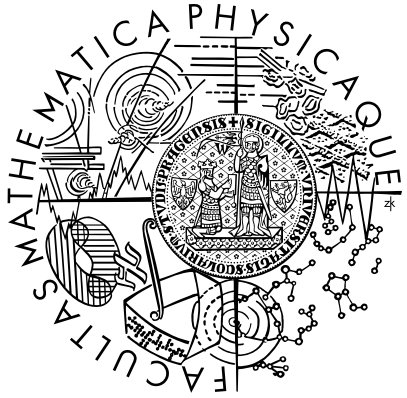
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Jan Kofroň

Behavior Protocols Extensions

Department of Software Engineering
Advisor: Prof. Ing. František Plášil, DrSc.

Abstract

Title:	Behavior Protocols Extensions
Author:	Jan Kofroň
	e-mail: jan.kofron@dsrg.mff.cuni.cz
	phone: +420 2 2191 4285
Department:	Department of Software Engineering
	Faculty of Mathematics and Physics
	Charles University in Prague, Czech Republic
Advisor:	Prof. František Plášil
	e-mail: frantisek.plasil@dsrg.mff.cuni.cz
	phone: +420 2 2191 4266
Mailing address (both Author and Advisor):	
	Dept. of SW Engineering, Charles University in Prague
	Malostranské náměstí 25
	118 00 Prague, Czech Republic
WWW:	`http://dsrg.mff.cuni.cz`
This thesis:	`http://dsrg.mff.cuni.cz/~kofron/phd-thesis`

Abstract

Formal verification of behavior of a component application requires a suitable specification language. It is necessary that the specification language captures all important aspects of the future implementation with respect to desired properties. Behavior Protocols have been proven to be a suitable component behavior specification platform if one is interested in absence of communication errors.

In this thesis, we (1) propose a new specification language based on Behavior Protocols and (2) address the issue of insufficient performance of BPChecker—a proprietary tool for verification of absence of communication errors in Behavior Protocols. Motivated by issues raised during specification of a real-life-sized case study aiming at providing wireless Internet access at airports, we extended the original Behavior Protocols with support for method parameters, local variables, synchronization of more than two components, and specification of variable-controlled loops. To address the second issue, we propose a method for verification of Behavior Protocols via their transformation to Promela—the input language of the Spin model checker.

Keywords

Software components, behavior specification, model checking, behavior verification, behavior composition

Acknowledgement

I would like to thank all those who supported me in my doctoral study and the work on my thesis. I very appreciate the help and counseling received from my advisor Prof. František Plášil. For the various help they provided me, I also thank my colleagues; a particular thank goes to (in alphabetical order): Jiří Adámek, Petr Hnětynka, Pavel Ježek, Pavel Parízek, Tomáš Poch, and Ondřej Šerý.

My thanks also go to the institutions that provided financial support for my research work. Through my doctoral study, my work was partially supported by the Grant Agency of the Czech Republic projects GD201/05/H014 and 201/06/0770.

Last but not least, I am in debt to my parents and Eddie, whose support and patience made this work possible.

Contents

Chapter 1

Introduction

1.1 Software components

Construction of software applications by assembling reusable pieces together belongs to modern trends of software development. Reusable software pieces, usually referred to as *software components*, from various vendors may be combined to build an application featuring the desired functionality. This approach both speeds up the development process and lowers the development costs. Furthermore, with support of an underlying layer (i.e., a middleware and an operating system), a component application can be executed in a distributed way thus allowing for exploitation of the power of multiple computer if the performance of the application becomes important.

A *component* is a piece of software (implementation) with well-defined functionality and interface providing access to it. Often, a component is viewed as a black box with provided and required parts called *ports* or *interfaces*. Using these parts, components can be connected with each other thus forming an application or a *composite component* providing some more complex functionality.

A *component model* is a set of rules defining abstractions for components and relations between those abstractions. A *component system* is a realization of a component model. From one point of view, there are two groups of component models differing on whether they allow component nesting—*flat component models* (e.g. COM/DCOM [65], Corba Component Model [63], and EJB [67]) disallow component nesting, while *hierarchical component models* (e.g. Darwin [43], Wright [3], SOFA [71], SOFA 2.0 [12, 30], and Fractal [11]) allow for it. In the latter case, the components directly implemented in a programming language (e.g. Java, and C++) are denoted as *primitive components*, while the components created by composing other ones are referred to as *composite components*. The hierarchical component models are more general, as the flat ones can be seen as a special case of hierarchical component models exhibiting no component nesting; thus, in this work, we will focus on hierarchical component models only.

1.2 Verification of software component properties

Verifying various properties of software applications may be important regardless of the fact whether the application is built of components or not. However, there is a special property addressing component applications only—*behavior compliance* [54]. This property describes a compatibility relation between two components. A component being behaviorally compliant to another one can replace this component safely, i.e., the communication of the new component after the update will not yield any communication errors as long as the communication of the original component with other components has not yield any communication errors. In hierarchical component models, supposing that behavior of each component (primitive or composite) is specified, the notion of behavior compliance can be extended in the sense of communication correctness between a composite component and its subcomponents. To assure no communication errors will appear during an execution of a component application, the behavior compliance between all components as well as the compliance between each composite component and its subcomponents should be verified.

When building a component application, properties of particular components have to be formally specified and verified to assure that the component application will not yield errors during execution; this is especially true when combining components from various vendors. Our experience shows that comparison at a syntactic level (e.g. the types of exported interfaces that our bound to each other) is not sufficient; a more thorough specification is actually needed.

As the implementation of a software component is usually too complex to be handled by automated verification tools, a model of the component behavior is needed. Behavior of a software component is typically modeled as a *labeled transition system* (LTS)—a (possibly infinite) graph with nodes representing the states of the software being modeled, and transitions between the nodes labeled by events performed by the component when changing its state. The model becomes an *abstraction* of the component—states and transitions not relevant to verification of properties under consideration (e.g. the behavior compliance) may be omitted to reduce the size of the model. Moreover, in most cases, to keep the verification of properties feasible, we often have to stick with finite state models making the model construction even more difficult.

The main problem of the verification of software components properties (and software in general) is the *state space explosion problem*. This problems denotes an enormous number of states of a model being verified and is usually caused by parallel composition of several software components (or parts of a software application).

1.3 Behavior Protocols

Behavior protocols (BP) are a platform for component behavior specification. They are used in several component models, e.g. SOFA [71], SOFA 2.0 [12, 30], and Fractal [11]. They model the component behavior at the level of abstraction allowing evaluation of behavior compliance.

With each component of an application, a behavior protocol is associated defining the allowed sequences of events that may occur on the component provided and required interfaces. A behavior protocol takes the form of an expression consisting of events (emits and accepts of method calls requests and responses) combined via regular and special operators. It does not contain any notion of data. Hence, BP provide a reasonable level of abstraction able to be handled by tools in a reasonable time.

For evaluation of the compliance relation, the behavior protocols associated with communicating components are combined via a special composition operator *consent* [2]. This operator is basically a parallel composition operator able to capture, besides the traces corresponding to correct communication, also traces containing communication errors. The most important types of errors are *bad activity*, denoting a situation when an emitted event cannot be accepted, and *no activity*, denoting the deadlock. The compliance relation is evaluated in an automated way using a proprietary tool BPChecker [42].

1.4 Problem statement

We used BP for the specification of a component based application aimed at providing access to the Internet at airports [1] consisting of approximately twenty components. We have identified several problems, which can be divided into two main groups:

1. Behavior protocols provide expressive power that is too weak to model several common pattern used in implementation. Moreover, in some cases where the expression power is sufficient, the resulting specification is unreadable and not easy to understand. This is a crucial property of a specification when error fixing takes place.

2. The memory and time requirements of the BPChecker [42] are too high in some cases; therefore, a simplification of the specification have to be done to make the verification feasible. This, of course, lowers the practical applicability of BP.

On the other hand, behavior protocols provide a suitable specification platform if a component-application designer is interested in behavior compliance

1.5 Goals of the thesis

There are two general goals of the thesis reflecting the aforementioned issues:

1. To extend the behavior protocols formalism to be able to model commonly used programming construct in a simple way thus providing an easy-to-use behavior specification platform.

2. To solve the performance issues of the proprietary BPChecker [42] either by (1) employing optimization and other approaches than the ones currently used or (2) using another (model-checking) tool to evaluate the behavior compliance relation.

1.6 Structure of the thesis

The rest of the thesis is structured in the following way: Chapter 2 provides the reader with information about component models considered in this thesis as well as with semantics of process algebras used for component behavior specification. Moreover, several languages aiming at modeling and description of component behavior are discussed. Finally, the tools verifying specification written in these languages are briefly described. Chapter 3 focuses at description of *Extended Behavior Protocols*—a new way of component behavior specification proposed in this thesis. In Chapter 4, we present the details on translation of EBP specification into Promela [32]. Chapter 5 compares the proposed formalism of EBP with original BP and discusses the properties of the proposed specification language. Finally, Chapter 6 concludes the thesis and proposes direction for future research.

1.7 Contributions and publications

The approach to implementation of an algorithm for evaluation of behavior compliance as well as the architecture of BPChecker along with performance comparison of a Python and Java implementations was published in the International Journal of Computer and Information Science, Vol. 6, Number 1 [42].

The experience with modeling a real-life component application being the primary motivation for this thesis was published in Electronic Notes in Theoretical Computer Science, Vol. 160 [34].

Extensions to Behavior Protocols proposed in this thesis were published in Tech. Report No. 2006/2, Dep. of SW Engineering, Charles University [36].

A transformation of behavior protocols to the Promela [32] modeling language and using the Spin model checker [32] for evaluating the behavior compliance relation was described in and published in the proceedings of the SAC'07 conference [38]. Technical details are described in Tech. Report No. 2006/11, Dep. of SW Engineering, Charles University in Prague [37]. This version expects the Behavior Protocols to be deterministic, which is rather restrictive. Therefore, in this thesis, we propose a more general algorithm being able to correctly translate also nondeterministic BPs, i.e., those ones corresponding to a general NFA.

Reviewed papers

[42] M. Mach, F. Plasil, and J. Kofron. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 6(1):22-30, The International Association for Computer and Information Science (ACIS), ISSN: 1525-9293, 2005.

[34] P. Jezek, J. Kofron, and F. Plasil. Model checking of component behavior specification: A real life experience. In *Electronic Notes in Theoretical Computer Science*, volume 160, pages 197–210, Elsevier, ISSN: 1571-0661, 2006.

[38] J. Kofron. Checking software component behavior using Behavior Protocols and Spin. In *Proceedings of Applied Computing 2007*, pages 1513–1517, ACM Press, ISBN: 1-59593-480-4, 2007.

[53] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06), pages 133–141, Los Alamitos, CA, USA, IEEE Computer Society, ISSN: 1550-6215, ISBN: 0-7695-2624-1, 2006.

Technical Reports

[36] J. Kofron. Extending Behavior Protocols With Data and Multisynchronization. Technical Report 2006/10, Dep. of SW Engineering, Charles University in Prague, October 2006.

[37] J. Kofron. Software Component Verification: On Translating Behavior Protocols to Promela. Technical Report 2006/11, Dep. of SW Engineering, Charles University in Prague, October 2006.

Presentations

J. Kofron, J. Adamek, T. Bures, P. Jezek, V. Mencl, P. Parizek, and F. Plasil. Checking Fractal component behavior using Behavior Protocols, presented at the Fractal Workshop (part of ECOOP'06) in Nantes, France, July 2006.

1.8 Note on conventions

The text of this work is partially based on the papers mentioned in the previous section. To denote the parts that were taken from the papers, corresponding paragraphs are marked with a vertical bar:

This is an example of a paragraph that was copied verbatim from a paper, therefore it is marked by a vertical side bar.

In some cases, the leading sentences of parts taken from the papers were slightly modified to fit into the rest of the text, and, due to obvious reasons, the phrase "in this paper" was replaced by the phrase "in this thesis".

Chapter 2

Background

In this chapter, we take a closer look at the component models considered in this thesis. As mentioned in Chapter 1, flat component models (CCM [63], EJB[67]) not allowing component nesting can be treated as a special case of hierarchical component models (Fractal [11], SOFA 2.0 [12]). The hierarchical component models are almost exclusively used in academia, while flat models mostly in industry. We focus on hierarchical component models in this thesis—in particular, we take the SOFA 2.0 and Fractal component models into account. Nonetheless, the results are not limited to these ones, but can be generalized and applied to any hierarchical component models where components communicate synchronously using provided and required interfaces.

2.1 Component models considered

2.1.1 SOFA 2.0

SOFA 2.0 (SOFtware Appliances) [12, 30] is a project providing a developer with a platform for designing and running software component applications. SOFA 2.0 provides a hierarchical component model, i.e., there are both primitive components (implemented in the Java programming language) and composite ones consisting of other components. The components can communicate using their exported—provided (server) and required (client) interfaces.

A *component frame* denotes the boundary of a component, i.e., the set of exported interfaces. There are two views on a component—a *black-box* view and a *grey-box* view. In the black-box view, only the component frame is considered and no internal structure of the component is taken into account, while the grey-box view reflects the frames of first-level-of-nesting subcomponents of the composite component and interface interconnections (ties) between them. This is referred to as the *component architecture*.

