool together with other source code, and this is where ACC comes in handy. However, altought LIME-C-tool doesn't actually run ACC by itself, the features of it must be taken into account in the tool.

ACC is a bit more work-in-progress than AspectJ, and thus doesn't support all the language features yet. The one that causes most problems is that static variables and functions can't be used in aspects. Limiting the interface specifications to only cover non-static functions would not make sense, so this limitation has been worked around. After running Doxygen and interpreting the source, LIME-C-tool takes the original code file and copies it for the user, at the same time creating accessory non-static functions for each static function it sees. These accessory functions just call the original function so they don't change the execution, but since they are non-static they can be used from the aspect. After this, the user can take these modified files and compile them with ACC without producing any unnecessary errors.

6 EXPERIMENTS WITH INTERFACE SPECIFICATIONS

The purpose of this section is to demonstrate generated safety aspects from example specifications. Sect. 6.1 demonstrates a basic call specification written with a regular expression. A return 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 A call 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. 10 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.

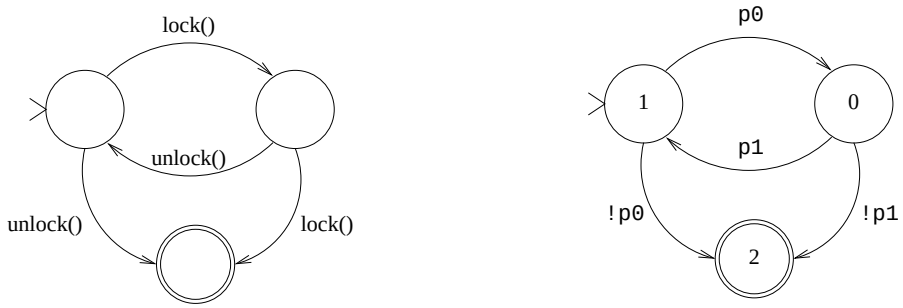

Figure 10: Proper use of a lock

Clearly in the terminology of the LIME specification language the guards on the transitions translate into call propositions in the specification. Therefore a `Lock` interface is written as follows:

```
package example_lock;

import fi.hut.ics.lime.aspectmonitor.annotation.
  CallSpecifications;

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

In Fig. 10 the automaton on the right illustrates the automaton synthesized in generated safety aspect (parts of which are presented in Fig. 11).

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();
  }
}
```

After the safety aspect corresponding to the call specification 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 10 which informs that StrictAlternation property has been violated:

```
Exception in thread "main" fi.hut.ics.lime.aspectmonitor.
    CallSpecificationException: StrictAlteration
        at example_lock.CALLSpecificationLockStrictAlteration.ajc$around$
            example_lock_CALLSpecificationLockStrictAlteration$2$fbbaaa31
            (CALLSpecificationLockStrictAlteration.aj:86)
        at example_lock.LockImpl.unlock(LockImpl.java:15)
        at example_lock.Main.main(Main.java:10)
```

This exception violation message will automatically be altered to be more readable by the provided tools, so the actual output the user will see is as follows:

```
example_lock/LockImpl.java:15: Call specification ‘StrictAlteration’
    violated at this point.
example_lock/LockImpl.java:15: Stack trace:
example_lock/LockImpl.java:15:    example_lock.LockImpl.unlock   <--
    violation happened here
example_lock/Main.java:10:        example_lock.Main.main
```

```
void around(example_lock.Lock callTarget)
        : (execution(public void example_lock.Lock+.lock(..)) &&
        target(callTarget)) {
    boolean p1 = true; // [CALL]: lock
    boolean p0 = false; // [CALL]: unlock
    refreshState(p0, p1);
    if (state == 0) {
        throw new fi.hut.ics.lime.aspectmonitor.
    CallSpecificationException("StrictAlteration");
    }
    proceed(callTarget);
}

void around(example_lock.Lock callTarget)
        : (execution(public void example_lock.Lock+.unlock(..)) &&
        target(callTarget)) {
    boolean p1 = false; // [CALL]: lock
    boolean p0 = true; // [CALL]: unlock
    refreshState(p0, p1);
    if (state == 0) {
        throw new fi.hut.ics.lime.aspectmonitor.
    CallSpecificationException("StrictAlteration");
    }
    proceed(callTarget);
}

private void refreshState(boolean p0, boolean p1) {
    int trDone=0;
    switch(state) {
        case 1:
            if((((!p0) && (!p1)) || p0)) { state = 0; trDone++; }
            if(p1) { state = 2; trDone++; }
            break;
        case 2:
            if((((!p0) && (!p1)) || p1)) { state = 0; trDone++; }
            if(p0) { state = 1; trDone++; }
            break;
        case 0:
            break;
    }
    if (trDone == 0)
        state = -1; // sink reject state
    else if (trDone > 1)
        throw new Error("Internal error");
}
```

