Checking Semantic Compatibility of SOFA/DCUP Components

Master Thesis

Stanislav Višňovský

Advisor: Miloslav Bešta
Department of Software Engineering
Prague, 1999
Acknowledgments

I would like to thank to my advisor Miloš Bešta for his important comments and longstanding support. I also thank to František Plášil, who really made possible to progress this thesis by long-lasting discussions. His rich experience in the field and everlasting optimism allow to complete the work.

I do declare that I have elaborated this master thesis on my own and that the references include all the sources of information I have exploited. I agree with lending of this master thesis.

Prague, April 23, 1999
Contents

1. Introduction .......................................................... 1

2. Related work ......................................................... 3
   2.1. Architecture description languages .......................... 3
       2.1.1 Wright ................................................. 3
       2.1.2 Aster .................................................. 3
   2.2. Communication description by event ordering ............... 4
       2.2.1 Path expressions ...................................... 4
       2.2.2 Procol ................................................ 4
       2.2.3 CSP .................................................. 5
       2.2.4 CCS .................................................. 5

3. Semantic description for SOFA components .......................... 6
   3.1. SOFA component model ..................................... 6
   3.2. Communication model ...................................... 8
   3.3. Behavior protocols ....................................... 9
   3.4. Interface protocol ....................................... 14
   3.5. Frame protocol ......................................... 15
   3.6. Architecture protocol ................................... 16

4. Verification of a component semantic description ................... 18
   4.1. Naming issues ............................................. 18
   4.2. Protocol conformance ..................................... 20
   4.3. Tests for component correctness .......................... 20
       4.3.1 Conformance of provides-interface and
            requires-interface .................................. 21
       4.3.2 Conformance of two provides-interfaces .......... 22
       4.3.3 Conformance of two requires-interfaces .......... 22
       4.3.4 Conformance of frame to provides-interfaces .... 22
       4.3.5 Conformance of frame to requires-interfaces .... 22
       4.3.6 Conformance of architecture to frame .......... 22
       4.3.7 Conformance of architecture to interfaces .... 23

5. Case study ............................................................ 24

6. Automation of correctness checking .................................. 33
   6.1.1 Atomic operand ......................................... 37
   6.1.3 Alternative ............................................. 38
   6.1.4 Repetition .............................................. 39
   6.1.5 And-parallel ........................................... 39
   6.1.6 Reentrancy ............................................. 40
   6.2. Event restriction on automaton ................................ 40
   6.3. Deterministic vs. non deterministic automaton ............ 43
   6.4. Reduction of automaton ................................... 43
   6.5. Language inclusion checking ................................ 44

7. Implementation ........................................................ 46
7.1. Representation of edges ................................................. 46
7.2. Representation of states ................................................. 48
7.3. Representation of automaton ........................................... 49
7.4. The Builder class ......................................................... 51
7.5. The Checker class ....................................................... 52

8. Evaluation and conclusion .................................................. 54

References ................................................................. 55
1. Introduction

Component-based programming is quickly becoming one of the major techniques for developing software systems. Its advantages include better level of reuse than in object-oriented approach. The frameworks, which provide the component-based approach, define the services provided resp. required by a software system by means of object interface signatures. But, only signature description does not provide enough information for component reuse and trading. As the reaction to this problem, there were defined frameworks and programming languages which allow to specify more than the signatures. One of the most typical approaches is the precondition/postcondition specification used in Eiffel [4]. A basic information needed for reuse is description of communication protocols, i.e., ordering of method invocations. The precondition/postcondition approach does not states this information explicitly. The area of protocols is well covered by different process algebras. Although many advantages of these methods have been claimed, it is not an easy task to employ these rigorous semantical methods. Also, these are not suitable for component trading, another important area the description should be used.

The goal of the SOFA project is to design a software environment to support software provider-user (consumer) relation. One area of the current research is the specification of component-based software systems. It is primary targeted at the design process of systems, but it should also provide useful information for component trading. The semantic description for the SOFA components is proposed in [19]. It is based on the behavior protocols. In this work, we use this description for proposing the testing of the designed system. For a practical use, it is necessary to provide a tool for automation of these tests. The second goal of this work is to design such automatic or at least a semiautomatic tool.

To meet the goals, the work is structured as follows. Section 2 discusses related work and possible approaches to the semantic description. In section 3 we present the description of the semantics and its deployment in SOFA component model. Section 4 is to specify the necessary
tests for verification of the specification. To illustrate the description, we present a case study of a web browser in Section 5. Sections 6 and 7 address the automation of the verification and present a prototype of an automatic tool. Section 8 summarizes the work and open issues.
2. Related work

This section presents an overview of related work. It can be divided into two areas. The first area comprises the architecture description languages, which allow to specify the architecture of a software system. SOFA’s Component Description Language (CDL) is also such a language. The second area is work on the semantic description of communication among software entities, i.e., protocols.

2.1. Architecture description languages

Architecture description languages (ADL) provide notation for specifying the software architectures in terms of hierarchical configurations [12]. We discuss two of them, which are the most closely related to this work, since they provide a support for semantic description.

2.1.1 Wright

Wright [12] is ADL, which allows to specify the hierarchies by means of components and connectors. Components are the computational entities and connectors are the mediating parts. Both components and connectors specify a set of interfaces called ports (resp. roles). These are symmetrical by means of direction of communication transmission. The semantic description is based on CSP [6]. Every component is described as a CSP process called computation, every connector is described by a process called glue. The ports (resp. roles) are also described by means of CSP. Wright allows static testing of the description to detect the most common design errors. The expressive power of CSP is also the most important drawback of Wright, since the formal approach is not suitable for software trading, where the readability is one of the basic requirements of semantic description.

2.1.2 Aster

The Aster [26, 27] language combines the CSP approach of Wright with another level of semantic description based on the first order logic. This allows for describing other properties
than communication semantics. Aster is able to specify fault-tolerance, transactions etc. The advantages and drawbacks of the Wright also hold for Aster. Although it gives more expressive power, it also makes the specification even less readable than in Wright.

2.2. Communication description by event ordering

There are a number of formalisms based on describing the communication by means of discrete atomic actions. We discuss two such formal notations, i.e., CSP and CCS, and two other protocol-based approaches closely related to behavior protocols.

2.2.1 Path expressions

Path expression [1] allows to describe the synchronization constraints by means of the ordering operators, such as sequence, alternative, parallel execution a reentrancy. At runtime, the procedure calls can be synchronized by the rules given. The purpose of synchronization does not require tests at the design time. Although the path expressions do not address description of semantics, it was the basis for research on method invocation ordering. The behavior protocols, as described in this work, are based on this work. Besides the only runtime checking, the biggest drawback of path expressions is the restriction on the use of operators, since the reentrancy and parallel execution are not allowed to nest. Thus, description of the possible situations is somehow limited.

2.2.2 Procol

PROCOL [3] is the object-oriented programming language, which allows to specify the protocol for an object interface. The protocol is used for automatic synchronization as well as for the semantic description. The protocol is closely tied to the object interface. Thus, it is not possible to specify a protocol to encompass more than one object. This is the main drawback, since, the component-based systems typically define more than one interface per component.
2.2.3 CSP

Hoare’s Communicating Sequential Processes [6] is the language for specifying the communication in a software system. It is used as the formal basis for some ADLs (Wright, Aster). The communication is specified as a set of equations and modeled by sequences of atomic events. CSP allows to specify almost any communication situation. The possibility of static checking of the description is limited. The best-known tool, FDR [13], provides such tests by exhaustive searching of possible sequences. Also, as the formal specification, CSP introduces the complexity of description, which leads to the hardly readable specifications.

2.2.4 CCS

A Calculus of Communicating Systems [5] is another algebraic model of communication. It is based on similar operators as the behavior protocols, i.e., sequencing, alternative, parallel communication and composition. The communication is modeled by sequences of atomic events. It introduces the idea of externally observable events and the internal events. Internal events are the origin of the internal events in communication model of the behavior protocols. As a formal description, CCS allows for exhaustive analysis of the modeled communication, but it is unusable for trading information, because the complex syntax.
3. Semantic description for SOFA components

In this section, we present the semantic description, as defined in [19]. First, we describe the SOFA component model followed by the formal basis of the description - behavior protocols. Their employment into SOFA components is then presented.

3.1. SOFA component model

The SOFA component model defines a component as a black-box computational entity, which defines its access points (called interfaces). An interface can be specified as a provides-interface or a requires-interface. A provides-interface specifies the service, which a component provides to environment. A requires-interface specifies a service, which component requires from environment for its functionality.

A software application is viewed as a hierarchy of components. Components can be either primitive or composed. Primitive components are specified in an underlying programming language and their structure is not specified in SOFA specification. They are the active computational entities. On the other hand, a composed component is specified as a set of components, whose interfaces are tied together. We denote a composed component as the parent component of all its subcomponents. An interface tie is a one-to-one relationship. There are three types of interface ties:

(1) **Bind.** It ties a requires-interface to a provides-interface. Thus, it satisfies a component’s requirement by specifying, which component will provide the service needed.

(2) **Delegate.** Delegation passes a provided service to a subcomponent by tying a provides-interface of a component to a provides-interface of a subcomponent.

(3) **Subsume.** To satisfy a requirement of a subcomponent, a component can subsume it as its own requirement. This is done by tying a subcomponent’s requires-interface to a component’s requires-interface.
In analogy with class and object instances, SOFA introduces a component template and a component. A template is a pair \(<\text{frame}, \text{architecture}>\). The frame abstraction allows for a black-box view of a component by hiding all implementation details and describing only the interfaces accessible by environment. The architecture abstraction allows for specifying a hierarchy of components. An architecture can be primitive or can be specified as a set of direct subcomponents and their ties. All subcomponents are specified on the frame abstraction level. It allows for recursive description of a system. Other important issues are better support for refinement process of the system design and it allows for an implementation of DCUP architecture [18]. DCUP is an extension of SOFA for runtime updating of components.