There are three kinds of ties between interfaces of distinct components:

- *Binding* connects a required interface of a component to a provided interface of another component. The connected components have to be (1) subcomponents of

the same composite component and both at the same level of nesting or (2) both components at the top level of nesting.

- *Delegation* is a tie between a provided interface of a composite component C_C and a provided interface of one of its subcomponents C_S. The calls on the interface of the C_C component are *delegated* to the interface of the subcomponent C_S.

- *Subsumption* denotes a connection between a required interface of a subcomponent C_S and a required interface of its parent component C_C (the parent component is the composite component C_C, whose subcomponent the component C_S is).

The terms *dynamic architecture* and *architecture reconfiguration* refer to changes of the application architecture at runtime. SOFA 2.0 provides a way to change the structure of the architecture. This is possible via the *factory pattern* creating new components. However, the factory pattern is limited in the following way: Only the component that requests creation of a new component can establish a binding with it. This factory pattern is to be extended in the future.

To be able to reason about compliance of components' behavior, a *behavior protocol* [54, 2] is associated with each component frame. A *behavior protocol* is an expression describing the behavior of a component in terms of sequences of events appearing on the component frame. Via application of the *consent* [2] composition operator onto behavior protocols, it is possible to detect incompatibility between behavior of components. The evaluation of behavior compliance is done automatically using a proprietary tool—BPChecker [42].

2.1.2 Fractal

Fractal component technology [11] provides, as well as SOFA 2.0, a component model with hierarchically nested components. It is similar to the SOFA 2.0 component model in many aspects, therefore we focus on the features not present in SOFA 2.0. In addition to standard (*business*) interfaces providing access to the component functionality, there are *controller interfaces*, shortly *controllers*, allowing for managing the component lifecycle (starting / stopping the component, setting attributes) as well as changing its internal structure in the sense of adding and removing subcomponents and bindings between them. Depending on the type of the component and its execution environment, there may be different count and types of controllers.

A Fractal component is composed of two parts—*controller* (membrane) and *content*. The membrane encapsulates the content and controls the incoming requests processed by the content. All request addressed to the component are queued in a buffer inside the membrane and processed in a FIFO manner.

For each exported interface (provided or required) there is its internal counterpart used for connection of the component with its subcomponent. The internal interface counterpart is of opposite type than the external interface is—for a provided interface, there is a required internal counterpart connected to a provided interface of a subcomponent (in the case of delegation) and vice versa (subsumption).

In addition to *primitive bindings* (connections) appearing also in SOFA 2.0, there are also *composite bindings* in Fractal. Primitive bindings serve for interconnection of two components together, while the composite bindings allow for communication of an arbitrary number of components regardless of the types (provided/required) of their interfaces. A composite binding is realized as a set of *binding components* and primitive bindings. A *binding component*, also called a *connector*, is, however, not a first-class entity of the Fractal component model.

An interface of a Fractal component can be declared as *optional*. An optional interface does not have to be bound to another interface. An interface can be also marked as *multiple* thus declaring an array of interfaces in fact.

Furthermore, the components in Fractal may be *shared*, i.e., several components may share a component as their subcomponent. This eases the reference passing and, generally, management of dynamic applications. On the other hand, however, it complicates the component model by making the architecture of Fractal applications harder to read and understand.

The Fractal specification defines four *levels of conformance*. Each level defines the requirements put on the application that have to be satisfied:

- *Level 0*: This level defines no requirements; thus, every software artifact is a Fractal component.
- *Level 1*: Component on this level has to provide the component introspection, i.e., a mechanism to discover all component interfaces.
- *Level 2*: On this level, additionally to the level 1, a component has to provide interface introspection, i.e., the information about interface cardinality, method names, parameter types, etc. has to be provided.
- *Level 3*: This level extends the level 2 by a type system, in particular by a subtyping relation.

Moreover, for each compliance level x, there is the x.1 sublevel defined requiring that each component provides a standard set of controllers.

Architecture reconfiguration in Fractal is possible using the control interfaces used to change the bindings and add new components (created using the *bootstrap component*) to add/remove components to/from the application architecture.

However, according to some people from the software components community, the architecture have been always treated as not only a set of bindings among particular parts, but also a prescription to which the structure of the application should obey. Allowing any changes of the application structure has to be therefore preceded by definition of changes that are allowed. As there is not, according to our knowledge, a common consensus on what is the set of changes that should be allowed, we omit the issue of dynamic reconfiguration in the rest of the work.

Julia

Julia is one of the Fractal implementation. It is implemented in Java and still being developed. As a result of the project *Component Reliability Extension for Fractal* [1], specification of Fractal components in Julia was extended with an option to specify component behavior using Behavior Protocols [2].

Fractive/ProActive

Fractive [7] is an implementation of the Fractal specification using ProActive [14] middleware for distribution. Features characterizing ProActive are asynchronous method calls, absence of shared memory, and transparency of distribution and migration.

ProActive is a Java implementation of distributed object with asynchronous method calls exploiting *future references*. A *future reference* is a reference to a result of a method call that is not yet ready but will be eventually evaluated; the future reference will be then updated with the result. The ProActive system is composed of several *activities*—active entities. Each activity has defined its entry point—the *active object*, which can be referenced (called) from outside, having its own execution thread. On the other hand, *passive objects* cannot be referenced directly from outside the component and they do not own a thread. To get an idea how a method call is processed, consider the following brief sequence of steps:

1. If a method call is performed on an active object, say $y = Ifc.m(x)$, the request (including a deep copy of all parameters—due to the absence of data sharing) is stored within the queue of the callee and a future reference y is immediately returned to the caller. The future reference is the promise of the asynchronous method call.

2. As soon as the callee decides to serve a request, it picks up the first item of the request queue and executes the requested method.

3. After finishing the method, the future reference previously returned to the caller is replaced with the result (value of y).

In case the caller tries to use the future reference before it has been replaced by the real result, the execution is blocked until the result is ready (*wait-by-necessity*). The ProActive computation model is defined by the ASP calculus [13].

In Fractive, the start and stop methods of the control interface are recursively propagated to subcomponents.

A primitive component in Fractive is composed of one activity, whose object implements the functionality provided by its interfaces. If a primitive component is stopped, the membrane ignores all request targeting the content (implementing the business logic of the component)—it filters such requests out, while processing only the controlling requests. Starting a Fractive component means running the thread of its active object while stopping the components means setting its active flag to false. The stopping of the active object

execution is implemented in a non-preemptive way, i.e., the active object should check the flag and behave accordingly.

In the case of a composite component, the membrane is an active object having its own request queue. Normally, if a component is started, requests from the outer world are propagated to the subcomponents along the bindings and the request from the subcomponents are similarly transfered to the membrane. If a composite component is stopped, it does not emit any functional (i.e., none-control) method calls.

The behavior of a Fractive system is modeled as a set of synchronised transition systems (LTSs). The information about bindings taken from ADL is used for determining the information about synchronization of particular events and lifecycle of each component. The synchronised product of all parts (control part, functional part, behavior of subcomponents) modeling the component behavior is called *controller automaton*. The construction of the controller automata is done in a bottom-up manner through the component hierarchy.

2.2 Modeling component behavior

In this section, we describe several approaches to specification of component behavior used in different component models and aiming at verification of different behavior properties. We are not going to describe all of them, however, we focus on the ones we believe are the most used/important for the rest of this thesis. Moreover, several tools supporting checking properties of the models will be discussed in the second part of this section.

2.2.1 Process Algebras

Process algebras focus on providing a high-level view on modeling of communication among parallel processes. In recent years, this approach was applied several times and several formalisms have evolved. In a process algebra, the basic entity is a process being able to perform various actions thus resulting in another process. Then, a system is described by a set of equation defining the behavior in the sense of observable actions of the particular parts of the system. The best known and most important members of process algebra family are CCS [49], CSP [31], ACP [9], and π-calculus extending the CCS by support for mobile processes. In the following paragraphs, we will briefly describe the most important ones.

CCS

CCS (stands for *Calculus of Communicating Systems*) was developed by Robin Milner around 1980 and published in [49]. It contains only few constructs, whose meaning is defined using operational semantics. In CCS, the basic entity is an *agent* able to perform *actions*. An *action* is an indivisible activity performed by an agent. Furthermore, there is a special action τ called the *silent* or *perfect* action.

Let us now describe the syntax of CCS as stated in [49]. Let $\mathcal{A}$ be a set of *names*, $\overline{\mathcal{A}}$ a set of *co-names*, and $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ set of *labels*. Here, $a, b, c, ...$ range over $\mathcal{A}$, $\overline{a}, \overline{b}, \overline{c}, ...$ range over $\overline{\mathcal{A}}$, and l, l' range over $\mathcal{L}$. Let $Act = \mathcal{L} \cup \{\tau\}$ be the set of *actions*; $\alpha, \beta, ...$ range over Act. We use K, L to denote subsets of $\mathcal{L}$, and $\overline{L}$ to denote the set of complements of labels in L. A *relabeling function* f is a function from $\mathcal{L}$ to $\mathcal{L}$ such that $f(\overline{l}) = \overline{f(l)}$; moreover, $f(\tau) = \tau$.

Further, let $\mathcal{X}$ be a set of *agents variables*, while $\mathcal{K}$ a set of *agent constants*; we let $X, Y, ...$ range over $\mathcal{X}$ and $A, B, ...$ range over $\mathcal{K}$. In some cases, when necessary, we use I and J to denote a set of indices (e.g. $\{E_i :\in I\}$. Finally, let $\mathcal{E}$ be a set of *agent expressions*, and let $E, F, ...$ range over $\mathcal{E}$, $\mathcal{E}$ is then the smallest set including $\mathcal{X}$ and $\mathcal{K}$ containing the following expressions, where E, E_i are already in $\mathcal{E}$:

(1) $\alpha.E$—a *Prefix*
(2) $\sum_{i \in I} E_i$—a *Summation*
(3) $E_1 \mid E_2$—a *Composition*
(4) $E\backslash L$—a *Restriction*
(5) $E[f]$—a *Relabeling*

Of the expressions above, only the expression (2) needs our further attention. It denotes the sum of all expressions $E_i, i \in I$; in the case $I = \{i, j\}$ we can write $E_i + E_j$. In cases when I is understood, the summation can be abbreviated to $\sum_i E_i$. If the set I is empty, the expression $\sum_i E_i$ denotes an inactive agent—an agent unable to perform any actions. As this agent is important, a special name $\mathbf{0}$ was introduced representing this agent; i.e., $\mathbf{0} = \sum_{i \in \emptyset} E_i$.

To decrease the number of parentheses and thus improve the readability of the expressions, a convention of different binding power of combinators was adopted. The order from the tightest to the lowest binding power follows: Restriction, Relabeling, Prefix, Composition, and Summation. To demonstrate this fact, consider the following example:

$$R + a.P \mid b.Q\backslash L \qquad \text{stands for} \qquad R + ((a.P) \mid (b.(Q\backslash L)))$$

The meaning to the language is given using well-known formalism of *labeled transition system*

$$(S, T, \{\stackrel{t}{\rightarrow} : t \in T\}),$$

where S is a set of states, T a set of transition labels, and $\stackrel{t}{\rightarrow} \subseteq S \times S, t \in T$ a transition relation.

In our case, we take S to be ϵ (the agent expressions) and T to be Act (the actions). The semantics is defined via a set of transition rules, which take the following form:

$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E|F \stackrel{\alpha}{\rightarrow} E'|F}$$

This is to express the following:

$$\text{From} \quad E \xrightarrow{\alpha} E' \quad \text{infer} \quad E \mid F \xrightarrow{\alpha} E' \mid F$$

The set of transition rules follows:

$$\textbf{Act} \quad \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Sum}_\textbf{j} \quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j} \ (j \in I)$$

$$\textbf{Com}_\textbf{1} \quad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \textbf{Com}_\textbf{2} \quad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\textbf{Com}_\textbf{3} \quad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\bar{l}} F'}{E|F \xrightarrow{\tau} E'|F'} \qquad \textbf{Res} \quad \frac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L} \ (\alpha, \bar{\alpha} \notin L)$$

$$\textbf{Rel} \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \qquad \textbf{Con} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \ (A \stackrel{def}{=} P)$$

The rule $\textbf{Sum}_\textbf{j}$ can be also expressed in its simpler form if we consider the I set to be finite, which is enough for most practical purposes:

$$\textbf{Sum}_\textbf{1} \quad \frac{E_1 \xrightarrow{\alpha} E'_1}{E_1 + E_2 \xrightarrow{\alpha} E'_1} \qquad \textbf{Sum}_\textbf{2} \quad \frac{E_2 \xrightarrow{\alpha} E'_2}{E_1 + E_2 \xrightarrow{\alpha} E'_2}$$

As we assume there are no other transitions except for those inferable from these rules, we say that the set of rules is complete. Furthermore, using of Restriction and Composition together, the internal communication can be easily modeled.

Let us now describe, how values can be incorporated into expressions inferred from the rules above. First, consider the following agent constants $Prod$ and $Cons$ modeling producer-consumer situation:

$$Prod = \overline{out}(x).Prod$$
$$Cons = in(x).((process.in(y).process) + (in(y).process.process)).Cons$$

Agent $Prod$—Producer—is able to send a value x to its output port $\overline{out}$ after which it becomes agent $Prod$ again (and is able to send further values). Agent $Cons$—Consumer—is able to receive at most two values without processing them. If only a single value is received, it can be processed before the other one is received and processed. After processing both values, it becomes agent $Cons$ again (and is able to receive further values). The behavior of the $Cons$ agent can be also seen as if there would be a buffer able to keep two values at a time.

