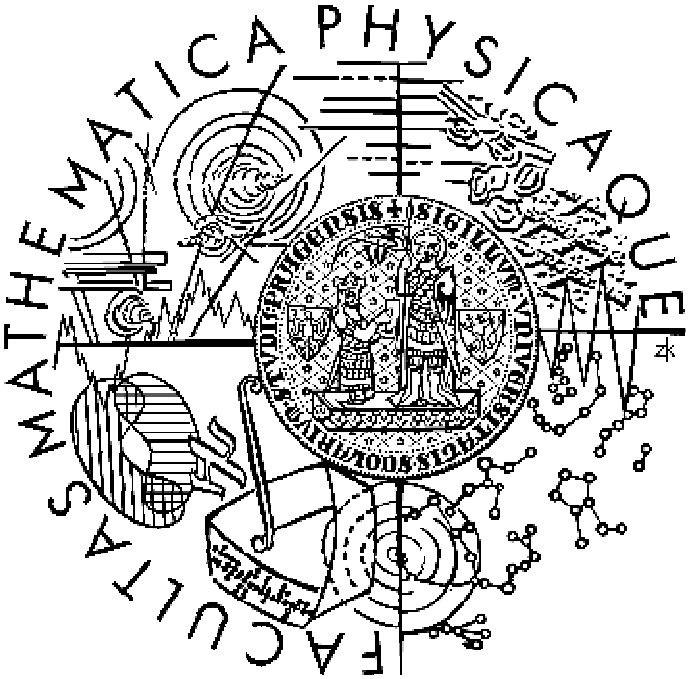
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Ondřej Šerý

Model Checking and Reduction of Behavior Protocols

Department of Software Engineering
Supervisor: Prof. František Plášil
Study Program: Computer Science, Software Systems

First of all, I would like to thank my advisor for his valuable suggestions and observations, Jan Kofroň and Tomáš Poch for fruitful discussions and last but not least my family and Lucie Šmídová for their support in my life and studies.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 21st April 2006 Ondřej Šerý

……

Contents

1 INTRODUCTION 4

1.1 SOFTWARE COMPONENTS 4
1.2 MODEL CHECKING 5
1.3 BEHAVIOR PROTOCOLS 7
1.4 PROBLEM STATEMENT 8
1.5 GOALS 8
1.6 STRUCTURE OF THIS THESIS 9

2 BEHAVIOR PROTOCOLS 10

2.1 BASICS 10
2.2 COMPLIANCE OF COMPONENTS 12
2.3 SUBSTITUTABILITY 13

3 EXPRESSING A PROPERTY 15

3.1 LTL 15
3.2 CTL 17
3.3 HENNESSY-MILNER LOGIC 18
3.4 DISCUSSION 19
3.5 TARGETING BEHAVIOR PROTOCOLS 21

4 REDUCTION OF PROTOCOLS 24

4.1 MOTIVATION EXAMPLE 24
4.2 REDUCTION PREORDER 26
4.3 TERM REWRITING 28
4.4 REDUCTION WITH RESPECT TO COMPOSITION 33
4.5 REDUCTION WITH RESPECT TO PROPERTY 36
4.6 DEPENDENCY GRAPH 38

5 PROTOTYPE 44

5.1 ORIGINAL CHECKER 44
5.2 LTL CHECKING 46
5.3 REDUCTION WITH RESPECT TO COMPOSITION 49
5.4 REDUCTION WITH RESPECT TO PROPERTY 51

6 CASE STUDY 52

6.1 DEMO APPLICATION 52
6.2 COMPONENT REPOSITORY 57

7 RELATED WORK 59

8 EVALUATION 61

9 CONCLUSION AND FUTURE WORK 64

10 REFERENCES 65

APPENDIX 68

RUNNING THE CHECKER 68
COMPILING THE SOURCES 69

Název práce: Model Checking and Reduction of Behavior Protocols
Autor: Ondřej Šerý
Katedra: Katedra Softwarového Inženýrství
Vedoucí diplomové práce: Prof. František Plášil
e-mail vedoucího: plasil@nenya.ms.mff.cuni.cz
Abstrakt:

Behavior protokol je formalismus pro specifikaci chování softwarových komponent. V syntaxi podobné regulárním výrazům jsou definovány přípustné sekvence volání metod, přičemž se abstrahuje od vnitřních dat komponent. Jde sice o rozumnou úroveň abstrakce pro ověření bezchybnosti komunikace softwarových komponent, nicméně pro člověka může být jeho přečtení a pochopení obtížné.

Tato práce se snaží pomoci softwarovému návrháři pochopit specifikaci chování komponent. Předkládá způsob automatického ověřování platnosti obecných časových vlastností vyjádřených v lineární temporální logice spolu s dvěmi technikami redukce behavior protokolů. Redukce vzhledem ke kompozici odstraní ty části protokolu, které nejsou použity v dané kompozici komponent, a zdůrazní tak skutečné role všech komponent. Redukce vzhledem k vlastnosti vypustí ty části protokolů, které nejsou podstatné pro danou vlastnost. Takto redukovaný protokol by měl zdůraznit části, které zapříčiňují platnost dané vlastnosti.

Klíčová slova: behavior protokol, lineární temporální logika, softwarové komponenty, model checking

Title: Model Checking and Reduction of Behavior Protocols
Author: Ondřej Šerý
Department: Department of Software Engineering
Supervisor: Prof. František Plášil
Supervisor's e-mail address: plasil@nenya.ms.mff.cuni.cz
Abstract:

Behavior protocol is a formalism used for behavior specification of software components. In a regular-expression like syntax, admissible sequences of method invocations are specified abstracting from components' internal data. While it seems to be a reasonable level of abstraction for checking correctness of communication of the software components, it can be still quite difficult for a human to read and understand.

This thesis aims to help the software designer to understand the behavior specification of components more easily. An approach to automatic verification of the general temporal properties stated in Linear Temporal Logic is presented along with two techniques for reduction of behavior protocols. Reduction with respect to composition prunes out those parts of the protocols that are not used in the particular composition and clarifies the actual role of each component. Reduction with respect to property removes the parts of the protocols that are irrelevant to the given property. The behavior protocols reduced in this manner should emphasize which part of the protocol makes the given property satisfied.

Keywords: behavior protocol, linear temporal logic, software components, model checking

1 Introduction

Complexity of software systems is growing permanently. In this process, two issues are having more attention. One of them is maintainability of software systems. A complex system can be hard to maintain and modify in a form of a huge monolithic code. The component paradigm, dividing software into functional parts with well-defined interfaces, is a promising approach.

The second issue is correctness of systems. With a growing size of code and extensive use of parallelism, it is impossible to prove correctness by hand. Model checking [9] is a method for proving correctness of a system automatically. It is based on traversing an abstract model of the system and verifying correctness properties stated in a suitable formalism such as temporal logic.

Behavior protocol [35] is a formalism used to describe abstract model of software components by a set of admissible sequences of method calls. Predefined correctness property, absence of communication errors [2] in a hierarchy of components, can be verified on their behavior protocols.

This thesis aims to extend the limited set of the predefined properties that can be verified on behavior protocols. It incorporates common temporal properties stated in the Linear Temporal Logic (LTL). Second contribution of this thesis is the development of reduction methods that trim away the "unimportant" parts of behavior protocols depending on either component composition or a property being verified.

1.1 Software components

Software components may be perceived as an extension of the *Object Oriented Programming* (OOP) paradigm. The concept of components shares many ideas with OOP, like encapsulation, and may be difficult to differentiate at first glance. Component is usually an object, but not every object is a component. In the component systems, independent deployment and easy reuse is emphasized. In general, a component is a building block of software and is well defined via its interfaces and possibly a behavior specification (nothing else should be expected both by the component and its environment). A component should be treated as a black box by its environment, so that it can be easily substituted by another component with appropriate interfaces and behavior. On the other hand, to facilitate components' reuse, components should make no assumptions of their future environment additional to the interface definition. The vision of programming with reusable software components is like playing with LEGO. It is just putting matching prefabricated pieces together.

There are many component systems used in the software industry. To name few of them, Microsoft's component systems' family starts by OCX, ActiveX through COM and DCOM and ends with the .NET component model [28, 27]. Other popular component models are the Enterprise JavaBeans (EJB) by Sun Microsystems [45] and the CORBA Component Model (CCM) by Object Management Group [30]. In the rest

of this section, few concepts of the component models will be presented using the Software Appliances (SOFA) [32] component model's terminology.

Interface of a component is defined by its *frame* - set of its interfaces and a behavior specification. The behavior specification will be covered in Section 1.3. Interfaces are defined by a set of methods that can be called on them. They can be of two types: *provided* and *required*, which mainly specifies only a direction of method calls. Methods on provided interfaces are called on the component by its environment; they are provided to the environment. Methods on the required interface are called by the component on its environment; they are required by the component.

SOFA model allows also *hierarchical* or *composite* components, which are composed of one or more other components. Such a component does not contain actual code but only a specification of its *architecture*, which is a set of inner components and *bindings* of their interfaces. Provided interface of an inner component can be bound to either required interface of another inner component or to a provided interface of the frame of its parent – *delegation*. In a similar vein, a binding between the required interface of an inner component and required interface of the frame of its parent is called *subsumption*. Some of the interfaces can remain unbound, but there is a danger that sooner or later the component will call a method on it or that it will wait for a call, which could result in the runtime error or a deadlock. Component, which is not composite, is called *primitive*. Fig. 1 shows an example of the composite component.

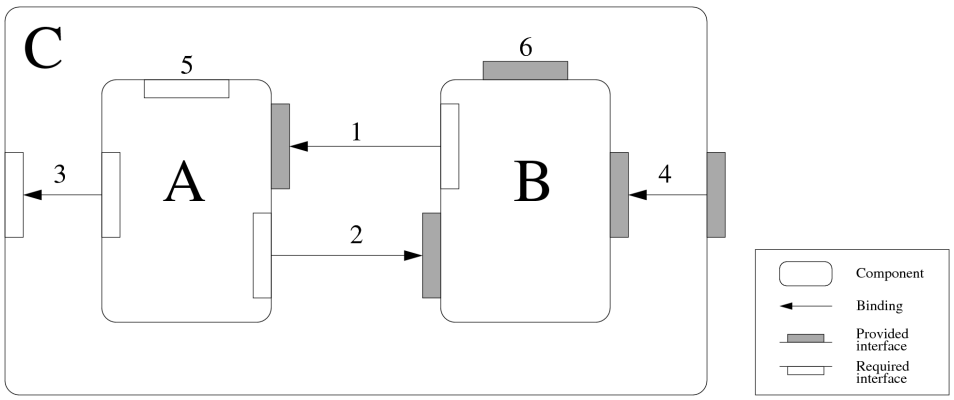


Figure 1. The composite component C consists of A and B, the internal bindings 1 and 2, the subsumption and delegation bindings 3 and 4, respectively. 5 and 6 are unbound interfaces

1.2 Model checking

Although it is a well-known fact that some properties of programs are algorithmically undecidable in general (for example the *halting problem*), the ability to automatically verify at least some properties under certain constraints is crucial in order to guarantee correctness of software systems. Model checking is a technique to achieve this goal; it uses an abstract model of a system to verify correctness properties. The model can be created manually or automatically and can use various levels of abstraction. It is usually finite to ensure decidability of the verification. Unlike testing and simulation, model checking can really prove that a given model satisfies particular correctness properties,

because it is based on exploration of the whole state space of the model. However, if the abstraction of the model is too far from the modeled system, the properties that hold on the model can be violated by the system itself.

The model of the system is usually represented as a *Kripke structure* or a *Labeled Transition System* (LTS), typically defined as follows:

Definition 1. Let AP be a finite set of atomic propositions, then a *Kripke structure* is a quadruple $(S, s_{init}, L, \rightarrow)$ such that:

- S is a finite set of states,
- s_{init} is an initial state,
- $L: S \rightarrow AP^2$ is a labeling function,
- $\rightarrow \subseteq S \times S$, where $\forall s \in S\ (\exists\ s' \in S: (s, s') \in \rightarrow)$, is a transition relation.

Definition 2. *Labeled Transition System* is a quadruple $(S, s_{init}, Act, \rightarrow)$ such that:

- S is a finite set of states,
- s_{init} is an initial state,
- Act is a finite set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation.

LTS is useful to describe action-based systems, where what matters are the performed actions. Kripke structure can represent systems, in which the internal state is more important then the visible actions. See the examples on Fig. 2.

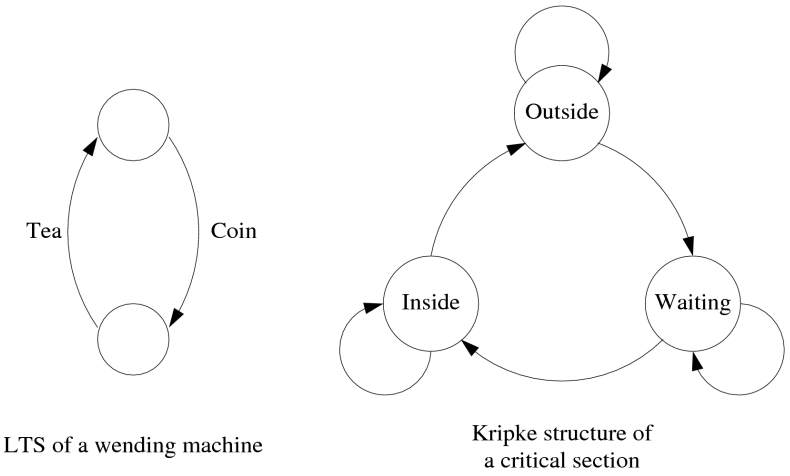


Figure 2. Examples of LTS and Kripke stricture

The correctness properties are most often specified in one (or more) of the following forms:

- Temporal logic formulas
- Assertions

- Pre-conditions and post-conditions

Temporal logic formulas will be covered in Section 3. They are formulas of a logic that is equipped with temporal operators to be able to express changes of the system in time. Assertions are expressions added to the specification of the model (or to the code). They work very like the assertions known from the programming languages, except for the fact that the model checkers can detect violation of assertions in any of the possible system's execution paths. Methods of a system can be also annotated by pre-conditions and post-conditions. Here, an advantage over assertions is the fact that pre- and post-conditions can be used to check only parts of the software system that can be later combined together if the conditions match.

The most painful attribute of model checking is its complexity. When used on the concurrent systems, the models usually feature state spaces exponential in the size of the specification. This problem is known as *state explosion problem* and many techniques are trying to cope with it.

There are several model checking tools available. For example, the model checker SPIN [43] can verify models described in its specification language Promela. For verifying Java code, there are Java PathFinder [21] originally developed at NASA and Bandera [39]. The Zing model checker [3] by Microsoft can verify models specified again in its own specification language.

1.3 Behavior protocols

Behavior protocol [35] is formalism for describing behavior of a software component. As already said, a component is viewed as a black box that can accept method calls on its provided interfaces and issue calls on its required interfaces. From this perspective, it is reasonable to specify behavior of a component by a set of admissible sequences of method calls. Behavior protocol is an expression with a regular-expression like syntax (in addition to "+" and "*" operators, it features also operators for expressing parallel interleaving "|" and "||"). Roughly speaking, the language generated by a protocol defines the set of all valid sequences of variously interleaved method calls on all interfaces of the component.

Having the behavior specification of a component in this form, it is natural to ask whether a communication of two or more components is in a sense correct – protocols are *compliant*. For example, if one component calls a method on the other one, which is not prepared to accept it, then the situation can be perceived as a communication error. However, the opinion, on where the border between the correct and the erroneous communication should be, has developed over time. Moreover, it is very likely to change in the future. This work is based on the definitions from [2], which are given in Section 2 that introduces behavior protocols more formally.

In verifying correctness of component systems, there are two major tasks to accomplish. First, correctness of communication between components has to be verified. And second, behavior protocols of primitive components have to be checked for matching with their code. There is already a behavior protocol checker that can automatically verify correctness of composition, developed as a part of SOFA. There is

also an attempt to verify the primitive components against their protocols using Java PathFinder. See [34] for more information on this topic.

1.4 Problem statement

(1) As already said in Section 1.3, there is an existing implementation of the model checker for behavior protocols. However, by detecting communication errors it can verify only correctness of communication, not supporting verification of general temporal properties, though. The need for the latter can be justified as follows: In a typical case, software designer wants to reuse a component stored in a component repository. Having its behavior protocol, he or she may want to check whether the component meets the application's requirements, which cannot be expressed via protocol compliance (e.g. whether an acceptance of a call of `a` will be in the future followed by a call of `b` provided a call of `c` is accepted in the meantime). The choice of a suitable formalism is crucial. It should be chosen with respect to the specifics of behavior protocols.

(2) Reusable software components usually provide more functionality than is actually used in a concrete application. Behavior protocols of such components tend to be very complicated and their integration to the application floods the resulting architecture with an unused behavior. Reduction of the unused parts of the frame protocols of participating components, so that only the actually used part of the behavior is captured, could significantly simplify understanding the behavior specification.

(3) If it was possible to check some general temporal property of a component's behavior, then it would be also useful to be able to reduce the behavior protocol to contain only parts that are relevant with respect to that property. This feature should allow designer to understand a possibly complex behavior specification more easily.

1.5 Goals

From the open problems listed in Section 1.4, this thesis lays out the following goals:

- The first goal is to address the problem (1) – verification of a general temporal property. This should involve comparison of possible alternative formalisms for expressing the property, discussion on the choice and description of incorporating into behavior protocols. A prototype implementation for verification of the properties stated in the chosen formalism should be developed.
- As a second goal, problems two and three should be addressed. The algorithms for both reduction with respect to composition (2) and reduction with respect to property (3) should be designed and a prototype implementation should be created. Since the aim is to design reduction methods applicable in a checker, any solution with time complexity higher then complexity of the compliance

or property checking, respectively, is unacceptable. The checking process is already exponential in a size of protocols, because the state space generated by protocols is generally exponential. In other words, even a solution performing suboptimal reduction is favorable over an optimal reduction that would require too much time.

1.6 Structure of this thesis

This thesis reflects the goals as follows. In Section 2, a deeper introduction to behavior protocols is given for the sake of completeness. Section 3 discusses choice of a suitable formalism for expressing properties to be checked and presents modifications to be done, in order to apply the formalism to behavior protocols. Section 4 covers reduction of protocols with respect to both composition and property, along with all necessary prerequisites. Later in Section 5, the prototype implementation in a form of extension to the existing protocol checker [38] is described. A brief case study follows in Section 6. It focuses on the use of the reduction on a real application and discusses a typical scenario for the use of the verification of a general property and reduction with respect to that property. Related work is listed in Section 7 followed by an evaluation in Section 8 discussing fulfillment of the goals. The thesis is concluded in Section 9.

2 Behavior protocols

An informal idea of behavior protocols, as the expressions representing valid sequences of the method calls, was already sketched. This section defines behavior protocols more formally. The presented view is based on the *consent operator* and the *consensual compliance* presented in [1, 2].

2.1 Basics

First of all, the syntax of the behavior protocols has to be established.

Definition 3. Let M be a set of methods, then syntactically correct *behavior protocols* are recursively defined as follows:

$$
\begin{aligned}
\wp \;::=\; & \mathsf{NULL} \mid (\wp)^* \\
& \mid (\wp ; \wp) \mid (\wp + \wp) \mid (\wp \mid \wp) \mid (\wp \parallel \wp) \\
& \mid \mathsf{!m\hat{}} \mid \mathsf{!m\$} \mid \mathsf{?m\hat{}} \mid \mathsf{?m\$} \mid \tau\mathsf{m\hat{}} \mid \tau\mathsf{m\$} \\
& \mid \mathsf{!m} \mid \mathsf{?m} \mid \tau\mathsf{m} \\
& \mid \mathsf{!m}\{\wp\} \mid \mathsf{?m}\{\wp\} \mid \tau\mathsf{m}\{\wp\} \qquad \text{for all } \mathsf{m} \in M.
\end{aligned}
$$

The key stone of the behavior protocol is an event, which is atomic. An event represents either method call *request* (start of the call) or *response* (return from the call). There are six basic events: !m^, !m$, ?m^, ?m$, τm^ and τm$. The active call of a method (!m^), passive acceptance of the call (?m^), active return from the call (!m$) and acceptance of the return notification (^m$). Events starting with τ represent the inner events not visible to the environment. It is a completed communication on the inner interfaces. The events !m^ and ?m^ yield into τm^ and !m$ and ?m$ into τm$.

There are four basic operators: ";", "*", "+" and "|". Concatenation ";" concatenates two behavior protocols (first behavior is followed by the second). Operator "+" is the alternative, any of the behaviors described by the operands is considered valid. The finite sequencing operator "*" describes any finite number of repetitions of the behavior specified by the operand. It includes also no repetition, which is an empty protocol "NULL". En empty protocol just specifies a component, which does nothing. At last, operator "|" specifies all parallel interleaving of the two protocols.