SOFA defines its own specification language called CDL (Component Definition Language [21]). CDL is based on CORBA IDL, which is enhanced by syntactic constructs for specifying frames and architectures. The following simple example shows the syntax of interface, frame and architecture constructs by specifying a component for a FIFO buffer. For other examples, see case study in Section 5.

```plaintext
interface bufferAccess {
    Long Get( void);
    void Put( in Long item);
    Short isEmpty( void);
};

interface memoryManager {
    Long Alloc( in Long size);
    void Free( in Long handle);
    Sequence Access( in Long handle);
};

frame Buffer {
    provides:
    bufferAccess buf;
    requires:
    memoryAccess mem;
};
```

```plaintext
architecture Buffer version "simple" {
    primitive;
};
```
3.2. Communication model

To specify semantics of components by means of communication, it is necessary to define the formal basis of for communication by basic abstractions.

A description of components’ communication is based on describing the sequences of atomic communication events. To support the client/server communication of component-based systems, events are either a request or a response. A component handles an event by one of following ways:
(1) emitting an event
(2) absorbing an event
(3) handling an internal event

An event can be either internal or external. An event is internal to the component, if it is emitted and absorbed in its subcomponents. Thus, a component can an event either emit, i.e., send to the other component, or absorb an event, i.e., receive it from other component. These support the ability to describe the point of view of a component. Nesting of components is represented by the idea that handling of an event is represented as communication of all components, which are parent components of the actual primitive component that either emits or absorbs the event.

The sequence of event handling by a component is called the activity of a component. The behavior of a component is the set of all possible activities of a component.

Events are represented as action tokens. Action token is a triple <‘?’ or ‘!’ or ‘τ’, event identification, ‘!’ or ‘!’>. The interrogation point represents absorbing an event, the exclamation point represents emitting an event and ‘τ’ represents handling of an internal event. The activity is then represented as the sequence of the action tokens called trace. The set of all traces is the language of a component. It represents the behavior of a component.
The behavior of a component grasps the communication on all interfaces of the component. To allow description of different views on a component, for example, the black-box view, we need to define the behavior on a set of interfaces. Thus, we introduce following definitions.

**Definition.** Let A be a component and CS be a set of its interfaces. The set of all possible activities of A is the behavior of A on CS. By convention, the behavior of A on CS is represented as a set of traces—the language of A on CS (denoted by \( L_{A|CS} \)).

**Definition.** The event restriction of a language L on a set of event names N is a function \( \phi_N : L \rightarrow L' \), such that \( \phi_N(\alpha_0 x_1 \alpha_1 x_2 \alpha_2 \ldots x_n \alpha_n) = x_1 x_2 \ldots x_n \), where \( \alpha_0 x_1 \alpha_1 x_2 \alpha_2 \ldots x_n \alpha_n \in L, \forall i x_i \in E_N, \alpha_i \in (E_L \setminus E_N)^* \). \( E_N \) is the set of all possible action tokens the identifiers of which are in N and \( E_L \) is the set of all action tokens in L. In other words, the event restriction is a function that, from every trace of the language L, omits all action token identifiers of which are not in N. The resulting set of words forms the language L’.

Let \( L_{A|CS} \) be a language of A on CS. The language L on a subset C of set of interfaces CS is the language \( L/C = \phi_{CE}(L_{A|CS}) \), where CE is the set of all event names, identifiers of which are qualified by an identification of an interface from C.

### 3.3. Behavior protocols

The language of a component can be, in general case, infinite. For useful semantic description based on behavior of a component, we need a finite form of behavior representation. As a such representation, we introduce behavior protocols, syntactic regular expression-like notation. A behavior protocol is built from action tokens, empty protocol NULL and behavior protocol operators. Formally, behavior protocol is a word generated by a Behavior Protocol Grammar BPG:

\[
BPG = ( \{ A, B, C, D, E, F, S, M, N, I \}, \text{Events} \cup \{ \uparrow, \downarrow, ?, !, +, ;, (,), ^{^*}, |, \|, \cap, \{, \} \}, \text{Rules}, \text{S} )
\]

where Events is a set of identifiers (event names) and Rules are defined as follows:
\[ S \rightarrow S \cap A \quad S \rightarrow A \]
\[ A \rightarrow A + B \quad A \rightarrow B \]
\[ B \rightarrow B ; C \quad B \rightarrow C \]
\[ C \rightarrow C | D \quad C \rightarrow C \| D \quad C \rightarrow D \]
\[ D \rightarrow E^\wedge \quad D \rightarrow E^* \]
\[ E \rightarrow N \quad E \rightarrow (S) \quad E \rightarrow N \{S\} \]
\[ N \rightarrow !M \quad N \rightarrow ?M \quad N \rightarrow M \quad N \rightarrow \text{NULL} \]
\[ M \rightarrow I! \quad M \rightarrow I \]
\[ I \rightarrow \text{event_identifier} \]

A behavior protocol generates the language. The following rules specify the semantics of behavior protocol operators by the language they generate. We denote the language generated by the protocol \( \alpha \) as \( L(\alpha) \), \( a, b \) denote event names, \( \alpha \) and \( \beta \) are protocols and \( t_\alpha \in L(\alpha) \), \( t_\beta \in L(\beta) \) are traces.

**Empty protocol (NULL)**
The resulting set of traces is empty. This is useful for specifying special situations like zero or one occurrence of a protocol.
\[ L(\text{NULL}) = \emptyset. \]

**Sequence (\( \alpha ; \beta \))**
A trace generated by \( \alpha;\beta \) is a concatenation of a trace generated by \( \alpha \) and a trace generated by \( \beta \). This represents the communication defined as \( \alpha \) to be followed by communication specified by \( \beta \).
\[ L(\alpha;\beta) = \{ t_\alpha \cdot t_\beta : t_\alpha \in L(\alpha) \& t_\beta \in L(\beta) \} \]

**Nested incoming call (\( ?a\{\alpha\} \))**
Curly braces are an abbreviation \( ?a\{\alpha\} = ?a! ; \alpha ; !a! \). It allows for description of a reaction to an incoming method invocation, where \( \alpha \) represents the internal computation (more precisely, the communication generated by an internal computation). After finishing communication defined by \( \alpha \) the response event is emitted.
\[ L( \texttt{?a\{ \alpha \} } ) = L( \texttt{?a}! ; \alpha ; !\texttt{a}! ) \]

**Simple incoming call ( ?event_identifier )**
An event name prefixed by ‘?’ and without the request/response symbol is an abbreviation used for describing an incoming method invocation, for which the response is immediately emitted.
\[ \texttt{?a} = \texttt{?a\{} = \texttt{?a}! ; !\texttt{a}! \]
\[ L( \texttt{?a} ) = \{ \texttt{?a}!, !\texttt{a}! \} \]

**Simple outgoing call ( !event_identifier )**
An event name prefixed by ‘!’ and without the request/response symbol is an abbreviation used for describing an outgoing method invocation.
\[ \texttt{!a} = !\texttt{a}! ; \texttt{?a}! \]
\[ L( \texttt{?a} ) = \{ !\texttt{a}!, \texttt{?a}! \} \]

**Alternative ( \alpha + \beta )**
The communication described by \( \alpha \) or by \( \beta \) can be exhibited, but not both \( \alpha \) and \( \beta \). Thus, it allows to specify two distinct possibilities of the communication.
\[ L( \alpha + \beta ) = L( \alpha ) \cup L( \beta ) \]

**Repetition ( \alpha^* )**
Traces generated by \( \alpha \) are repeated zero or more times. Communication specified by \( \alpha \) is to be repeated any number of times.
\[ L( \alpha^* ) = L( \alpha ) \cup L( \alpha ; \alpha ) \cup L( \alpha ; \alpha ; \alpha ) \cup ... \]

**And-Parallel execution ( \alpha | \beta )**
The resulting language consists of all possible interleaving of traces from \( L( \alpha ) \) and \( L( \beta ) \). In principle, \( \alpha \) and \( \beta \) represent two independent communication patterns. This is essential for describing parallel processes.
\[ L(\alpha | \beta) = \{ \texttt{a}_1 \texttt{b}_1 \texttt{a}_2 \texttt{b}_2 ... \texttt{a}_n \texttt{b}_n; \texttt{a}_1 \texttt{a}_2 ... \texttt{a}_n \in L(\alpha) \& \texttt{b}_1 \texttt{b}_2 ... \texttt{b}_n \in L(\beta) \} \]
Or-parallel operator (\( \alpha \parallel \beta \))

The resulting language consists of interleaved traces generated by the protocol \( \alpha \), or \( \beta \), or both. This is not the case of the \(|\) operator which requires traces from both protocols \( \alpha \) and \( \beta \) to appear in the resulting trace.

\[
L( \alpha \parallel \beta ) = L( \alpha + \beta + \alpha \mid \beta )
\]

Reentrancy (\( \alpha^\wedge \))

The operand \( \alpha \) is reentrant. This allows to specify the reentrant access, i.e., the possibility of any number of concurrent communication sequences defined by \( \alpha \).

\[
L( \alpha^\wedge ) = L( \alpha \mid \alpha \mid \ldots \mid \alpha )
\]

Composition (\( \alpha \cap \beta \))

Let \( \tau_\alpha = A_0e_1A_1e_2A_2\ldots e_nA_n \in L(\alpha) \) and \( \tau_\beta = B_0e_1B_1e_2B_2\ldots e_nB_n \in L(\beta) \), \( S_\lambda \) is the set of all events used in protocol \( A \), \( A_i \in (E_\alpha \cap (E_\alpha \cap E_\beta))^* \), \( B_i \in (E_\beta \cap (E_\alpha \cap E_\beta))^* \), \( e_i \) is an event, which is observed in \( \tau_\alpha \) and is emitted in \( \tau_\beta \), or vice versa. The set of resulting traces will be \( L( A_0^p \mid B_0^p ; \tau_1 ; A_1^p \mid B_1^p ; \tau_2 \ldots \tau_n ; A_n^p \mid B_n^p ) \), where \( \tau \) represents handling of the internal event \( e \) as internal, and \( A_i^p \) denotes protocol derived from trace \( A_i \) by sequencing the events from \( A_i \).