To capture the meaning of these agent expressions formally in the sense of the definitions above, consider the x and y range over V. $Prod$ and $Cons$ agent constants can be then considered as *families* of constants: i.e., $\overline{out}(x)$ becomes a family $\overline{out}_x$, one for each value $x \in V$, $in(y)$ becomes a family in_y for $y \in V$, etc. Then, the producer agent constant becomes a family of constants:

$Prod = \sum_{x \in V} \overline{out}_x.Prod$

Similar approach may be applied in the case of the consumer agent.

In [49], Milner discusses several equivalence relations based on behavior of agents and their derivation trees. The basic requirement put on the relations is that two agents P and Q should be equivalent if the distinction between them cannot be detected by an external agent interacting with P and Q. Depending on whether the internal action τ is considered as observable or not, the equivalence relation varies. Milner argues that the equivalence relation should not be too restrictive (e.g. to make equivalent agents with isomorphic derivation trees only), on the other hand, equivalence based on the possible sequences of action taken from the automata theory is denoted to be too weak—e.g. agents A and B defined as following:

$$
\begin{array}{llll}
A \stackrel{def}{=} a.A_1 & B \stackrel{def}{=} a.B_1 + a.B_1' & & \\
A_1 \stackrel{def}{=} b.A_a + c.A_3 & B_1 \stackrel{def}{=} b.B_2 & B_1' \stackrel{def}{=} c.B_3 & \\
A_2 \stackrel{def}{=} \mathbf{0} & B_2 \stackrel{def}{=} \mathbf{0} & & \\
A_3 \stackrel{def}{=} d.A & B_3 \stackrel{def}{=} d.B & &
\end{array}
$$

are equivalent in this relation, although we would like them not to be. After performing the action a, the A agent becomes A_1 and is able to perform either action b or action c. The agent B, however, according to something (that is not known nor important for our argumentation) chooses a branch at the beginning and after performing the action a, it is able to perform either the b or the c action, but cannot choose an action at this point any more—the executable action has been already determined in the first step.

To satisfy this feeling of what properties should the equivalence relation have, Milner defines a relation referred to as *strong bisimulation* $\sim$ as follows:

$P \sim Q$ iff, for all $\alpha \in Act$:
(i) Whenever $P \stackrel{\alpha}{\rightarrow} P'$ then, for some Q', $Q \stackrel{\alpha}{\rightarrow} Q'$ and $P' \sim Q'$, and
(ii) Whenever $Q \stackrel{\alpha}{\rightarrow} Q'$ then, for some P', $P \stackrel{\alpha}{\rightarrow} P'$ and $P' \sim Q'$

Further, Milner shows that this relation is also a strong congruence, i.e., it is substitutive under all combinators, and recursive definition:

Let $P_1 \sim P_2$. Then

(1) $\alpha.P_1 \sim \alpha.P_2$
(2) $P_1 + Q \sim P_2 + Q$
(3) $P_1 \mid Q \sim P_2 \mid Q$
(4) $P_1 \backslash L \sim P_2 \backslash L$
(5) $P_1[f] \sim P_2[f]$

π-calculus

The π-calculus [50] is an extension of CCS supporting dynamic reconfiguration of the agents. The reconfiguration means changing the structure of the system dynamically. The information about new structure of (linkage among) agents can be even carried by the communication among agents. As dynamic reconfiguration is not addressed by this thesis, we omit the details about it and refer the reader to e.g. [50].

ACP

A basic issue in theory of concurrency is the modeling of communication. Apart from the basic entities similar to CCS, the *Algebra of Communicating Processes* (ACP) [9] defines the communication in the following way: Let γ is an associative and commutative partial binary function. If $\gamma(a, b) = c$ is defined, in a composition of processes A and B, the process A performing the action a is able to *communicate* with the process B performing the action b resulting in the action c (observed from outside). In CCS [49], the communication can be seen as a special case of this, in particular that $\gamma(a, \overline{a}) = \tau$ is defined for each a. On the contrary, in CSP [31], the communication can be described as $\gamma(a, a) = a$ for all a.

Networks of communicating automata

Networks of communicating automata were introduced by Maurice Nivat in 1979 in [52] at a seminar of the French company *Thomsom-CSF*. The formalism describes communicating processes as interacting finite automata. Each process is modeled as a finite automaton with labels associated with particular transitions.

To talk about communication and synchronization, first, Arnold and Nivat defined *synchronization constraints* in [5] in the following way: Let $A_1, \ldots, A_n$ be alphabets representing actions or events. A *synchronization constraint* is then a subset of the Cartesian product $A_1 \times \ldots \times A_n$.

Next, they propose a notion of *free product of transition systems* being a parallel composition of several finite automata without any constraints.

Finally, they defined *synchronous product* of finite automata as free product, whose global transitions are limited to those allowed by (i.e., contained in) a given synchronization constraint.

Symbolic transition graphs

The formalism of *Symbolic transition graphs* [27] is due to M. Hennessy and H. Lin. They focused on description of interprocess communication where value passing of data of unlimited domains takes place [27]. Although a value-passing version of CCS can be used for description of such situations, the resulting transition systems may be infinite when using infinite data domains; consider Fig. 2.1 as an example. Such transition systems cannot be then processed by tools performing bisimulation checking [49]. Symbolic transition

graphs aim at description of such models using only finite structures to enable automated reasoning about bisimilarity of such processes.

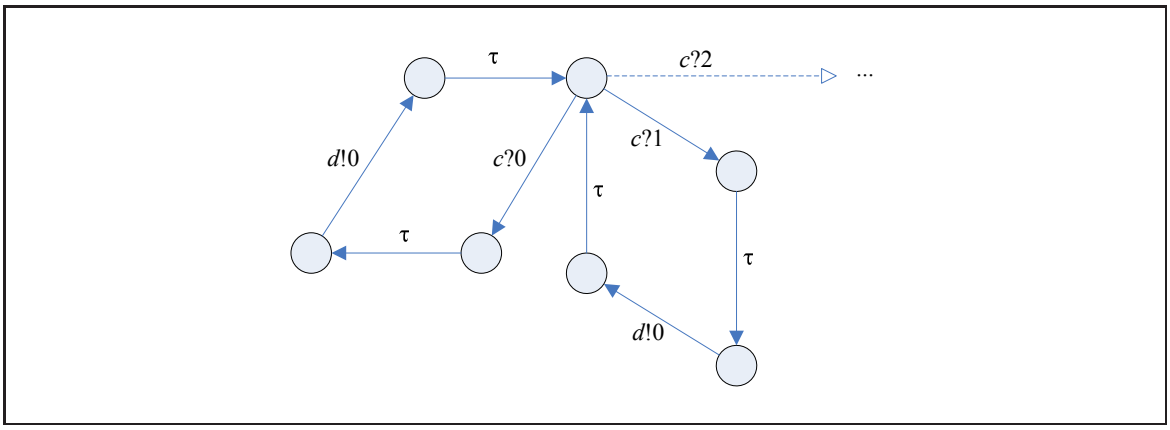


Figure 2.1: A standard transition graph for the CCS process $S = c?x\,.\,\tau\,.\,d!\lfloor x/2\rfloor\,.\,\tau\,.\,S$ where x ranges over natural numbers. Note that the graph is infinite (for each natural number value there is a distinct cycle within the graph).

A symbolic transition graph (STG) is more abstract description of processes than classical LTS; symbolic transition graph uses symbolic actions as the transition labels. As an example of such a description, consider the graph in Fig. 2.2 modeling the same situation as the one in Fig. 2.1. The problem of infinite number of values of variable x is solved by using this variable directly in the transition graph.

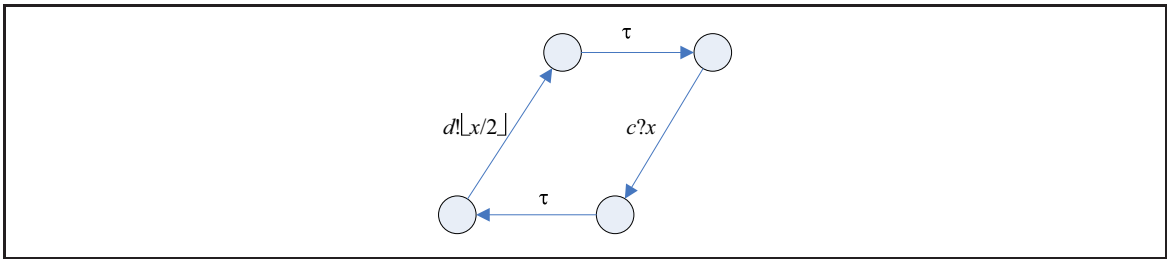


Figure 2.2: A symbolic transition graph for the CCS process $S = c?x\,.\,\tau\,.\,d!\lfloor x/2\rfloor\,.\,\tau\,.\,S$ where x ranges over any arbitrary countable domain.

To describe STG in a formal way, several notations have to be defined first. First, let Var be a countable set of variables, $Var = \{x_0, x_1, \ldots\}$ and V a countable set of values. Let ρ be an evaluation function, i.e., a total function from Var to V. The expression $\rho[v/x]$ denotes the evaluation ρ' differing from ρ only in the mapping of variable x to v. σ denotes a substitution function, while $\sigma[x \mapsto y]$ denotes the substitution differing from σ in an obvious way. The expression $new(W)$ denotes a new variable not present in W, i.e., the variable v_{i+1} where v_i is the last variable (with respect to the ordering of the set Var) in W.

Further, the set of expression Exp, ranged over by e, includes both Var and V. Each expression e has associated a set $fv(e)$ denoting the set of free variables of the expression. Evaluation and substitution behave with respect to fv in an expected manner, i.e. for $e \in Exp : fv(e\sigma) = \sigma(fv(e))$. $BExp$ denotes a set of boolean expressions ranged over by b.

Having the field prepared by the definitions above, Hennessy and Lin define the class of graphs forming the desired set of interest. They are arbitrary directed graphs where each node is labeled by a set of variables—the free variables, and each edge is labeled by a *guarded action*, being a pair of a boolean expression and an action. The action may be either an *input* action, $c?x$, where $c \in Chan$ is a channel, an *output* action, $c!e$, or a *neutral* action from $NAct$, e.g. τ. Let $SyAct$ denotes the set of symbolic actions:

$$SyAct = \{c?x, c!e \mid c \in Chan\} \cup NAct$$

The sets of free and bound variables are defined naturally: $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$, and otherwise both $fv(\alpha)$, $bv(\alpha)$ are empty. The set guarded actions $GuAct$ is the defines as follows:

$$GuAct = \{(b, \alpha) \mid b \in BExp,\ \alpha \in SyAct\}$$

With respect to the facts denotations and definitions above, Hennessy and Lin define the STG in the following way:

A *symbolic transition graph* is a directed graph in which every node n is labeled by a set of variables $fv(n)$ and every edge is labeled by a guarded action such that if a branch labeled by (b, α) goes from node m to n, which we write as $m \xrightarrow{b,\alpha} n$, then $fv(b) \cup fv(\alpha) \subseteq fv(m)$, and $fv(n) \subseteq fv(m) \cup bv(\alpha)$.

Hennessy and Lin proposed definitions for both *early* and *late symbolic operational semantics* where symbolic actions such as $c?x$ and $c!e$ and their residuals are associated with *open terms*. After assigning values to all the free variables, concrete operational semantics is determined which results in a concrete bisimulation equivalence. They also provide algorithm for checking both types (i.e., early and late) of bisimulation equivalence of two processes.

In [40], the formalism of symbolic transition graphs is further extended with *assignments* as parts of transition labels. To explain the motivation behind this, first consider the following process definition:

$$P(x) \stackrel{def}{=} c!x \,.\, P(x+1)$$

Assuming that x is of the integer type, this defines an infinite (countable) set of processes $P(x)$ that cannot be described via a finite STG. The purpose of Lin's work is to enable description of such sets of processes using finite structures. He achieves this goal via extending the transition labels to the form $n \xrightarrow{b, \overline{x}:=\overline{e}, \alpha} n'$; this denotes a transition from the state n to the state n', where if b is evaluated to true, the action α is fired and in n' the free

variable $\overline{x}$ will have the value $\overline{e}$. The author denotes these transition graphs as *symbolic transition graphs with assignment* (STGA). Using STGA, the process in the example above will be associated with a transition graph having only one state and one (cyclic) transition. STG can be viewed as a special case of STGA where the assignment is identity mapping. To reason about similarity of processes, Lin defined bisimulation equivalence between STG and STGA processes for both early and late bisimulation semantics.

2.2.2 Languages

Process algebras provide a suitable semantics for modeling behavior of computational systems. However, to be practically usable, one also needs a suitable way for expressing this semantics. In this section, we present several specification languages aimed at description of communication among computational entities.

Promela

Promela [32] is an acronym for *PROcess MEta-LAnguage*. It was developed around 1980 by G. J. Holzmann. It combines the C programming language with some CSP features. It is much more like a programming language in comparison with process algebras described in Sect. 2.2.1; nonetheless, using it as a programming language usually results in models of enormous size that cannot be verified due to their time and memory requirements.