Figure 11: Parts of a generated call specification aspect

6.2 A return specification for a file interface

The following return specification experiment is based on Example 7 given on page 17 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 call specification, i.e., that `write` is not called with null argument, to ensure as well as the redundant methods to the return specification have been left out from this experiment.

```
package example_file;

import fi.hut.ics.lime.aspectmonitor.annotation.
  ReturnSpecifications;
import fi.hut.ics.lime.aspectmonitor.annotation.
  Observe;

@ReturnSpecifications(
  pltl = {
    "ProperWrites ::= "+
      "G(<{ #pre(#s.length() + #this.length()) "+
      "== #this.length() }>)"
  }
)
public interface LogFile {
  @Observe(
    specs = { "ProperWrites" },
    returnException = RuntimeException.class
  )
  public void write(String s);
  public int length();
}
```

Fig. 12 shows parts of 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 observing policy (see Def. 19 on page 20). Even though the specification declaration references `length()` method in the `LogFile` interface, the calls to it do not observe the specification. This is because the reference is done through a value proposition and not through a call proposition as the default observing policy would require.

```
void around(java.lang.String s, example_file.LogFile callTarget)
        : ((execution(public void example_file.LogFile+.write(..)) &&
        args(s)) &&
        target(callTarget)) {
    int pre0 = (s.length() + callTarget.length());
    try {
        proceed(s, callTarget);
    } finally {
        // [VALUE]: #pre(#s.length()+#this.length())==#this.length()
        Boolean p0 = (pre0 == callTarget.length());
        refreshState(p0);
        if (state == 1) {
                throw new java.lang.RuntimeException("ProperWrites");
        }
    }
}

private void refreshState(boolean p0) {
    int trDone=0;
    switch(state) {
        case 1:
            break;
        case 0:
            if(p0) { state = 0; trDone++; }
            if((!p0)) { state = 1; trDone++; }
            break;
    }
    if (trDone == 0)
        state = -1; // sink reject state
    else if (trDone > 1)
        throw new Error("Internal error");
}
```

Figure 12: Parts of a generated return specification aspect

6.3 PLTL specification with a past time subformula

This experiment demonstrates how past time subformulas are handled in PLTL specifications. The call 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;

import fi.hut.ics.aspectmonitor.annotation.
  CallSpecifications;
import fi.hut.ics.aspectmonitor.annotation.
  Observe;

@CallSpecifications(
  pltl = {
   "ProperStarts ::= G (start() -> O(ignite()))"
  }
)

public interface Car {
  @Observe(
    specs = {"ProperStarts"},
    callException = RuntimeException.class
  )
  public void start();

  @Observe(
    specs = {"ProperStarts"},
    callException = RuntimeException.class
  )
  public void ignite();
}
```

Fig. 13 shows some parts of 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. 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.

```
public CALLSpecificationCarProperStarts() {
    preProperStarts = new boolean[1];
    nowProperStarts = new boolean[1];
    state = 0;
    preProperStarts[0] = false; // ONCE
}

void around(example_past.Car callTarget)
        : (execution(public void example_past.Car+.ignite(..)) &&
        target(callTarget)) {
    boolean p1 = true; // [CALL]: ignite
    boolean p0 = false; // [CALL]: start
    refreshState(p0, p1);
    if (state == 1) {
        throw new java.lang.RuntimeException("ProperStarts");
    }
    proceed(callTarget);
}

void around(example_past.Car callTarget)
        : (execution(public void example_past.Car+.start(..)) &&
        target(callTarget)) {
    boolean p1 = false; // [CALL]: ignite
    boolean p0 = true; // [CALL]: start
    refreshState(p0, p1);
    if (state == 1) {
        throw new java.lang.RuntimeException("ProperStarts");
    }
    proceed(callTarget);
}

private void refreshState(boolean p0, boolean p1) {
    nowProperStarts[0] = p1 || preProperStarts[0]; // ONCE
    preProperStarts[0] = nowProperStarts[0];
    int trDone=0;
    switch(state) {
        case 1:
            break;
        case 0:
            if(((!p0) || nowProperStarts[0])) { state = 0; trDone++; }
            if((p0 && (!nowProperStarts[0]))) { state = 1; trDone++; }
            break;
    }
    if (trDone == 0)
        state = -1; // sink reject state
    else if (trDone > 1)
        throw new Error("Internal error");
}
```

Figure 13: Parts of a generated PLTL call specification aspect

7 CONCLUSIONS

We have presented a specification language that can used to establish richer correctness requirements for software components. The specification methodology described can be seamlessly added to a software process to complement traditional means for quality assurance. Components specified in this manner can be completely or partially implemented, and their correctness requirements can independently refined. The reader is requested to consult [10] for a more concise presentation of the LIME interface specification language and the motivation behind its development. The automated Java testing tool also developed in the LIME project is documented in [9].

REFERENCES

[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] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, 2005.

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

[7] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

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

[9] Kari Kähkönen. Automated test generation for software components. Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, December 2009.

[10] Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä. The LIME interface specification language and runtime monitoring tool. In Saddek Bensalem and Doron Peled, editors, *Proceedings of the 9th International Workshop on Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 93–100, 2009.

[11] Timo Latvala. Efficient model checking of safety properties. In T. Ball and S. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop*, pages 74–88. Springer, 2003.

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

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

[14] Grigore Rosu. 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 *LNCS*, pages 332–345. Springer-Verlag, 2007.

A THE SYNTAX FOR NFA CHECKERS

This is what the annotation for a Java method looks like. The requirement that the strings be one-line and concatenated with + is due to the Java limitation of not having multi-line strings.