**Definition.** Given a behavior protocol \( BP \), the \( BP \) restriction on \( N \) (denoted as \( BP/N \)) is the behavior protocol constructed from \( BP \) by systematic replacement of all action tokens the identifiers of which are not prefixed by element from \( N \) by \( NULL \). In the following text \( L(BP) \) denotes the language generated by \( BP \).

Not all languages can be expressed by a behavior protocol. For example, the language, which consists of the following traces cannot be generated by a behavior protocol.

\[
\begin{align*}
a, b \\
a, b, a, b, b \\
a, b, a, b, b, a, b, b, b \\
\ldots
\end{align*}
\]
But behavior protocols can approximate any useful language of a component “closely enough.” The exact meaning of “closely enough” can be different for different component models. As presented in [19], a definition suitable for SOFA is to define protocol in the way, that the behavior of a component can allow “more” on provides-interfaces and it has to want “less” on requires-interfaces. This is formally defined as follows.

**Definition.** Let C be a component and P<sub>C</sub> resp. R<sub>C</sub> be the sets of all C’s provides-interfaces resp. requires-interfaces. We say that the behavior represented as language L<sub>C</sub> of the component C on a set of its interfaces SI is bounded by a protocol BP if both of the following inclusions hold:

\[
\begin{align*}
(1) & \quad L(BP)/(P_C \cap SI) \subseteq L_C/(P_C \cap SI) \\
(2) & \quad L(BP)/(R_C \cap SI) \supseteq L_C/(R_C \cap SI)
\end{align*}
\]

To show expressive power of behavior protocols, let us specify a protocol for incoming calls on an interface, which is to provide a database access service by methods *Open, Close, Insert, Delete* and *Query*.

The intended use of this interface can be to call the method *Open* first, then do a modification of the database by invocation of *Insert, Delete* and *Query*, and finally to finish the work with the database by invoking *Close*. The corresponding protocol can take the form 

\[ ?\text{Open} \; ( ?\text{Insert} \; + \; ?\text{Delete} \; + \; ?\text{Query} \; )^* ; \; ?\text{Close} \].

If *Insert, Delete* and *Query* methods are designed to handle their parallel execution, we could specify this intention by 

\[ ?\text{Open} \; ( ?\text{Insert} \; \| \; ?\text{Delete} \; \| \; ?\text{Query} \; ) \; * ; \; ?\text{Close} \].

This protocol indicates that parallel run of the methods *Insert, Delete* and *Query* is possible, but any two invocations of *Insert* must be done sequentially. The same argument holds for *Delete* and *Query*. To specify that completely parallel access to these methods is allowed, the reentrancy (^) operator is to be used instead of the repetition (*). The protocol takes the form 

\[ ?\text{Open} \; ( ?\text{Insert} \; \| \; ?\text{Delete} \; \| \; ?\text{Query} )^\land \; ?\text{Close} \].

Since we have no other clue than interface signature, there could be also used a protocol *Query^\land* ( *Open* ; ( *Insert* + *Delete* )^* ; *Close* )^*. This specifies that the database can be queried any time and one has to use *Open* and *Close* methods for indication of database modification. Modifications can be done only sequentially.
The following sections present employing the behavior protocols into CDL, as presented in [19]. To employ behavior protocols in design process of a software architecture, we associate every interface, frame, and architecture with a protocol to express how the behavior on the different levels should be bounded. Intuitively, these protocols can be seen as the specification of how a particular interface of a template can be used and how instances of the template will behave on their interfaces.

3.4. Interface protocol

As mentioned in Section 3.1, a component’s provides-interface can be viewed as a service definition. Although there is a possibility of more cooperating interfaces, the correct design should allow to use each of them as a separate part of a service. As a result, the idea of semantic description at the level of a component interface is straightforward. The interface protocol allows for describing the ordering dependencies among the invocations on a single interface.

The key question for employing the interface protocols is whether should be a protocol associated with an interface type or with an interface instance. If an interface protocol is associated with an interface instance, definition of interface type cannot be used as a definition of a service. Different services can have the same signature, for example, the signatures of a stack and a buffer consist of Get and Put methods. Since the abstraction of an interface as a service is one of the basic building blocks of component-based programming abstractions, the interface protocol is to be associated with an interface type.

When specifying a protocol for a provides-interface, the protocol should be specified with use of incoming method invocations. On the other hand, a protocol for a requires-interface describes outgoing method invocations, since it specifies the way the interface is going to be used.

These two interface protocols are very similar. They differ only in the direction of invocation, i.e., whether an invocation is incoming or outgoing one. In principle, they describe the same situation from different points of view. As the interface protocol is associated with interface type, the protocol should take into account both situations. Thus, an interface protocol takes a generic
form, where the outgoing/incoming symbols are omitted and the protocol associated with a particular interface in a component template is the simple syntactic, automatically generated, modification of this protocol. For example, the interface protocol $Init;\ Print^*$ is in a generic form. For a requires-interface, it takes the form $!Init;\ !Print^*$ and for a provides-interface it is $?Init;\ ?Print^*$. The following example shows employing the interface protocol into interface type specification.

```plaintext
interface bufferAccess {
    int Get( void);
    void Put( in int item );
    int isEmpty( void );
protocol:
    (Get + Put + isEmpty)^*
};
```

3.5. Frame protocol

The frame protocol allows for description of the black-box view of a component. The black-box view is one of the basic properties of the component abstraction definition. Thus, semantic description, which allows such a view, is one of the most important.

A frame protocol allows to depict the ordering of method invocations among component’s outmost interfaces, it grasps the following dependencies:

1) **The ordering dependencies between a provides-interface and requires-interfaces.** This is one of the main achievements of description presented here. This shows the visible dependencies among invocation of provided methods and invocations on requires-interfaces, so the resulting description is something like “shadow” black-box view, since it allows for a reasoning on a component, but one cannot do any presumptions upon the implementation of a component. For example, the protocol $!a | !b$ allows for reasoning, that the component is to be multithreaded, since $b$ invocations are completely independent of invocations of $a$ by means of invocation ordering.
(2) **Ordering dependencies among requires-interfaces.** These dependencies allow for reasoning on the interface use. This can provide useful information for special situations, like an automatic creation of an adaptor, when tied interfaces do not conform, but are similar enough.

(3) **Ordering dependencies among provides-interfaces.** These dependencies allow for describing the cooperation among the interfaces, which form a service. There is a long list of possible usage, for example, the supervisor and client interfaces, cooperating client interfaces, etc.

To illustrate the syntax of frame protocols, we present the frame *Buffer* with a frame protocol.

```plaintext
frame Buffer {
  provides:
    bufferAccess buf;
  requires:
    memoryAccess mem;
  protocol:
}
```

### 3.6. Architecture protocol

Architecture protocol supports the grey-box view of a component, as the architecture is the basic abstraction in SOFA component model for specifying hierarchies of components (Section 3.1). The architecture protocol, as proposed in [19], is automatically generated protocol allowing to check the correctness of an architecture specification.

We choose to automatic generate the architecture protocol, since we argue, that to specify an architecture protocol by hand can be a very difficult task, mainly because the invocation dependencies implied by bind ties in an architecture.
The generated architecture protocol of a component C is a composition of frame protocols of all C’s subcomponents. The composition operator is equivalent to the and-parallel operator in case of external events. This can be a bit restrictive, but this is a consequence of the independence of subcomponents. In general, the generated protocol depicts the minimal invocation ordering dependencies between the component’s interfaces and the interfaces of its subcomponents. Thus, the cooperation across the hierarchy of components can be grasped.
4. Verification of a component semantic description

The goal of the semantic description is to specify components more precisely. It should also allow automatic testing, whether the specification is correct. Not only the signatures of tied interfaces have to agree, but also the protocols specified can be checked, whether they specify the same behavior. Thus, we introduce the tests to be used to detect the most common design errors, such as the incorrect use of an interface.

In SOFA component model, the component correctness can be defined on two abstract levels. These are the level of component template specification and the level of component use in specified environment.

Verification of the component template correctness is checking of template specification for soundness, i.e., if the protocols specified in the template agree on the communication they describe. Component is used correctly if the way environment, the component will be tied to, communicates with the component via provides-interfaces and component correctly use its requires-interfaces, i.e., it correctly communicates with environment.

These checks are based on protocols specified for a given template and its interfaces. As the protocols are only part of the specification, we must ensure also the signature compatibility of interfaces and the completeness of the architecture ties. However, these checks are already built-in in the CDL compiler. For interfaces, the signature checking is based on the subtype relationship. For checking of protocol agreement, we use the protocol conformance relationship, as defined in [19].

4.1. Naming issues

As presented in previous sections, interface, frame and architecture protocols do not use the same event identification for the same event. It is consequence of the information available at the design time. This naming problem should be solved by automatic tools. The user should be forced to use only absolutely necessary unique identification. Thus, interface protocol should
identify events only by method names, frame protocol should use the qualification by the identification of an interface. Without renaming, architecture protocol could contain an event qualified by different interface names.

The idea of semi-instances of protocols is to be used [19]. Basically, a semi-instance of a protocol is the protocol modified by qualifying event names by the interface identification available. We need to define following three special cases of semi-instances, which would be used for protocol checking. Let \( <F,A> \) be a template.

(1) The semi-instance of an interface protocol \( P \) to reflect the knowledge gained from a frame \( F \) (to reflect in \( P \) the identification and the requires/provides role of the interface instances specified in \( F \) if \( P \) is associated with an interface specified in \( F \)).

(2) The semi-instance of an interface protocol \( P \) to reflect the knowledge gained from an architecture \( A \) (to reflect in \( P \) the identification and the requires/provides role of the interface instances specified in \( A \) if \( P \) is associated with an interface specified in \( A \)).

(3) The semi-instance of a frame protocol of \( F \) to reflect the knowledge gained from an architecture \( A \) (to reflect the ties specified in \( A \) and the identification of subcomponents in \( A \) if \( P \) is associated with \( F \) which is declared as a subcomponent of \( A \)).