A Promela model consist of type, variable, channel, and process type declarations.

Type declarations are used for defining user types using keyword `mtype` followed by an explicit enumeration of the new-type members.

Variables can be of a built-in or an user type; the built-in types include integer, short, byte, bit, and boolean; moreover, arrays and records can be used as in common programming languages.

A *process* is the active entity of a Promela model; it is an instance of a *process type*. The *process type* consists of a name, formal parameters, local variable declarations and a sequence of statements called message body. The statements of an instantiated process are executed sequentially and statements of arbitrary processes are interleaved. Furthermore, Promela contains constructs for creating atomic (non-interruptible) sequences that are very useful for decreasing the size of models and allowing implementation of synchronization primitives.

Channels are intended to be used for interprocess communication. With each channel, a message buffer is associated that holds all the not-yet-received messages sent through this channel. The size of the buffer (i.e., the count of messages it can hold at a time) is specified at the beginning and cannot be modified during the computation. The size of the associated buffer can be zero; in such a case, a message can be sent through this channel if and only if there is process waiting for a message from this channel. For each channel, as a part of its declaration, a structure of the messages intended to be sent through this channel is defined; it is usually a tuple of built-in or user types. Unless a channel has declared an exclusive-sender, an arbitrary process may send a message to the channel. Similarly, if a

channel has not its exclusive receiver, any process (even the sending one) may receive the messages from the channel. A Promela model can be executed in two modes regarding the behavior of message channels—in the first mode, a sending statement within a process is blocked in case the buffer (associated with the channel to whoch the process is sending a message) becomes full, while in the second mode, the sending statement does not block but, conversely, the message is lost. Depending on the area on which the model is targeted, the proper mode can be selected.

For illustration consider the following simple Promela code modeling the typical producer-consumer situation:

```
chan c = [2] of {byte, bit};

active proctype producer()
{
    bit parity = 0;
    byte data = 0;

    do
    :: c!data, parity ->
        data++;
        parity++;
        printf("Produce\n");
    od
}

active proctype consumer()
{
    bit parity = 0;
    bit recv_bit;
    byte data;

    do
    :: c?data, recv_bit ->
        assert(recv_bit == parity);
        parity++;
        printf("Consume\n");
    od
}
```

In this model, the `producer` process uses the channel `c` to send messages (numbers 0 - 255) to the `receiver` process. The messages are augmented with a parity bit, whose

value is checked at the receiver side via the `assert` statement. The channel `c` has a buffer associated, whose capacity is 2 messages.

Parallel Assignment Language

The parallel assignment language is the input language of one of the best symbolic model checkers—Symbolic Model Verifier [47]. Using a set of equation, it directly describes a transition system where each state is characterized by values of several variables. The set of equation can be divided into several parts called modules thus modeling several "independent" (up to communication) entities.

To provide an example, we present the following piece of code modeling the same situation as in the Promela example above:

```
MODULE main
VAR
  channel1 : {0, 1, 2, 3, 4, 5};
  channel2 : {0, 1, 2, 3, 4, 5};
  prod : process producer(channel1, channel2);
  cons : process consumer(channel1, channel2);

ASSIGN
  init(channel1) := 0;
  init(channel2) := 0;

SPEC
  -- properties to check expressed in CTL

MODULE producer(chan1, chan2)
VAR
  state : {nothing, moving};
  data : {0, 1, 2, 3, 4, 5};

ASSIGN
  init(data) := 0;
  next(data) :=
    case
      (data = 5) : 1;
      1 : data + 1;
    esac;

  init(state) := nothing;
  next(state) :=
```

```
      case
        ((chan1 = 0) & (!chan2 = 0)) : moving;
        1 : nothing;
      esac;

    next(chan1) :=
      case
        (state = moving) : chan2;
        1 : {data, 0};
      esac;

    next(chan2) :=
      case
        (state = moving) : 0;
        ((state = nothing) & (!chan1 = 0)) : {data, 0};
      esac;

  MODULE consumer(chan1, chan2)
  ASSIGN
    next(chan1) := 0;
    next(chan2) := {0, chan2};
```

In this model, three *modules* are defined: `producer`, `consumer`, and `main`. The `producer` and `consumer` modules are instantiated in the `main` module via the `process` statement; defined this way, in each step, a module is nondeterministically chosen for execution. With each module, a set of variable is associated—in the case of producer, `data` and `state`. Furthermore, the modules can access variables provided as parameters during instantiation (`chan1` and `chan2`). Initial values of variables (not parameters, of course) are determined by the `init` statement. The execution of the entire model is divided into steps executed atomically. Within a step, only one module is executed. The execution inheres in assigning new values to the variables according to the equation defined by the `next` statement.

In our example, the `producer` process is responsible for consistency of the buffer—i.e., it avoids the state that there are some data at position 2 (`channel2`) and none at position 1 (`channel1`). As the SMV input language does not employ any numerical types as *integer* or *byte*, to model similar data domains as in e.g. Promela, we have to define them explicitly—the `data` and `channelx` variables in this model—which may become inconvenient in some cases.

Although the aforementioned modeling languages all succeeded in the task of modeling behavior, none of them focuses on software components. Even though almost either can be used for specification of software component behavior, there is no direct support for expressing or verification of behavior compliance (Sect. 1)—if verification of this property

is needed, the application designer has to be aware of this fact from the very beginning; a try to achieve this in Promela has led into a hard-to-read and large model.

A specification language aimed at behavior specification of software components needs to be built upon primitives forming a suitable level of abstraction both straightly usable and easily readable and maintainable. Depending on the properties the designer is interested in, the primitives may differ a lot—from byte-code instruction through method calls to e.g. sending messages or taking some high level actions.

Wright

Wright [3] is an architecture description language (ADL) developed by Robert Allen and David Garlan at Carnegie Mellon University, USA in 1997. It aims at description of the architecture of a component application. Wright introduces two basic abstractions—a *component* and a *connector*. Components are entities that are connected (communicates) using connectors. A component is defined by a *component type* that provides and requests ports (communication points). A connector is similarly described by a *connector type* that is defined by a set of *roles* and a *glue* specification. While instantiating a component system, bindings between components' ports and connectors' roles are declared thus connecting the parts together. As an example, consider the following skeleton of an ADL specification describing a simple client-server system (the example was taken from [3]):

```
System SimpleExample

  component Server =
    port provide [provide protocol]
    spec [Server specification]

  component Client =
    port request [request protocol]
    spec [Client specification]


  connector C-S-connector =
    role client [client protocol]
    role server [server protocol]
    glue [glue protocol]

  Instances
    s: Server
    c: Client
    cs: C-S-connector
```

```
    Attachments
      s.provide as cs.server
      c.request as cs.client

  end SimpleExample.
```

The behavior is described using the *interacting protocols*—a subset of CSP [31]. From the large set of constructs provided by CSP [31], only few of them are allowed in Wright. Besides processes and events the following constructs are included in Wright:

- **Prefixing:** The notation $e \rightarrow P$ denotes a process that can perform the event e and then behaves as P.
- **Alternative:** $P \,\Box\, Q$ denotes a process that behaves as P or Q, where the choice is made by the "environment", i.e., by a process interacting with $P \,\Box\, Q$. This is also referred to as the *external choice*.
- **Decision:** $P \sqcap Q$ denotes a process that behaves as P or Q, but the choice is made by the process itself. This is also denoted as the *internal choice*.
- **Named processes:** A process name can be associated with a process expression, however, unlike CSP, Wright does not allow an infinite number of processes.
- **Parallel composition:** The notation $P \parallel Q$ denotes a process that behaves in the following way: It can perform events lying in the alphabet of either P or Q, however, the events lying in the intersection of the alphabets can be performed only if both processes can perform the event. This operator is not used in behavior specification of processes nor ports, however, it is used when combining their behaviors.

Furthermore, there are three special terms: $STOP$ denotes a process unable to perform any event, $\surd$ denotes the "success" event, and § represents successfully terminating process, i.e, $\S \stackrel{def}{=} \surd \rightarrow STOP$.

Next, process-scope expressions can be defined: **let** $Q = expr$ **in** R defines a process Q that behaves like *expr* in the scope of R.

Finally, labeling of events and processes is provided; the event e labeled with l is denoted by $l.e$. The operator ":" is used to label all of the process events: $l : P$. Then, Σ represents the set of all unlabeled events.

Formally, in CSP, a process P is defined as a triple (A, F, D), where A is the alphabet of P, F is a set of "failures", and D is a set of "divergences". The set of failures is a set of pairs, each pair is formed by a trace and a set of events the process can "refuse" to participate in after executing this trace. The divergences are the set of traces of P, after execution of which the process can exhibit any arbitrary behavior (i.e., perform any events).

As an example, consider the full specification [3] of the *C-S-connector* from the aforementioned example:

connector C-S-connector =
  role Client = (request!x → result?y → Client) ⊓ §
  role Server = (invoke?x → return!y → Server) □ §
  glue = (Client.request?x → Server.invoke!x → Server.return?y → Client.result!y
    → **glue**) □ §

This declaration defines the expected behavior on both server and client sides (the **role** statements) as well as the way these behaviors should be combined (the **glue** statement). Let us now describe this specification in more detail.

The communication behavior of the client is defined as a process that first *request*s a service and then obtains a *result*. Since the internal choice operator is used, it is up to the client process to decide whether to emit a *request* or to terminate successfully (§).

The communication behavior of the server can be denoted as "dual"—first, a request is *invoke*d on the server, after which a computed value is *return*ed. Unlike the previous case, the definition of server behavior takes advantage of the external choice operator thus modeling the fact that the server should offer its service as long as its environment (a client connected through a glue) uses it.

The glue combines the server and client behaviors together—first a *request* with a value x is accepted from a client, which is used to *invoke* the server. After that, the server *return*s a value y that is as a *result* sent to the client. The entire sequence of events may be performed again because of the use of recursion. Again, the external choice operator says us that the glue will not decide on termination on its own, but this decision is left to its environment (again, a client connected to this glue).

Having informally explained the simple example above, we are ready to define the connector description formally.

The *meaning of a connector description* with roles $R_1, R_2, \ldots, R_n$ and glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1 : R_1 \parallel \mathrm{R}_2 : R_2 \parallel \ldots \parallel \mathrm{R}_\mathrm{n} : R_n)$$

where R_i is the name of role R_i, and the alphabet of *Glue* is:

$$\alpha Glue = \bigcup_i (\mathrm{R}_\mathrm{i} : \Sigma) \cup \{\surd\}.$$

Similarly, the behavior of a port can be also described as an protocol:

component DataUser =
  port DataRead = get → DataRead ⊓ §

After associating with roles, port protocols take the place of the role protocols in resulting system. The main reason for separation of ports and roles is enabling of connector reuse in a wider field of cases. Putting the things down like this, indeed, the question "when is a port *compatible* with a role?" arises.

Wright defines the *compatibility* between ports and roles in the following way: A port is compatible with a role if its process is substitutable for the role process, i.e, the rest of the connector is not able to detect such a replacement. In CSP, this notion is formally captured by the *refinement relationship*—a process P is *refined by* a process Q, written $P \sqsubseteq Q$, if the following three conditions are satisfied:

1. alphabets of P and Q are the same,
2. the set of failures of P is a superset of the failures of Q, and
3. the set of divergences of P is a superset of the divergences of Q.

This definition is actually too restrictive for practical purposes for two reasons: First, the alphabet of a role process differs in most cases from the alphabet of a port process. Second, from the methodological point of view, we want to make a port able to fill as broad set of roles as possible. In some cases, a port and a role having the same alphabet are incompatible, because an incompatible behavior is possible in general, but would never arise in the context in which the port is used. Thus, in Wright, the compatibility relation is based on *traces described by the role*. For more details, we refer the reader to [3].

Now, we are ready to present how the compatibility relation can be used in practice in Wright. A situation we want to avoid in a system composed of parts is that some parts are waiting for interaction but no part is able/wants to perform it. This situation is denoted as *deadlock*. However, usually, we want to allow all the parts (and glue) to agree on success, i.e., end up with the $\surd$ event. Furthermore, the authors define the conditions under which two components can be considered as compatible; the deadlock-freedom is preserved after replacing a port with a compatible one.

As to automatic compatibility checking, the authors use FDR [68], a commercial tool for checking of refinement conditions for finite CSP processes. As the FDR tool accepts CSP processes as input, the authors of Wright provide a tool translating specifications in Wright into the CSP [31] language.

Wright does not support dynamic reconfiguration (e.g. adding a new process) of the system architecture nor passing process names via messages; however, a dynamic update of a component is supported through the compatibility checks of the new component and the role to which it should be attached.

Darwin (Tracta)

Darwin [43] is another language used for specification of hierarchical component-based systems. It is a general-purpose declarative language with support for description of dynamic structures evolving during the execution. The basic primitives upon which the semantics of Darwin is built are components and services.

The components in Darwin are viewed as basic building blocks both providing and requiring services. A component is defined in a *context independent* way, i.e., regardless of other components (its environment) with which the component is going to interact.