An active method call (!m) is an abbreviation of a protocol: !m^; ?m$. That is, a component actively emits a start of the method event (!m^) and then passively waits for the event notifying the method end (?m$). In a similar vein, the abbreviation !m{℘} represents protocol: !m^; ℘; ?m$, which dictates behavior specified by ℘ during the method call m. Other abbreviations are listed in the following:

Definition 4. Let ℘ and ρ be behavior protocols and m an identifier of a method, then the abbreviations can be expressed as follows:

$!m \equiv (!m\hat{}; ?m\$)$

$?m \equiv (?m\hat{}; !m\$)$

$\tau m \equiv (\tau m\hat{}; \tau m\$)$

$!m\{\wp\} \equiv (!m\hat{}; \wp; ?m\$)$

$?m\{\wp\} \equiv (?m\hat{}; \wp; !m\$)$

$\tau m\{\wp\} \equiv (\tau m\hat{}; \wp; \tau m\$)$

$\wp \parallel \rho \equiv (\wp) + (\wp \mid \rho) + (\rho)$

As already mentioned, each behavior protocol defines (or *generates*) a set of admissible traces - *language*. Only finite traces are considered, but the language is not necessarily finite. Two protocols are considered semantically equivalent, if they generate the same language.

Definition 5. Let u, v and w be finite traces and n be the length of u, then u is an *interleaving* of v and w if there exists a function f: $\{1, 2, \ldots, n\} \rightarrow \{0, 1\}$, such that trace defined by symbols of u indexed by numbers, on which f equals to 0, is equal to v and trace defined by symbols of u indexed by numbers, on which f equals to 1, is equal to w.

Definition 6. Let M be a set of methods and $\wp$ and ρ behavior protocols, then the *language of protocol* $\wp$, denoted as $\mathscr{L}(\wp)$, is recursively defined as follows:

$\mathscr{L}(\text{NULL}) = \{<>\}$

$\mathscr{L}(x) = \{<x>\}$ for any $x \in \{!m\hat{}, !m\$, ?m\hat{}, ?m\$, \tau m\hat{}, \tau m\$: m \in M\}$

$\mathscr{L}(\wp + \rho) = \mathscr{L}(\wp) \cup \mathscr{L}(\rho)$

$\mathscr{L}(\wp; \rho) = \{uv: u \in \mathscr{L}(\wp) \wedge v \in \mathscr{L}(\rho)\}$

$\mathscr{L}(\wp^*) = \{<>, u, uu, uuu, \ldots: u \in \mathscr{L}(\wp)\}$

$\mathscr{L}(\wp \mid \rho) = \{u: \exists v \in \mathscr{L}(\wp), w \in \mathscr{L}(\rho): u \text{ is an interleaving of } v \text{ and } w\}$

It can be easily seen, that the expressive power of behavior protocols is equal to the power of regular languages. Examples of behavior protocols and languages they generate follow:

$\mathscr{L}(?m\{!a + !b\}) = \mathscr{L}(?m\{!a\} + ?m\{!b\}) =$
$\{<?m\hat{}, !a\hat{}, ?a\$, !m\$>, <?m\hat{}, !b\hat{}, ?b\$, !m\$>\}$

$\mathscr{L}(?a \mid ?b) = \{<?a\hat{}, !a\$, ?b\hat{}, !b\$>, <?a\hat{}, ?b\hat{}, !a\$, !b\$>,$
$<?a\hat{}, ?b\hat{}, !b\$, !a\$>, <?b\hat{}, !b\$, ?a\hat{}, !a\$>, <?b\hat{}, ?a\hat{}, !b\$, !a\$>,$
$<?b\hat{}, ?a\hat{}, !a\$, !b\$>\}$

$\mathscr{L}(?a\{!b^*\}) = \{<?a\hat{}, !a\$>, <?a\hat{}, !b\hat{}, ?b\$, !a\$>,$
$<?a\hat{}, !b\hat{}, ?b\$, !b\hat{}, ?b\$, !a\$>, \ldots\}$

2.2 Compliance of components

Having a composition of components annotated with their behavior protocols, compliance of the protocols can be studied and possible communication errors can be detected. In [2], three types of the communication errors are distinguished:

- *Bad activity* – this communication error occurs when a call is issued on a component which is according to its behavior protocol not prepared to accept it. For example, consider protocols: ?a; ?b and !b; !a. The former describes a component, that is able to accept call of the method a and then b. The latter describes a component that first calls b and then a. This result in a bad activity error on the first call of b.
- *No activity* (or *deadlock*) – it happens when at least one component waits for an event (and cannot finish), but no component is able to emit any. For example, protocols: ?a; !b and ?b; !a describe behavior of components that would result in no activity error, when composed together. The first component waits for the call of a and the second waits for the call of b. Neither one can emit an event and neither one can finish – $<> \notin \mathscr{L}(?a; !b)$ and $<> \notin \mathscr{L}(?b; !a)$.
- *Infinite activity* (or *divergence*) – this error occurs when there is a sequence of events, such that there is no suffix that would allow all components to finish at the same time or that would result in no activity error. Consider the following exemplary protocols: (!a; ?b)* and ?a; (!b; ?a)*. The communication of such component will never produce neither bad activity nor no activity error. They can call the methods a and b forever, but both components will never be able to finish at the same time – the former can finish only after a call of b or at the very beginning and the latter can finish only after a call of a.

The additional *consent operator* ∇ is able to add error tokens representing all types of communication errors to the composition. The precise definition is given in the aforementioned paper [2]. For purposes of this work, only an intuitive idea should suffice. For behavior protocols $\wp$ and ρ and a set of methods X, $\mathscr{L}(\wp \nabla_X \rho)$ contains all interleaved traces of $\wp$ and ρ synchronized on events associated with methods from X. The synchronization means that for each $m \in X$, any event of a form ?m^ or ?m$ will always wait for the corresponding !m^ or !m$ resulting into an internal action τm^ or τm$. In addition, $\mathscr{L}(\wp \nabla_X \rho)$ can also contain paths ending with an error token representing corresponding communication errors, if present. Bad activity is denoted as εm^ or εm$, where m depends on the event that caused the error. No activity and infinite activity are denoted as $\varepsilon\oslash$ and $\varepsilon\infty$, respectively. The set X contains methods that are used for communication. Those are methods of all interfaces that are bound between the two components. Few examples for clarifying the idea follow:

$$\mathscr{L}(?a; !b \nabla_{\{a, b\}} !a; ?b) = \{<\tau a^, \tau a\$, \tau b^, \tau b\$>\}$$

$$\mathscr{L}(?a; !b \nabla_{\{a\}} !a; ?c) = \{<\tau a^, \tau a\$, !b^, ?b\$, ?c^, !c\$>,$$
$$<\tau a^, \tau a\$, !b^, ?c^, ?b\$, !c\$>, <\tau a^, \tau a\$, !b^, ?c^, !c\$, ?b\$>,$$

$$\langle \tau a\hat{}, \tau a\$, ?c\hat{}, !c\$, !b\hat{}, ?b\$\rangle,\ \langle \tau a\hat{}, \tau a\$, ?c\hat{}, !b\hat{}, !c\$, ?b\$\rangle,$$
$$\langle \tau a\hat{}, \tau a\$, ?c\hat{}, !b\hat{}, ?b\$, !c\$\rangle\}$$

$$\mathscr{L}(?a;\ ?b \nabla_{\{a,\,b\}} !b;\ !a) = \{\langle \varepsilon b\hat{}\rangle\}$$

$$\mathscr{L}(?a;\ ?b \nabla_{\{a,\,b,\,c\}} !a;\ ?c) = \{\langle \tau a\hat{}, \tau a\$, \varepsilon\oslash\rangle\}$$

$$\mathscr{L}(\ (!a;\ ?b)^* \nabla_{\{a,\,b\}} ?a;\ (!b;\ ?a)^*\ ) = \{\langle \varepsilon\infty\rangle,\ \langle \tau a\hat{}, \varepsilon\infty\rangle,\ \langle \tau a\hat{}, \tau a\$, \varepsilon\infty\rangle,$$
$$\langle \tau a\hat{}, \tau a\$, \tau b\hat{}, \varepsilon\infty\rangle,\ \langle \tau a\hat{}, \tau a\$, \tau b\hat{}, \tau b\$, \varepsilon\infty\rangle,\ \ldots\}$$

Because the consent operator explicitly adds traces with the communication errors, it can be used to define compliance of components. There are basically two types of compliance: *horizontal* and *vertical*. Given a composite component from Fig. 1, the horizontal compliance relates to the communication on one level of the component hierarchy, between the components A and B. On the other hand, vertical compliance expresses substitutability relation between the architecture (component A and B) and the frame bounding the architecture (C). Both these ideas can be described using the consent operator. Informally, the components A and B are horizontally compliant, if $\mathscr{L}(\text{protocol}_A \nabla_X \text{protocol}_B)$ does not contain any trace with an error token. The set X contains all methods on interfaces bound by 1 and 2.

For definition of the vertical compliance, the *inverted frame* trick is used. The frame protocol of C is inverted; all "!" are substituted by "?" and vice versa. The inverted protocol can be perceived as an independent component representing the environment (see Fig. 3) and the consent operator can be used in the same way as in the case of the horizontal compliance. So the architecture of A and B is vertically compliant with the frame C, if $\mathscr{L}(\ \text{protocol}_C^{-1} \nabla_Y (\text{protocol}_A \nabla_X \text{protocol}_B))$ does not contain any trace with an error token. The set Y contains all methods on subsumed, delegated or unbound, interfaces.

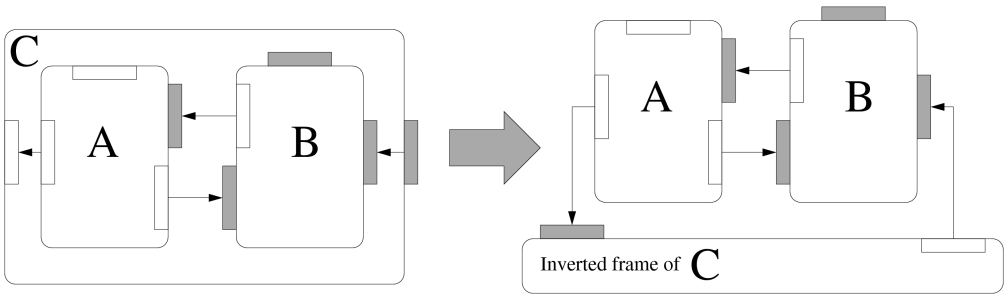


Figure 3. An idea of the inverted frame representing the environment

2.3 Substitutability

For purposes of the reduction of protocols, the precise notion of a *substitutability* of the component by another one will be necessary. It may be viewed as a special case of the

vertical compliance, when the architecture is defined by just a single component and all common interfaces are bound (subsumed and delegated).

Definition 7. Let A and B be components and α and β their behavior protocols, then A is *substitutable* for B, if $\mathscr{L}(\alpha \nabla_M \beta^{-1})$ does not contain any trace with an error token, where *M* is a set of all methods.

The concept of substitutability will be applied to the components and their protocols interchangeably. The relation of substitutability defined in this way is evidently reflexive (for any protocol α, $\mathscr{L}(\alpha \nabla_M \alpha^{-1})$ does not contain any trace with a communication error). Unfortunately, the relation is not transitive. If α, β and γ are behavior protocols, such that α is substitutable for β and β is substitutable for γ, then α is not substitutable for γ in general. It can be proved that $\mathscr{L}(\alpha \nabla_M \gamma^{-1})$ cannot contain trace with a bad activity error; however it can still contain the no activity or infinite activity error. As an example, consider protocols:

α = ?b, β = !a + ?b and γ = !a, or

α = (!a; !a)*, β = !a; (!a; !a)* + (!a; !a)* and γ = !a; (!a; !a)*

Protocol α is substitutable for β and β is substitutable for γ, but $\alpha \nabla_{\{a, b\}} \gamma^{-1}$ contains no activity or infinite activity communication error, respectively.

This is a known issue inherent in using the consent operator on definition of the vertical compliance. It represents an unfortunate inconsistency that may cause a loss of the communication errors in a hierarchy of components, which constitutes a false negative. However, targeting this issue is out of scope of this thesis, which will be based on the available vertical compliance and substitutability as defined above.

3 Expressing a property

This section discusses the choice of a suitable formalism for expressing a general temporal property. The formalism has to be chosen with respect to specifics of behavior protocols. First, three possible choices are introduced, two popular temporal logics: *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL) (for more information see the fundamental papers [37, 24, 8]) and modal *Hennessy-Milner logic* [44]. Choice of LTL and CTL is based on [9] and the fact that these temporal logics are used in the real-life model checkers such as Spin [43] and SMV [7]. Many case studies were made on applying temporal logic based model checking tools to nontrivial verification tasks (to name few of them [4, 13, 42]). LTL and CTL are thus regarded as well-tried. It also reinforces the hope that designers will sooner or later become familiar with these temporal logics. For this reasons, exploiting one of them sounds reasonable.

The third possible choice presented is Hennessy-Milner logic. It is designed to express properties of processes in *process algebras* (see [5]). The main reason, for which it is considered, is its action nature. Both CTL and LTL are designed to specify properties of systems based on validity of atomic propositions in particular states. Transition is important only as a way to other state, no information is associated with it. From this point of view, behavior protocols are closer to process algebras. They are based on sending events. In behavior protocols, states carry no information but transitions are labeled by the appropriate events.

3.1 LTL

Linear temporal logic, sometimes also Linear-time temporal logic, is based on propositional logic, which is enriched by the temporal operators that allow for expressing changes of a system in time.

Definition 8. Let AP be a set of atomic propositions, then *syntactically correct LTL formulas* are recursively defined as follows:

$$\begin{aligned}\phi \ ::= \ & P \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \\ & \mid (\mathrm{X}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{G}\,\phi) \\ & \mid (\phi \ \mathrm{U} \ \phi) \mid (\phi \ \mathrm{R} \ \phi) \\ P \ ::= \ & \mathrm{true} \mid \mathrm{false} \mid p_1 \mid p_2 \mid \ldots \qquad \text{where } p_1, p_2, \ldots \in AP.\end{aligned}$$

From this definition, operators: true, false, $\neg$, $\vee$, X and U are the basic operators and operators: $\wedge$, $\rightarrow$, $\leftrightarrow$, F, G and R are derived. Derived operators are only a syntactic sugar that makes writing LTL formulas a little bit easier. They can be expressed using the basic operators.

Definition 9. Let ϕ and ψ be LTL formulas, then the *derived operators* can be expressed as follows:

$\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$

$\phi \rightarrow \psi \equiv \neg\phi \vee \psi$

$\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$

$\phi \text{ R } \psi \equiv \neg(\neg\phi \text{ U } \neg\psi)$

$\text{F}\,\phi \equiv \mathit{true} \text{ U } \phi$

$\text{G}\,\phi \equiv \neg\text{F}\,\neg\phi \equiv \text{false R } \phi.$

syntax	*name*	*informal description*
X ϕ	next	Formula ϕ will hold in the next state.
F ϕ	finally/eventually	Eventually ϕ will hold.
G ϕ	globally	Formula ϕ holds globally.
ϕ U ψ	until	Eventually ψ will hold, ϕ holds until then.
ϕ R ψ	release	Need for true of ψ is released by validity of ϕ.

Figure 4. Informal description of temporal operator

See Fig. 4 for names and informal meaning of the temporal operators. Proper formal definition of the semantics of LTL formulas is given in the definition below. LTL formulas are interpreted over infinite words of the alphabet $\Sigma=2^{AP}$. Any character of this alphabet represents a possible state of the system in a discrete time moment by the evaluation of the atomic prepositions. An infinite word then describes changes of the examined system's state in time.

Definition 10. Let $w = s_0, s_1, s_2, \ldots$ be an infinite word from Σ^ω and ϕ a LTL formula. Then *satisfaction* of ϕ by the word w ($w \models \phi$) is recursively defined as:

$w \models \text{true} \quad \text{and} \quad \neg(w \models \text{false})$

$\forall p \in AP\text{: } (w \models p \Leftrightarrow p \in s_0)$

$w \models \neg\phi \Leftrightarrow \neg(w \models \phi)$

$w \models \phi \vee \psi \Leftrightarrow (w \models \phi) \vee (w \models \psi)$

$w \models \text{X}\,\phi \Leftrightarrow s_1, s_2, \ldots \models \phi$

$w \models \phi \text{ U } \psi \Leftrightarrow \exists i \geq 0\text{: } (s_i, s_{i+1}, \ldots \models \psi \wedge \forall 0 \leq j < i\text{: } s_j, s_{j+1}, \ldots \models \phi).$

Definition 11. Let $M = (S, s_{init}, L, \rightarrow)$ be a Kripke structure and $T \subseteq \Sigma^\omega$ a set of infinite paths that can be traversed in M starting in the state s_{init} (states are mapped to the characters of Σ via the labeling function L). Kripke structure *M satisfies* a LTL formula ϕ ($M \models \phi$) if $w \models \phi$ for all $w \in T$.

In less formal words, a system satisfies a LTL formula if the formula is satisfied by all possible runs (traces) of the system. Each run of the system is judged separately. Hence, the name “linear”. This feature can be called *trace semantics* and is the fundamental difference in comparison to CTL.

3.2 CTL

Computational tree logic has a little bit more complicated syntax then LTL.

Definition 12. Let AP be a set of atomic propositions, then *syntactically correct CTL formulas* are recursively defined as follows:

$$\begin{aligned}\phi \ ::= \ & P \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \\ & \mid (\mathrm{AX}\,\phi) \mid (\mathrm{EX}\,\phi) \\ & \mid (\mathrm{AF}\,\phi) \mid (\mathrm{EF}\,\phi) \\ & \mid (\mathrm{AG}\,\phi) \mid (\mathrm{EG}\,\phi) \\ & \mid \mathrm{A}[\phi \ \mathrm{U} \ \phi] \mid \mathrm{E}[\phi \ \mathrm{U} \ \phi] \\ P \ ::= \ & \mathrm{true} \mid \mathrm{false} \mid p_1 \mid p_2 \mid \ldots \qquad \text{where } p_1, p_2, \ldots \in AP.\end{aligned}$$

As well as in the case of LTL, some operators are basic: true, false, ¬, ∨, EX, EG and EU. The others: ∧, →, ↔, AX, AF, EF, AG and AU are derived and can be expressed using the basic ones.

Definition 13. Let ϕ and ψ be CTL formulas, then the *derived operators* can be expressed as follows:

$$\begin{aligned}&\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi) \\ &\phi \rightarrow \psi \equiv \neg\phi \vee \psi \\ &\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \\ &\mathrm{AX}\,\phi \equiv \neg\mathrm{EX}(\neg\phi) \\ &\mathrm{EF}\,\phi \equiv \mathrm{E}[\mathrm{true} \ \mathrm{U} \ \phi] \\ &\mathrm{AF}\,\phi \equiv \neg\mathrm{EG}(\neg\phi) \\ &\mathrm{AG}\,\phi \equiv \neg\mathrm{EF}(\neg\phi) \\ &\mathrm{A}[\phi \ \mathrm{U} \ \psi] \equiv \neg\mathrm{EG}(\neg\psi) \wedge \neg\mathrm{E}[\neg\psi \ \mathrm{U} \ (\neg\phi \wedge \neg\psi)].\end{aligned}$$

Informally, A and E mean “along all paths” and “along at least one path” respectively. X, E, G and U refer to “next state”, “some future state”, “all future states” and “until”. CTL formulas are interpreted over Kripke structures.

Definition 14. Let $M = (S, s_{init}, L, \rightarrow)$ be a Kripke structure, $s \in S$ its state and ϕ a CTL formula. *Satisfaction* of ϕ for a structure M and state s ($M, s \models \phi$) is recursively defined:

$M, s \models \text{true} \quad \text{and} \quad \neg(M, s \models \text{false})$

$\forall p \in AP: \ (M, s \models p \ \Leftrightarrow \ p \in L(s))$

$M, s \models \neg\phi \ \Leftrightarrow \ \neg(M, s \models \phi)$

$M, s \models \phi \vee \psi \ \Leftrightarrow \ (M, s \models \phi) \vee (M, s \models \psi)$

$M, s \models \text{EX}\ \phi \ \Leftrightarrow \ \exists t \in S: \ (\langle s, t\rangle \in \rightarrow \wedge M, t \models \phi)$

$M, s \models \text{EG}\ \phi \ \Leftrightarrow \ \exists s_0, s_1, \ldots \in S: \ (s = s_0 \wedge$

$\forall i \geq 0: \ (\langle s_i , s_{i+1}\rangle \in \rightarrow \wedge M, s_i \models \phi))$

$\text{M, s} \models \text{E}[\phi \ \text{U} \ \psi] \ \Leftrightarrow \ \exists i \geq 0, s_0, s_1, \ldots, s_i \in \text{S}: \ (s = s_0 \wedge M, s_i \models \psi \wedge$

$\forall 0 \leq j < i: \ (\langle s_j, s_{j+1}\rangle \in \rightarrow \wedge \ M, s_j \models \phi)).$

Definition 15. Let $M = (S, s_{init}, L, \rightarrow)$ be a Kripke structure and ϕ a CTL formula. Kripke structure *M satisfies* the formula ϕ ($M \models \phi$) if $M, s_{init} \models \phi$.