In [19] we propose the full-chain form for identification of the events. Since events are transmitted via a chain of tied interfaces, we identify an event by qualifying it by \( <c_1;r_1;c_2;r_2;\ldots;c_n;p_n;\ldots;c_m;p_m> \), where \( c_i \) is a component identification as specified by its parent component’s architecture, \( r_i \) (resp. \( p_i \)) is an identification of a requires-interface (resp. a provides-interface) as specified by a frame of \( c_i \). The semi-instance of the full-chain form identification is only part of the full identification. The case study presented in Section 5 illustrates generated semi-instances of protocols.
4.2. Protocol conformance

To formalize the fact that two protocols “agree” on a communication they specify, we introduce
the protocol conformance relationship. Following definitions (taken from [19]) specify the
protocol conformance for different protocols in component specification.

**Definition.** Let T1 and T2 be interface types and P_{T1} resp. P_{T2} are the interface protocols
associated with T1 resp. T2. We say that the interface protocol of T1 conforms to the interface
protocol of T2 if T1 is a subtype of T2 in the classical sense and L(P_{T1}) ⊆ L(P_{T2}).

**Definition.** We say the frame protocol of F conforms to the interface protocols of interfaces of
F if, for every provides-interface P in F with an interface protocol PP, holds L(PP') ⊆ L(P)/{P},
and, for every requires-interface R in F with an interface protocol PR, holds L(P)/{R} ⊆ L(PR'),
where PP’ (resp. PR’) denotes the semi-instance of PP (resp. PR) with respect to F.

**Definition.** Let <F, A> be a template specification. Let P_{A} be the architecture protocol of A, P_{F}
the frame protocol of F, and PS resp. RS the set of the provides-interfaces resp. requires-
interfaces of F (being also the outmost interfaces of A). We say the architecture protocol of A
conforms to the frame protocol of F if L(P_{F})/PS ⊆ L(P_{A})/PS and L(P_{A})/RS ⊆ L(P_{F})/RS.

To make the list complete, we provide following definition:

**Definition.** Let <F, A> be a template specification. Let P_{A} be the architecture protocol of A and
P (resp. R) be a provides-interface (resp. requires-interface) in F with an interface protocol PP
(resp. RP). We say the architecture protocol of A conforms to the interface protocol of P if L(PP)
⊆ L(P)/{P} and to the interface protocol of R if L(P)/{R} ⊆ L(RP).

4.3. Tests for component correctness

The conformance relationship is the basis for defining the component correctness tests. In this
section we present their definitions.
Let $<F,A>$ be a component template $<F,A>$. To formalize the testing of the specification correctness, we define following sets:

(1)$I_{IS}$ is the set of all interfaces of $F$ subsumed from subcomponent in $A$
(2)$I_{ID}$ is the set of all interfaces of $F$ delegated to subcomponents in $A$

These sets consist only of interfaces of direct subcomponents of $<F,A>$. For testing the correct use, we define following additional sets, which describe only the external ties of $A$, i.e., ties to environment, in which the template should be used. Such environment is formed by components to which is $C$ will be tied to.

(3)$I_{S}$ is the set of all subsumed interfaces of $F$
(4)$I_{D}$ is the set of all delegated interfaces of $F$
(5)$I_{BP}$ is the set of all bound provides-interfaces of $F$
(6)$I_{BR}$ is the set of all bound requires-interfaces of $F$

We do not specify special sets for bound interfaces of the template. Testing for them is equivalent to testing those interfaces in sets $I_{BP}$ and $I_{BR}$ defined for the frames of subcomponents specified in $A$.

In following subsections, we define all tests for verification of a template specification. All of them are based on the conformance relationship as defined in Section 4.2.

4.3.1 Conformance of provides-interface and requires-interface

It checks a bind tie. Let $P$ be an element of $I_{BP}$ and $R$ be a requires-interface, which is bound to $P$. The interface protocol of $R$ has to conform to the interface protocol of $P$. This ensures the environment will use the service provided by $P$ correctly. On the other hand, let $R$ be an element of $I_{BR}$ and $P$ be a provides-interface $R$ is bound to. The interface protocol of $R$ has to conform to the interface protocol of $P$ to ensure the component uses correctly the service provided by environment.
4.3.2 Conformance of two provides-interfaces

It is the test for delegate ties. Let P be an interface from I_D and R be an interface, which is delegated to P by the parent component. The interface protocol of R has to conform to the interface protocol of P. This ensures that the delegation does not limit the functionality provided by the parent component.

4.3.3 Conformance of two requires-interfaces

It is the test for subsume ties. Let P be an interface from of I_s and R be an interface of the parent computer, to which is P subsumed. The interface protocol of P has to conform to the interface protocol of R. This ensures the subsuming will not violate the way R specified to use the service provided by environment. P will not use environment more extensively than R would.

4.3.4 Conformance of frame to provides-interfaces

Let P be a provides-interface of F. For all P, the frame protocol of F has to conform to the interface protocol of P. The frame protocol of F has to allow at least the communication specified by the interface protocol of P. Thus, F implements the service, which would be as much available as specified by P.

4.3.5 Conformance of frame to requires-interfaces

Let R be a requires-interface of F. For all R, the frame protocol of F has to conform to the interface protocol of R, i.e., it does not use the interface R more extensively than specified by the interface protocol of R.

4.3.6 Conformance of architecture to frame

Let \(<F,A>\) be a template. The architecture protocol of A has to conform to the frame protocol of F. This ensures that A implementing F can replace any other architecture, which conforms to F. It is a consequence of the relationship between the frame and the architecture abstractions. A has
to use requires-interfaces as much as specified by the frame protocol of F. A has to provide the service at least as defined by the frame protocol of F.

4.3.7 Conformance of architecture to interfaces

Checking of the conformance of the architecture protocol to the protocols of interfaces is redundant. If the conformance of the architecture to the frame and the conformance of the frame to the interfaces hold, it is easy to prove that the architecture protocol conforms to the interface protocols because of transitivity of the language inclusion.
5. **Case study**

In this section, we illustrate protocol-based description on an example of a simple web browser. As a first step, we have to design the interfaces needed. A browser would provide an interface called *Browse*, which consist of two high-level methods *Open* and *Download*. The *Open* method should open a URL in a browser and *Download* should fetch a file from the network and save it to a file system. The access to these methods will be serialized. Thus, we define the interface as follows:

```java
interface Browse {
    void Open( in String URL );
    void Download( in String URL );

    protocol:
    ( Open + Download )*}
```

For implementation of functionality defined by the *Browse* interface, the browser needs to use a network subsystem and a file system. Therefore, we have to specify the interfaces for accessing these services.

The *FileSystem* interface specifies the access to the file system. It consists of methods for accessing files. For simplicity, we list only the basic ones. The interface protocol specifies, that for every file, it must be firstly opened by calling the *Open* method, then data can be read (resp. write) by *Read* (resp. *Write*) method any number of times. Then, the *Close* method is to be invoked to close the file. The reentrancy operator allows for parallel access to the file system, but the access to one opened file has to be serialized.

```java
interface FileSystem {
    Long Open( in String filename );
    void Close( in Long handle );

    Long Read( in Long handle, out sequence Buffer, in Long size );
    Long Write( in Long handle, in sequence Buffer, in Long size );
```
protocol:
    ( Open ; ( Read + Write )* ; Close )^{
    
    
};

The NetworkSystem interface specifies the access to the network subsystem. Its structure is similar to the structure of the FileSystem interface. The protocol is similar too, just the names of the methods are different.

interface NetworkSystem {
    Long Connect( in String host );
    Long Send( in Long handle, in sequence buffer, in Long size );
    Long Receive( in Long handle, out sequence buffer, in Long size );
    void Close( int Long handle );
protocol:
    ( Connect ; ( Send + Receive )* ; Close )^{
    
    
};

With all the necessary interface types defined, we can specify the frame for a browser. A browser component has to provide the Browse interface and it needs access to the network by the instance of the NetworkSystem interface and to the file system by the instance of the FileSystem interface. The frame protocol states that for every request of opening URL, the component will use the file system and the network unspecified number of times, but every time it will use it correctly by obeying the protocol for a particular file or a network connection. On the other hand, it will use the file system any number of times, but the network will be used only once for every invocation of the Download method. Since we use the or-parallel operator, it is not necessary to use the network at all. Design of this protocol allows implementations not to be forced by the use of the file system and the network, as long as they obey the access protocols of these subsystems.
frame Browser {
    provides:
        Browse browserAccess;
    requires:
        FileSystem filesystemAccess;
        NetworkSystem networkAccess;
    protocol:
        (?browserAccess.Open{
            !filesystemAccess.Open;(!filesystemAccess.Read +
            !filesystemAccess.Write)*;!filesystemAccess.Close)^ ||
            !networkAccess.Close)^}+
        ?browserAccess.Download{
            !filesystemAccess.Open;(!filesystemAccess.Read +
            !filesystemAccess.Write)*;!filesystemAccess.Close)^ ||
            !networkAccess.Close)
        })*
};

At this point, we can test the conformance of the frame protocol to the interface protocols of the browserAccess, filesystemAccess and networkAccess interfaces. We illustrate the conformance on the networkAccess interface. Testing of other interfaces would be similar. The first step to be taken is to generate the semi-instance of the interface protocol with respect to the Browser frame. The networkAccess interface is declared as a requires-interface, thus we must prefix all the action tokens in interface protocol of the Network interface by interrogation point to generate the protocol

    (!Connect;( !Send + !Receive)*; !Close)^
To generate the semi-instance, we have to reflect the identification of an interface in a frame, thus we transform the protocol for `networkAccess` to the final form:

```

With the semi-instance generated, we can restrict the frame protocol of `Browser` to the `networkAccess` interface. The resulting protocol is