This simplifies both reuse of the component and replacement with another one during maintenance. The basic purpose of the Darwin language is to describe composition of components (i.e. how instances of various types are connected together) in a declarative way resulting in composite components that can be composed again. As an example of specification of composite component consider the following example:

```
component pipeline(int n) {
  provide output;
  require input;

  array F[n]: filter;
  forall k:0..n-1 {
    inst F[k] @ k+1;
    when k < n-1;
      bind F[k+1].input -- F[k].output;
  }
  bind
    F[0].input -- input;
    output -- F[n-1].output;
}
```

A component defined this way is a pipeline, where the number of subcomponents is passed as an argument `n`. Output of each but the last subcomponent of type `filter` is bound to the input of the following subcomponent (`bind F[k+1].input -- F[k].output`)[1]. The input of the first subcomponent is bound to the input of the composite component (`F[0].input -- input`); similarly the output of the last subcomponent is bound to the output of the composite component (`output -- F[n-1].output`). This way, architecture of composite components is defined.

To reason about evolving architectures, Darwin uses the π-calculus. The π-calculus is a process algebra built upon the Milner's CCS [49] extending it by support for mobile agents—thus, dynamic reconfiguration of a running system can be described. The authors of Darwin have chosen the simple monadic form of the calculus. The system is modeled as a collection of independent processes communicating via channels. Channels are referred to by name. Processes are built from the names via application of the following rules:

[1]Note that this is possible due to the declarative nature of Darwin.

action terms ::=	$\overline{x}z.P$	Output the name z along the link named x; then execute process P.
	$x(y).P$	Input a name, call it y, along the link x and then execute P (binds all free occurrences of y in P).
terms::=	$A_1 + \ldots + A_n$	Alternative of actions $n \geq 0$, execute one of A_i.
	$P_1 \mid P_2$	Composition—P_1 and P_2 are executed concurrently.
	$(\nu\, y)P$	Restriction—introduces a new name y with scope P (binds all free occurrences of y in P).
	$!P$	Replication—provide any number of copies of P. It satisfied the equation $!P = P \mid !P$.

Computation in the π-calculus is then expressed by the following reduction rule:

$$(\ldots + x(y).P_1 \ldots) \mid (\ldots + \overline{x}z.P_2 + \ldots) \rightarrow P_1\{z/y\} \mid P_2$$

Sending z along channel x reduces the left hand side to $P_1 \mid P_2$ and replaces all free occurrences of y in P_1 with z.

A declaration of a provided service *provide* p is then modeled as the agent $\mathrm{Prov}(p, s) \stackrel{def}{=} !(p(x).\overline{x}s)$, where s is a reference to the service provided by the component that has to be implemented, x is a location at which s is required, and p is the access name. Note that the use of ! at the beginning of the expression assures availability for several clients. Similarly, a required service *require* r is modeled as $\mathrm{Req}(r, l) \stackrel{def}{=} r(y).\overline{y}l$, where l is a location of the service provision, y is the name of the service provider, and r is the access name. Finally, a binding *bind* $r - p$ is modeled as $\mathrm{Bind}(r, p) \stackrel{def}{=} \overline{r}p$. The result of composition is $\overline{l}s \mid \mathrm{Prov}(p, s)$—the name of the service s is sent to the place l where it is required.

In Darwin, the behavioral description is provided using the Tracta [26] approach. Tracta is based on the formalism of Labeled Transition Systems (LTS) with specifications expressed in FSP [44] (finite state processes). This way, a specification is provided for each primitive component described in Darwin; behavior of a composite component is then derived from the behavior of its subcomponents by application of parallel composition on particular LTSs. This composition is defined by the following set of derivation rules:

$$\frac{P \stackrel{a}{\rightarrow} P'}{P \| Q \stackrel{a}{\rightarrow} P' \| Q}\ a \notin \alpha Q \qquad \frac{Q \stackrel{a}{\rightarrow} Q'}{P \| Q \stackrel{a}{\rightarrow} P \| Q'}\ a \notin \alpha P \qquad \frac{P \stackrel{a}{\rightarrow} P' \quad Q \stackrel{a}{\rightarrow} Q'}{P \| Q \stackrel{a}{\rightarrow} P' \| Q'}\ a \neq \tau$$

where αX denotes the alphabet of the process X. The order in which the component are composed is not important as the composition operator $\|$ is both associative and commutative. The composed components synchronize on shared actions; since the actions are not "internalized" (transformed to τ), more than two components may be synchronized on an action. The private (i.e., not shared) actions from various components are interleaved

in the same way as in common parallel composition. Sometimes, however, it is convenient to hide "internal" actions (those not taking part in external communication) from being visible at a higher level of component composition. Therefore, Tracta defines *hiding* and *relabeling* operators similar to those in CCS [49]. On each level of component composition, the actions on bindings are relabeled and "internalized" in order to be hidden for higher composition levels—the LTS describing behavior of a composite component is minimized with respect to *weak semantic equivalence* defined in [49].

Tracta supports verification of both *safety* and *liveness* properties. Safety properties are expressed as deterministic LTS without τ actions modeling the expected behavior. A component system S satisfies a property P if:

$$\text{traces}(S)\backslash\alpha P \subseteq \text{traces}(P)$$

Informally, the behavior (all traces) of a component restricted to the actions contained in the alphabet of P has to be also included in P.

As to the liveness properties, these are specified using *Büchi automata*. To cope with the distinction between LTS (no information within particular states available) and Büchi automata (information about accepting states stored within states), Büchi automata are extended with special transitions from accepting states. There are several restrictions put on the Büchi automaton B describing a liveness property:

- B has to be deterministic,
- B has to be complete, i.e., at each state there is a transition for each $a \in \alpha B$, and
- the choices taken in the system S are assumed to be fair.

Again, the system S satisfies the property modeled by B if the automaton B accepts all infinite executions of the system S.

To verify Tracta properties, the tool *Labeled Transition Systems Analyser* (LTSA) [44] can be used.

LOTOS

LOTOS (Language of Temporal Ordering Specification[2]) [70] is one of the FDT (Formal Definition techniques); it was developed within the International Standards Organization (ISO) during the years 1981-1986.

LOTOS aims at description of a system viewed as a *hierarchy of processes*. A process is an active entity that may perform both *external* (observable) and *internal* (hidden) *actions* (atomic interactions, events); an external action may be a subject to interprocess communication. Each external action is thought to appear at a *gate*—an interaction point. When describing behavior of a process, the other processes (possibly interacting with this process) are referred to as its *environment*. Additionally to the process and its environment,

[2]Despite its name, LOTOS has nothing to do with temporal logic—it is based on the formalism of process algebras.

there is a special process *observer*, which can always consume any external action the rest of the system may perform, and does not exhibit any further external nor internal activity.

LOTOS specification has the following syntax assuming that B, $B1$, and $B2$ are behavior expressions:

(1)	**stop**	inaction
(2)	$\mathbf{i}; B$	internal action
(3)	$g; B$	external action
(4)	$B1[]B2$	choice
(5.1)	$B1 \mid [g_1, \ldots, g_n] \mid B2$	general parallel composition
(5.2)	$B1 \mid\mid\mid B2$	pure interleaving
(5.3)	$B1 \mid\mid B2$	full synchronization
(6)	**hide** $g_1, \ldots, g_n$ **in** B	hiding
(7)	$p\,[g_1, \ldots, g_n]$	instantiation of a process
(8)	**exit**	successful termination
(9)	$B1 >> B2$	sequential composition
(10)	$B1\,[> B2$	disabling

The **stop** process denotes a process that is not able to perform any action; in some process algebra (e.g. ACP), such a process is denoted as the *deadlock* process.

The internal action **i** is equivalent to the τ internal event from process algebras.

The expression $B1[]B2$ denotes a process that is able to behave as $B1$ or as $B2$. The choice is made according to the process environment—if a process within the environment is able to perform the initial action of $B1$, then $B1$ is chosen to be executed; similarly, indeed, for $B2$. If there is an action of the environment common to both $B1$ and $B2$, one of them is nondeterministically chosen for execution.

$B1 \mid [S] \mid B2$ denotes a process composed of expressions $B1$, $B2$ synchronized on the set of gates S common to both $B1$ and $B2$; that is, the process may perform either an action at a gate in S (both $B1$ and $B2$ perform this action) or an action at a gate not in S that may be performed either by $B1$ or $B2$. In other words, if one of $B1$ and $B2$ is able to perform an action at a gate in S, it has to wait for the other one until the other one will be also able to perform the same action.

$B1 \mid\mid B2$ is equivalent to $B1 \mid [S] \mid B2$ where S is the set of all gates common to $B1$ and $B2$ while $B1 \mid\mid\mid B2$ is equivalent to $B1 \mid [S] \mid B2$ with S being the empty set.

The hiding operator **hide** $g_1, \ldots, g_n$ **in** B is used to "internalize" the actions g_i in B, that is, it converts the g_i actions in B into the internal action **i**.

Process p can be instantiated using the parameters $g_1, \ldots, g_n$ via the expression $p\,[g_1, \ldots, g_n]$; recursion can be achieved via instantiation of a process within its own behavior expression.

The expression **exit** denotes a nullary operator used for successful termination of a process. After this termination, the process becomes the dead process **stop**.

In sequential composition $B1 >> B2$, $B2$ is enabled, i.e., executed, only after successful termination of the process $B1$.

Disabling $B1 [> B2$ denotes behavior where $B1$ is executed as long as the initial action of $B2$ is not allowed to be executed; if it becomes to be executable, the execution of $B1$ is interrupted and the control is transfered to $B2$. If the initial action of $B2$ is not executable before $B1$ termination, $B2$ is disabled and never executed.

The LOTOS language exists in two variants—*basic LOTOS*, whose syntax and semantics have been just described, and *full LOTOS* (or simply LOTOS), which is an extension of basic LOTOS adding the ability of data representation. Unlike in basic LOTOS where actions and gates, at which the actions happen, coincide, in full LOTOS, each action has the form $g < v_1, \ldots, v_n >$ where g is a gate and v_i are values. The values are of *abstract data types*; the data types are based on ACT ONE [19]—a specification language for abstract data types.

Processes in full LOTOS can be parameterized not only by formal gates, but also via a parameter list declaring new variables. As an example of a full LOTOS specification, consider the following prescription taken from [70]:

```
process compare[in, out] (min, max: int) : noexit :=
  in ?x:int;
  (    [min < x < max] --> out !x; compare [in, out] (min, max)
    [] [x <= min] --> out !min; compare [in, out] (x, max)
    [] [x >= max] --> out !max; compare [in, out] (min, x)
  )
endproc
```

This process models a filter parametrized by two values `min` and `max`. It accepts a value `x` at the gate `in` and in case the value is between `min` and `max`, the value of `x` is sent to the gate `out` and the filter continues working with the same parameter as before. If `x` is less than `min`, `min` is sent to the output gate `out` and the filter lower limit is set to `x`. Similarly with the upper limit. The keyword `noexit` expresses that this process is intended to never successfully stop.

Parametrized contracts

Design-by-contract is a specification technique for software defined by Bertrand Meyer e.g. in [48]. A contract between a client and a supplier of e.g. a service is composed of two obligations:

(i) The client has to satisfy the supplier's precondition and
(ii) the supplier has to fulfill its postcondition if its precondition has been satisfied by the client.

Taking into account software components communicating through their provided and required interfaces, we can look at the required interfaces of a component as at its requirements, i.e., as the precondition of the supplier, while the provided interfaces can be viewed as its postcondition. Ralf Reussner et al. described this approach in e.g. in [55].

The concept of parametrized contracts [55] exploits the fact that even though a given environment E of a component C (i.e., the set of components communicating with the component C) does not satisfy the precondition of the component C, i.e., not all required interfaces of the component C are bound, the component may still provide a reasonable subset of its functionality. This is especially true in cases of composite components offering a service with a lot of variations (e.g. the DHCP server in [1]).

A *parametrized contract* is a mapping $p : 2^P \rightarrow 2^R$ where P is the set of provided interfaces and R is the set of required interfaces. Informally, for each subset S_P of provided interfaces, the contract p defines a set of required interfaces necessary to be bound in order that the component will be able to provide the functionality of S_P. Similarly, the inverse mapping $p^{-1} : 2^R \rightarrow 2^P$ makes sense and, then, for each subset S_R of required interfaces of a component being satisfied by (bound to) the environment we get the set of provided interfaces of the component that can be used in this particular environment. The contract p is denoted as the *provides-parameterized contract* while the contract p^{-1} as the *requires-parameterized contract*.

To reason about behavior of a component, the authors use *component protocols* — description of valid sequences of calls to services supported by the component. With each provided interface, a *provides protocol* is associated, while a *requires protocol* defines the valid sequences of each required interface. A protocol is modeled as a finite state machine (P-FSM and R-FSM for provides and requires protocols, respectively). Further, each method s provided by the component is associated with a finite state machine *SE-FSM*$_s$ (Service Effect FSM) which describes all possible sequences of calls to other methods when the method s is called.

Now, each edge (transition) of a P-FSM corresponding to a method s can be substituted with the SE-FSM$_s$ resulting in a FSM containing all the SE-FSMs in the order they can be called by a client after a provided method is called. If the substitution is marked within the resulting FSM, we can obtain the original P-FSM by removing the substituted parts. This way, a set of provided interfaces that can be used in a given environment can be computed.