In contrast to LTL, value of CTL formula depends to big extend on the structure of system's state space. Given a Kripke structure of the system, possible branching of the execution paths forms a *computational tree*, see the example on Fig. 5. Semantics of CTL is best imagined on this tree.

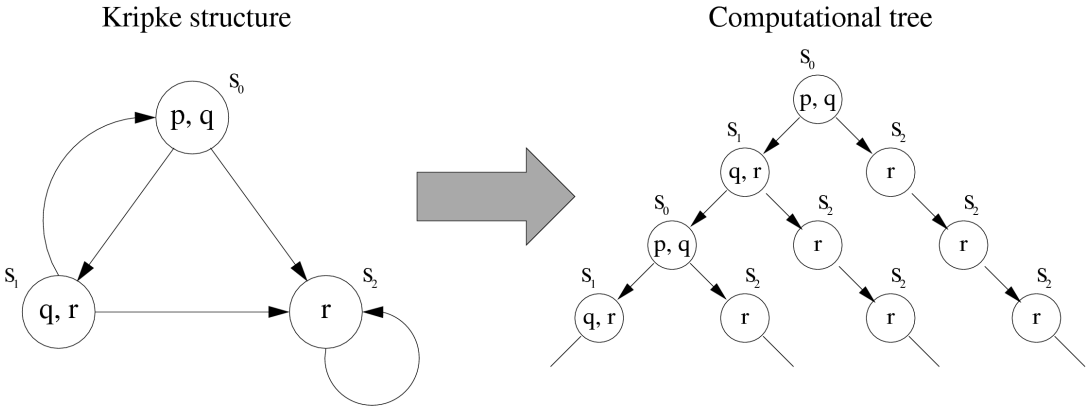


Figure 5. Kripke structure unwound into a computational tree

3.3 Hennessy-Milner logic

Unlike the LTL or CTL introduced above, the Hennessy-Milner modal logic is used to specify properties of processes from the realm of process algebras such as *Calculus of Communicating Systems* (CCS [29]) or *Communicating Sequential Processes* (CSP [19]).

Process can be viewed as an entity capable of performing actions. After performing an action (notation $\rightarrow^{\mathsf{a}}$ for an action a), process may act as another process. For example, process Clock = tick.Clock is capable of repeatedly performing the action tick. Process Alarm = ring.Clock can perform the action ring and then acts as the

process Clock. Models for CCS and CSP are Label Transition Systems. On Fig. 6, the LTS representing processes Alarm and Clock is depicted.

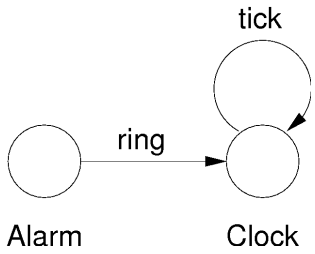


Figure 6. LTS representing processes Clock = tick.Clock and Alarm = ring.Clock

Hennessy-Milner logic has a very simple syntax compared to CTL or LTL. Its semantics is also quite comprehensible.

Definition 16. Let *ACT* be set of all possible actions. *Syntactically correct formulas of Hennessy-Milner logic* are recursively defined as follows:

$$\phi ::= \text{true} \mid \text{false}$$
$$\mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi)$$
$$\mid [K]\ \phi \mid \langle K \rangle \phi \qquad \text{where } K \subseteq ACT.$$

Definition 17. Let E be a process, then *satisfaction* relation $\models$ is recursively defined:

$$E \models \text{true} \quad \text{and} \quad \neg(E \models \text{false})$$
$$E \models \neg\phi \Leftrightarrow \neg(E \models \phi)$$
$$E \models \phi \vee \psi \Leftrightarrow (E \models \phi) \vee (E \models \psi)$$
$$E \models [K]\ \phi \Leftrightarrow \forall F \in \{E' : E \rightarrow^{a} E' \wedge a \in K\} : F \models \phi$$
$$E \models \langle K \rangle \phi \Leftrightarrow \exists F \in \{E' : E \rightarrow^{a} E' \wedge a \in K\} : F \models \phi$$

Informally, $[K]\ \phi$ says that if the process performs an action from the set K, then the formula ϕ must hold. Dual to that, $\langle K \rangle\ \phi$ says that the process is able to perform at least one action from K and ϕ would hold afterwards. For example, formula $\langle K \rangle$ true requires the process to be able to perform at least one action from set K. On the other hand, formula $[K]$ false is satisfied by a process unable to perform any action from K. Not surprisingly, $\langle K \rangle\ \phi \equiv \neg[K]\neg\phi$ for any formula ϕ. As can be seen from the examples, satisfaction of the Hennessy-Milner logic formulas again depends very much on the structure of processes (or LTS that represent it).

3.4 Discussion

With possible options introduced, the ultimate choice has to be made. When comparing different formalisms the following criteria should be kept in mind:

- Suitability for specifying properties of behavior protocols and
- Expressiveness.

The first criterion encourages use of such formalism, whose model and semantics is close to behavior protocols. The reason for this criterion is evident. Formalism that is used to operate over similar structures with similar meaning is likely to match also the needs of behavior protocols. Just to remind, behavior protocols' syntax can evoke processes from process algebras. The common thing is the action nature of both. Both processes and protocols can be view as entities that are capable of performing actions (sending events and receiving events, in case of protocols). This suggests that use of the same formalism to specify the property could be reasonable.

However, the semantics of behavior protocols makes the difference. For a given behavior protocol, the semantics is defined over set of traces generated by the protocol. Thus, two behavior protocols are semantically equivalent if they generate the same set of traces. This is what was denoted as the trace semantics and what was shared only with the LTL. Both CTL and Hennessy-Milner logic formulas depend on the structure of the examined system. It does not mean that these two could not be used at all. It rather says that the use of them could be confusing.

According to the second criterion, expressiveness of the formalisms should be discussed. However, this is not meant to aim the exact formal expressive power (expressiveness of CTL and LTL are know to be incomparable) but rather to investigate whether are the formalisms able to express frequently used properties. For this purpose, project Specification patterns at SAnToS laboratory [40] is considered. The project targets to enumerate frequently used specification patterns and to give their notation in different formalisms: LTL, CTL, *Graphical Interval Logic* (GIL [12]), *Quantified Regular Expressions* (QRE [33]) and *INCA queries* [41]. For example, such frequently used patterns are: "A is absent between B and C", "P is true globally after Q" or "between X and Y, Z is true at most twice". As can be seen from results of the project, both CTL and LTL can be successfully used to express these frequently used specification patterns, although the resulting formulas are usually not easy to read. From this point of view, CTL and LTL can be thought of as quite suitable.

Specification patterns project does not consider Hennessy-Milner logic, but it can be seen that it is not suitable to express this kind of properties without being seriously extended. In the basic form, it is not even able to express that the process should perform action tick for ever. By $\phi = (\langle ACT \rangle \text{ true}) \wedge ([ACT\text{-}\{\text{tick}\}] \text{ false})$, one can say that the process is obliged to perform action tick as its first action. Generally, n-times $[ACT]$ followed by ϕ requires that action number $n + 1$ (if any) can be only the action tick. Then by finite number of conjunctions, first n actions are forced to be tick.

$$((\langle ACT \rangle \text{ true}) \wedge ([ACT\text{-}\{\text{tick}\}] \text{ false})) \wedge$$
$$[ACT]\ ((\langle ACT \rangle \text{ true}) \wedge ([ACT\text{-}\{\text{tick}\}] \text{ false})) \wedge$$
$$[ACT]\ [ACT]\ ((\langle ACT \rangle \text{ true}) \wedge ([ACT\text{-}\{\text{tick}\}] \text{ false})) \wedge \ldots \wedge$$
$$[ACT]^n\ ((\langle ACT \rangle \text{ true}) \wedge ([ACT\text{-}\{\text{tick}\}] \text{ false}))$$

First, this constrains only a finite number of first actions. Second, this is definitely not a formula one would like to write or read.

At the bottom line, Hennessy-Milner logic, despite its action nature close to behavior protocols, is disqualified by its low expressiveness. When comparing CTL with LTL, both are expressive enough and both lack the action nature of behavior protocols, so they cannot be used without some changes in the semantics. Difference is in the trace semantics of LTL, which makes it a bit more suitable. The ultimate decision is to use LTL with some necessary semantic changes that will be described in the following section.

3.5 Targeting behavior protocols

There are two major issues in using LTL on behavior protocols. First, the LTL is defined over a finite set of atomic propositions *AP*. In every state of the Kripke structure that represents the examined system, any subset of *AP* can hold. On the contrary, states of component described by a behavior protocol do not carry any special information; the transitions are labeled by appropriate events. In other words, from the point of view of Kripke structure, the run of a system is a sequence of visited states labeled by subsets of *AP*. From the point of view of behavior protocol, it is a sequence of transitions labeled by corresponding events. The natural solution is to map atomic propositions to events of behavior protocols. The set of atomic propositions *AP* can be chosen to be equal to the set of possible events. When a particular event is emitted, the associated atomic proposition is considered to be true and all others are false.

In fact, there is one more improvement based on the following observation. When monitoring behavior of a composition of components, any emitted event is either accepted by a component bound on the corresponding interface (possibly out of the composition in case of subsumption binding) or it generates a bad activity error. Bad activity is always wrong and is detected by the compliance checker. Thus, it is not necessary to make difference between emitting and accepting an event, because it does not provide any additional information. From this reason, atomic propositions are mapped to m^ and m$ for every method m. Where m^ is considered to be true, when either of events ?m^, !m^ or τm^ is generated. Analogously m$ is true for events ?m$, !m$ and τm$.

The second issue is in finiteness of traces of behavior protocols. Semantics of LTL is defined over infinite traces, whereas semantics of behavior protocols uses only finite traces. This suggests modification of LTL definitions to work only over finite words. In [14], three possible semantics of LTL over finite traces are presented: weak, strong and neutral, and their relations are studied. The reason for this division is in problematic X operator on finite traces. What value should have the formula X ϕ on the end of the trace? Consider formulas X true and X false and property $\neg$X ϕ = X $\neg\phi$ that holds in the classical LTL. In [17], it is shown that problems with different semantics can be, not surprisingly, consistently solved by using LTL_{-X}, which is LTL without the X operator. Because of the simplicity, we adopt this approach and continue with the LTL_{-X}. The definitions of LTL_{-X} over finite traces follow. Definitions of derived operators do not change.

Definition 18. Let M be a set of methods, then *syntactically correct LTL$_{-X}$ formulas* are recursively defined as follows:

$$\phi ::= P \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi)$$
$$\mid (\mathrm{F}\,\phi) \mid (\mathrm{G}\,\phi)$$
$$\mid (\phi\ \mathrm{U}\ \phi) \mid (\phi\ \mathrm{R}\ \phi)$$
$$P ::= \text{true} \mid \text{false} \mid \mathsf{m}_1\text{^} \mid \mathsf{m}_1\$ \mid \mathsf{m}_2\text{^} \mid \mathsf{m}_2\$ \mid \ldots \qquad \text{where } \mathsf{m}_1, \mathsf{m}_2, \ldots \in M.$$

Definition 19. Let $w = \langle e_1, e_2, \ldots, e_n \rangle$ be a finite trace of behavior protocol events and ϕ a LTL$_{-X}$ formula. Then *satisfaction* of ϕ by w ($w \models \phi$) is recursively defined as:

$w \models \text{true}$ and $\neg(w \models \text{false})$

$\forall\, \mathsf{m} \in M$: $(w \models \mathsf{m}\text{^} \Leftrightarrow n > 0 \wedge e_1 \in \{?\mathsf{m}\text{^}, !\mathsf{m}\text{^}, \tau\mathsf{m}\text{^}\})$

$\forall\, \mathsf{m} \in M$: $(\mathrm{w} \models \mathsf{m}\$ \Leftrightarrow n > 0 \wedge e_1 \in \{?\mathsf{m}\$, !\mathsf{m}\$, \tau\mathsf{m}\$\})$

$w \models \neg\phi \Leftrightarrow \neg(w \models \phi)$

$w \models \phi \vee \psi \Leftrightarrow (w \models \phi) \vee (w \models \psi)$

$w \models \phi\ \mathrm{U}\ \psi \Leftrightarrow \exists\, \min(1, n) \le i \le n$: $(\langle \mathrm{e}_i, \mathrm{e}_{i+1}, \ldots, \mathrm{e}_n \rangle \models \psi \wedge$

$$\forall\, 1 \le j < i\text{:}\ \langle \mathrm{e}_j, \mathrm{e}_{j+1}, \ldots, \mathrm{e}_n \rangle \models \phi).$$

Definition 20. Let $\wp$ be a behavior protocol with language $\mathscr{L}(\wp)$. Protocol $\wp$ *satisfies* a LTL$_{-X}$ formula ϕ ($\wp \models \phi$) if $w \models \phi$ for all $w \in \mathscr{L}(\wp)$.

In order to design an algorithm for checking the modified LTL$_{-X}$ formula on behavior protocols, the algorithm for checking the original LTL formula on Kripke structure was consulted (it can be found in [9]). The keystone of the original algorithm is translation of the LTL formula into *Büchi automaton*, which accepts exactly the traces that satisfy the formula. Büchi automaton is a finite automaton that accepts infinite traces. An infinite trace is accepted if the automaton visits an accepting state infinite number of times along the trace. One of the possible translation algorithms is described in [16]. The algorithm for checking the LTL formula ϕ on Kripke structure M works in four steps:

- Büchi automaton $\mathrm{B_M}$, which accepts exactly traces corresponding to possible runs of system described by Kripke structure M, is constructed.
- Formula ϕ is negated and Büchi automaton $\mathrm{B}_{\neg\phi}$, which accepts exactly traces violating formula ϕ, is constructed.
- Intersection of $\mathrm{B_M}$ and $\mathrm{B}_{\neg\phi}$ is constructed.
- Language accepted by intersection of $\mathrm{B_M}$ and $\mathrm{B}_{\neg\phi}$ is sought for emptiness. If the language is nonempty then the system does not satisfy the formula ϕ, because there is a trace of system M that violates ϕ.

To prove that the language accepted by a given Büchi automaton is nonempty, one has to find a cycle reachable from the initial state and containing at least one

accepting state. This can be done in time linear in the size of the automaton by a single depth first search traversal.

The question is if the algorithm is usable also to the modified LTL and behavior protocols. Model for behavior protocol (or composition of protocols) is a finite automaton. If it was possible to construct a finite automaton that would accept exactly traces satisfying a given modified LTL formula, then the algorithm above could be used with just substituting Büchi automata by finite automata. Fortunately, the main idea of the LTL to Büchi automaton tableau-based translation algorithm can be used unchanged to generate a nondeterministic automaton accepting exactly traces, which satisfy a given modified LTL formula. Thus, the algorithm for checking whether a behavior protocol (or a composition of protocols) satisfies a modified LTL formula is almost the same as in the previous case sharing also the time complexity. The fourth step is done again in time linear in the size of the intersection automata, which is generally exponential in both length of the formula and length of the protocol. The translation algorithm is described in Section 5.2.

Expressive power of the modified LTL is evidently a subset of regular languages, because any LTL formula can be translated into a finite automaton. It is reasonable to ask if the introduction of LTL really brought something new to the world of behavior protocols, whose power is also regular. There are at least two reasons for answering: "yes". First, behavior protocols contain neither wildcards nor conjunction, so expressing property of a type: "A happens sometime after B" or "protocol satisfies both conditions A and B", is theoretically possible, but very unpractical. Second, LTL with almost unchanged syntax is arguably better choice then introduction of "just another" specification language for component designer to learn.

The last note is on differences between finite and infinite trace semantics of LTL_{-X}. Consider the following infinite trace π and LTL_{-X} formula ϕ:

$$\pi = \langle a, x, b, a, y, x, b, a, y, x, b, a, y, x, \ldots \rangle$$

$$\phi = G((a \rightarrow F\, b) \wedge (x \rightarrow F\, y))$$

Formula ϕ says: "After each occurrence of a, b eventually follows. And after each occurrence of x, y eventually follows." This is evidently true when considering π with the infinite semantics. However, no nonempty prefix of π satisfies this formula with the finite semantics – one of the obligations will always be disobeyed. The second example of difference is formula ψ:

$$\psi = F((G\, a) \vee (G \neg a))$$

Formula ψ says: "Eventually, either a will be emitted for ever or a will not be emitted any more." In the infinite trace semantics, this really puts a constraint on some kind of "stabilization" of the system. However, in the modified semantics, this holds for any trace including the empty one. Informally, on any non-empty finite trace, there is the last symbol, which is either a or not. Evidently, from this symbol on – that means just for this symbol – either G a or G ¬a holds. Satisfaction on empty trace follows directly from the definition.

4 Reduction of protocols

Depending on the environment of a software component, some parts of its functionality are never used. For example, some methods are never invoked. In the language of behavior protocols this means that some traces defined by component's protocol are not used in the composition. These unused traces can be omitted to improve readability of the protocol – *reduction with respect to composition*.

Another case of protocol reduction is *reduction with respect to property*. Having a LTL formula, some aspects of the components' behavior do not influence validity of the formula. Omitting parts of behavior protocols that represents these aspects and leaving only parts relevant to the property should make the comprehension of the behavior specification easier.

However, the important requirement on both reductions is that the original protocol should be substitutable for the reduced protocol. In other words, any component that can be correctly described by the original protocol can be also described by the reduced protocol.

4.1 Motivation example

In order to clarify the basic idea, examples of the two different types of reduction will follow. The first one demonstrates use of the reduction with respect to composition. Fig. 7 shows composition of two components: A and B with the following behavior protocols:

- Component A: ?a{!x}* | (?b)*
- Component B: !a{?x + ?y}*

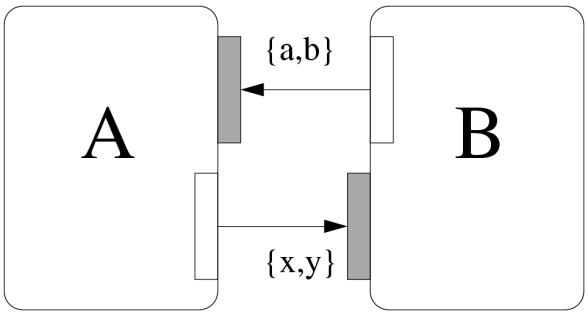


Figure 7. Example of reduction with respect to composition

The component A can accept arbitrary numbers of calls of the methods a and b in parallel. During the invocation of the method a it calls the method x on B. On the other side, the component B repeatedly calls the method a on A and during this call it must receive call of either x or y.

Given the composition from Fig. 7, one can easily see that the methods b and y will never be called. Even though, both components can accept these calls, they will never be issued in this particular composition. So the behavior protocols of both components can be reduced to contain only the parts that are really used:

- Component A: ?a{!x}*
- Component B: !a{?x}*

The important thing to note is the fact that this reduction just removed the unnecessary parts of both protocols. It did not reduce the behavior of the composed components in any way, since the behavior was restricted by the composition. This is not the case in the following example that presents reduction with respect to given property.

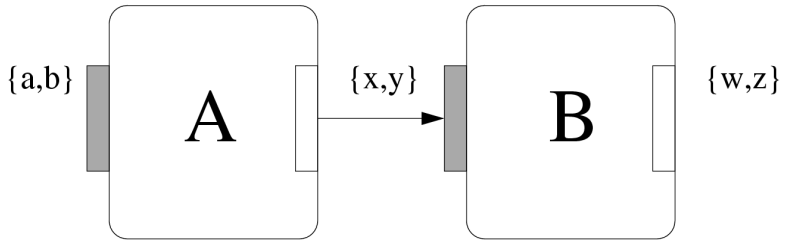


Figure 8. Example of reduction with respect to property

On Fig. 8, there are again the components A and B. Each of them has also one external interface (to be used be the environment). Assume that the behavior protocols of the components are:

- Component A: (?a{!x})* | (?b{!y})*
- Component B: (?x{!z})* | (?y{!w})*

The component A can accept arbitrary numbers of calls of the methods a and b in parallel. During invocation of a and b, it calls the method x and y on B, respectively. In the same manner, the component B forwards parallel calls of the methods x and y to the external calls of z and w.

Let ϕ = G (a^ → F z^) be the property. That is: "Each call of the method a (begin of the invocation) is eventually followed by a call of the method z." Validity of this property is trivial, because every call of the method a is postponed to z via x. In contrast, parallel invocation of the method b and consequently y and w does not affect validity of this property by any means and can be omitted. The reduced protocols contain only parts relevant to the given property:

- Component A: (?a{!x})*
- Component B: (?x{!z})*

Unlike the previous type of reduction, this restricts behavior of the components. In this case, the reduction is achieved only for a promise that the method b will never be called. This is a fundamental difference between these two types of reduction. If reduction with respect to composition is applied, the result can be used to substitute the original – only the unused behavior was removed. However, the result of reduction with

respect to property is generally weaker than the original. Also some parts that possibly could be used by the environment are omitted so the result can be used only in those environments that keep this promise.