```
```

By a definition of the conformance, the frame protocol has to generate a “narrower” language than a requires-interface. Since the semi-instance of the `Network` interface protocol allows completely reentrant access to the network by means of one connection, the frame protocol conforms to the interface protocol.

So far, we have designed the frame of the browser component. We should now design an architecture of the component. In fact, we will design two different architectures to illustrate the possibilities of specification.

The first one is simpler one. It contains only one subcomponent with the frame `BrowserBody.Simple`. This frame contains the whole implementation. Therefore, it specifies all the provides-interfaces and all the requires-interfaces as the `Browser` frame. The protocol specifies the behavior, where for every invocation of the method on the `browserAccess` interface only one access to the network is used. Thus, there are no caching and no possibility of accessing the file system.
frame BrowserBodySimple {
    provides:
        Browse browserAccess;
    requires:
        FileSystem fsAccess;
        NetworkSystem networkAccess;
    protocol:
        )*
};

Following definition specifies the architecture of the browser frame called “Simple” as containing only one subcomponent of the BrowserBodySimple frame and all necessary subsume and delegate ties.

architecture Browser version “Simple” {
    inst BrowserBodySimple body;
    delegate browserAccess to body.browserAccess;
    subsume body.fsAccess to fsAccess;
    subsume body.networkAccess to networkAccess;
};

Let us now introduce the caching mechanism. We define the CacheAccess interface for accessing the cache. It is a simple cache, thus only Get, Put and inCache are specified. The protocol allows completely independent invocations of methods.
interface CacheAccess {
    Long Get( in String filename );
    void Put( in Long handle, in String filename );
    bool inCache( in String filename );
}
protocol:
  ( Get || Put || inCache )^
frame BrowserBody {
    provides:
        Browse browserAccess;
    requires:
        CacheAccess cacheAccess;
    protocol:
        (?browserAccess.Open !cacheAccess.Get ^)
        +
    )*
};

The frame *SmartCache* needs an access to the file system as well as to the network. It provides
only interface for accessing the cache. The protocol is complicated, since the cache has to know
about the network and the file system. For getting the URL, it is necessary to allow an access to
files stored in the file system and in the network. For putting the file into a cache, it is slightly
easier to specify the sub-protocol to capture the communication, since the putting files into a
cache need only to put it to the file system. No network access is needed. An invocation of
*inCache* does not need to access the network, since only a local cache in file system can be
queried.

frame SmartCache {
    provides:
        CacheAccess cacheAccess;
    requires:
        FileSystem filesystemAccess;
        Network networkAccess;
    protocol:
        (?cacheAccess.get !filesystemAccess.Open; (?filesystemAccess.Read + !filesystemAccess.Write) *;
        !filesystemAccess.Close) ^
        ||
}
To finish our smart cache browser, we specify the architecture SmartCache with only two subcomponents, one for browser logic (body) and one for a smart cache (cache), which has to be the instance of the frame SmartCache.

architecture Browser version "SmartCache" {  
  inst BrowserBody body;  
  inst SmartCache cache;  

  delegate browserAccess to body.browserAccess;  
  bind body.cacheAccess to cache.cacheAccess;  
  subsume cache.networkAccess to networkAccess;  
  subsume cache.filesystemAccess to filesystemAccess;  
};

We finish this section by illustrating the architecture protocol of the SmartCache version of Browser. To generate the architecture protocol of the SmartCache architecture, we need semi-instances of the frame protocols of frames BrowserBody and SmartCache with respect to this architecture. Then, the semi-instances are joined by the composition operator, as illustrated by the following architecture protocol. Note that the semi-instance of SmartCache is shorten to improve readability:
+
(?<browserAccess-body:browserAccess>.Download{
!<!body:cacheAccess~cache:cacheAccess>.Get }
})*
||
(?<body:cacheAccess~cache:cacheAccess>.get{
<!cache:filesystemAccess-filesystemAccess>.Open;
(!<!cache:filesystemAccess-filesystemAccess>.Read
+
<!cache:filesystemAccess-filesystemAccess>.Write
}*;
||
(!<cache:networkAccess-networkAccess>.Send + !<cache:networkAccess-
}
|| ...
||
(?<body:cacheAccess~cache:cacheAccess>.inCache{
<cache:filesystemAccess-filesystemAccess>.Open;
(!<!cache:filesystemAccess-filesystemAccess>.Read
+
<!cache:filesystemAccess-filesystemAccess>.Write
}*;
}
)^
6. **Automation of correctness checking**

For practical use of the semantic description, it is necessary to develop tools for automatic verification of specifications. The CDL compiler checks the specification in terms of CDL language and the completeness of a specification. Thus, we need only to provide tool for automatic checking of the protocol part of specifications. Since all the correctness checks are based on the language inclusion of protocol-generated languages, this is the only functionality needed to allow complete correctness tests as defined in Section 4.3.

For simplicity of use, the Behavior Protocol Grammar defines a few abbreviations. We eliminate all the abbreviations by replacing them by the basic operators for description of algorithms and for discussion on their correctness. The basic operations are sequence, alternative, and-parallel, repetition and reentrancy. These operators are basic in the sense that none can be replaced by combination of others. As the atomic operands, we specify the NULL and action tokens. Replaced abbreviations are simple incoming call, simple outgoing call, nested incoming call and the or-parallel operator.

For conformance testing, there is no need for the composition operator to be supported by an automatic tool. It is only used for architecture protocols and these are tested only on a set of provides-interfaces or a set of requires-interfaces at the time. The composition operator is used only to force the ordering among provides-interfaces and requires-interfaces and these are eliminated by the conformance test definitions. Thus, we do not take the composition operator into account for further design.

To design an automatic tool, it is necessary to identify how complex the generated languages are. The elementary NULL and action token protocols with sequence, repetition and alternative operators define standard regular expressions. Thus, without any other operator these would generate the regular languages, for which is the inclusion testing a well-known and successfully solved problem. The class of regular languages is the same as the one accepted by finite-state machines. In Section 6.1.5 we present how to construct the finite-state machine for the and-
parallel operator. Thus, the protocols, which do not use the reentrancy operator still generate regular languages.

The reentrancy operator damages the regularity of the resulting language, as can be easily proved for language generated by the protocol \((a;b;c)^\omega\). The language is not even context-free. When speaking in terms of finite-state automata, an automaton can be constructed for any number of parallel execution, but the maximal number must be known at automaton construction time. Such an automaton is just the and-parallel operator used specified number of times.

Thus, we design a tool only the subset of the problem. The idea is to require for both protocols, which are tested for inclusion, to include equivalent reentrant subprotocols. If one protocol includes a reentrant sub-protocol, other protocol has to include an equivalent reentrant sub-protocol. These sub-protocols are then replaced by special events. If the protocols use the and-parallel operator for these special events, we would need to specially handle this kind of parallelism in protocol. Thus, we further reduce the subset of tested protocols to only those, which contain only equivalent subprotocols containing reentrant subprotocols used as operands to the and-parallel operator. The whole subprotocols are then replaced be the special action token. For protocol \(\alpha\) we denote \(\alpha'\) the modified protocols. The replaced subprotocols must be equivalent in both tested protocols. A protocol \(A|B\) is equivalent with a protocol \(C|D\) if \(A\) is equivalent with \(B\) and \(C\) is equivalent with \(D\) or \(A\) is equivalent with \(D\) and \(B\) is equivalent with \(D\).

For example, the protocol \(a;(b|c)\) is transformed to the protocol \(a;x\), where \(x\) represents a special event. But the protocol \(a;(b|c)\) would remain the same, since either operand of the and-parallel operator does not contain a reentrant sub-protocol.

Formally, we modify the semantics of the behavior protocols. We generate modified language \(L'\) for given behavior protocol by adding a new special kind of action token to be used in resulting traces. Intuitively, they represent subprotocols by means of parallelism. Thus, we define a new semantics for the and-parallel operator and for the reentrancy operator in following way, semantics of other operators is the same.
And-Parallel execution (α | β)

If L(α) and L(β) do not contain a trace with an occurrence of a special action token, the resulting language consists of all possible interleaving of traces from L′(α) and L′(β). Otherwise, the resulting language consists of only one trace. It is formed by the new special action token e_0.

Reentrancy (α^)

L′(α^) = { e_α }, where e_α is special action token unique for the equivalent reentrant automata.

The question is, whether the class defined by modified protocols is equivalent to the class of protocols we address. Formally, we need to prove following claim.

Claim. Let α and β be protocols from our class, α′ and β′ be the modified protocols. Then

L(α) ⊆ L(β) ⇔ L′(α′) ⊆ L′(β′)

Proof sketch. Firstly, let us discuss the implication L(α) ⊆ L(β) ⇒ L′(α′) ⊆ L′(β′). We use a proof by contradiction. Let there be a trace t such that t ∈ L(α) ⊆ L(β) and t′ ∈ L′(α′) & t′ ∈ L′(β′), where t′ is the trace, to which t corresponds. It is the trace, which can be generated from t by replacing all sub-traces generated by eliminated parts of α by the corresponding special action tokens. It can be also defined as t′ generates t by replacing all special action tokens by a sequences of standard action tokens.

t′ must contain at least one special action token. If not, then t=t′, since L(α) ⊆ L(β) and all traces, which does not contain special action tokens are also in L′(α′) resp. L′(β′). Now, there are two possibilities. Let t′_β be the trace which can be generated in the same way as t′, but for protocol β,

(1) t′ contains the same special action tokens, but in different order than t′_β
(2) t′ contains a different special action token, which is not in t′_β
(1) \( t' = a_1, a_2, ..., x_1, ..., x_2, ..., a_n \) and \( t'_\beta = a_1, a_2, ..., x_2, ..., x_1, ..., a_n \). But both of the traces generate the same resulting trace \( t \) in \( L(\alpha) \). Thus, \( x_1 \) and \( x_2 \) are equivalent and \( t' = t'_\beta \in L'(\beta') \). This is a contradiction.

(2) \( t' = a_1, a_2, ..., x, ..., a_n \) and \( t'_\beta \) does not contain \( x \) and both of them generate the same trace \( t \). Thus, \( t'_\beta = a_1, a_2, ..., y_1, y_2, ..., a_n \), where \( y_1, y_2 \) generate the same trace as \( x \). But \( x \) represents and-parallel operator or reentrancy operator. None of them can be replaced by a sequence of two protocols. Therefore, there is no such \( x \). This is a contradiction.

For the other implication \( L(\alpha) \subseteq L(\beta) \Leftrightarrow L'(\alpha') \subseteq L'(\beta') \) we use similar proof. Let assume there are traces \( t' \in L'(\alpha') \subseteq L'(\beta') \) and \( t'_\alpha \not\in L(\alpha) \) & \( t'_\beta \not\in L(\beta) \). \( t' \) has to contain special action tokens. \( t'_\alpha \) is a corresponding trace in \( L(\alpha) \) and \( t'_\beta \) is a corresponding trace in \( L(\beta) \), \( t'_\alpha \neq t'_\beta \). Let \( x \) be a special action token in \( t' \). It is represented as the same sequence of action tokens in both \( t'_\alpha \) and \( t'_\beta \). But the resulting traces are not the same. There are two possibilities:

(1) \( x \) is represented as a sub-trace, i.e., it represents a sequence \( b_1, b_2, ..., b_m \) in a form of ..., \( a_j, b_1, ..., b_m, a_{j+1}, ... \)

(2) \( x \) is represented as a set of sub-traces, i.e., it represents a sequence \( b_1, b_2, ..., b_m \) in a form of ..., \( a_j, b_1, ..., b_p, a_k, ..., a_l, b_{p+1}, ..., b_m, a_{k+1}, ... \)

In case (1), if \( x \) is represented in this way, it is done so in both \( t'_\alpha \) and \( t'_\beta \). Therefore, \( t'_\alpha = t'_\beta \). This is a contradiction. In case (2), \( x \) is represented as an interleaving of the action tokens. But this is possible only by the reentrant operator and the and-parallel operator. Thus, \( x \) is an operand of one of these operators, thus, it cannot be contained in \( L'(\alpha') \) or \( L'(\beta') \).

At this point, we have to design a tool for inclusion testing of these modified languages. In remaining subsections a protocol denotes a modified protocol. By elimination of the reentrancy operator we reduce our problem to the inclusion test for regular expressions. Thus, we test this relation by constructing acceptors for languages and we test their equivalence.
6.1. **Construction of automaton**

The automaton, which accepts the language generated by a given modified behavior protocol, can be created by the step-by-step building process based on the derivation tree of the protocol. In following sections we discuss how to build an automaton based on the automata of the operator’s operands. Formally, we represent an automaton as the quintuple \( <Q, X, F, q_0, \delta> \), where \( Q \) is the set of all states, \( X \) is the input alphabet, \( F \) is the set of accepting states, \( q_0 \) is the starting state and \( \delta : Q \times X \rightarrow \{Q\} \) is the non deterministic transition function.

6.1.1 **Atomic operand**

The basic building blocks of behavior protocols are action tokens and NULL (empty protocol). The automata created for such elementary protocols are very simple.

For the NULL protocol, it contains only one state without any edges. This is the starting and the accepting state. Formally, the resulting automaton is defined as ( \{q\}, \varnothing, \{q\}, q, \varnothing).

For protocol containing only one action token \( e \), the automaton consists of the state \( q \) and the state \( q' \), which are connected by one edge from \( q \) to \( q' \). The transition is labeled by \( e \). The state \( q \) is the starting state and \( q' \) is the only accepting state. Formally, it is ( \{q,q'\}, \{e\}, \{q'\}, q, \delta). where \( \delta(q, e) = q' \) and \( \delta \) is not defined elsewhere.

6.1.2 **Sequence**

Let \( \alpha \) and \( \beta \) be protocols, \( A_\alpha = (Q_\alpha, X_\alpha, F_\alpha, q_{0,\alpha}, \delta_\alpha) \) and \( A_\beta = (Q_\beta, X_\beta, F_\beta, q_{0,\beta}, \delta_\beta) \) their automata. The automaton of the protocol \( \alpha ; \beta \) is constructed as follows. For every accepting state of \( A_\alpha \) we create its own copy of \( A_\beta \). The starting state of this copy is then merged with the corresponding accepting state of \( A_\alpha \). The starting state of the resulting automaton is the starting state of \( A_\alpha \). The set of accepting states is the union of accepting states of all copies of \( A_\beta \).
Formally, it is \((Q, X, F, q_0, \delta)\), where

\[
\begin{align*}
Q &= Q_\alpha^\emptyset F_\alpha x Q_\beta \\
F &= \{ (q, q') ; q' \in F_\beta \land (q, q') \in Q \} \\
X &= X_\alpha^\emptyset X_\beta \\
q_0 &= q_{0,\alpha} \\
&= (q_{0,\alpha}, q_{0,\beta}) \\
\delta(x, e) &= \{ q ; q \in \delta_\alpha(x, e) \land q \in F_\alpha \} \\
&\cup \{ (q, q_{0,\beta}) ; q \in \delta_\alpha(x, e) \land q \notin F_\alpha \} \\
\delta(x, e) &= \{ (q_1, q') ; q' \in \delta_\beta(q_2, e) \} \\
&\text{if } x = (q_1, q_2) \in F_\alpha x Q_\beta
\end{align*}
\]

6.1.3 Alternative

Let \(\alpha\) and \(\beta\) be protocols, \(A_\alpha = (Q_\alpha^\emptyset F_\alpha X_\alpha q_{0,\alpha} \delta_\alpha)\) and \(A_\beta = (Q_\beta^\emptyset F_\beta X_\beta q_{0,\beta} \delta_\beta)\) their automata. The automaton for accepting the protocol \(\alpha + \beta\) is constructed by the merging the starting states of \(A_\alpha\) and \(A_\beta\). The resulting state is the starting state of \(A_{\alpha + \beta}\). The set of accepting states comprises all accepting states of \(A_\alpha\) and \(A_\beta\).

The necessary precondition for this construction is the property (noted as ALT), that \(A_\alpha\) (resp. \(A_\beta\)) does not contain an edge incoming into the starting state of \(A_\alpha\) (resp. \(A_\beta\)). To ensure this property, automata built by the repetition operator must be created in slightly ineffective way (in terms of size). No other operator can damage the property, since it holds for elementary protocols and none of the sequence, alternative and and-parallel introduce edges incoming into the starting state of the resulting automaton.
Formally, the automaton for $\alpha+\beta$ is $(Q,X,F,q_0,\delta)$, where

\[
\begin{align*}
Q & = Q_\alpha \cup Q_\beta \setminus \{q_{0,\alpha}\} & \text{if } q_{0,\beta} \notin F_eta \\
F & = F_\alpha \cup F_\beta \\
      & = F_\alpha \cup F_\beta \setminus \{q_{0,\beta}\} \cup \{q_{0,\alpha}\} & \text{if } q_{0,\beta} \notin F_eta \\
X & = X_\alpha \cup X_\beta \\
q_0 & = q_{0,\alpha}
\end{align*}
\]

\[
\delta(x,e) = \begin{cases} 
\delta_\alpha(x,e) & \text{if } x \in Q_\alpha \setminus \{q_{0,\alpha}\} \text{ and } e \in X_\alpha \\
\delta_\beta(x,e) & \text{if } x \in Q_\beta \setminus \{q_{0,\beta}\} \text{ and } e \in X_\beta \\
\delta_\alpha(q_{0,\alpha},e) \cup \delta_\beta(q_{0,\alpha},e) & \text{if } x = q_{0,\alpha} \text{ and } e \in X_\beta
\end{cases}
\]

### 6.1.4 Repetition

Let $\alpha$ be a protocol, $A_\alpha = (Q_\alpha, X_\alpha, F_\alpha, q_{0,\alpha}, \delta_\alpha)$ its automaton. An automaton for the protocol $\alpha^*$ is constructed as follows to hold the ALT property. To every accepting state of $A_\alpha$, are added copies of all edges of starting state. The starting state also becomes another accepting state.

Formally, it is $(Q,X,F,q_0,\delta)$, where

\[
\begin{align*}
Q & = Q_\alpha \\
F & = F_\alpha \cup \{q_{0,\alpha}\} \\
X & = X_\alpha \\
q_0 & = q_{0,\alpha}
\end{align*}
\]

\[
\delta(x,e) = \begin{cases} 
\delta_\alpha(x,e) & \text{if } x \in F_\alpha \\
\delta_\alpha(x,e) & \text{if } x \notin F_\alpha
\end{cases}
\]

### 6.1.5 And-parallel

Let $\alpha$ and $\beta$ be protocols, $A_\alpha = (Q_\alpha, X_\alpha, F_\alpha, q_{0,\alpha}, \delta_\alpha)$ and $A_\beta = (Q_\beta, X_\beta, F_\beta, q_{0,\beta}, \delta_\beta)$ their automata. If $\alpha$ or $\beta$ contain a reentrant subprotocols, then the resulting automaton is $A = (Q, \{q^*, q_{\epsilon_\beta}\}, \{q^*\}, q, \delta)$, where $\delta(q,e_{\epsilon_\beta}) = q^*$. If $\alpha$ or $\beta$ do not contain any reentrant subprotocols the automaton for the protocol $\alpha \parallel \beta$ is defined on the set of states $Q_\alpha \times Q_\beta$. The resulting automaton contains an edge from state $S_1 = (q_{1,1}, q_{1,2})$ to the state $S_2 = (q_{2,1}, q_{2,2})$ by the transition $e$ only if $A_\alpha$ contains the transition
from $q_{1, 1}$ to $q_{2, 1}$ by $e$ or $A_\beta$ contains the transition from $q_{1, 2}$ to $q_{2, 2}$ by $e$, but not both of them. Thus, the event triggers the transition only in one of $A_\alpha, A_\beta$ exclusively.

Formally, it is $(Q, X, F, q_0, \delta)$, where

$$Q = Q_\alpha \times Q_\beta$$

$$F = \{ (q, q'); q \in F_\alpha \& q' \in F_\beta \}$$

$$X = X_\alpha \cup X_\beta$$

$$q_0 = (q_{0, \alpha}, q_{0, \beta})$$

$$\delta((q, q'), e) = \{ (s, q'); s \in \delta_\alpha(q, e) \} \cup \{ (q, s); s \in \delta_\beta(q', e) \}$$

6.1.6 Reentrancy

Let $\alpha$ be a protocol. The automaton for the protocol $\alpha^\wedge$ is defined as $(\{q, q'\}, \{e_\alpha\}, \{q'\}, q, \delta)$, where $\delta(q, e_\alpha) = q'$ and $\delta$ is not defined elsewhere.

6.2. Event restriction on automaton

One of the basic operations required for conformance checking is an implementation of the event restriction. It is to be implemented on the language generated by a protocol. As we represent the language in form of the automaton, we have to implement event restriction on an automaton.

The idea is to eliminate the edges, which are based on the action tokens to be omitted. By eliminating all the edges, we get the automaton for the resulting language of the event restriction. It is not enough to eliminate the edge and to join the target of this edge with the origin state of the edge, since the target state can be the target of other edges but the origin state is not, for example, see Figure 2a. Figure 2b shows the result of simply merging the origin and target states.
To solve the problem, we split the target state into two. Let us consider the situation on Figure 2a. The edge $x$ is to be eliminated. The idea is to split the state $B$ into two states, $B_1$ and $B_2$ (Figure 3a). Then, the states $A$ and $B_2$ are merged in the way, that all edges outgoing from $B_2$ become outgoing edges of $A$. But, as we can see on Figure 3b, the edge $e$ creates the problem. At this point, the state $A$ is not equivalent with the original state $A$, because the automaton from this state accepts also suffixes starting with a sequence of $e$. To solve such situations, we need to “unwind” one repetition of $e$ by creating a new edge $e$ from $A$ to $B$, as can be seen of Figure 3c.
Formally, we can specify the resulting automaton as follows. Let $A=(Q,X,F,q_0,\delta)$ an automaton, $d$ is a state, which contains outgoing edge $e \in R$ and $d'=\delta(d,e)$. The resulting automaton $(Q,X,F',q_0,\delta')$ is defined as

$$
\begin{align*}
F' &= F \cup \{d'\} & \text{if } d \in F \\
&= F & \text{if } d \notin F \\
\delta'(d,x) &= \delta(d',x) & \text{if } x \neq e \\
\delta'(q,x) &= \delta(q,x) & \text{if } q = d
\end{align*}
$$

To eliminate all edges, we can repeat this algorithm for eliminating one edge, till there are no edges left for elimination. This is not correct, since the “unwinding” creates new edges, which are to be possibly eliminated too. Thus, the algorithm could be infinite. But, if we do not create these, the number of edges to be eliminated does not increase, thus, the modified algorithm is finite.

The algorithm eliminates edges, if they should be eliminated. But, in case of special action token it is not clear, whether to eliminate an edge or do not. For special tokens which represent reentrant sub-protocol, the edge can be eliminated if the whole sub-protocol should be eliminated. For special tokens which represent and-parallel sub-protocol, the same reasoning would eliminate important case, where restriction eliminates the reason for and-parallel sub-protocol to be represented by a special action token. In this case, the transition has to be replaced by a sub-automaton.

We could formally define the following modification for an automaton to specify the replacement of the special action token $x$ from the state $d$ to the state $d'$:

$$
\begin{align*}
\delta(d, e) &= \delta(\alpha, e) \\
\delta(x, e) &= \delta(d', e) & x \in F_{\alpha}
\end{align*}
$$

where $\alpha$ subscript denotes parts defined by the restricted sub-automaton.
6.3. **Deterministic vs. non deterministic automaton**

The determinism is a key requirement for finding isomorphism of two automata. But the construction as described in Section 6.1 does not ensure the determinism of the resulting automaton. Thus, the general algorithm is needed for transforming a non-deterministic automaton into a deterministic one. We use the algorithm, which creates the automaton with a new set of states, which represent a set of all states reachable by a transition. We create the new automaton, whose states represent power set of original set of states. The transitions are defined as follows. If the S is the a subset represented by the state $s_i$, the transition by e is defined to the state $s'_i$ which represents the set $S'$. $S'$ comprises all states of the original automaton reachable by e from states in S. Formally, $S' = \{ s'; \exists s \in S \ s \in \delta(s,e) \}$.

This gives an exponential time and space complexity of the algorithm by means of number of states, since the number of new states can be exponential. But, the worst case scenario is very unlikely, since protocols usually do not create a large number of transitions from every state. The most expensive operator by means of size of the resulting automaton is the and-parallel operator.

6.4. **Reduction of automaton**

To identify the automata, which accept the same language, we need to eliminate redundant states. Idea is to find the sets of equivalent states and then create the automaton, where each set will be represented by the single state. Two states are equivalent, if they accept the same set of suffixes.

Formally, we need to find maximal equivalence $\sim$, such that $q$ is equivalent with $q'$ by means of standard finite-state machines. But, as the equivalence of special action token transitions is defined by means of equivalence of automata, the equivalence is defined recursively. To find such equivalence, we deploy the following algorithm. As the precondition, we require the automaton A to be deterministic.

We will define the equivalence by repeating the refining of the sets of equivalent states in following way. At the beginning, there are two sets of equivalent states. One consists of all
accepting states and other contains all other states. In every repetition, we define new equivalence set by:

\[ q \sim_{1,1} q' \land \forall x \in X \delta(q,x) \sim_{1,1} \delta(q',x) \]

The algorithm stops if \( \sim_{1,1} \) is the same as \( \sim_{1} \).

### 6.5. Language inclusion checking

Based on algorithms presented in previous sections, we can define an algorithm for language inclusion checking, i.e., to answer the question \( L(P_1) \subseteq L(P_2) \) for behavior protocols \( P_1 \) and \( P_2 \).

Again, we take the idea from finite-state machines, for which the inclusion checking can be done by constructing the automaton for union of languages \( L(P_1) \cup L(P_2) \) and then checking the equivalence of the automata for \( L(P_2) \) and \( L(P_1) \cup L(P_2) \). For creating the automaton for union of languages, we use the alternative operator, i.e., we query the equivalence of the protocols \( P_2 \) and \( P_1 + P_2 \). Next, we reduce the automata for \( L(P_2) \) and \( L(P_1 + P_2) \) and we try to lookup an isomorphism between them. If it exists, then the languages are in inclusion. Otherwise we claim that they are not.

The correctness of this algorithm for finite-state machines is based on claim, that all reduced automata, which accepts the same language are isomorphic. Time and space complexity of the algorithm is exponential, because of complexity of the algorithm for transforming a non deterministic automaton into deterministic one.

This algorithm solves the inclusion test for the class of behavior protocols as defined at the beginning of this section. As said in Section 6.3, the transformation of a non deterministic automaton into deterministic one can be expensive operation. But, in an average case it is not as much expensive. Another important complexity issue is the equivalence of action tokens. All previous sections use the equivalence operator. But the equivalence test for special action token introduces the recursive invocation of the automaton equivalence algorithm. There can be two invocations at one level of nesting of special action tokens for two parallel special action tokens.. This gives us \( 2^l \) invocations, where \( l \) is number of parallel or reentrant operators nesting.
7. Implementation

We designed a tool called a conformance verifier to incorporate the algorithms discussed in previous sections. It verifies, whether a protocol conforms to another one. Both protocols are specified as input of the utility. Optionally, a user can specify sets of event names to restrict the resulting languages. The output is a boolean value indicating, whether the isomorphism was found or not.

This tool is to be incorporated into the CDL compiler. The compiler should use it to verify the conformance relations specified in Section 4.3. This is less complex solution than to program a complete verifier of the correctness of a specification. It would introduce the complete parser of the CDL language and double the parts of the CDL compiler. One important result of this solution is the way to solve naming issues discussed in Section 4.1. It is necessary to incorporate information of the whole specification in CDL. Thus, the CDL compiler should be responsible for transforming a protocol to the correct semi-instance. As the implementation presented here is only a prototype, we do not take the naming into the account for further discussion.

We choose the Java programming language as an implementation language for its system independence and the build-in garbage collector. Also, the object-oriented approach allows for easier implementation.

The basis of implementation is the representation of an automaton. We choose to represent it as a general graph. Thus, we introduce the classes for representing an edge, a state and an automaton itself. In following sections, we present the most important implementation details.

7.1. Representation of edges

The representation of edges is not as straightforward, as one can think. Section 6 has introduced three different kinds of transitions one for standard action tokens and two for special action tokens. We distinguish between special action tokens for reentrant subprotocols and for parallel subprotocols containing special action tokens. This is only for implementation purposes, since
we represent the parallel subprotocols as two subprotocols for easier implementation of the special action token equivalence verification.

The access to the transition is defined by an interface (G H). It declares the methods, which are common for all three kinds of edges. The method `7DUJ HW` is the access method for querying the target of an edge. Method `LV7UDQVLWLRQ TXLYDOHQMQR` is supposed to return `WUXH`, if this edge has the same transition action token as the one passed as an argument. This method should be redefined for every implementation of this interface. The last method specified is for querying an edge, whether it should be eliminated by the event restriction algorithm.