Provides-parameterized contracts are to be used when designing a new system. The system designer selects components providing the desired functionality and using their provides-parameterized contracts, he/she computes their requirements. Note here that not the entire functionality of the selected components may be needed resulting in weaker requirements of the selected components.

When inserting a new component into an existing system either due to an update (or component replacement) or extending the current functionality of the system, requires-parameterized contracts are to be used. In these situations, we can ask whether the requirements of the update component are not higher than those provided by its environment or what functionality will be provided by the newly inserted component in its environment.

The concept of parametrized contracts is general and, if extended, it can be used for predicting reliability of component applications, which is of major interest in many cases. Informally, the reliability of a component means the probability of returning correct results after a method of the component is invoked. As there are usually several methods (services)

provided by a component, the reliability of the component depends on the *usage profile*, i.e., on the frequency particular methods are called. To capture this fact, the authors use Markov chains [61]. The P-FSM is extended with the information about probabilities that particular methods are called (*the usage profile*). Given that the reliabilities of the methods required by a component are known, it is possible to compute the reliability of particular methods provided by the component and the overall reliability of the component under a given usage profile.

To illustrate this specification technique, consider the P-FSM in Fig. 2.3 taken from [55].

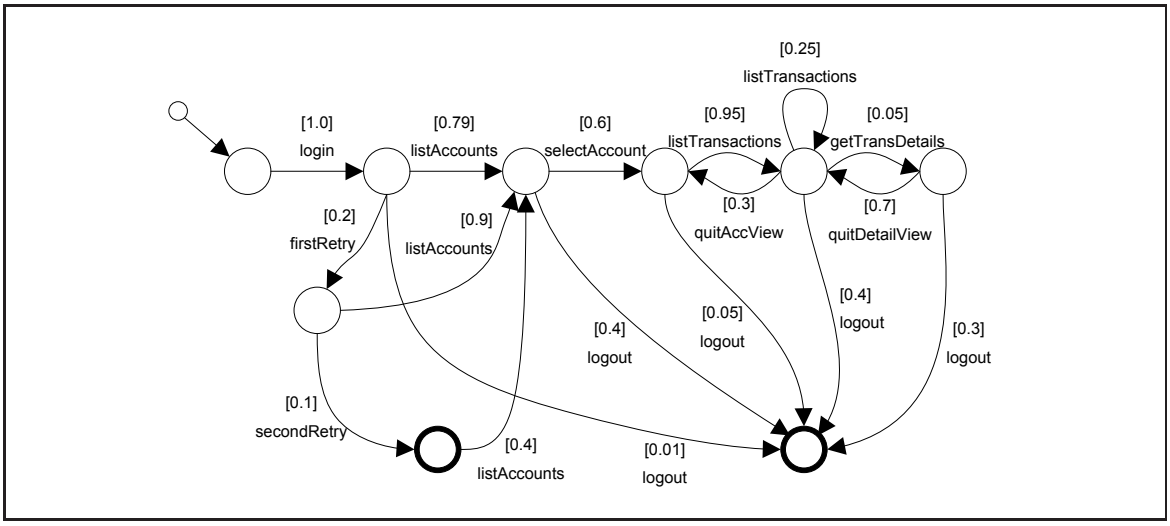


Figure 2.3: Example of P-FSM with a usage profile.

The P-FSM models behavior of an *OnlineAccountManager* component; the component is able to accept several sequences of method calls, e.g. *login, listAccounts, logout*. The final states, i.e., the states denoting successful completing of the service, are denoted by circles with a thick border. The transitions are labeled not only with method names, but also with a number denoting the probability that a particular transition is taken. The sum of the probabilities associated with the transitions leading from each state has to be equal to one (except for the final states where the probability sum has to be less or equal to one).

In case of a composite component, besides the usage profile, information about reliability of particular ties (i.e., mappings and bindings) between interfaces of components is needed to reason about reliability of the composite component. Details on evaluation of the component reliability can be found in [55].

The idea of parametrized contracts is used and further extended in the *Palladio Component Model* [8]. Here, SE-FSM is extended by information about loop iteration numbers, resource usage, and parameter dependencies to allow more accurate performance prediction.

Accepting the fact, that the usage profile of a component is known, the component developer is not able to provide information about performance of the component as whole in the sense of constant values. However, he/she may be able to provide such information

as a function of parameters passed to component methods (provided services). Since it is sometimes impossible to state e.g. the exact number of interations of a loop even as a function of a parameter, Palladio uses *random variables* and provides also some basic operation upon values of random parameters. As an example, consider the diagram in Fig. 2.4 taken from [8], where a *Resource Demanding Service-Effect Specification* of a shipping service of the online-store component is depicted. The shipping service calls another service which depends on the order cost—if the order cost is below 100EUR, full shipping fee is charged, if the cost is 100-200EUR, reduced fee is used, while the orders above 200EUR are shipped free of charge. Given a usage profile, i.e., the distribution function of orders' costs, we can deduce the probabilities of particular branches. Similarly, number of loop iterations and parameter dependencies are modeled.

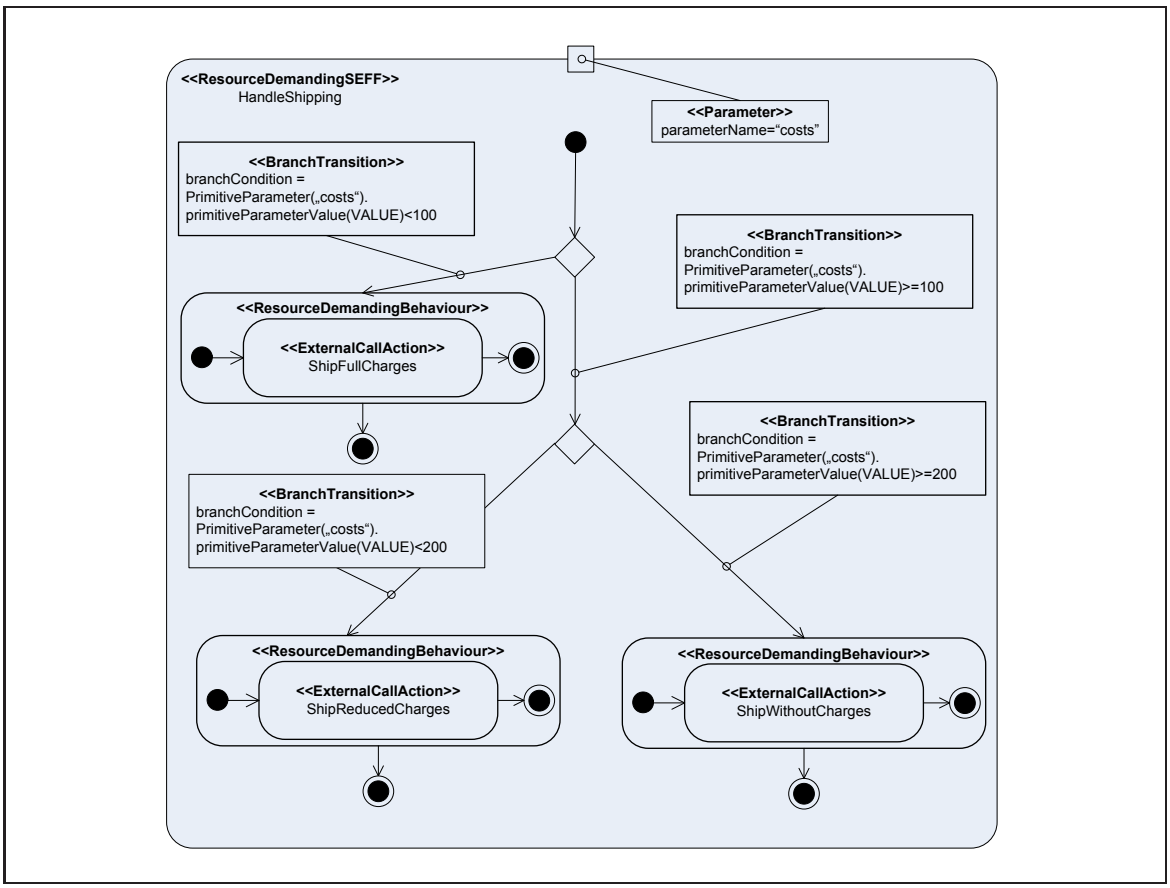


Figure 2.4: Example of branch conditions.

As to the limitations of the model, Palladio expects the application architecture to be static, it does not support dynamic architecture reconfiguration. Regarding the modeling of data, Palladio does not model the return values of methods, which may be also a factor influencing behavior (and consequently performance of the application, of course). The component model has also a limited support for concurrency, since the performance of an application run on a multiprocessor has hard to predict.

Component-interaction automata

Component-interaction automata (CI automata) introduced by Brim et al. in [10] represent an approach to specification and verification process of component-based systems. CI automata are a specification language for modeling behavior of components in (hierarchical) component models. The actions of the language are not a priori associated with any kind of events as well as the composition of particular specifications is not predefined. This way, the authors aim at applicability on a wide spectrum of component models.

Each component in a system is associated with an CI automaton. The components are supposed to communicate with each other only through its interfaces (services, messages, etc.). The communication is possible if two communicating components are able to perform an action of the same name. Although only two components may synchronize their execution in this way (based on the client-server principle), more sophisticated synchronization can be achieved using connectors. The primitive components in a system are identified with natural numbers like (1) and (((1))), while the denotation of a composite may look as $(((1)), (2))$ and $(2, 1)$.

A *hierarchy of component names* is then an n-tuple $H = (H_1, \ldots, H_n)$ where H_i are either (pairwise distinct) component names (i.e. natural numbers) or also hierarchies of component names. A set of hierarchies of component names is denoted by $\mathfrak{H}$, while S_H denotes the set of component names corresponding to H.

A *component-interaction automaton* is a 5-tuple $C = (Q, Act, \delta, I, H)$ where

- Q is a finite set of states,
- Act is a finite set of actions where the set of labels $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\}))\backslash(\{-\} \times Act \times \{-\})$,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of labeled transitions,
- $I \subseteq Q$ is a nonempty set of initial states, and
- $H \in \mathfrak{H}$ is a hierarchy of component names.

The particular parts of the labels have the following meaning: given a label (X, A, Y), X denotes a name of the component that outputs the action A, while Y denotes a name of the component that inputs the action A. Given a CI automaton C, a set of paths, i.e. finite and infinite alternating sequences of states and transitions starting and ending with a state, is denoted by $Path(C)$. Although CI automata address synchronization of (communication between) only two components, the formalism can be extended to *Multi CI automata* with labels taking the form (C, A, D), where C and D are sets of components.

A set of CI automata can be composed thus forming a CI automaton modeling behavior of this set. To do so, the automata have to be *composable*, i.e., they have to be denoted with distinct natural numbers.

Now, we are ready to describe the composition of CI automata. Let $I = \{i_1, \ldots, i_n\}$ be a nonempty set of natural numbers and Q_i for each i be a set. Then $\Pi_{i \in I} Q_i$ denotes the set $\{(q_{i_1}, \ldots, q_{i_n}) \mid \forall j \in \{1, \ldots, n\} : q_{i_j} \in Q_j\}$. The function $proj_j : \Pi_{i \in I} Q_i \to Q_j$ returns the j-th item of the tuple. Let $S = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i \in I}$ be a composable

set of CI automata. The complete transition function for S is defined by the relation $\Delta_S = \Delta_{S,old} \cup \Delta_{S,new}$ where

- $\Delta_{S,old} = \{(q, x, q') \mid q, q' \in \Pi_{i \in I} Q_i, \exists j \in I : [(proj_j(q), x, proj_j(q')) \in \delta_j \wedge \forall i \in I \backslash \{j\} : proj_i(q) = proj_i(q')]\}$
- $\Delta_{S,new} = \{(q, (n_1, a, n_2), q') \mid q, q' \in \Pi_{i \in I} Q_i \wedge \exists j_1, j_2 \in I, j_1 \neq j_2 : [(proj_{j_1}(q), (n_1, a, -), proj_{j_i}(q')) \in \delta_{j_1} \wedge (proj_{j_2}(q), (-, a, n_2), proj_{j_2}(q')) \in \delta_{j_2} \wedge \forall i \in I \backslash \{j_1, j_2\} : proj_i(q) = proj_i(q')]\}$

The preceding definition is a rather general; it defines cartesian product of the automaton with possible (but not required) pairwise synchronization of automata on the same actions. In practice, however, a more restrictive composition that e.g. hides internal actions may be desired. Therefore, the authors define an unary composition operator restricting the transition relation in the following way: Let T be a set of transitions. Then $\otimes_T$ denotes a unary composition operator applicable on a composable set of CI automata such that if $S = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i \in I}$ is a composable set of CI automata, then:

$$\otimes_T S = (\Pi_{i \in I} Q_i, \cup_{i \in I} Act_i, \Delta_S \cap T, \Pi_{i \in I} I_i, (H_i)_{i \in I}).$$

The automaton $\otimes_T S$ is again a CI automaton and thus can be member of another set of CI automata that are composed using the $\otimes_{T'}$ operator for some T'. For illustration, consider the component application (taken from [15]) depicted in Fig. 2.5.