Motivation of the reduction is to help the designer with comprehension of a specification, by hiding insignificant parts of it. Reduction with respect to composition hides parts that are not used in the given composition, whereas reduction with respect to property goes further. It hides also parts that might be used, but are not significant to the given LTL formula.

Although the basic idea of protocol reduction is quite easy to understand, a more formal approach has to be taken in the next sections to clearly present the proposed solution.

4.2 Reduction preorder

This section focuses on formal definition of possible reductions of a single behavior protocol. Given a behavior protocol, the question is: "What protocols can be thought of as its reductions?"

The basic condition was already mentioned. The original protocol has to be substitutable for the reduced protocol. So any component that can be described by the original protocol can be also described by the reduced protocol. This condition follows from the fact that the reduced protocol is only a weaker description of the same implementation.

Definition 21. Let $\wp_1$ and $\wp_2$ be behavior protocols, then $\wp_1 \leq_R \wp_2$ if $\mathcal{L}(\wp_1) \subseteq \mathcal{L}(\wp_2)$ and $\wp_2$ is substitutable for $\wp_1$.

Informally, the reduced protocol can generate only traces generated by the original one – not necessarily all of them. Only the traces whose choice does not depend on the described component but rather on its environment (the environment has to emit an event) can be omitted. Otherwise, trace that can be chosen according to the original protocol is not possible any more in the reduced protocol and the original would be no longer substitutable for the reduced protocol. This is a sort of "contract" that the traces can be omitted provided that some methods are not called in some situations.

Lemma 1. Relation $\leq_R$ on the set of behavior protocols is a *preorder*. It is called *reduction preorder*.

Proof In order to show that the relation $\leq_R$ is a preorder, it has to be shown that the relation is both transitive and reflexive. Reflexivity is immediate. For any behavior protocol $\wp$, $\mathcal{L}(\wp) \subseteq \mathcal{L}(\wp)$ and $\wp$ is always substitutable for itself, because the substitutability relation (Definition 7) is trivially reflexive.

For transitivity, we have protocols $\wp_1$, $\wp_2$ and $\wp_3$ such that $\wp_1 \leq_R \wp_2$ and $\wp_2 \leq_R \wp_3$. It is to be shown that $\wp_1 \leq_R \wp_3$. Obviously, $\wp_1 \leq_R \wp_2$ and $\wp_2 \leq_R \wp_3$ implies that $\mathcal{L}(\wp_1) \subseteq \mathcal{L}(\wp_2) \subseteq \mathcal{L}(\wp_3)$. However, substitutability of $\wp_3$ for $\wp_1$ is not so

obvious. Suppose that $\mathscr{L}(\wp_3 \nabla_M \wp_1^{-1})$, where M is a set of all methods, contains traces with communication errors. Let π be one of the error traces.

Suppose that π represents the bad activity error. Then there is π', such that π' is a prefix of some trace generated by $\wp_3$ and continuing by an emit token and $\wp_1$ generates one or more traces with this prefix but none continuing with this particular token. Because of $\mathscr{L}(\wp_1) \subseteq \mathscr{L}(\wp_2) \subseteq \mathscr{L}(\wp_3)$, also $\wp_2$ generates one ore more traces with prefix π' (since $\wp_1$ do). There are two possibilities. Either $\wp_2$ generates also a trace with prefix π' continuing by the particular emit token or not. However, the first case contradicts substitutability of $\wp_2$ for $\wp_1$ and the second contradicts substitutability of $\wp_3$ for $\wp_2$. In both cases, the error trace π would occur.

Cases with no activity and infinite activity can be proven analogously. ❑

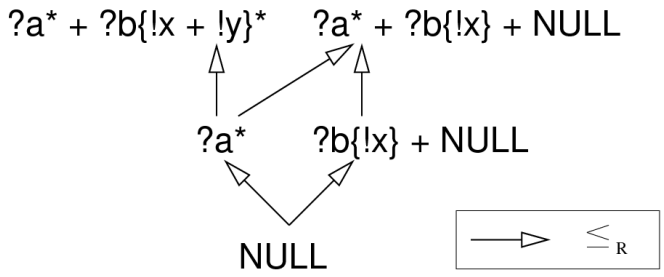


Figure 9. An example of the reduction preorder (it is incomplete – an infinite number of protocols could be inserted into the graph)

See Fig. 9 for a brief example of the reduction preorder. After the reduction preorder is well defined, some other important concepts should be stated precisely, as well.

Definition 22. Behavior protocol $\wp_1$ is a *reduction* of behavior protocol $\wp_2$ if $\wp_1 \leq_R \wp_2$. The *set of reductions* of protocol $\wp$ is a set $\Re_\wp = \{\rho: \ \rho$ is a behavior protocol and $\rho \leq_R \wp\}$.

In the set of reductions, there are all possible reductions of a particular behavior protocol. Elements of this set, for which there is no smaller element present, are the most dummy protocols that are still compliant with the original one. They offer the same but perform less. However, usually the situation is more complicated. There is often a set of traces that are in some sense important and should be present also in the reduced protocols.

Definition 23. Let $\wp$ be a behavior protocol and $T \subseteq \mathscr{L}(\wp)$ a set of traces. Behavior protocol ρ_1 is a *minimal reduction of $\wp$ containing T* if $\rho_1 \leq_R \wp$, $T \subseteq \mathscr{L}(\rho_1)$ and there is no protocol ρ_2 such that $\rho_2 \leq_R \rho_1$ and $T \subseteq \mathscr{L}(\rho_2) \subset \mathscr{L}(\rho_1)$.

Set T is a set of important traces that should be preserved in the reduced protocol. What traces are important depends on the purpose of reduction. This definition is quite straightforward but unfortunately has some drawbacks. First, the minimal reduction is not unique.

Lemma 2. Let $\wp$ be a behavior protocol and $T \subseteq \mathscr{L}(\wp)$ a set of traces and ρ_1 a minimal reduction of $\wp$ containing T. There exist a protocol $\rho_2 \neq \rho_1$ which is also a minimal reduction of $\wp$ containing T.

Proof Let $\rho_2 = \rho_1$ + NULL. Claim follows immediately from $\mathscr{L}(\rho_1) = \mathscr{L}(\rho_2)$. ❑

It can be easily seen that this way, an infinite set of minimal reductions can be generated if at least one exists. However, also more minimal reductions with different languages can exist in general. If $\wp$ = ?a + ?b and T = Ø then both protocols ?a and ?b are minimal reductions of $\wp$ containing T.

Second problem is that no minimal reduction is obliged to exist at all if the set of traces T is not a regular language. For example if $\wp$ = ?a*; ?b* and $T = \mathscr{L}(?a^n; ?b^n)$ for $n \in \mathbb{N}$, no minimal reduction of $\wp$ containing T exists. If we stick to the regular T, this problem does not occur.

Unfortunately, Definition 23 does not enforce any restrictions on the resulting syntactic structure of the minimal reductions. However, the original goal was to help the designer with reading and reusing the protocol. Even a suboptimal solution preserving the syntactic structure of the original protocol would be preferred over the optimal one. Such a solution will be described in the next section.

Because of the principle of inverted frame protocol, introduced in Section 2.2, also notion of inverted reduction will be useful in the following.

Definition 24. Let $\wp_1$ and $\wp_2$ be behavior protocols, then $\wp_1 \leq_{IR} \wp_2$ if $\mathscr{L}(\wp_1) \subseteq \mathscr{L}(\wp_2)$ and $\wp_1$ is substitutable for $\wp_2$.

Definition 25. Behavior protocol $\wp_1$ is an *inverted reduction* of behavior protocol $\wp_2$ if $\wp_1 \leq_{IR} \wp_2$. *Set of inverted reductions* of protocol $\wp$ is a set $\Re_\wp = \{\rho:\ \rho$ is a behavior protocol and $\rho \leq_R \wp\}$.

4.3 Term rewriting

In this section, an approach that operates on the syntactic structure of the protocol – the *term rewriting* [11] – is presented. We define a set of rewriting rules that are repeatedly applied to the original protocol to perform the reduction. These rules will enforce preservation of both syntactic structure and substitutability of the original protocol. This way, any result of the rewriting will always be a reduction of the original protocol in accordance with Definition 22.

First of all, a few definitions have to be stated.

Definition 26. Behavior protocol or subprotocol of a behavior protocol $\wp$ is *nullable* (predicate nullable($\wp$) is true) if an empty trace <> $\in \mathscr{L}(\wp)$.

Definition 27. Behavior protocol or subprotocol of a behavior protocol $\wp$ is *passive* (predicate passive($\wp$) is true) if no trace $\pi \in \mathscr{L}(\wp)$ starts with an emit event (event of a form !a^ or !a$, where a is a method).

The point of passive and nullable protocols is simple. Whenever a subprotocol only waits for en event – is passive – it is a prospective place to be reduced. In other words, environment can choose not to send any of the events the component is waiting for and this part can be reduced. If a nullable subprotocol is to be reduced, it can be reduced to NULL protocol.

For example protocol ?a{!x} is passive but protocol !a{?x}* is not. The first one is not nullable but the second one is (note the * operator).

Definitions 26 and 27 can be stated also inductively as they are really used in the implementation. These definitions follow for the sake of completeness.

Definition 28. Let $\wp$ and ρ be behavior protocols and let a be a method:

- !a^, !a$, ?a^ and ?a$ are not *nullable*
- NULL is *nullable*
- $\wp^*$ is *nullable*
- $\wp + \rho$ is *nullable* if $\wp$ or ρ is *nullable*
- $\wp \mid \rho$ and $\wp ; \rho$ are *nullable* if both $\wp$ and ρ are *nullable*

Definition 29. Let $\wp$ and ρ be behavior protocols and let a be a method:

- !a^ and !a$ are not *passive*
- ?a^ and ?a$ are *passive*
- NULL is *passive*
- $\wp^*$ is *passive* if $\wp$ is *passive.*
- $\wp + \rho$ and $\wp \mid \rho$ are *passive* if both $\wp$ and ρ are *passive*
- $\wp ; \rho$ is *passive* if $\wp$ is *passive* and either $\wp$ is not *nullable* or ρ is also *passive*

Syntactic abbreviations like method calls or the or-parallel operator can be derived intuitively. Also brackets despite being part of the syntax of behavior protocols are ignored, to stay comprehensible.

With all necessary concepts defined, the reduction rules can be formulated. Let $\wp$ and ρ be behavior protocols. The rewriting rules for protocol reduction follow:

$\wp$	→	NULL	if $\wp \neq$ NULL and nullable($\wp$) and passive($\wp$)
$\wp + \rho$	→	$\wp$	if passive(ρ)
$\wp + \rho$	→	ρ	if passive($\wp$)
$\wp \mid \rho$	→	$\wp$	if passive(ρ) and nullable(ρ)
$\wp \mid \rho$	→	ρ	if passive($\wp$) and nullable($\wp$)
$\wp \mid \rho$	→	$\wp + \rho$	if passive($\wp$) and passive(ρ) and nullable($\wp$) and nullable(ρ)
$\wp ; \rho$	→	$\wp$	if passive(ρ) and nullable(ρ)

$\wp ; \rho \quad \rightarrow \quad \rho$ if passive($\wp$) and nullable($\wp$)

Rules can be applied on any subprotocol of the protocol being rewritten. A subprotocol is formally any substring that is itself syntactically a behavior protocol. It has to be shown that application of the rules generates reduction of the original protocol.

Lemma 3. Let $\wp_1$ and $\wp_2$ be behavior protocols and $\wp_2$ be derived from $\wp_1$ by application of one or more rewriting rules to some of its subprotocols. Then $\wp_2$ is a reduction of $\wp_1$.

Proof First, evidently $\mathcal{L}(\wp_2) \subseteq \mathcal{L}(\wp_1)$, because no rule adds traces that would not be present in the original protocol. Second, $\wp_1$ is substitutable for $\wp_2$, because any omitted subprotocol is passive. It means that only parts stating requirements put on the component are omitted. So the protocol $\wp_2$ has lower requirements on the implementing component than the protocol $\wp_1$. Since no changes on the provided side are made, $\wp_1$ is clearly substitutable for $\wp_2$. ❑

The rewriting rules are monotonic. Each rule makes the protocol shorter (less tokens) except the first one, which may remove no tokens but which cannot be used more than once on the same subprotocol. This ensures that after a finite number of rewriting no rewriting rule matches any more. Thus, at least one final reduction always exists. Depending on the choice of the rules, more such final reductions can exist. For example, consider again the protocol ?a + ?b. It can be reduced to either ?a or ?b that both cannot be reduced any further.

As mentioned in the previous section, there is often a set of important traces that should be preserved by the reduction. This set depends on the purpose of reduction and is rather conceptional. It is not mentioned to be enumerated and given to the rewriting algorithm as an input – it is usually infinite. Among others, the rewriting algorithm has no notion about traces. Better way how to make the rewriting process aware of the important traces is to forbid the use of a particular rewriting rule if it could possibly cause removal of an important trace. For this purpose, both types of reductions are obliged to provide a mechanism that would specify whether a concrete rewriting rule can be applied to a particular subprotocol or not.

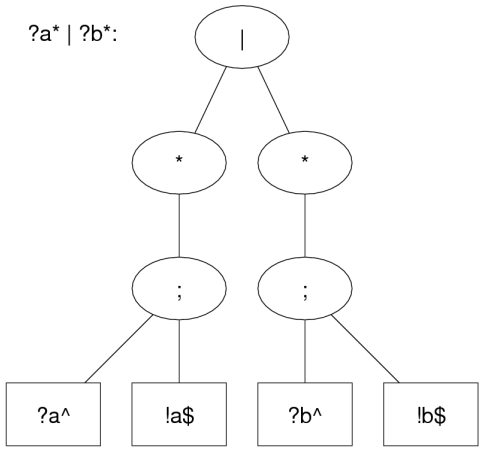


Figure 10. An example of the parse tree for the protocol ?a* | ?b*

As described in the following, both algorithms for reduction with respect to composition and for reduction with respect to LTL formula work over the parse trees of the protocols. An example of a parse tree can be seen on Fig. 10. The nodes of the parse trees are naturally the place, where to put the information about importance. Informally, each node of the parse tree is marked as *important* if the corresponding subprotocol can be used to generate an important trace. Moreover, the parallel operator nodes are marked as *parallel-important* if the corresponding interleaved subprotocols can be used to generate an important trace. The rewriting rules are then altered as follows:

$\wp$	→ NULL	if $\wp \neq$ NULL and nullable($\wp$) and passive($\wp$) and not important($\wp$)
$\wp + \rho$	→ $\wp$	if passive(ρ) and not important(ρ)
$\wp + \rho$	→ ρ	if passive($\wp$) and not important($\wp$)
$\wp \mid \rho$	→ $\wp$	if passive(ρ) and nullable(ρ) and not important(ρ)
$\wp \mid \rho$	→ ρ	if passive($\wp$) and nullable($\wp$) and not important($\wp$)
$\wp \mid \rho$	→ $\wp + \rho$	if passive($\wp$) and passive(ρ) and nullable($\wp$) and nullable(ρ) and not parallel-important($\wp \mid \rho$)
$\wp ; \rho$	→ $\wp$	if passive(ρ) and nullable(ρ) and not important(ρ)
$\wp ; \rho$	→ ρ	if passive($\wp$) and nullable($\wp$) and not important($\wp$)

On Fig. 11, example of the protocol reduction is given. The protocol is ?a* | ?b* and the set of important traces is equal to {<?a^, !a$>, <?a^, !a$, ?a^, !a$>, …}. The important nodes are marked by black marks. It is easy to see, that all important traces are generated only using the left subtree. By application of the fourth rule, the ?a* protocol in acquired.

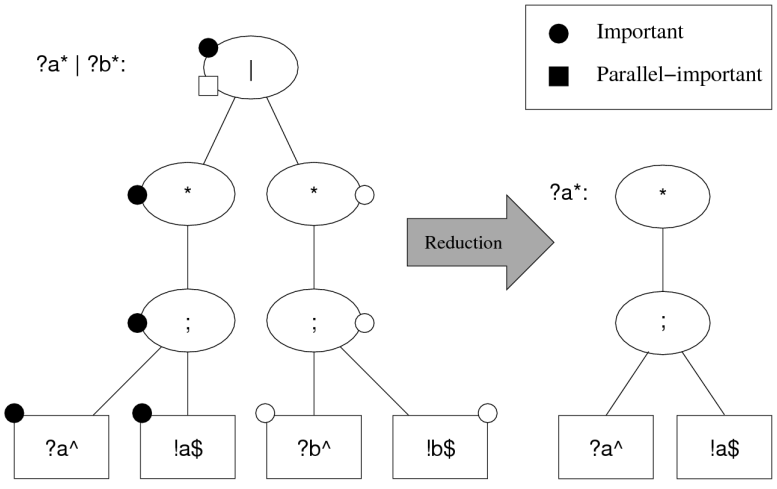


Figure 11. Rewriting of the protocol ?a* | ?b* with the set of important traces {<?a^, !a$>, <?a^, !a$, ?a^, !a$>, …}

Another example can be seen on Fig. 12. The protocol is the same and the set of important traces is equal to {<?a^, !a$>, <?a^, !a$, ?a^, !a$>, …, <?b^, !b$>, <?b^, !b$, ?b^, !b$>, …}. Both subtrees are necessary for generation of important traces; however, the parallel interleaving is not. The sixth rule can be applied to get the protocol ?a* + ?b*.

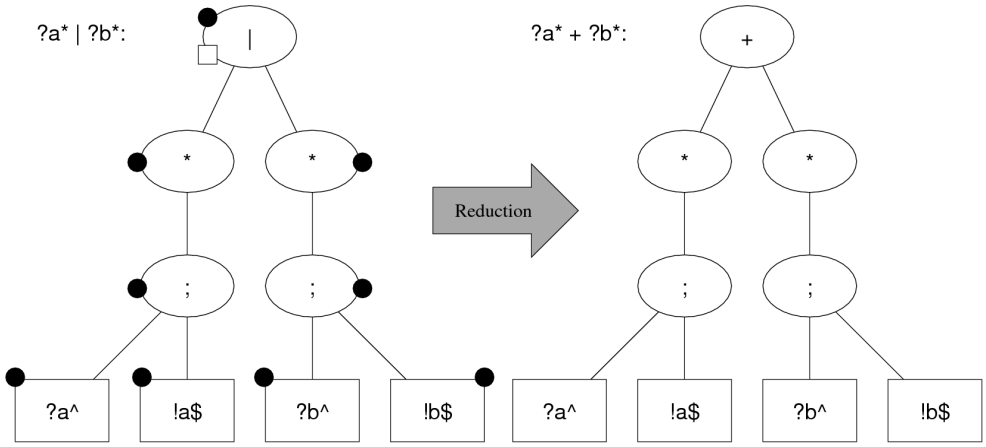


Figure 12. Rewriting of the protocol ?a* | ?b* with the set of important traces {<?a^, !a$>, <?a^, !a$, ?a^, !a$>, …, <?b^, !b$>, <?b^, !b$, ?b^, !b$>, …}

Suboptimality of this approach is hidden in two aspects. First and obvious, we stick to the structure of the original protocol and thus automatically refuse solutions that are potentially better. Consider protocol ?a* with the set of important traces {<?a^, !a$, ?a^, !a$>}. Protocol ?a; ?a is arguably the best reduction, but the structure is different; no sequence of the rewriting rules can result in this protocol. In this case, the protocol will not be reduced by the rewriting rules at all. Second, the reduction can be suboptimal when the protocols are not deterministic.

Definition 30. We say that a behavior protocol is *deterministic* if for each nonempty prefix of any of its traces, the leaf of the protocol parse tree that have generated the last event can be identified unambiguously.

For example, protocol ?a* + ?a* is not deterministic, since for a prefix <?a^>, there are two leaves of the parse tree that could have generated the event ?a^. Because all nodes of the parse trees that could be used to generate an important trace are marked as important, all ambiguous nodes are marked. For the protocol ?a* + ?a* and a set of important traces equal to its whole language, all nodes of the parse tree will be marked as important due to the nondeterminism (all nodes can be used to generate an important trace). However, the best reduction would be ?a*. This behavior is acceptable because nondeterministic protocols are rather rare in practice.

As already indicated, the notion of inverse reduction will be useful in the following. In order to generate an inverse reduction using the rewriting rules, concept of passive protocols has to be substituted by a dual concept of active protocols.

Definition 31. Behavior protocol or subprotocol of a behavior protocol $\wp$ is *active* (predicate active($\wp$) is true) if no trace $\pi \in \mathcal{L}(\wp)$ starts with an accept event (event of a form ?a^ or ?a$, where a is a method).

Rewriting rules have to be altered. All occurrences of the predicate passive have to be substituted by the predicate active. Modified rules can be used to generate inverse reductions.