```
LQWHUFLFHV (G H)
  6WDWH 7DUJ HW
  ERROHDQ LV7UDQVLWLRQ TXLYDOHQMQR (G H H
  ERROHDQ HQLPLQDWHFPA SHWWULFVLRQ 6HW UHVVVLFW
```

As said above, we use three different implementations of the (G H) interface. `H6LPSOH` is to implement a standard action token. Thus, for constructor one can pass the action token as a string. For equivalence of transition it only queries the case sensitive equality of strings. The elimination query is also done by the simple string-based comparing.

The `H5HHQVUUDQW` class is the implementation of reentrant special action token. As the identification of the token is not important, we only represent the reentrant sub-protocol as the whole automaton. Thus, the implementation of `LV7UDQVLWLRQ TXLYDOHQW` is done be querying the equivalence of the sub-protocol automata. The instance of the `H5HHQVUUDQW` can be eliminated by the event restriction if the restricted sub-protocol automaton will contain only one state without edges, i.e., it only accepts the empty trace.

`H3DOOHOHO` represents the special action token for parallel run of two protocols, which contain special tokens. Therefore, it contains reference to two automata as its operands. The equivalence of the transition is done by verifying the equivalence of each automaton as defined in Section 6. The edge should be eliminated if both automata should be eliminated. This is not

47
the most efficient implementation, since the restriction used on sub-protocol automata could eliminate the special action tokens. Thus, the whole special action token could be replaced by a new sub-automaton build in the standard way.

All three implementations also include the possibility of printing an object to the standard output for debugging purposes. This is done by implementing the interface 3ULQWHU.

For manipulating edges, we implement the (G H) DFVRUA class, which allows to create or copy a new edges without necessary code for each implementation.

7.2. Representation of states

A representation of a state consists of just data, which are encapsulated by the object and its methods. Thus, the interface 6VDWH, which specifies all possible operation on states specifies mostly methods for accessing the state data.

One obvious data item is the list of all outgoing edges. Each state has an ID, which is unique in the automaton. This allows for easier printing of an automaton and, also the FRQMDLQV method of PDFKLQH, PSO needs this information for comparing two references to states, even when they are not the references to the same object.

The implementation of an automaton and its frequent use of BFS (Breadth-first search) algorithm introduces the need for marking visited states. For this purpose, the interface specifies access methods to the internal mark of the state. The last important method of the 6VDWH interface is the HGJ HPA 7UDQVLWLQRQ which uses the LV7UDQVLWLQRQ TXLYDQHQW method for lookup of the edge with a given transition.
The implementation of this interface is the class `WVDW`, `PSO`. It includes two constructors - the constructor for a new state and the copy constructor. The edges are stored in a linked list.