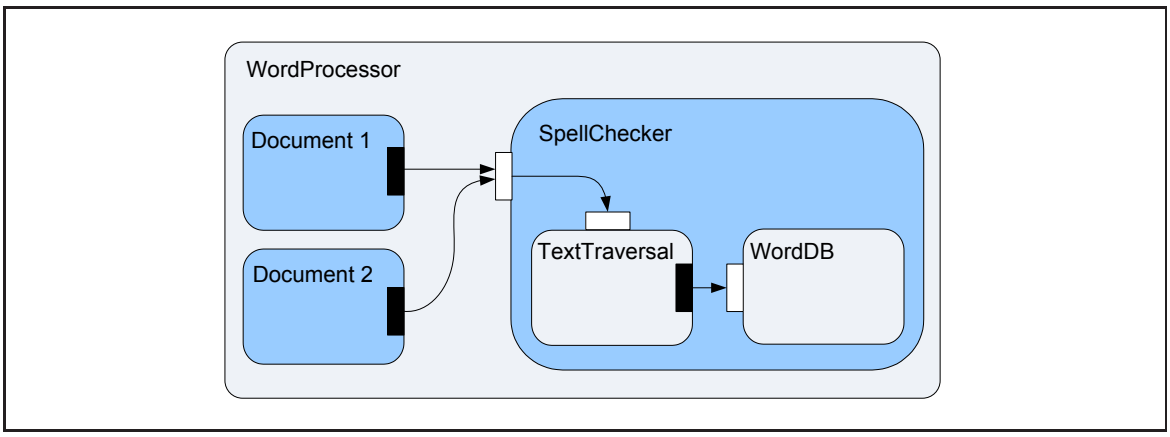


Figure 2.5: Architecture of the WordProcessor composite component.

The WordProcessor component is composed of three components—*Document1*, *Document2*, and *SpellChecker*. The *SpellChecker* component is again a composite component consisting of the *TextTraversal* and *WordDB* components. Let us denote the primitive components *Document1*, *Document2*, *TextTraversal*, and *WordDB* as (1), (2), (3), and (4), respectively. The behavior of (1), (2), and (3, 4) (i.e., the SpellChecker component) is modeled as CI automata in Fig. 2.6.

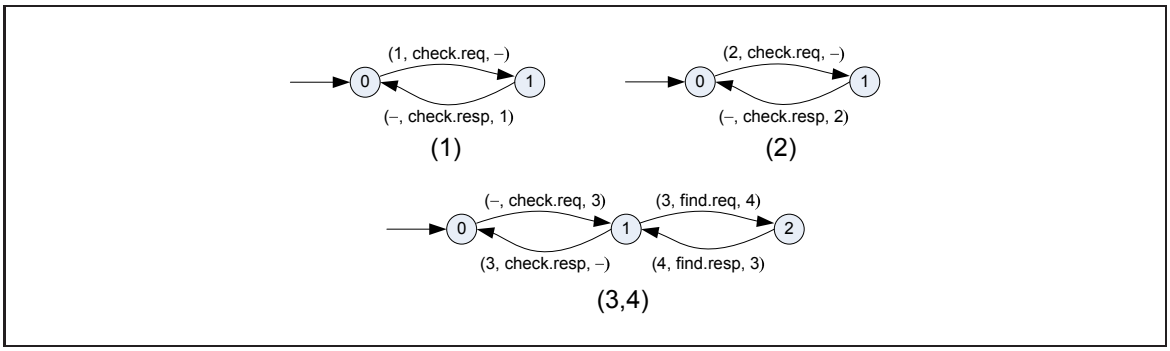


Figure 2.6: CI automata for the Document1 (1), Document2 (2), and SpellChecker (3, 4) components.

Consider now the situation, when the WordProcessor is a closed system in the sense that it does not offer any functionality to its environment. Therefore, we want to restrict the set of transitions in the way that no other component may interact with any component inside the WordProcessor component. This can be achieved via application of the composition operator $\otimes_T$ to (1), (2), and (3,4) to the composable set of CI automata $\{(1), (2), (3, 4)\}$ where $T = \{(1, check.req, 3), (3, check.resp, 1), (2, check.req, 3), (3, check.resp, 2), (3, find.req, 4), (4, find.resp, 3)\}$. The CI automaton $\otimes_T\{(1), (2), (3, 4)\}$ is then depicted in Fig. 2.7. As an aside, the complete transition space of the composite CI automaton has twelve states.

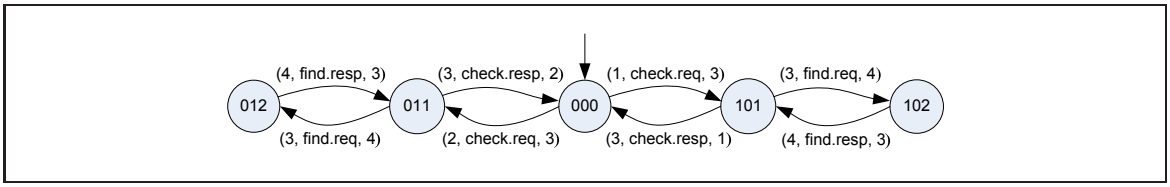


Figure 2.7: CI automaton modeling the composition ((1), (2), (3, 4)).

To touch a few particular aspects of this formalism, we mention the communication. The formalism of CI automata allows for modeling several communication styles—in particular it allows modeling non-blocking method calls as well as blocking message passing, in both synchronous and asynchronous ways. Since it does not specify any kind of communication errors (whose detection is a subject of consecutive model checking), in this sense, it does not limit the set of component models to which it can be applied. Furthermore, communication types not directly supported by the formalism (e.g. multicast) can be modeled using connectors [15].

Unified Modeling Language

Unified Modeling Language [73], or UML for short, is probably the most used specification platform in the industry. Featuring a variety of specification diagrams, it allows for describing almost all aspects of application design. It provides thirteen diagram types divided into three groups:

1. Structure diagrams
   - Class Diagram
   - Component Diagram
   - Composite Structure Diagram
   - Object Diagram
   - Package Diagram
   - Deployment Diagram
2. Behavior diagrams
   - Activity Diagram
   - State Machine Diagram
   - Use Case Diagram
3. Interaction diagrams
   - Communication Diagram
   - Interaction Overview Diagram
   - Sequence Diagram
   - Timing Diagram

Since the meaning of the diagram types of the first group is well-known, we briefly describe meaning of the diagram types from the second and third group.
Activity diagram aims at modeling of actions of a component/class from a start point to a finish. Various options are modeled using decision points.
State machine diagram models behavior of a single object as an entity responding to events issued by its environment, i.e., other parts of the system interacting with it.
Use case diagram aims at description of (several) execution scenarios in the form of interaction sequences of particular system parts.
Communication diagram aims at capturing of communicating object relationships.
Interaction overview diagram is a special case of *activity diagram* where the basic entities (nodes of the diagram) represent interaction diagrams (sequence, communication, interaction overview and timing diagrams).
Sequence diagram is a special form of *activity diagram* where the lifelines of several interacting objects are shown. It aims at visualization of the fact with which objects each object interacts. Creation and destroying of components instances/objects can be expressed.
Timing diagram describes changes of object states and variable values in time. It may also describe interaction among objects with respect to the timing constraints put on the system.

Despite its wide usage, the specification of arbitrary parts of UML is written in an informal way. This makes adopting of UML as a platform for formal specification and verification very hard [20] in general, speaking nothing of possible misunderstandings when sharing the specification among several designers. On the other hand, it allows definition of meaning of particular details for precise modeling in each specific case. Moreover, thanks to its illustrative power, UML can be advantageously used as a communication platform between a customer and a system designer.

Behavior protocols

Behavior protocols [2] are an approach to specification of software component behavior based on expressions corresponding to finite automata. They are used in several hierarchical component models (SOFA [71], SOFA 2.0 [12], Julia implementation of Fractal [11]).

Behavior protocols (BP) describe behavior of a software component by means of events appearing on the component exported (i.e., server and client) interfaces. BP distinguish four types of events appearing on the frame, which have the following syntax:

- $!interface.method\hat{}$ denotes emitting of a method request,
- $?interface.method\hat{}$ denotes accepting of a method request,
- $!interface.method\$$ denotes emitting of a method response, and
- $?interface.method\$$ denotes accepting of a method response.

These basic terms are combined using operators building up the expressions. The operators allowed in BP include ';' (sequencing), '+' (alternative), '*' (finite repetition), and '|' (parallel composition). Furthermode, to make BP expressions more readable, three syntactic abbreviations are defined:

- $?interface.method$ stands for $?interface.method\hat{}$; $!interface.method\$$,
- $!interface.method$ stands for $!interface.method\hat{}$; $?interface.method\$$, and
- $?interface.method\,\{expr\}$ stands for $?interface.method\hat{}$; $expr$; $!interface.method\$$.

As an example, consider the behavior protocol in Fig. 2.8. This behavior protocol defines behavior of a component providing access to files in the following way: first, the component expects the *open* method call on its *i* interface. Then, an arbitrary but finite number of times the methods *read* and *write* may be called; before returning the result of the *write* method call, the method *log* on the interface *lg* is called. Eventually, the *close* method is supposed to be called on the interface *i*. All the time, in parallel, the *status* method on the interface *ctrl* may be called to retrieve information about a file. Note that for the sake of simplicity, this specification allows to work with one file only.

$$(?i.open;\ (?i.read + ?i.write\ \{!lg.log\})*;\ ?i.close) \mid ?ctrl.status$$

Figure 2.8: Example of a behavior protocol

As described in Sect. 2.1.1, the set of exported interfaces of a component forms the component *frame*. With each frame, a *frame protocol* is associated defining the behavior of the component. Further, the composition of the behavior protocols of all first-level-of-nesting subcomponents of a component forms the *architecture protocol*.

Having specified behavior of all components, two kinds of compatibility relations can be verified in a hierarchical component model: (1) horizontal compliance referring to the absence of communication errors in a composition of behavior specifications of the component architecture (i.e., within the architecture protocol) and (2) vertical compliance[3] denoting the absence of communication errors between the frame and architecture protocols of a (composite) component.

In the case of BP, the *consent* composition operator [2] is used to detect the communication errors in a composition of BPs. *Consent* is basically a parallel composition joining the complementary events into internal τ-events. It is able to detect four types of communication errors:

- *Bad activity* refers to a state in the protocol composition when a request is being emitted but it is not being accepted at this moment; in other words, the internal τ-event could not be created.

- *No activity* refers to a deadlock, i.e., to a state where no further event may appear but this state is not a final (accepting) one—not all BP in the composition have been successfully accomplished.

- *Divergence* denotes a loop in a composition that cannot be exited and there is no final (accepting) state among the states in the loop.

- *Unbound-requires error* denotes a situation when an event is emitted on an unbound required (client) interface.

The main benefit of BP is that the correctness of communication can be verified at the design stage of the development process, i.e., before an implementation is available.

Behavior protocols have been successfully used for specification of real-life applications like in [1]. However, as BP disregard data at all, the behavior model is sometimes too coarse thus hiding some important behavioral aspects; also, the set of properties that can be verified is limited to the four types of communication errors mentioned above.

2.3 Tools

There are several tools supporting verification of various properties; however, only few of them aim at verification of software component behavior, in particular on behavior compliance. In this section, we discuss the ones that are with respect to our aims the most important ones.

[3]Technically, in the case of vertical compliance, the frame protocol is "inverted" (the emitting and accepting events are swapped, i.e., $!e$ is replaced by $?e$ and vice versa) and composed together with the architecture protocol. Then, vertical compliance is verified in the same way as the horizontal compliance.

2.3.1 Spin

Spin was developed by Gerard J. Holzmann in Bell Labs around 1980. Eleven years later, in 1991, Spin has been made freely available and it is still being developed. Nowadays, it is a state-of-the-art explicit state model checker and thus provides a very stable and convenient verification platform for concurrent processes. It is written in the C language and does not require any special libraries, therefore it is available on almost all platforms. To be friendly to common users, Spin comes with a graphical user interface (in addition to the command-line version) *XSpin* based on the Tcl/Tk [72] library that is again supported on a wide variety of platforms—MS Windows, Linux, Mac OS, HPUX, SCO Unix, FreeBSD, OS/390, and others.

The principle of Spin inheres in creating a C source file containing a verifier for the input model written in Promela—a specification language close to C. Afterward, the verifier is compiled and run. Spin offers a lot of options affecting the verification process. First, the user can choose the correctness properties to be checked—either safety or liveness properties can be chosen. The safety properties involve checking for absence of invalid end states and assertion violation, while the liveness properties focus on non-progress and acceptance cycles. Additionally, Spin is also able to verify properties expressed in LTL_{-X}—*Linear Time Logic* [39] without the X operator.

There are three modes of verification in Spin—exhaustive, hash-compact, and bit-state-hashing:

- Exhaustive—in this mode, the entire reachable state space is traversed, i.e., each reachable state of the model is visited.

- Bitstate hashing—the set of visited state in this mode is represented via a hashtable containing only zeros and ones. If a state is visited, a hash function is applied to the state vector representing the state and at the resulting position in the hashtable one is stored. A state is considered visited, if and only if there is a one at the corresponding table address. Hence for each state, only one bit of memory is needed; however, as the hash collisions are not solved, the hashtable should be at least one hundred times larger (in bits) than the size of the state space to keep a reasonable level of reliability that there are no hash collisions (and thus all reachable states are visited). For improving the reliability, two hash function are used in parallel and a state is considered visited if and only if there are ones at both addresses in the hashtable. The method is thoroughly described in [33].