In this section, rewriting rules for reduction of behavior protocol were defined. However, they can be used only on a single protocol with the parse tree nodes marked as important or parallel-important according to the set of important traces. Next sections extend this idea to a composition of more components and define what the important traces are with respect to composition and with respect to property.

4.4 Reduction with respect to composition

As said before, composition of components is defined by frame protocol of each participating components (A and B on Fig. 13), frame protocol of the resulting composite component (C), bindings between inner components (1 and 2), subsumption (3) and delegation (4) bindings and unbound interfaces (5 and 6).

The goal is to reduce protocols of inner components (A and B in the example) and the frame protocol (C) to leave out the behavior unused in this particular composition. The unreachable parts of the individual protocols should be omitted but the behavior of the whole architecture should remain unchanged. If α, β and γ are behavior protocols of components A, B and C respectively, then the goal is to create protocols α', β' and γ' such that $\alpha' \leq_R \alpha$, $\beta' \leq_R \beta$ and $\gamma' \leq_{IR} \gamma$. This should guarantee that any implementation of component A (or B) that is compliant with the protocol α (β) is compliant also with the new protocol α' (β'). Also $\gamma' \leq_{IR} \gamma$ should imply that γ' remains substitutable for γ, so no communication errors are introduced in the higher levels of hierarchy. This claim would be true if the substitutability relation was transitive, which is actually not as discussed in Section 2.3. Protocols α', β' and γ' can be imagined to define virtual wrapper components A', B' and C' as depicted on Fig. 14.

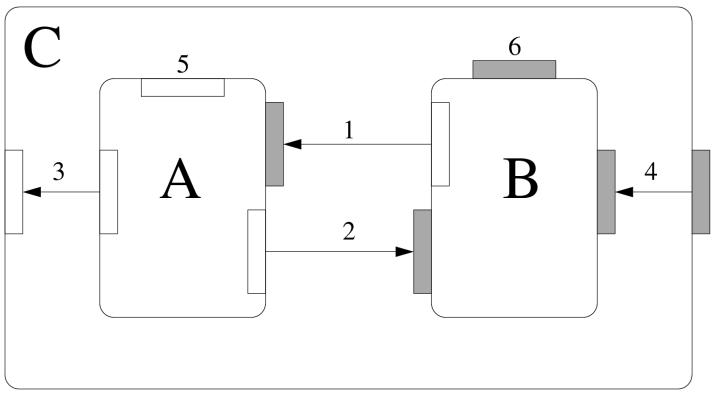


Figure 13. The composite component C consists of A and B, the internal (1 and 2), subsumption (3) and delegation (4) bindings and unbound interfaces (5 and 6)

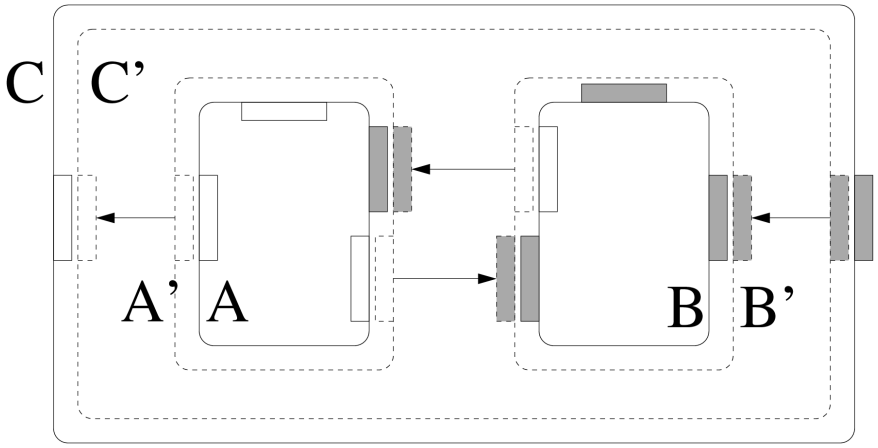


Figure 14. Idea of the virtual wrapper components. The components A, A', B, B', C and C' have the behavior protocols α, α', β, β', γ and γ' respectively, where $\alpha' \leq_R \alpha$, $\beta' \leq_R \beta$ and $\gamma' \leq_{IR} \gamma$

The last and most evident requirement is that if the composition A, B and C is communication error free then also the composition of virtual components A', B' and C' should contain no communication errors. Fortunately, this property is for free if the whole behavior is unchanged as required. It means that both compositions contain the same set of traces and also the same set of error traces. Thus, if the set of error traces of the composition A, B and C is empty then the set of error traces of the virtual composition is empty too.

To create reduced protocols of inner components (α' and β'), rewriting rules from the previous section are used. In case of the frame protocol (γ'), the rules for inverse reduction are applied. Both sets of rules operate on a single protocol, so the sets of important traces have to be defined per each protocol. Informally, a trace of a protocol is important and should be preserved if it is really used – the consent operator does not discard it. Instead of constructing these sets, the implementation uses the second approach. It marks nodes of the protocol parse trees as important or parallel-important. This information is actually used in the rewriting rules.

The algorithm is divided into two parts. In the first part, traversal of the state space of the whole composition (using the consent operator) is performed and reachable nodes of each parse tree are marked as important. The parallel operator nodes are marked as parallel-important if a nontrivial interleaving of its subprotocols is ever used. This phase is very similar to the compliance checking. Instead of finding communication errors the algorithm is just marking reachable nodes of the parse trees, during the traversal. In fact, this part of the algorithm is really done together with the compliance checking in a single pass. Fig. 15 illustrates the first phase.

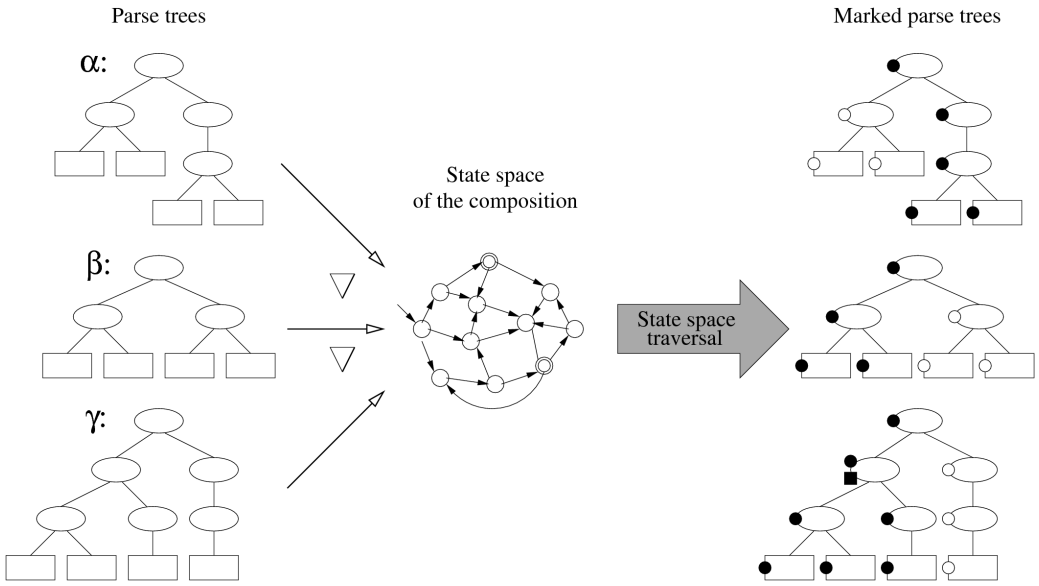


Figure 15. First phase of the algorithm for reduction with respect to composition. In this phase, parse trees of individual components are marked during the traversal of the state space of the whole composition

In the second phase, the rewriting rules are applied to each individual protocol using the information gathered in the first phase. See Fig. 16.

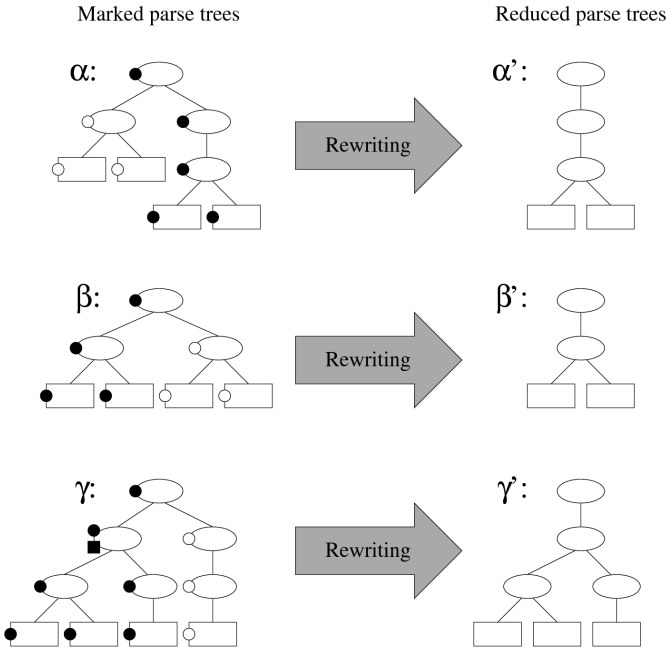


Figure 16. Second phase of the algorithm for reduction with respect to composition. In this phase, parse trees of individual components are separately reduced using the rewriting rules and marks gathered in the first phase

4.5 Reduction with respect to property

An approach to checking a general temporal property stated in modified LTL_{-X} was already presented in Section 3.5. It can be used both on a single protocol or a composition of more protocols. This section will focus on reducing the architecture to contain only archetypal traces that do influence validity of the LTL formula.

Formally speaking, if the LTL formula is successfully verified then it is by Definition 20 true on all traces. So there is no trace that is really influencing the validity. LTL formula will stay true no matter which traces are removed. However, it is still possible to identify traces that are representative for different ways of fulfilling the property stated as the LTL formula.

As described in Section 3.5, negation of the LTL formula is translated into a non-deterministic finite state machine that accepts all traces that violate the formula. In this state machine, transitions are labeled either with a single event token (only a single event can be emitted at one time moment) or by a possibly empty set of negated event tokens. The second type of transition can accept any event token that is not in the set. Recall the example of LTL formula from Section 4.1: G (a^ → F z^). State machine corresponding to its negation can be seen on Fig. 17. This automaton accepts any word that starts with an arbitrary prefix (self loop in the state 0) and continues with a^ and any suffix without z^.

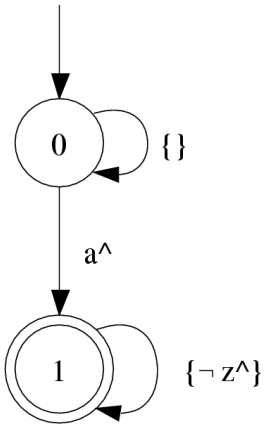


Figure 17. Non-deterministic finite state machine accepting all traces that violate the LTL formula: G (a^ → F z^)

Let π be a trace not accepted by this automaton. Informally, symbols of the trace π can be classified as important or unimportant. The symbol can be seen as important if it allows different run of this automaton then any symbol not mentioned in the formula or if it disallows some of runs otherwise possible. For example in the state 0, symbol a^ is important because it allows traversal to the state 1, which is not possible otherwise. In the state 1, symbol z^ is important because it disallows use of the self-loop.

Definition 32. Let M be a non-deterministic finite state machine, π a trace of length n and x_i, where $1 \leq i \leq n$, symbol on the trace π. Let S_i be a set of states of M that are

reachable by i - 1 long prefix of π. Then the symbol x_i is *important* if there is a transition from any state in S_i marked by either the symbol x_i itself or by a set of negated symbols including negation of x_i.

Using this definition and the automaton from the previous example, the underlined symbols in the following trace are important:

<?a^, ?b^, !a$, !z^, !b$, ?z$, !z^, ?z$, ?a^, !a$>

The next step is identification of important traces from the set of traces generated by a behavior protocol. Just stating that important traces are those that contain important symbols is not enough. Use of for example parallel operator can mix in unwanted symbols. So the important traces have to be defined as in some meaning "shortest" traces containing the same important symbols – representatives.

Definition 33. Let $\wp$ be a behavior protocol, $\pi \in \mathscr{L}(\wp)$ one of its traces and n length of π. Trace π is *important* if there is no nonempty set $I \subseteq \{1, 2, \ldots, n\}$ such that:

1) If m is the smallest element in I then x_m, the m-th symbol of π, is an accept event (event of a form ?a^ or ?a$, where a is a method) and
2) For each $i \in I$, x_i is not an important symbol of π and
3) There is $\rho \in \mathscr{L}(\wp)$ created from π by removal of all x_i, for $i \in I$.

For example, consider again the LTL formula G (a^ → F z^) with the corresponding finit automaton (from Fig. 17) and a behavior protocol (?a{!z})* | (?b{!w})*. Traces of this protocol are parallel interleavings of calls to methods a and b. More precisely, traces of this protocol are exactly all interleavings of two sequences α and β, where $\alpha \in \{$<>, <?a^, !z^, ?z$, !a$>, <?a^, !z^, ?z$, !a$, ?a^, !z^, ?z$, !a$>, …$\}$ and $\beta \in \{$<>, <?b^, !w^, ?w$, !b$>, <?b^, !w^, ?w$, !b$, ?b^, !w^, ?w$, !b$>, …$\}$.

By Definition 33, all traces for which β = <> are important, because they cannot be shorten whithout omitting some important symbols. Not so obviously, no trace for which $\beta \neq$ <> is important. Whenever $\beta \neq$ <> for some trace π, there exists a nonempty set I of indices of first appearances of symbols ?b^, !w^, ?w$ and !b$ in π. No symbol with index from I is important. First index identifies an event ?b^ which is the accept event. Trace π' created by removal of symbols indexed by I is interleaving of β' and unchanged α, where β' is derived from β by omitting one repetition of sequence <?b^, !w^, ?w$, !b$>. In other words, π' is also a trace of the protocol (?a{!z})* | (?b{!w})*. Thus, such a π is not an important trace by Definition 33. To summarize, important traces for this example are just traces: $\{$<>, <?a^, !z^, ?z$, !a$>, <?a^, !z^, ?z$, !a$, ?a^, !z^, ?z$, !a$>, …$\}$ and the protocol could be reduced to (?a{!z})*.

In the light of these definitions, the task is to reduce all the participating frame protocols of the individual components in the architecture and the frame protocol of the resulting compound component, so that the resulting composition would not contain any communication errors but preserve all important traces. Recall the example from Section 4.4, which describes composite component C consisting of components A and

B (see Fig. 13). If α, β and γ are behavior protocols of components A, B and C respectively, then the goal is to create protocols α', β' and γ' such that $\alpha' \leq_R \alpha$, $\beta' \leq_R \beta$ and $\gamma' \leq_R \gamma$. Note the difference in requirements put on the γ'. Reduction with respect to composition requires $\gamma' \leq_{IR} \gamma$. This difference corresponds to the fact already mentioned in the motivation. In general, reduction with respect to LTL formula limits behavior of the whole composition. That means, component with the reduced behavior γ' can be unusable on some places, where the original component C can be used. Protocol γ' may provide less than protocol γ.

Protocols α', β' and γ' can be imagined to define virtual wrapper components A', B' and C' respectively. See Fig. 18 in contrast to Fig. 14.

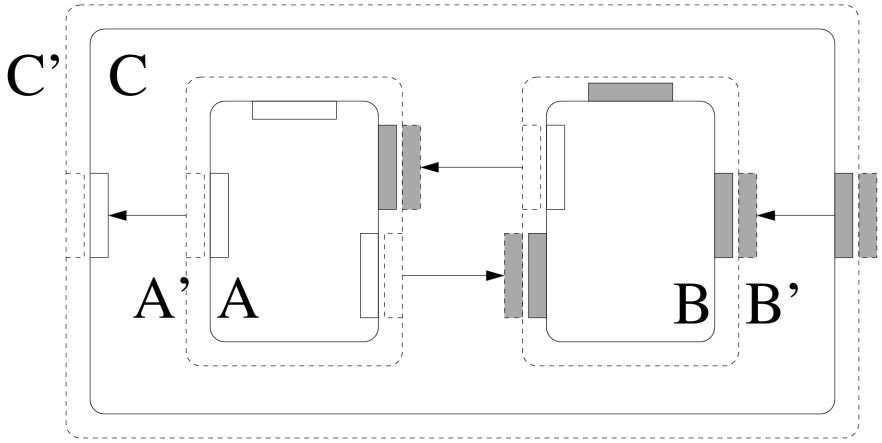


Figure 18. Idea of the virtual wrapper components for reduction with respect to property. The components A, A', B, B', C and C' have the behavior protocols α, α', β, β', γ and γ', respectively, where $\alpha' \leq_R \alpha$, $\beta' \leq_R \beta$ and $\gamma' \leq_R \gamma$

4.6 Dependency graph

Because of the threat of the state explosion problem, any acceptable solution should not increase the time complexity necessary to check the LTL formula. It is linear in size of the connected state space of the behavior protocols and the finite automaton representing the LTL formula. These are generally exponential in sizes of the protocols and the formula, respectively. The proposed solution is suboptimal. It does not necessarily find the best reduction. However, it can manage without increasing the time complexity.

As well as the algorithm for reduction with respect to composition, the algorithm first marks the parse trees of participating behavior protocols and then applies the rewriting rules. However, the situation is not so easy, because reachability is not enough in this case. For marking nodes of the parse trees, the algorithm creates a *dependency graph* derived from the parse trees and possible communication. Dependency graph is an oriented graph. Each vertex of the graph represents a group of nodes from a parse tree that share the importance mark. Edge in the dependency graph represents implication of importance (and thus preservation). If the source of the edge is marked as important then also the destination is marked.

The purpose of the graph is to represent the communication dependencies between protocols. It should be designed to have the following property. Let ?a^ be a leaf of a parse tree and *x* be a vertex of the dependency graph representing a group containing the leaf. Suppose that each symbol generated by the leave ?a^ is important. If all nodes represented by all vertices reachable from *x* were marked as important (and also parallel-important in case of parallel operator nodes) then a consequential reduction would preserve all important traces with respect to Definition 33 and no communication errors would be introduced. Having this property, the marking of the parse tree nodes can be done during the LTL checking by just marking corresponding vertices of dependency graph whenever an important symbol is generated.

The algorithm itself is divided into three steps. First, parse trees are partitioned into groups of nodes that define vertices of the dependency graph. The graph is initialized by edges that are implied by the parse trees' structure (see Fig. 19). Second, during the state space traversal of the checking process, vertices of dependency graph are marked as important if any of their nodes generates an important symbol and some additional edges may be added depending on the components' communication (see Fig. 20). Third, the vertices reachable from the marked vertices are also marked as important. Node of the parse tree is marked as important if it belongs to the group of an important vertex. Then, the rewriting rules are applied to perform the reduction.

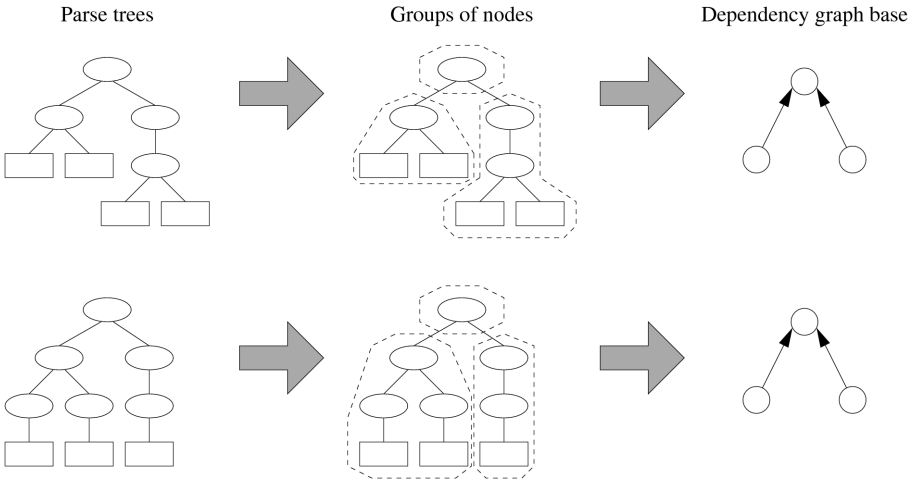


Figure 19. First phase of the algorithm for reduction with respect to property. Nodes of the parse trees are partitioned into groups that form vertices of the dependency graph. Edges implied by the structure of parse trees are also added in this phase

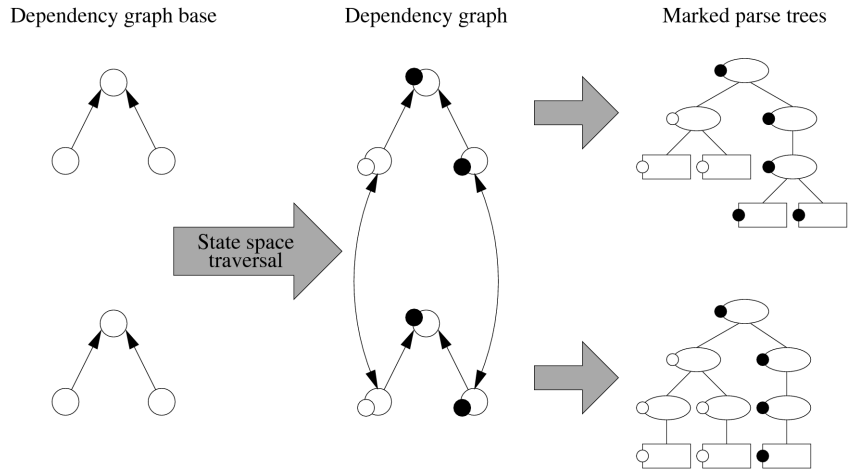


Figure 20. During LTL checking, edges implied by the communication are inserted and vertices of the dependency graph are marked as important, whenever an important symbol is generated (second phase). Marks for parse tree nodes are derived from the marks in the dependency graph (third phase)

The partitioning of each parse tree is done by a recursive depth first search traversal starting from the root node. It exploits predicates *nullable* and *passive* defined in Section 4.3. Domain of the predicates is intuitively extended to the parse tree nodes. The node is nullable (resp. passive) if the subprotocol represented by the node is nullable (resp. passive). Five basic rules are used to decide whether to add the node to his parent's group or to start a new group. The first matching rule is applied:

- If the current node is not passive then add it to the parent's group
- If the current node is nullable then add it to a newly created group
- If the parent node is "*" then add the current node to the parent's group
- If there is no nullable ancestor of this node or there is at least one "|" or ";" node between this node and the closest nullable ancestor then add the current node to the parent's group
- Otherwise add the current node to a newly created group

Meaning of the first two rules is quite straightforward. The first says that no subprotocol that can spontaneously emit an event will be omitted without omitting its parent too. No such rewriting rule exists, so there is no need to create a new group. The second rule enforces creation of a new group for the passive subprotocol that is substitutable by a NULL protocol, because it could possibly be left out and should be treated individually. The third rule is just a technicality that prevents creation of the unnecessary single member groups for "*" nodes. The fourth rule is quite tricky. It is chosen to distinguish between two following situations.