```
@CallSpecifications(
  nfa = {
    "always_nfa {" +
    "  state1_init:" +
    "    if" +
    "      :: ((a && b) || c) -> goto state2;" +
    "      :: (c && d) -> goto reject_state3;" +
    "      :: (1) -> goto state1_init;" +
    "    fi;" +
    "  state2:" +
    "    if" +
    "      :: ((a || b) || c) -> goto state1_init;" +
    "      :: (c && d) -> goto state2;" +
    "      :: (1) -> goto reject_state3;" +
    "    fi;" +
    "  reject_state3:" +
    "    if" +
    "      :: ((!(a || b)) || c) -> goto state2;" +
    "      :: (c && d) -> goto reject_state3;" +
    "      :: (1) -> goto state1_init;" +
    "    fi;" +
    " }" }
)
```

The NFA format is similar to Spin syntax for never-claims, but the meaning is reversed. The automaton specifies good behavior for the program. To reflect this difference, the statement is named "always_nfa".

If the automaton ever ends up in a state where no current states are accepting (i.e. all current states are rejecting), the specification is considered to fail the check. States are accepting and non-initial by default.

Initial states are named <something>_init, rejecting states reject_<something>. If a state is both initial and rejecting, it is named reject_<something>_init. Note that since we only consider the prefixes of the execution, it generally only makes sense to have automata where at least one of the initial states is also an accepting state.

The condition for transitioning into a state must always be in parentheses.

1. Atomic propositions (a, b, c):

   1.1. Call and value propositions:

      All call and value propositions, including escaped Java boolean propositions, need to be made into named propositions. Only named propositions can be referenced in the NFA.

The format is the same as for regexp and pltl:

```
@CallSpecification(
  valuePropositions = {
    "properData ::= (#s != null)"
  },
  nfa = { " ... (properData) || a && ..." }
)
```

1.2. Literals

0, 1 represent false and true

2. Propositional logic combinations of atomic propositions

```
write || 1

a && b

open && b

open == 0

open != 1

(!error) && (consistent)
```

Operators: !, == (equivalence), != (non-equivalence), &&, ||
Parentheseses: ()

The precedences of the operators are the same as in the C language, with negation having the highest precedence and logical OR having a lower precedence than logical AND:

	Operator	Associativity
higher precedence	!	right-to-left
	==, !=	left-to-right
	&&	left-to-right
lower precedence	||	left-to-right

However, to ensure compatibility with Spin, use enough parentheses to make the formulae unambiguous.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R15 Eerika Savia, Kai Puolamäki, Samuel Kaski
On Two-Way Grouping by One-Way Topic Models. May 2009.

TKK-ICS-R16 Antti E. J. Hyvärinen
Approaches to Grid-Based SAT Solving. June 2009.

TKK-ICS-R17 Tuomas Launiainen
Model checking PSL safety properties. August 2009.

TKK-ICS-R18 Roland Kindermann
Testing a Java Card applet using the LIME Interface Test Bench: A case study. September 2009.

TKK-ICS-R19 Kalle J. Palomäki, Ulpu Remes, Mikko Kurimo (Eds.)
Studies on Noise Robust Automatic Speech Recognition. September 2009.

TKK-ICS-R20 Kristian Nybo, Juuso Parkkinen, Samuel Kaski
Graph Visualization With Latent Variable Models. September 2009.

TKK-ICS-R21 Sami Hanhijärvi, Kai Puolamäki, Gemma C. Garriga
Multiple Hypothesis Testing in Pattern Discovery. November 2009.

TKK-ICS-R22 Antti E. J. Hyvärinen, Tommi Junttila, Ilkka Niemelä
Partitioning Search Spaces of a Randomized Search. November 2009.

TKK-ICS-R23 Matti Pöllä, Timo Honkela, Teuvo Kohonen
Bibliography of Self-Organizing Map (SOM) Papers: 2002-2005 Addendum. December 2009.

TKK-ICS-R24 Timo Honkela, Nina Janasik, Krista Lagus, Tiina Lindh-Knuutila, Mika Pantzar, Juha Raitio
Modeling communities of experts. December 2009.

ISBN 978-952-248-278-5 (Print)
ISBN 978-952-248-279-2 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)