```
FODV VVDW; PSO LPSCHPHQOW 3ULQMHU 6WVDW ^
    SULYVVDW / LVW HG| HV
    SULYVVDW LQW LG
    SXEOLF VVDW, PSO
    SXEOLF VVDW, PSO 6WVDW Q
    SXEOLF YRLG 3ULQW
    SULYVVDW ERROHDOQ PDUN
```

### 7.3. Representation of automaton

The interface `0DFKLO` specifies the methods available for any implementation of an automaton. It consists of the access methods for manipulating states and the methods for transformations. Transformations are represented by methods `5HDFKDEOH` `5HWULFW` `'HWHUPLQL'` `H`, `5HGXFH` `LV`, `VRPRUKLF7R` and `LV( TXLYDOHQWPR` methods.

 `'HWHUPLQL` `H` transforms a possibly non deterministic automaton into a deterministic one. The `5HGXFH` method should reduce the automaton by means of normal reduction of finite-state machines. `LV`, `VRPRUKLF7R` should return true, if the automaton and one passed as an argument are isomorphic. The higher-level comparing method is `LV( TXLYDOHQWPR`, which is the shortcut
for reducing and isomorphism lookup. 5HDFKDEQH method just eliminates all unreachable states of the automaton. 5HWULFW should apply the event restriction, where the argument represents the set of action tokens, which should be eliminated.

A verifier uses the class PDFKLOH, PSO as the implementation of the 0DFKLOH interface. It implements all algorithms discussed in Section 6 as follows.

' HWHUPLQL] H and 5HGXFH implements the standard finite-state machine algorithm, but for identifying the equivalent transitions they use the LV7UDQVLWLQ( TXLYDHOQW This is essential for implementing the inclusion for class defined in Section 6.

The method LV, VRPRUSKLF7R uses BFS for definition of the isomorphism, which is then checked for accuracy on every state and edge. It is also used by the LV( TXLYDHOQW method, which creates the copies of the automata and then applies ' HWHUPLQL] H, 5HGXFH and LV, VRPRUSKLF7R methods.

The PDFKLOH, PSO class also implements the 3ULQVHU interface. Again, there are two constructors. The copy constructor is more complicated, since it has to copy all the states and all the edges of the original automaton. The 5RS\6VWVH method is used for copying one state with all the outgoing edges. The implementation of the copying the outgoing edges requires a recurrent algorithm.
7.4. The Builder class

To use the representation given in previous sections, it is necessary to provide a class for building the automaton from a behavior protocol using the algorithms from Section 6.1. The class \texttt{LOGHU} is designed to do just that. A protocol is passed not to the constructor, but as the parameter of the \texttt{LOG} method. \texttt{LOG} is the only public method of this class. Thus, one instance of \texttt{LOGHU} can be used for creating any number of automata.

The \texttt{LOG} method creates the automata using standard LL(1) parsing. The tokenization of an input reader is done by the method \texttt{QH[ WRNHQ} which returns the string containing the operator or an action token. Tokens are used by \texttt{OWIYLYH}, \texttt{VHTXQFH}, \texttt{DQSDOCHO}, \texttt{RUSDODOHO}, \texttt{WHUP} and \texttt{DFVRU} methods for building the resulting automaton. Since the automaton of \texttt{RUSDODOHO} and \texttt{OWIYLYH} are complicated and these algorithms are used also for \texttt{RUSDODOHO}, we introduced \texttt{FUSHDHTDOCHO} and \texttt{FUSHDHOVWQDNYH} methods. All the building methods return a reference to an instance of \texttt{DFKLQH}.

As the methods of Builder class do the parsing, most of them can throw the \texttt{6QW}( \texttt{UURU} [ \texttt{FHSWLRO} or, 2( [ \texttt{FHSWLRO} for the badly formed protocols. 
6\ QMD( ( UURU( [ FHSWLRO H] WHQGV ( [ FHSWLRO ^
6\ QMD( ( UURU( [ FHSWLRO
6\ QMD( ( UURU( [ FHSWLRO 6WULQ) V
```