Protocol (?a + ?b)* (see Fig. 21) generates set of traces {<>, <?a^, !a$>, <?b^, !b$>, <?a^, !a$, ?a^, !a$>, <?a^, !a$, ?b^, !b$>, …}. Suppose that each occurrence of a symbol !b$ is important (as underlined). Then only traces {<>, <?b^, !b$>, <?b^, !b$, ?b^, !b$>, …} are important by Definition 33. That is why the separate vertices

are needed for the parse tree nodes for ?a and ?b, because these nodes can be reduced independently. Fifth rule is applied.

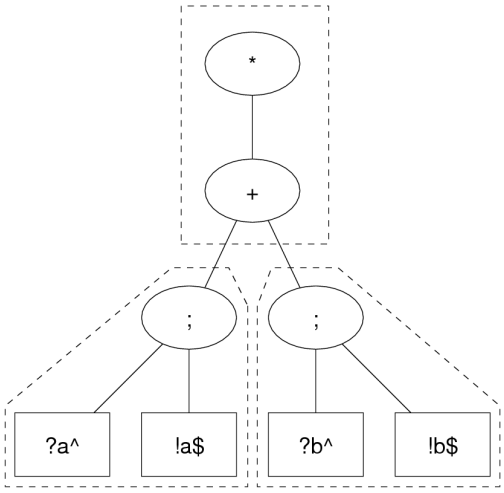


Figure 21. Parse tree for the behavior protocol: (?a + ?b)*. Groups of nodes define vertices of the dependency graph

On the other hand, protocol ((?a + b?); ?c)* generates set of traces {<>, <?a^, !a$, ?c^, !c$>, <?b^, !b$, ?c^, !c$>, <?a^, !a$, ?c^, !c$, ?a^, !a$, ?c^, !c$>, …}. If every occurrence of ?c^ is important (as underlined) then all these traces are important too, by definition. In this case, parse tree nodes for ?a and ?b should belong to the same group – the third rule is applied and the group of the parent node is shared (see Fig. 22).

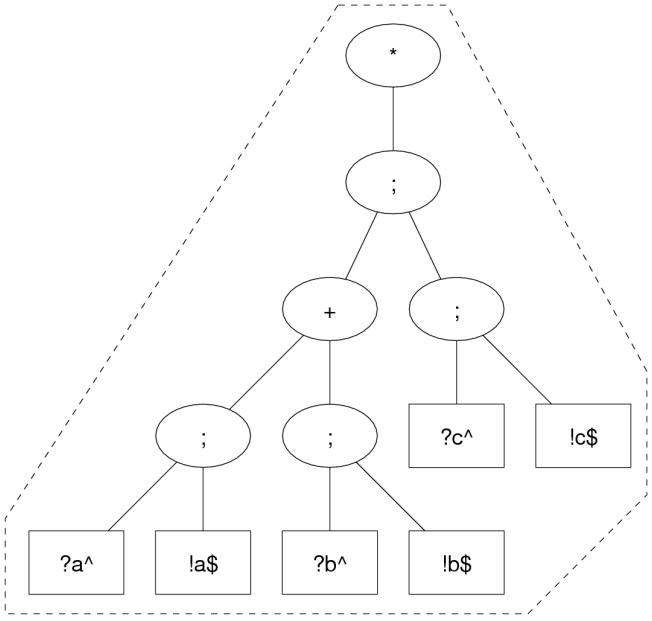


Figure 22. Parse tree for the behavior protocol: ((?a + ?b); ?c)*. The single group of nodes defines the only vertex of the dependency graph

Oriented edges of the dependency graph represent preservation dependence. If the source of the edge is preserved in the resulting protocol then the destination has to be preserved as well. The protocol: ?a{?b*}*, generates two vertices. One for ?a{…}* and one for ?b*. Additionally an edge is created from ?b* to ?a{…}*. If the ?b* part is preserved then also the ?a{…}* part is needed and the result is ?a{?b*}*. However, the result ?a* is also possible (no edge goes the opposite direction).

There are two rules for adding edges between two dependency graph vertices. Let *x* and *y* be distinct vertices of the dependency graph:

- If there are *a* and *b* nodes of the parse trees, such that *a* is parent of *b* and *a* (resp. *b*) is in the group represented by *x* (resp. *y*), then there will be an edge coming from *y* to *x*.
- If there are *a* and *b* leaves of the parse trees, such that *a* (resp. *b*) is in the group represented by *x* (resp. *y*) and the composition contains a trace in which *a* and *b* communicate (generate events that are joined into a tau event), then there will be two edges coming from *y* to *x* and from *x* to *y*.

Edges enforced by the first rule are added during the initialization phase because they depend only on the syntactic structure of the protocols. Other edges are added continuously during the second step of the algorithm – the state space traversal. The first rule says that whenever a particular node generates an important symbol, then the node and its ancestors cannot be completely reduced out. This surely preserves all important traces that contain this symbol generated by this node in the single protocol. However these traces could be still reduced out if they waited for an event that would not be emitted. The second rule transitively enforces existence of all traces that are necessary in the other protocols to preserve the important traces. The second rule also prevents introduction of the communication errors, provided that the original composition was error free. Complete proof of this claim is very technical and will not be presented here. Just to sketch it out, the construction of the dependency graph guarantees, that a leaf of the parse tree survives the reduction if and only if the associated vertex was marked as important. Thus, if a bad activity error is present in the reduced composition, it was also present in the original one. Otherwise, the second rule would ensure, that the vertex associated with the accept event preventing the bad activity in the original protocols will be marked as important. Absence of the other communication errors can be proved in a similar but more complicated way.

The time complexity of the first and the third phase of the algorithm is clearly polynomial in the size of the protocols, because they need only depth first search traversals of the parse trees and the dependency graph (the rewriting can be also done by a single DFS traversal). The second phase is done during the LTL checking and adds only a constant overhead per each visited state and each used transition. Thus, it is still linear in the size of the connected state space of the protocols and automaton representing the formula. Thus, it is exponential in the size of the protocols and the LTL formula.

It was already said, that this solution is suboptimal. To show an example, consider again the protocol ((?a + b?); ?c)*. As shown before, it is represented by a single vertex in the dependency graph. For this reason, the algorithm can reduce it to either NULL protocol or not at all. However, if every occurrence of the event ?b^ was important, then a protocol (?b; ?c)* would be a reasonable reduction containing all

important traces (the presented algorithm cannot achieve this reduction). The problem is inherent in the fact that the decision, whether a node is used on any important trace depends on a trace, not just the state. To make it properly, one would have to analyze all different traces (without cycling), number of which is generally exponential in the size of the state space. This would yield in the unbearable time complexity.

5 Prototype

As a part of the thesis, a prototype implementation of the described techniques was developed. This section presents the prototype and justifies the design decisions that were made.

An early decision was to base the prototype implementation on one of the existing behavior protocol checkers. Both verification of LTL and reduction of protocols are based on the traversal of the state space, which is also a keystone of the behavior protocol checkers. There were two choices described in the following subsection; both written in Java. This fact determined the use of Java also for all the extensions.

5.1 Original checker

Although, there were already several generations of behavior protocol checkers, only two possibilities are considered. First choice is the current behavior protocol checker that is part of the SOFA and Fractal component models' reliability extensions [32, 10]. The second – DChecker – is being developed as a part of a master thesis focusing distributed checking of behavior protocols [38]. The checker, despite being in early development stage, involves some promising optimizations and gives very good results in the local version that was only available at the time of writing.

Each behavior protocol can be equivalently imagined as a finite automaton accepting exactly the traces that the protocol can generate. A parallel composition of protocols using the consent operator can be again represented by a finite automaton. To check compliance of this composition, state space of the automaton has to be sought for communication errors. However, the state space of this automaton can be – and usually is – exponential in a size of participating protocols. This problem is known as the *state explosion* problem. Both checkers differ in a way they cope with it.

The approach of the first checker is described in [23]. It generates the automaton representing the composition *on-the-fly* (states and transitions are created during the checking process as necessary) from the structure called *parse tree automaton* (PTA). PTA is a tree structure isomorphic to the protocol parse tree, extended as follows: (i) in each leaf, there is a primitive automaton representing an event of the protocol (i.e. it has two states an initial and a final one) and (ii) each inner node combines its children PTAs using a protocol operator. The state of this automaton is then determined by a state of all primitive automata in leaves and some additional information from other nodes (for example, identification of the branch of the + operator that is used, if any). The state identification is a selective concatenation of states of primitive automaton and the additional information. Unfortunately, this yields to state identification with variable length. Moreover, the length is far from optimal in the worst case. Consider a protocol consisting of n consequent events. The minimal automaton representing this protocol has $n + 1$ states with n transitions, each representing one event. Evidently, state of this automaton can be represented in $\lceil \log_2(n + 1) \rceil$ bits. However, the state identification generated by PTA is n bit long.

Another problem follows from the fact that PTAs are generally nondeterministic. Since the behavior protocols have the trace semantics, one has to consider all states reachable by the same prefix at the same time. For this reason, each PTA is associated with a special node that converts information from PTA to the deterministic form. State identification is a concatenation of all states reachable by already processed prefix. These deterministic nodes are connected using the consent nodes.

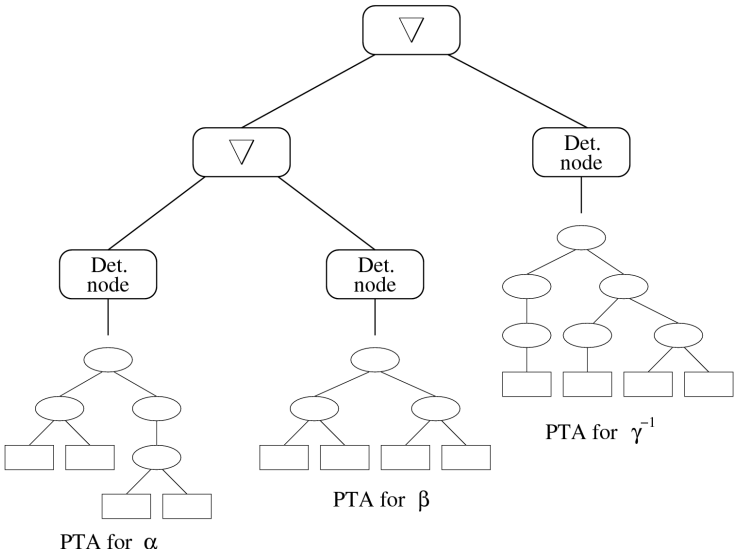


Figure 23. The tree structure for on-the-fly state space generation in the PTA-based checker. Depicted structure is used during the vertical compliance checking of the composition of the components A and B against frame C. A, B and C are specified by the protocols α, β and γ, respectively

Fig. 23 shows the whole structure that is used for on-the-fly state space generation. In each step of the algorithm, the whole structure has to be traversed to create a set of possible transitions. This is a trade of between memory consumption and time necessary for the checking. The PTA-based checker can check really huge protocols without running out of memory, but it is rather slow.

The approach of the second checker is based on the following observation: "The main source of the state explosion is the parallel composition of protocols by a consent operator, whereas the state spaces of the individual protocols are usually feasible." This observation is arguable in presence of parallel operators, however it allows for very elegant optimizations.

DChecker translates the individual behavior protocols into the corresponding nondeterministic finite state machines and then to the deterministic finite state machines, which are used instead of PTAs. This approach has two advantages. First, number of states is known in advance, so the state identification can be coded optimally and is of a fixed size. Second, no deterministic node is needed, as was the case with PTAs. The resulting structure used for on-the-fly state space generation can be seen on Fig. 24.

Because the FSMs can be accessed very efficiently, the checker is quite fast. Of course, the price is that it can check only protocols, whose FMSs fit into memory.

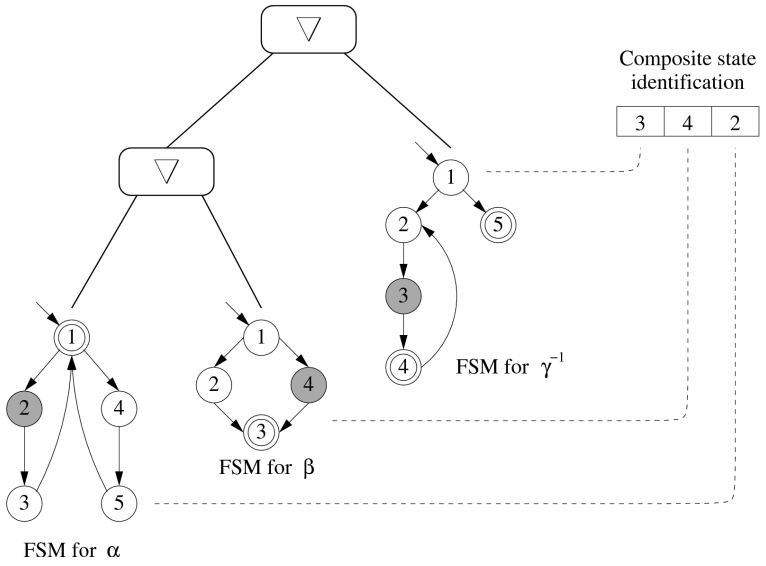


Figure 24. The tree structure for on-the-fly state space generation in DChecker. This structure is used for checking vertical compliance of the composition of the components A and B against the frame C. A, B and C are specified by the protocols α, β and γ, respectively

For the performance reasons discussed above, DChecker was chosen to be a base for the LTL checking and protocol reduction extensions. There was one more reason, the PTA-based checker uses concept of the *atomic actions* introduced in [22], which is not yet fully stabilized and which changes semantics of behavior protocols. This fact may cause problems, since this work does not take atomic actions into account. Fig. 25 shows the table with performance comparison of the checkers. Input protocols can be found on the attached CD ROM in the directory `/examples/evaluation`.

	`tiny.bp`	`medium.bp`	`large.bp`
PTA-based checker	3 sec. 6 084 states	439 sec. 456 976 states	1 039 sec. 1 001 832 states
DChecker	1 sec. 3 969 states	15 sec. 194 481 states	37 sec. 500 094 states

Figure 25. Performance test of the PTA-based checker and DChecker. Protocols are available on the attached CD. All test were run on Pentium 4 3 GHz with 1 024 MB RAM (600 MB for Sun JVM – build 1.5.0_06-b05), Windows XP SP2

5.2 LTL checking

The local version of DChecker was extended to support checking of the modified LTL_{-X} formulas and reduction of behavior protocols both with respect to composition and with respect to property as described in Sections 3 and 4. In this section, the LTL checking extension is described. It consists of two parts: translation of modified LTL_{-X} formula

into a nondeterministic automaton and modification of the on-the-fly state space generation structure needed for the LTL checking itself.

A simple parser of the LTL formula was generated using JavaCC [20], the Java tool for easy generation of parsers. The code of the parser resides in the package `bpchecker.ltl.parser` along with the grammar definition file `ltl.jj`.

The translation of the modified LTL_{-X} formula into a nondeterministic finite automaton is defined in the package `bpchecker.ltl.translation`. During the development, also the library LTL2BA4J [26], which is the Java binding of the C tool LTL2BA for LTL to Büchi automata translation [25], was used for the translation (the option is still available). However, the result was a Büchi automaton with the infinite trace semantics. In many cases, the Büchi automaton could be directly used as a nondeterministic finite automaton for expressing also a property in the modified LTL_{-X}, but it is not true in general. The translation algorithm was implemented using the same interface as the LTL2BA4J library, to allow easy substitutability. It is based on the tableau-based algorithm described in [16]. The main idea is to create states of the automaton describing different ways how to satisfy the LTL formula. Each state is represented by the following attributes:

Incoming	Set of states. From each of the states, there should be a transition coming to this one.
New	Set of subformulas that must hold in this state and have not yet been processed.
Next	Set of subformulas that must hold in any immediate successor of this state.
Forbidden	Set of events that cannot be used to transit to this state.
Obliged	Single event that can be used to transit to this state. Either *Forbidden* or *Obliged* is empty.

Let ϕ be a modified LTL_{-X} formula to be translated. First of all, ϕ is rewritten to contain only the operators: $\wedge$, $\vee$, U, R and $\neg$, where negation can be used only before the event tokens. This can be done using relations from Definition 9 and De Morgan's laws.

Initialization: The algorithm starts by creating a dummy initial state and one working state. The *Incoming* attribute of the working state contains the initial state and the *New* attribute is set to $\{\phi\}$. The working state is added to the set of unprocessed states.

Core: While there is an unprocessed state, remove it from the set of unprocessed states and do the following, until its *New* attribute gets empty. Remove one formula ψ from its *New* attribute and:

- ψ is an event – if the *Forbidden* attribute contains this event or the *Obliged* attribute is a different event, then discard this state, because the entering condition cannot be met. Otherwise, set *Obliged* to this event and empty the *Forbidden* attribute, because all other events are already forbidden by setting up *Obliged*.

- ψ is a negation of an event – if *Obliged* contains this event, then discard the state. Otherwise add the event to the *Forbidden* set.
- ψ is a conjunction – add both subformulas to the *New* set.
- ψ is of a form $\phi_1 \vee \phi_2$, ϕ_1 U ϕ_2 or ϕ_1 R ϕ_2 – split the state and add corresponding formulas to the *New* and *Next* sets of the two states as depicted on Fig. 26.

When there is no more formulas in the *New* attribute, look to the set of processed states whether there is a state with the same *Next*, *Obliged* and *Forbidden* attributes. If so, add states from the *Incoming* set to the *Incoming* attribute of the already processed state and discard the newer one. Otherwise, add the current state to the set of processed states and create one unprocessed state initialized by setting *New* equal to the *Next* set of the current one and by adding it to the *Incoming* set.

Finalization: It is left to say what states are accepting. With the only exception of the dummy initial state, the state is accepting if it does not contain any formula of a form ϕ_1 U ϕ_2 in its *Next* set. This is, because an eventuality, specified by ϕ_2, is yet to be satisfied. The initial state is accepting if the original formula ϕ is satisfied by an empty trace. This can be easily found out using directly Definition 19.

ψ	New_1	$Next_1$	New_2
$\phi_1 \vee \phi_2$	ϕ_1	$\emptyset$	ϕ_1
ϕ_1 U ϕ_2	ϕ_1	ϕ_1 U ϕ_2	ϕ_2
ϕ_1 R ϕ_2	ϕ_2	ϕ_1 R ϕ_2	ϕ_1, ϕ_2

Figure 26. State split. Formulas to be added to the *New* and *Next* sets of the new states

Because the translation algorithm is not substantially changed, it shares some important properties. Probably the most important one is that the resulting automaton is generally exponential in a size of the formula. Thus, the time and space complexity is also exponential in the worst case. For example, consider a formula of a form:

$$(\mathrm{F}\, e_1) \wedge (\mathrm{F}\, e_2) \wedge (\mathrm{F}\, e_3) \wedge \ldots \wedge (\mathrm{F}\, e_n)$$

The size of a finite automaton representing this formula has to be exponential, since in every state, it has to somehow remember what eventualities were already satisfied. Number of the different subsets is equal to 2^n.

In the implementation, one useful optimization is added (based on the paper [17]). The drawback of the straightforward algorithm above is that all transitions going into one state are required to be labeled by the same labels, derived from the contents of the *Obliged* and *Forbidden* sets. However, two states should be perceived as equal if the same traces start from them. Based on this observation, two states are considered equal if they share contents of the *Next* set. To ensure the correctness, states in the *Incoming* set have to be annotated with information determining the future label of the transition.

When it comes to the checking process itself, the structure used by DChecker for on-the-fly state space generation was modified by injecting a new root node on top of the tree (class `bpchecker.fly.FlyLTL`) as depicted on Fig. 27. This node makes the intersection between state spaces of the behavior protocol composition and automaton representing the negation of the checked LTL formula. Whenever an accepting state of the intersection is reached, the formula is not satisfied.

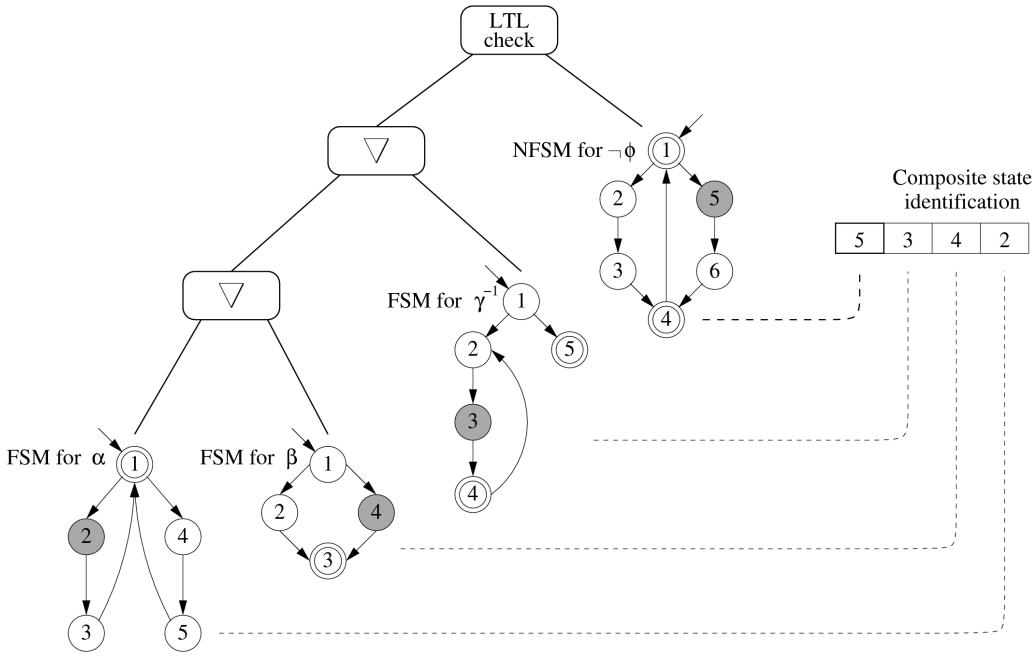


Figure 27. Modification of the DChecker's on-the-fly state space generation structure

5.3 Reduction with respect to composition

This section describes the protocol reduction extensions of DChecker. Both reside mainly in the package `bpchecker.reduction`. First part covers reduction with respect to composition. As described in Section 4.4, the algorithm first marks reachable parts of the protocols, and then performs the reduction using the rewriting rules.

In DChecker, states and transitions of the finite automata representing the state spaces of the individual protocols do not carry information, from which part of the protocol they originated. This information has to be added and preserved during the behavior protocol to finite automata translation process. The process consists of three steps: (i) the protocol is translated into a nondeterministic finite automaton (`bpchecker.fsm.NondeterministicFSM`) and (ii) a deterministic automaton (`bpchecker.fsm.DeterministicFSM`) is created from the nondeterministic and (iii) a concise representation of the finite automaton for fast on-the-fly state space generation is created (`bpchecker.fly.SimpleFlyFSM`).

These structures were altered to support user information added to the states and transitions (classes: `StateInfo` and `TransitionInfo`). This user information must support merging, because in the second step of the translation, more states or more

transitions can be merged together (method `merge`). Later during the state space traversal, these classes, if present, are called back, whenever the state is visited or accepted or when the transition is used. This is enough to implement adding marks to the reachable parse tree nodes.

Each parse tree node has an initially unset *important* marker (`NodeMarker`) and also a *parallel-important* marker, in case of the parallel operator node. In the first step of the translation algorithm, these markers are added to the user information of states and transitions, to be set up on the appropriate callbacks. The following rule is used: The important marker of a parse tree node is added to each transition that leads out of the subtree represented by the node, and to all accepting states (to be marked on the `accepted` callback) within the subtree of the node. Moreover, the parallel-important marker is added in both cases if the transition's source or the accepting state represents position in both subtrees of the parallel operator node (child protocols were interleaved).

During the second step of the algorithm, more states or more transitions can be merged together. In this case, sets of markers associated with the particular states or transitions are also merged. The third step does not affect structure of the automata, so the user information is just passed on. An example of the first two steps of the algorithm is shown on Fig. 28. The non-deterministic protocol ?a*; ?a is used to demonstrate merging of the markers in the second step.

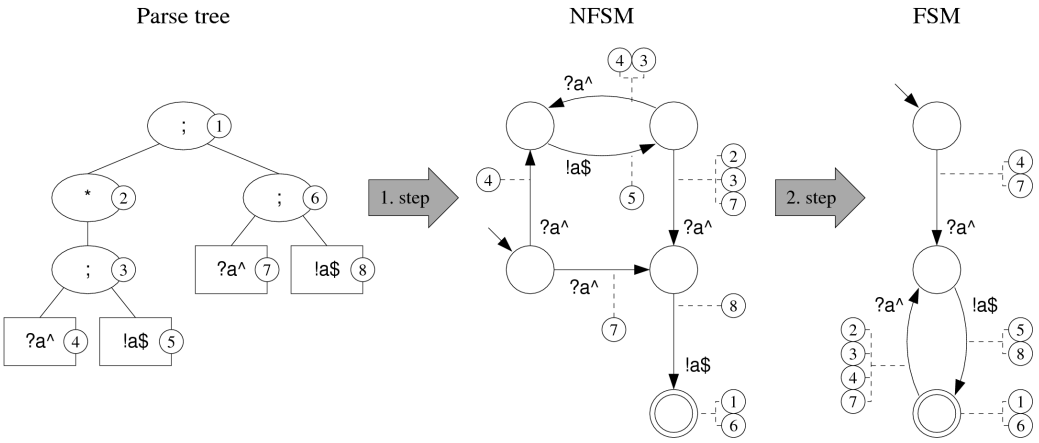


Figure 28. First two phases of the behavior protocol to deterministic finite state machine translation process for protocol ?a*; ?a. Importance markers are gathered in the user information associated with states and transitions of the automata

Later during the state space traversal, all met markers are just set up as visited via appropriate callbacks on `StateInfo` and `TransitionInfo`. For this purpose, the depth first search traversal class (`FlyDFSTraversal`) was altered to make the callbacks on the user classes.

After the state space traversal, when all reachable parse tree nodes are marked, the reduction takes place. In the implementation, the rewriting rules can be used in just a single pass through the parse trees. The *visitor pattern* (see [15] for introduction to the design patterns) is used to perform the reduction in a single recursive traversal through the parse trees (`bpchecker.parser.traversal.ReductionActions`).

5.4 Reduction with respect to property

This section focuses on reduction with respect to property. As described in Sections 4.5 and 4.6, the parse trees are first partitioned into groups of nodes that later form vertices of the dependency graph (class `DependencyGraph`). The partitioning and the base dependency graph creation are done again by a single recursive pass through the parse trees using the `DependencyGraphActions` visitor from the package `bpchecker.parser.traversal`. This way, each node of the parse tree is associated with a vertex of the dependency graph.

During the behavior protocol to a finite automaton translation process, the dependency graph vertices representing protocol leaves are associated with the automaton transitions in a same manner as the markers above. This allows adding of additional edges to the dependency graph during the state space traversal, whenever a communication takes place (a tau event is created) and also marking dependency graph vertices as important, if the important symbol is generated. After the traversal, the dependency graph vertices reachable from the important ones are also marked as important and the reduction is done by using the same `ReductionActions` parse tree visitor.

6 Case study

This section presents a short case study focusing on the use of reduction with respect to composition on the real-life protocols developed as a part of the already mentioned project Component Reliability Extensions for Fractal Component Model [10]. The second part of this section discusses a possible use of LTL checking of behavior protocols and reduction with respect to property to facilitate search for a suitable component in a big component repository. Unfortunately, no such a big component repository annotated by behavior protocols is available, so no real examples are given.

6.1 Demo application

As a part of the aforementioned project, a demo component application was designed. The components' behavior was specified using behavior protocols and components were checked for both horizontal and vertical compliance. The demo application constitutes the airport service for providing wireless internet connection. The connection is granted to the owners of the first class or business class tickets and to the owners of the Frequent Flayers Card if they have a valid ticket. Other passengers can prepay the connection time by a credit card.

The system consists of about twenty components including virtual components that are used to model the synchronization. The Fractal component model [31] is hierarchical – the components are either primitive or composite. Each component is associated with its frame protocol. The overview of components can be found on Fig. 29 borrowed from the project's documentation available on its homepage.

To briefly present the main components, the Firewall component represents the firewall for blocking unauthorized internet connections and redirecting them to the login page. The FlyTicketDatabase and FrequentFlyerDatabase components mediate the access to the databases of the airlines companies. CardCenter communicates with the bank credit card services and the AccountDatabase component encapsulates the database of accounts with prepaid internet connection. The Token component is a dynamically created entity representing a single logged user. All communication is orchestrated by the Arbitrator component. The last main component is the DhcpServer. It represents a DHCP server for a dynamic IP address allocation with support for the use of the permanent IP address database. This database mapping Mac to IP addresses could be connected via the IIpMacPermanentDb interface and its use can be triggered on via the IManagement interface. However, both these interfaces are left unbound and the feature of permanent IP address allocation is not used in the demo application.

As already said, some of the components are composite. In the rest of this section, only the highest level of hierarchy will be studied. The behavior protocols of the main components will follow. The Firewall component provides interface IFirewall for the management of the port blocking and redirection and features the following behavior protocol:

```
(
        ?IFirewall.EnablePortBlock_1*
        |
        ?IFirewall.EnablePortBlock_2*
        |
        ?IFirewall.EnablePortBlock_3*
        |
        ?IFirewall.DisablePortBlock*
)
```

The component FlyTicketDatabase provides interfaces IFlyTicketDb for the uniform access to the airline companies' ticket databases and IFlyTicketAuth for the login confirmation based on the first or business class ticket identification:

```
(
        ?IFlyTicketAuth.CreateToken_1
        +
        ?IFlyTicketAuth.CreateToken_2
        +
        ?IFlyTicketDb.GetFlyTicketsByFrequentFlyerId
)*
```

FrequentFlyerDatabase provides IFrequentFlyerAuth for login confirmation based on the frequent flyer identification.

```
(
        ?IFrequentFlyerAuth.CreateToken {
                (
                        !IFlyTicketDb.GetFlyTicketsByFrequentFlyerId;
                        (!IFlyTicketAuth.CreateToken_2 + NULL)
                ) + NULL
        }
)*
```

CardCenter provides ICardCenter for money withdrawal requests:

```
(
        ?ICardCenter.Withdraw*
)
```

The component AccountDatabase provides two interfaces IAccountAuth for the login confirmation based on the user account identification and password and IAccount for the account management:

```
(
        (
                ?IAccount.GenerateRandomAccountId
                +
                ?IAccount.CreateAccount
                +
                ?IAccount.RechargeAccount {
                        !ICardCenter.Withdraw
                }
        )*
        |
        ?IAccount.AdjustAccountPrepaidTime_1*
        |
        ?IAccount.AdjustAccountPrepaidTime_2*
        |
        ?IAccount.AdjustAccountPrepaidTime_3*
```

```
        |
        ?IAccountAuth.CreateToken*
)
```

Token provides IToken interface to allow control over its validity:

```
(
        ?IToken.InvalidateAndSave_1 {
                (!IAccount.AdjustAccountPrepaidTime_1 + NULL);
                !ITokenCallback.TokenInvalidated_1
        }*
        |
        ?IToken.InvalidateAndSave_2 {
                (!IAccount.AdjustAccountPrepaidTime_2 + NULL);
                !ITokenCallback.TokenInvalidated_2
        }*
        |
        (
                (!IAccount.AdjustAccountPrepaidTime_3 + NULL);
                !ITokenCallback.TokenInvalidated_3
        )*
)
```

The Arbitrator component provides ILogin for logging users in and out, IDhcpCallback to accept information from DHCP Server and ITokenCallback to be informed whenever the validity of a token expires:

```
(
        (
                ?ILogin.GetTokenIdFromIpAddress
                +
                ?ILogin.LoginWithFlyTicketId {
                        !IFlyTicketAuth.CreateToken_1;
                        (!IFirewall.DisablePortBlock + NULL)
                }
                +
                ?ILogin.LoginWithFrequentFlyerId {
                        !IFrequentFlyerAuth.CreateToken;
                        (!IFirewall.DisablePortBlock + NULL)
                }
                +
                ?ILogin.LoginWithAccountId {
                        !IAccountAuth.CreateToken;
                        (!IFirewall.DisablePortBlock + NULL)
                }
                +
                ?ILogin.Logout {
                        !IToken.InvalidateAndSave_1 + NULL
                }
        )*
        |
        ?ITokenCallback.TokenInvalidated_1 {
                !IFirewall.EnablePortBlock_1
        }*
        |
        ?ITokenCallback.TokenInvalidated_2 {
                !IFirewall.EnablePortBlock_2
        }*
        |
        ?ITokenCallback.TokenInvalidated_3 {
                !IFirewall.EnablePortBlock_3
        }*
        |
```

```
        ?IDhcpCallback.IpAddressInvalidated {
                !IToken.InvalidateAndSave_2 + NULL
        }*
)
```

DhcpServer provides IManagement interface which is left unbound:

```
(
        !IDhcpCallback.IpAddressInvalidated*
        |
        (
                ?IManagement.UsePermanentIpDatabase^; (
                        !IIpMacPermanentDb.GetIpAddress*
                        |
                        (
                                !IManagement.UsePermanentIpDatabase$;
                                IManagement.StopUsingPermanentIpDatabase^
                        )
                ); !IManagement.StopUsingPermanentIpDatabase$
        )*
)
```

The frame protocol of the whole composition representing the environment follows:

```
(
                ?ILogin.GetTokenIdFromIpAddress
                +
                ?ILogin.LoginWithFlyTicketId
                +
                ?ILogin.LoginWithFrequentFlyerId
                +
                ?ILogin.LoginWithAccountId
                +
                ?ILogin.Logout
                +
                ?IAccount.GenerateRandomAccountId
                +
                ?IAccount.CreateAccount
                +
                ?IAccount.RechargeAccount
)*
```

The state space generated by this composition features above 4.5 millions states and the correctness can be verified by the DChecker in about 3 minutes (on Pentium 4 3 GHz, 1 024 MB RAM, 600 MB for Sun JVM – build 1.5.0_06-b05, Windows XP SP2). Now, suppose that the wireless internet providing application should be reused in another environment (e.g. public garden), where the parts specific to the airport location will not be used. The frame protocols representing the environment can be manually altered to contain only inputs valid in the new location (only a credit card payment is now possible):

```
(
                ?ILogin.GetTokenIdFromIpAddress
                +
                ?ILogin.LoginWithAccountId
                +
                ?ILogin.Logout
                +
                ?IAccount.GenerateRandomAccountId
```

```
        +
        ?IAccount.CreateAccount
        +
        ?IAccount.RechargeAccount
)*
```

Behavior protocols of the components can be now automatically pruned by running the reduction extensions of the DChecker. Protocols of the components Token, Firewall, CardCenter and AccountDatabase remain unchanged. On the other hand, protocols of the components FlyTicketDatabase and FrequentFlyerDatabase are reduced to NULL which means that the components are never used and can be safely left out from the composition. The protocols of the components Arbitrator and DhcpServer are reduced partially. The airport specific login calls are omitted in the protocol of Arbitrator:

```
(
    (
        ?ILogin.GetTokenIdFromIpAddress
        +
        ?ILogin.LoginWithAccountId {
            !IAccountAuth.CreateToken;
            (!IFirewall.DisablePortBlock + NULL)
        }
        +
        ?ILogin.Logout {
            !IToken.InvalidateAndSave_1 + NULL
        }
    )*
    |
    ?ITokenCallback.TokenInvalidated_1 {
        !IFirewall.EnablePortBlock_1
    }*
    |
    ?ITokenCallback.TokenInvalidated_2 {
        !IFirewall.EnablePortBlock_2
    }*
    |
    ?ITokenCallback.TokenInvalidated_3 {
        !IFirewall.EnablePortBlock_3
    }*
    |
    ?IDhcpCallback.IpAddressInvalidated {
        !IToken.InvalidateAndSave_2 + NULL
    }*
)
```

In case of DhcpServer, the unused feature of permanent addresses is left out:

```
(
    !IDhcpCallback.IpAddressInvalidated*
)
```

The behavior specifications of both the whole compositions and the composition with the simplified environment can be found on the accompanying CD in the directory `/examples/case_study`.

6.2 Component repository

The airport internet service demo is not very useful for demonstration of the LTL checking and reduction with respect to property. The typical use of these techniques would be searching a large component repository for a suitable component, when designing a component application. Each component in the repository should be annotated by its behavior protocol.

Consider a designer trying to find one of many data storage components present in the repository. The component should provide the IDataAccess interface with the methods insert, delete and query. Moreover, it should also provide the interface: IDataStorageManagement, which would allow configuring the component at runtime. On the other hand, the component should require the IFileSystem and ILogger interfaces. Methods of the IFileSystem interface provide access to a persistent storage, on which tables with data should be eternalized. The ILogger interface provides access to the system log. Suppose that the designer wants to find one of potentially many data storage components which would log each call of the methods on the IDataAccess interface (insert, delete and query).

Without the LTL checking, the designer would define a behavior protocol that would best suit his needs, and try to find a component substitutable for it. However, this way, the designer would be forced to specify also the interplay on the IFileSystem interface as a reaction on the insert, delete and query method calls. This would require creation of some kind of a super-protocol compliant with any acceptable behavior.

With the LTL checking, the designer can specify just the concrete condition that should be satisfied:

$$G\ ((\text{IDataAccess.insert}\hat{}\ \vee\ \text{IDataAccess.delete}\hat{}\ \vee\ \text{IDataAccess.query}\hat{}) \rightarrow F\ \text{ILogger.log}\hat{})$$

The LTL extension of the checker can be used to identify those components that satisfy the condition. Of course, only the components that provide the necessary interfaces have to undergo the verification process. Moreover, all protocols of the chosen set can be reduced with respect to the specified formula and the designer can read the logging interplay without being confused by parts of the protocols unimportant to the logging. Probably nothing more than the parallel calls on the IDataStorageManagement interface will be reduced in this case.