6\ QMD( ( UURU( [ FHSWLRO] is raised for protocol syntax errors like badly paired parentheses, etc.
, 2( [ FHSWLRO] is raised in case of an unexpected end of input reader. These exceptions are not
caught by any class of verifier. Thus, the execution of the FKHFNHU class can result in an
unhandled exception. This is for easier integration into the CDL compiler and a more user-
friendly wrapper can be easily done.

SXEOLF FCDV %KLOGHU ^
  SULYDMH FKDQ QH[ WBKFDU
  SULYDMH 5HDGHU VRXUFH
  SULYDMH 6WULQ] QH[ WBWRNHQ
  SULYDMH ERROHDQ LV7RNYHQ,'
  SULYDMH 6WULQ] QH[ WWRNHQ
  SULYDMH 0DFKLOH FUHDMH$0WHUQDMLYH 0DFKLOH P 0DFKLOH P
  SULYDMH 0DFKLOH DQWUQDMLYH
  SULYDMH 0DFKLOH VHTXHQFRH
  SULYDMH 0DFKLOH FUHDMH$0G3DUCDOHO 0DFKLOH P 0DFKLOH P
  SULYDMH 0DFKLOH DGSDUCDOHO
  SULYDMH 0DFKLOH RUSDUCDOHO
  SULYDMH 0DFKLOH WHUP
  SULYDMH 0DFKLOH I DFWRU
  SXEOLF 0DFKLOH %KLOG 5HDGHU SURWRFRO
```

7.5. The Checker class

The entry point of the tool is the PDLQ method of the class FKHFNHU. It accepts the protocols and,
optionally, the restriction sets as the arguments. It also includes the constructor with the
parameters for two 5HDGHU references, which should contain protocols and two 6HW references
for restriction sets. This constructor is intended for CDL compiler interaction as well as the public method $FKHFN$, which returns a boolean value representing the result of the inclusion test.

The $PDLQ$ method of $&KHFNHU$ throws two kinds of exceptions, the $2( [FH5WLRQ]$ in case of I/O problems, and $6\{ QMD\} ( UURU) [FH5WLRQ]$ for badly formed protocols or sets. The automaton, which should recognize the union of two languages is constructed by the alternative operator as specified in Section 6.

$$SXEOLF \ FODV\ FKHFNHU \ ^\ $$
$$SXEOLF \ FKHFNHU \ 5HDGHU \ SURW \ 5HDGHU \ SURW \ 6HVW \ 6HVW \ $$
$$SXEOLF \ VDWLF \ YRLG \ PDLQ \ 6WULQJ \ >@ \ DU \ V \ WKURZV \ 2( [FH5WLRQ \ $$
$$6\{ QMD\} ( UURU) [FH5WLRQ \ $$
$$SULYDWH \ VDWLF \ ERROHDQ \ FRQI \ RUPV \ ODFKLQH \ P \ ODFKLQH \ P \ $$
$$SULYDWH \ VDWLF \ ODFKLQH \ SUDUOHO5XQ \ ODFKLQH \ P \ ODFKLQH \ P \ $$
$$SXEOLF \ ERROHDQ \ FKHFN \ WKURZV \ 6\{ QMD\} ( UURU) [FH5WLRQ \ $$
$$, 2( [FH5WLRQ \ $$
8. Evaluation and conclusion

To meet the first goal of this thesis, we propose the semantic description for the SOFA component model. The description is based on easy-to-read regular expression-like notation called behavior protocols. It satisfies all the requirements for description to be used for component trading. We employ the behavior protocols into CDL at three levels. The interface, frame and architecture protocols form a description hierarchy. The defined conformance relation allows for reasoning about template design and it supports the refinement design process.

A set of verification tests for CDL specification to be performed is specified. All of them are formally based on conformance relation between behavior protocol-based description of different parts of CDL specification. Each test addresses different part of the communication among component interfaces as grasped by different protocols. Granularity of tests ranges from the test for interface ties to the test for complete verification of architecture protocol against the frame protocol. The different tests allow to identify different design errors along with the locus of the potential problems.

The second goal of this thesis is to design a prototype of an automatization tool for verifying the semantic description proposed. As stated, the tests are based on the language inclusion and the languages generated by behavior protocols are not even context-free. Thus, the problem of an inclusion test is the main problem identified for automatization tool design. Because of problem complexity, we define narrower class of protocols, which can be tested for inclusion. Its definition is based on a possibility of elimination of reentrant subprotocols as the primary reason of the problem complexity by requiring the subprotocols to be equivalent. It is restrictive by the way of presence of the reentrant subprotocols, since both tested protocols have to contain the same reentrant subprotocols. To provide a more general solution for them remains the most important open issue. Also, the implementation provided is a prototype and further performance optimalizations are possible.
References