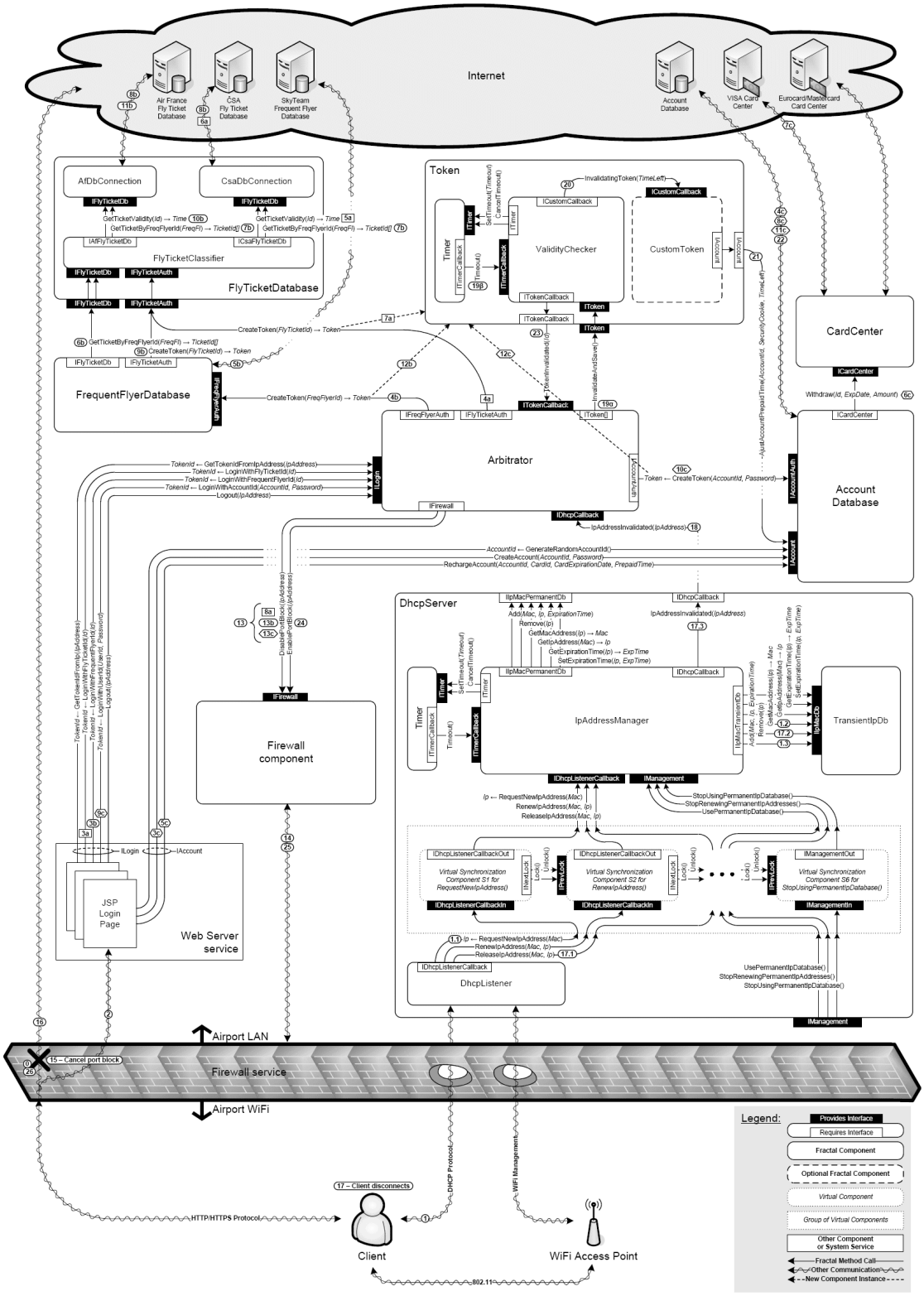


Figure 29. Overview of the airport internet providing demo application

7 Related work

This work is based on the concept of behavior protocols introduced in [35]. Since invention of behavior protocols, the notion of communication errors and compliance of components further evolved. This thesis is based on the consensual compliance presented in [1, 2]. There is an ongoing research on behavior protocols. The most significant works aim at two goals. First, a considerable effort is made to combine Java PathFinder with the checker for behavior protocols, in order to check primitive components against their behavior protocol specification [34]. Without it, there is no way to guarantee that the primitive components satisfy their behavior protocols and verification of compliance of the whole component hierarchy could be based on an unsatisfied assumption.

The second goal is motivated by the work on the demo application – the airport internet providing service – designed as a part of the project Component Reliability Extensions for Fractal Component Model [10]. Lesson taught is that the syntax of behavior protocols misses some practical features like exceptions or atomic actions. See [36, 22] for further details on this topic.

In Section 3, the finite trace semantics of LTL_{-X} is presented. Similar modifications to the semantics of LTL are done also in other works, e.g. [14, 17]. These works focus on the verification of LTL formulas using runtime analysis and simulation of the examined system. In both cases, the output is a finite trace that can be further studied. The LTL is useful for its trace semantics. In contrast with CTL, each trace can be verified separately. The authors faced similar problems with definition of the next operator. In [14], three different semantics are presented: weak, neutral and strong, and their relations are discussed. On the other hand, authors of [17] sacrifice the next operator; this approach is the closest to the solution presented in this thesis. However, our work applies LTL_{-X} on the action-based models and thus modifies the semantics even more. For example, the empty trace has a good meaning in the action-based semantics (no action was performed by the system) and LTL formulas have to be well defined on it.

In the aforementioned works, the finiteness of the traces is inherent to the methods of their acquisition. The run of a system is still perceived as an infinite trace, however an output of the simulation or the run-time analysis is always finite. The traces are in fact truncated. Thus, all possible suffixes have to be correctly discussed, which makes the goal harder. On the other hand, in our work, finiteness of traces of behavior protocols is given by definition and each finite trace represents a single complete run of the system.

The last branch of research that can be considered related to this thesis is *program slicing* (for surveys see [46, 6]). Program slicing is used to reduce a given program with respect to a certain criterion. For example, such a criterion can be a value of a variable at a particular line of the program. In that case, variables and execution paths of the program, which have no impact to that value, can be cut off. This technique is used also in the realm of model checking to reduce state space of the model before performing the verification.

As well as the protocol reduction, program slicing usually includes creation of a dependency graph with a similar meaning to the dependency graph from Section 4.6. However, there are also some major differences. First, unlike programs, behavior protocols do not contain any notion of variables or method parameters. Second, reduction of protocols is performed during the checking process and it is not primarily used to decrease the size of the state space.

8 Evaluation

In Section 1.5, the goals of this thesis were stated. This section discusses the achieved results with respect to these goals.

First goal was to target the verification of a general temporal property. This goal was reflected in Section 3, which presents an approach to verification of the properties stated in Linear Temporal Logic. LTL was chosen mainly for its trace semantics close to the semantics of behavior protocols. Also CTL and Hennessy-Milner logic were considered, but they did not qualify as discussed in Section 3.4. Since the classical LTL is defined over infinite traces and is rather state-based, modifications of the semantics were made to target finite traces of behavior protocols. To be precise, only LTL_{-X}, which is LTL without the next operator, was used. As a consequence of the changes in semantics, also the algorithm for verification had to be altered along with the LTL to Büchi automata translation algorithm.

The prototype implementation was created as an extension of the behavior protocol checker – DChecker [38]. Time complexity of the verification process in the worst case is exponential in both size of the protocols and size of the LTL formula. The following LTL formulas were verified on the behavior protocol below. The performance data are summarized on Fig. 30.

Formula 1: $G\,(c1.a\hat{} \rightarrow F\,c2.y\hat{})$

Formula 2: $G\,(c1.a\hat{} \rightarrow (F\,c1.a\$ \wedge (c2.x\hat{}\ R\ \neg c1.a\$)))$

Formula 3: $G\,(c1.a\hat{} \rightarrow (F\,c1.a\$ \wedge (c2.x\hat{}\ R\ \neg c1.a\$))) \wedge$
$G\,(c1.zz\hat{} \rightarrow (\neg c1.ww\hat{}\ U\ c1.zz\$))$

Formula 4: $G\,(c1.a\hat{} \rightarrow (F\,c1.a\$ \wedge (c2.x\hat{}\ R\ \neg c1.a\$))) \wedge$
$G\,(c1.zz\hat{} \rightarrow (\neg c1.ww\hat{}\ U\ c1.zz\$)) \wedge$
$(F\,c2.x\hat{} \leftrightarrow F\,c2.y\hat{})$

Formula 5: $G\,(c1.a\hat{} \rightarrow (F\,c1.a\$ \wedge (c2.x\hat{}\ R\ \neg c1.a\$))) \vee$
$G\,(c1.zz\hat{} \rightarrow (\neg c1.ww\hat{}\ U\ c1.zz\$)) \vee$
$(F\,c2.x\hat{} \leftrightarrow F\,c2.y\hat{})$

```
frame: ?c1.a* | ?c1.b* | ?c2.c* | ?c2.d*

# bound methods
sync{c1.a, c1.b, c2.c, c2.d}

# component 1
?c1.a{ !c2.x | !c2.y }* |
?c1.b{ !c2.xx* | !c2.yy* }* |
?c1.w* |
?c1.z* |
?c1.ww* |
?c1.zz*

# bound methods
sync{c2.x, c2.y, c1.w, c1.z, c2.xx, c2.yy, c1.zz, c1.ww}

# component 2
?c2.c{ !c1.w* | !c1.z* }* |
?c2.d{ !c1.ww* + !c1.zz* }* |
?c2.x* |
?c2.y* |
?c2.xx* |
?c2.yy*
```

	Time	*With reduction*	*States of NFSM for LTL*	*Visited states*
Compliance test	12 sec.	–	–	119 952
Formula 1	14 sec.	26 sec.	2	254 898
Formula 2	19 sec.	31 sec.	4	359 856
Formula 3	24 sec.	42 sec.	7	511 559
Formula 4	27 sec.	49 sec.	11	607 695
Formula 5	20 sec.	33 sec.	49	348 389

Figure 30. Performance summarization. All test were run on Pentium 4 3 GHz with 1 024 MB RAM (600 MB for Sun JVM – build 1.5.0_06-b05), Windows XP SP2

Second goal consisted of developing reduction techniques for behavior protocols and it was focused in Section 4. Two types of reduction were identified: reduction with respect to composition and reduction with respect to property. Both types of reduction should help the designer to understand the behavior specification by making the protocols easier to comprehend. The first type should prune out those parts of the protocols that are not used in the particular composition. It should clarify the actual role of each component. On the other hand, reduction with respect to property removes the parts of the protocols that are irrelevant to the given property. The behavior protocols reduced in this manner should emphasize which part of the protocol makes the given property satisfied.

A prototype implementation of both of the proposed reduction algorithms was developed. Both can be performed during the verification process without increasing its time complexity. The drawback is the suboptimality of reduction results; i.e. both algorithms can reduce less than is really possible, mainly when applied to the nondeterministic protocols or in cases where the decision would require consulting the whole trace which would increase the time complexity, as discussed in Section 4.6.

Usefulness of reduction with respect to composition was demonstrated on the real-life behavior protocols in Section 6 along with description of a typical use case for LTL checking and reduction with respect to property, without application on the real data in this case.

The bottom line is that the goals stated in Section 1.5 were fulfilled. However, a natural question reads: “Does it really help the designer?” When it comes to reduction with respect to composition, it is a very straightforward technique that works reasonably well on real-life protocols. Suboptimality in case of the nondeterministic protocols (discussed in 4.3) is not an issue, because the amount of nondeterminism in the meaningful protocols is usually very low.

When considering LTL checking and reduction with respect to property, it is important to note that the LTL checking can verify only properties that are really present in the abstract model of the behavior specification. For example, in the demo application, it would be very useful to be able to check whether the system behaves consistently to a single user (e.g. logout fails for the user that is not logged in), but the behavior specification of the components is stateless and this information is simply not present there. The properties that can be checked are only conditions on various implications of method calls without respect to their parameters. In fact, it is usually not so difficult to decide validity of the formulas by just looking into the protocols. However, it requires some level of understanding to the particular protocols and this is exactly the contribution of LTL checking and reduction with respect to property. These techniques can do the same automatically and can point out the important parts of the protocols even for the unqualified users interested in reuse.

9 Conclusion and Future work

To summarize, the first contribution of this work is incorporating LTL into behavior protocols. Modifications to the semantics of LTL to suit needs of behavior protocols were proposed along with modifications to the standard LTL verification algorithm. The second contribution is development of reduction techniques for behavior protocols in order to make the comprehension of the behavior specification easier. All proposed algorithms were implemented as an extension to the behavior protocols checker DChecker.

When it comes to future work, there are two tasks that might be aimed at in the future. First, the algorithm for translation of the modified LTL_{-X} formulas into the finite automata is just a modification of the original algorithm described in [16] and it does not contain any nontrivial optimization. Thus, the resulting automaton can be larger than necessary. Second, as mentioned in Section 5 on the prototype implementation, DChecker tool, on which all the extensions were based, was still under development at the time of writing. Some additional effort have to be spend, in order to keep the extensions compatible with the alpha version that has not been completed yet. In its alpha version, the DChecker should support distributed state space traversal, which is not considered by the presented extensions.

10 References

1. Adamek, J: *Enhancing Behavior Protocols*. Master Thesis, Department of Software Engineering, Charles University, 2001.
2. Adamek, J.; Plasil, F.: *Component Composition Errors and Update Atomicity: Static Analysis*. Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363–377, 2005.
3. Andrews, T.; Qadeer, S.; Rajamani, S. K.; Rehof, J.; Xie, Y.: *Zing: A Model Checker for Concurrent Software*. Technical Report, Microsoft Research, no. MSR-TR-2004-10. 2004.
4. Baumgartner, J.; Heyman, T.; Singhal, V.; Aziz, A.: *Model Checking the IBM Gigahertz Processor: An Abstraction for Enabling Model Checking of High-Performance Netlists*. In Proceedings of the 11th International Conference on Computer Aided Verification, LNCS, vol. 1633, pp. 72–83, 1999.
5. Bergstra, J. A.; Ponse, A.; Smolka, S. A.: *Handbook of Process Algebra*. Elsevier Science, 2001.
6. Binkley, D.; Gallagher, K. B.: *Program Slicing*. Advances in Computers, vol. 43, pp. 1–50, 1996.
7. Carnegie Mellon University: Symbolic Model Verifier, http://www.cs.cmu.edu/~modelcheck/smv.html.
8. Clarke, E. M.; Emerson, E. A.; Sistla, A. P.: *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 3, issue 2, pp. 244–263, 1986.
9. Clarke, E. M.; Grumberg, O.; Peled, D. A.: *Model Checking*. The MIT Press, 2000.
10. Component Reliability Extensions for Fractal Component Model, http://kraken.cs.cas.cz/ft/public/public_index.phtml.
11. Dershowitz, N.; Jouannaud, J.-P.: *Rewrite Systems*. In Handbook of Theoretical Computer Science: Formal Models and Sematics, vol. B, chapter 6, pp. 243–320, 1990.
12. Dillon, L. K.; Kutty, G.; Moser, L. E.; Melliar-Smith, P. M.; Ramakrishna, Y. S.: *A graphical interval logic for specifying concurrent systems*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 3, issue 2, pp. 131–165, 1994.
13. Eisner, C.: *Formal verification of software source code through semi-automatic modeling*. Software and System Modeling, vol. 4, issue 1, pp. 14–31, 2005.
14. Eisner, C.; Fisman, D.; Havlicek, J.; Lustig, Y.; McIsaac, A.; Van Campenhout, D.: *Reasoning with Temporal Logic On Truncated Paths*. In Proceedings of Internation Conference on Computer Aided Verification (CAV), LNCS, vol. 2725, pp. 27–39, 2003.

15. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

16. Gerth, R.; Peled, D.; Vardi, M. Y.;Wolper, P.: *Simple on-the-fly automatic verification of linear temporal logic*. In Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification, IFIP, vol. 38, pp. 3–18, 1995.

17. Giannakopoulou, D; Havelund, K.: *Automata-Based Verification of Temporal Properties on Running Programs*. In Proceedings of the 16th IEEE International Conference on Automated software engineering, ASE'01, pp. 412–416, 2001.

18. Graphviz – Graph Visualisation Software, http://www.graphviz.org.

19. Hoare, C. A. R.:*Communicating sequential processes*. Prentice-Hall, 1985.

20. Java Compiler Compiler, https://javacc.dev.java.net.

21. Java PathFinder, http://javapathfinder.sourceforge.net.

22. Kofron, J: *Enhancing Behavior Protocols with Atomic Actions*. Technical Report, Department of Software Engineering, Charles University, no. 2005/8, 2005.

23. Kofron, J: *Performance Improvements of Behavior Protocol Checking*. In Proceedings of the 14th Annual Conference of Doctoral Students, WDS'05, pp. 25–30, 2005.

24. Lamport, L.: *"Sometime" is sometimes "not never": on the temporal logic of programs*. In Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 174–185, 1980.

25. LTL2BA - fast algorithm from LTL to Büchi automata, http://www.liafa.jussieu.fr/~oddoux/ltl2ba.

26. LTL2BA4J – Java bridge to ltl2ba, http://www-i2.informatik.rwth-aachen.de/Research/RV/ltl2ba4j.

27. Microsoft: .NET, http://www.microsoft.com/net.

28. Microsoft: Component Object Model Technologies, http://www.microsoft.com/com.

29. Milner, R.: *A Calculus of Communicating Systems*. LNCS, Vol. 92, Springer-Verlag, 1980.

30. Object Management Group: CORBA Component Model, http://www.omg.org/technology/documents/formal/components.htm.

31. ObjectWeb Consortium: Fractal, http://fractal.objectweb.org.

32. ObjectWeb Consortium: SOFA Component Model, http://sofa.objectweb.org.

33. Olner, K. M.; Osterweil, L. J.: *Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation*. IEEE Transactions on Software Engineering, vol. 16, issue 3, pp. 268–280, 1990.

34. Parizek, P.; Plasil, F.; Kofron, J.: *Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker*. In

Proceedings of 30th IEEE/NASA Software Engineering Workshop, SEW'30, 2006.

35. Plasil, F.; Visnovsky, S.: *Behavior Protocols for Software Components*. IEEE Transactions on Software Engineering, vol. 28, issue. 11, pp. 1056–1076. 2002.
36. Plasil, F.; Holub, V.: *Exceptions in Component Interaction Protocols - a necessity*. In Proceedings of Dagstuhl Seminar 04511, Architecting Systems with Trustworthy Components, LNCS, vol. 3938, to appear in 2006.
37. Pnueli, A.: *The Temporal Logic of Programs*. In Proceedings of the 18th IEEE Symposium on Foundation of Computer Science, pp. 46–57, 1977.
38. Poch, T.: *Distributed Behavior Protocol Checker*. Master Thesis, Department of Software Engineering, Charles University, to be defended in 2006.
39. SAnToS laboratory: Bandera, http://bandera.projects.cis.ksu.edu.
40. SAnToS laboratory: Bandera Specification Patterns, http://patterns.projects.cis.ksu.edu.
41. Siegel, S. F.: *The INCA query language*. Technical Report CMPSCI TR 02-18, Department of Computer Science, University of Massachusetts, 2002.
42. Siegel, S. F.; Avrunin, G. S.: *Verification of MPI-Based Software for Scientific Computation*. Proceedings of 11th SPIN Workshop, LNCS, vol. 2989, pp. 286–303, 2004.
43. Spin Model Checker, http://spinroot.com.
44. Stirling, C: Modal and Temporal Properties of Processes. Springer-Verlag, 2001.
45. Sun Microsystems: Enterprise JavaBeans, http://java.sun.com/ejb.
46. Tip, F.: *A Survey of Program Slicing Techniques*. Journal of Programming Languages, vol. 3, pp. 121–189, 1995.

Appendix

This thesis is accompanied by the CD ROM containing binaries and source codes of the implementation and a set of examples. The CD ROM is organized as follows:

`/README.txt`
: Brief description of the contents of the CD ROM

`/doc/javadoc`
: Generated reference documentation (see `index.html`)

`/doc/thesis`
: Electronic version of this document along with all the figures

`/src/`
: Source codes of the extended DChecker

`/bin/dchecker.jar`
: Executable JAR archive containing build of the extended DChecker

`/bin/lib`
: LTL2BA4J library

`/examples/`
: Directory with the examples (see `EXAMPLES.txt`)

`/prerequisites/`
: Software prerequisites of the prototype: Sun Microsystems JRE 1.5, Ant tool and Java Compiler Compiler for both Linux and Windows

Running the checker

Prior to running the application, JRE 1.5 has to be installed on the target system. The installation files can be found in the `/prerequisites` directory on the CD. Use the shell scripts `dchecker.bat` or `dchecker.sh` to run the application from the supplied executable JAR file. The command line parameters are as follows:

`-h  --help`
: View the help for command line parameters

`-i  --input 'file'`
: Read input from the specified file

`-r  --reduce`
: Reduce protocols after verification. Type of the reduction depends on the type of verification (if `--ltl` or `--ltl-file` parameter is present, then reduction with respect to property is performed)

`-o  --output 'file'`
: Output the reduced protocols to the given file

`--ltl 'ltl'`
: Verify the given LTL formula

`--ltl-file 'ltl-file'`
Verify the LTL formula from the given file

`--dot-dump 'file'`
Dump an automaton representing the given LTL to the specified file in a DOT format (it can be visualized later by the Graphviz tool [18])

`--external-translation`
Use the LTL2BA4J library for the LTL to NFSM translation

The LTL operators have to be specified as follows:

¬	➡	!
∧	➡	&
∨	➡	|
→	➡	->
↔	➡	<->

An event have to be of a form Interface.Method^ or Interface.Method$.

To run the checker on the supplied examples, use of the Ant script `/examples/build.xml` is recommended. Installation files of the Ant tool are also available in the `/prerequisites` directory. The script can be executed by just typing '`ant`' in the `/examples` directory. It will print the necessary guidelines.

Compiling the sources

For the easy compilation process, the Ant script `build.xml` is provided in the directory `/src`. It contains following targets:

`build`
Build the sources (default)

`clean`
Delete the compilation output

`rebuild`
Build the sources from scratch

`run`
Execute the built application

`parsers`
Generate the LTL and behavior protocol parsers using JavaCC tool. The `JAVACC_HOME` environment variable has to be set to the valid home directory of the JavaCC installation (installation files are provided in the `/prerequisites` directory).

If the Ant tool is correctly installed, the script can be executed from the `/src` directory by simply typing (on the write-enabled file system):

`ant` *`target`*