- Hash-compact—to be able to have a hashtable a hundred times larger than the state space and to fit such a table into the operational memory, this method used the following approach: As the hashtable is sparse in ideal case (only about one percent are ones, the rest are zeros), another hash function to a much smaller hashtable can be applied to reduce the memory required. In this secondary hashing, the hash collisions are solved to keep the reliability of the method at the same level as bitstate hashing. Technically, using a hash function to e.g. 64-bits space as the hash function

in the "compacting" phase, the probability of undetected hash collision in this phase is even for large state spaces negligible. The advantage of this approach is the low probability of hash collisions and thus of omitting a part of the reachable state space while allowing traversal of larger state spaces than in the case of an exhaustive search. On the other hand, bitstate hashing can be applied on state spaces of an arbitrary size, with, however, reliability decreasing with the size of the state space. The hashcompact approach was introduced in [62].

In addition to verification, Spin also allows for simulation. This means that the model is executed in a common way—in contrast to verification, only a single interleaving of various processes' statements is considered and only one branch at non-deterministic points is taken. Together with visual representation of the channels, variable values, and execution description, the simulation mode provides a nice insight into the model.

2.3.2 Symbolic Model Verifier

The SMV (Symbolic Model Verifier) tool [47] was developed by K. L. McMillan at Carnegie Mellon University in 1992. It aims at traversal very large state spaces by using symbolic state space representation, in particular *ordered binary decisions diagrams* [17]. It uses a proprietary input language based on parallel assignment of variables' values describing almost directly a transition system via a set of equations. The SMV tool is due to the symbolic state manipulation able to traverse very large state spaces, on the other side, however, the input language provides almost no abstraction of the state space and therefore is not suitable (and probably not intended) to be used by designers of (component) applications. As the input language, it uses the parallel assignment language described in Sect. 2.2.2.

2.3.3 CADP

Cadp (Caesar/Aldébaran Development Package) [24] is a set of tools for protocol engineering with wide variety of features—modeling, simulation, and verification of properties. Cadp is able to accept three input languages:

1. high-level ISO specification language LOTOS [70],
2. low-level descriptions of protocols using Labeled Transition Systems in BCG graph format [23], and
3. intermediate-level Networks of communicating automata [52].

The set of languages can be easily extended due to the open architecture of the toolset.

The central part of Cadp is the open, language-independent Open/Caesar application programming interface providing the developer with simulation, execution, verification (partial, on-the-fly, etc.), and test generation. The input of Cadp is first transformed into

LTS represented by its initial state and a transition function. Such a representation is then processed by a simulator, a model checker, etc. In the following sections, we briefly describe the most important parts of the CADP toolset.

Aldébaran

Aldébaran [21] was jointly developed by the Vasy team and the Verimag laboratory. It aims at verification of communicating system represented as LTS. Its features include reduction of LTS according to (modulo) a number of equivalence relations (e.g. strong bisimulation, observation equivalence, and branching bisimulation). It uses several algorithm exploiting various approaches to make the verification process more efficient, such as on-the-fly state space generation and Ordered Binary Decision Diagrams [60].

The Caesar and Caesar.ADT compilers

Caesar [25] is a compiler translating the behavioral part of LOTOS specification into the C language or into LTS. In case of the C-language, the output can be used for simulation or verification. There are some minor restrictions (see [24] for details) put on the input language; the restricted language is, however, expressive and powerful enough for most real-life situations. The Caesar compiler works in several steps: first, it translates the input model from LOTOS into simplified process algebra called SubLOTOS, from which a Petri Net model is generated. Finally, after performing reachability analysis on the Petri Net model, LTS is produced. As to the performance and scalability of the tool, Caesar is able to process reasonable large models resulting in LTS of several millions od states.

The other Caesar compiler, Caesar.ADT [22], focuses on translation the data part of a LOTOS specification into the C language. The data language enable very fast prototyping directly in LOTOS. Again, Caesar.ADT has minor restrictions regarding the input language (see [24] for details), on the other hand, it is again able to process reasonable sized models.

The XTL model checker

XTL (eXecutable Temporal Language) [45] is a programming language with functional properties. It is intended for implementation of various temporal logic operators that are, by the XLT model checker, evaluated over LTS encoded in the BCG format. Since there is a working compiler for this language, a number of temporal logics have been implemented and used, e.g. HML [28], CTL [16], and ACTL [51].

BCG_MIN tool

BCG_MIN tool is used for minimization of not only standard but also "probabilistic" and "stochastic" LTSs. Although it implements only two bisimulations, namely the strong and branching ones, it can handle at least an order of magnitude larger LTS than Aldebaran can. Further, it is also more efficient regarding the memory requirements as it uses BCG

as its native format. Finally, BCG_MIN is able to print results of equivalence checking in a user friendly way by relating the nodes of the minimized graph to the original one.

Evaluator model checker

Evaluator [46], in CADP present in version 3.0, is a model checker performing verification of regular alternation-free μ-calculus formulas on LTS. The model checker first translates the model checking problem into a boolean equation systems, which is afterward solved using an efficient local algorithm [4, 58]. Details on the transformation process can be found in [46].

The input language of Evaluator allows for definition of user temporal operators, which turned to be very useful for writing a precise specification by ordinary users/system designers. As to the performance of the Evaluator, its third version clearly outperforms previous versions thanks to the incorporation of a new algorithm *SOLVE* for finding solutions of the boolean equational system. With the most recent version, verification of several liveness and safety properties of the well-known *alternating-bit protocol* model took from several seconds to several minutes [46].

Eucalyptus user interface

All the CADP tools are controlled by the user using the graphical user interface Eucalyptus. The user interface is written in Tcl/Tk [72] and can be used on a wide variety of platforms. Moreover, through Eucalyptus, the user may use the Autograph editor [56] for creation of graphical representations of LTSs.

2.3.4 Behavior Protocols Checker

Behavior Protocol Checker (BPC) [42] is a tool for checking compliance of behavior specifications of communicating components. In this section, we briefly describe the basic principles and the architecture of the tool.

Shortly, BPC is an explicit model checker using the on-the-fly state space generation. It allows evaluation of both pragmatic and consensual compliance relations. To tackle the state explosion problem in representation of behavior protocols, BPC uses *parse tree automata* as a state space representation technique. In the following paragraphs, BPC is referred to as the *Java verifier* and *Java implementation of PTA*.

Parse trees (also *syntax* or *expression trees*) are a common way to represent expressions in memory. They are mainly used to represent mathematic formulas and program source codes in compilers. Obviously, they are also capable to represent behavior protocols (Fig.2.9).

A parse tree is a tree structure that describes a given expression unambiguously. When representing behavior protocols, the parse tree features the following important properties:

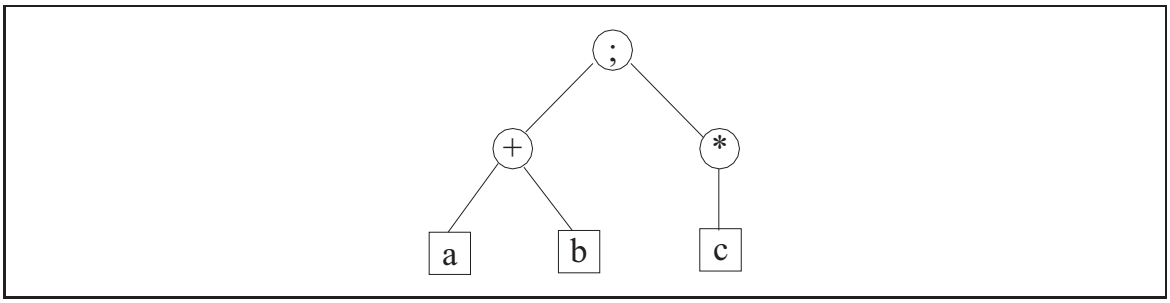


Figure 2.9: A parse tree representing (a+b); c*.

- Event symbols featuring in an expression appear only in the leaf nodes and operators in inner nodes of the corresponding parse tree.
- The operator nodes representing the repetition and restriction operators are unary; all others are binary.
- Every subtree describes an expression (valid behavior protocol).

The main advantage of parse trees is the size of representation, linearly dependent on the expression length and having no direct relation to the number of states. Also the building time is linear in the length of expression. Evaluation of access time and state identifiers' space requirement will be discussed later after we present parse tree-based representation technique (parse tree automata).

Parse tree automata (PTA). Construction of a PTA follows the idea of recursive state space creation in the explicit representation technique. As PTA is a symbolic technique, the actual full state space of PTA is never represented as a single complex data structure. On the contrary, the key idea is to (i) directly represent only the parse tree (PT) of the expression and the *primitive automata* which accept the event symbols in the leaves of the parse tree, (ii) introduce composed state identifiers allowing to denote the current state and avoid unnecessary multiple traversals of PTA states, and (iii) define the transition function of PTA via recursive rules determining the (direct) transitions from a state, given its composed identifier. An example of PTA and its correspondence to a parse tree is illustrated in Fig. 2.10.

We will demonstrate the idea on three simple examples: (1) representation of a primitive automaton, (2) implementation of automata composition driven by the sequence operator, and (3) implementation of automata composition driven by the parallel operator. Automata compositions driven by the other operators are implemented in a similar manner (a detailed description is in [41]).

A primitive automaton has two states (initial and accepting) and a single transition between them. The transition label is an event symbol (Fig. 2.11a).

The sequencing operator expresses concatenation of the languages accepted by the left- and right-hand automata PTA_L and PTA_R. To create the respective composed automaton $\mathrm{PTA}_;$, it is sufficient to establish implicit transitions (λ) from the accepting states of PTA_L

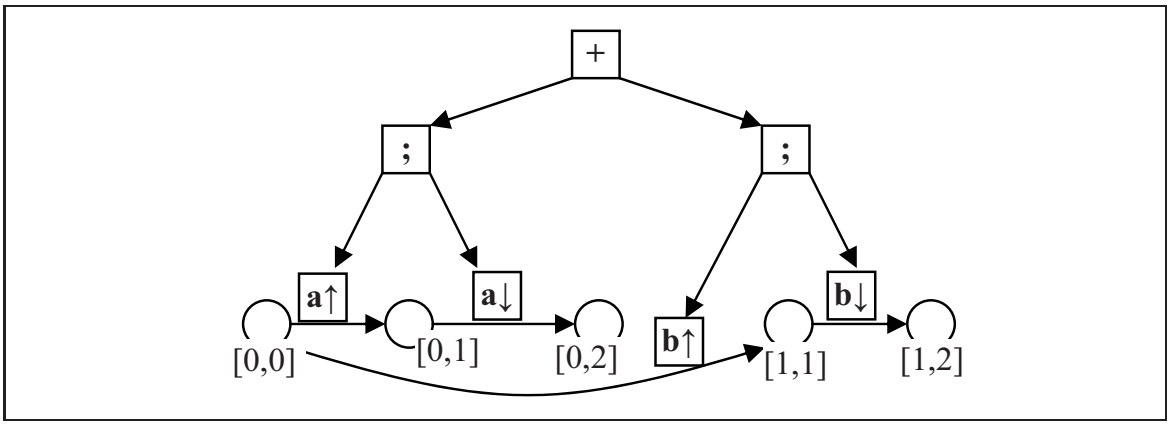


Figure 2.10: Generating states and transitions of PTA. Circles represent states. Squares represent nodes of PT; [0,0] denotes the initial state.

to the initial state of PTA_R (Fig. 2.11b). The resulting set of accepting states in $\text{PTA}_;$ consists of the accepting states of PTA_R. The accepting states of PTA_L are added only if the initial state of PTA_R is accepting. Obviously, modifications of PTA_L and PTA_R are not necessary, since the implicit transitions λ are added in the implementation of the sequencing operator in $\text{PTA}_;$.

The parallel operator expresses arbitrary interleaving of all the words of the languages accepted by the left- and right-hand automata PTA_L and PTA_R. In order to create the respective product automaton, it is sufficient to establish a state space "grid" and corresponding transitions as illustrated in Fig. 2.11c.

Composed state identifiers in PTA. To address the idea (ii) above, a state identifier must reflect the structure of the subtree of PT it is associated with and capture the state of the primitive automata within the subtree. For a specific PT, all the top-level identifiers will be of the same size (linear in the size of PT). As a technicality, memory allocation for state identifiers can cause substantial memory overhead. It is recommended to use an allocator that is optimized for allocating small memory chunks of the same size.

Time requirements for generating PTA transitions. The average time required is influenced by the number of PT nodes that have to be visited to calculate the list of transitions associated with a particular state. In each of these nodes some computation is necessary, as the potential transitions are determined on the fly. For each transition, also the state identifier of the target state has to be evaluated for keeping track of the states visited.

The number of visited PT nodes is greatly influenced by the actual operators encountered in PT. For example, for the standard regular expression operators only one subtree has to be visited. On the contrary, encountering a parallel operator means visiting both subtrees.

PTA optimizations. As discussed above, performance of PTA depends on the number of nodes in PT. If the number of PT nodes were reduced, performance would greatly improve. Therefore we experimented with several optimizations in PTA representation.

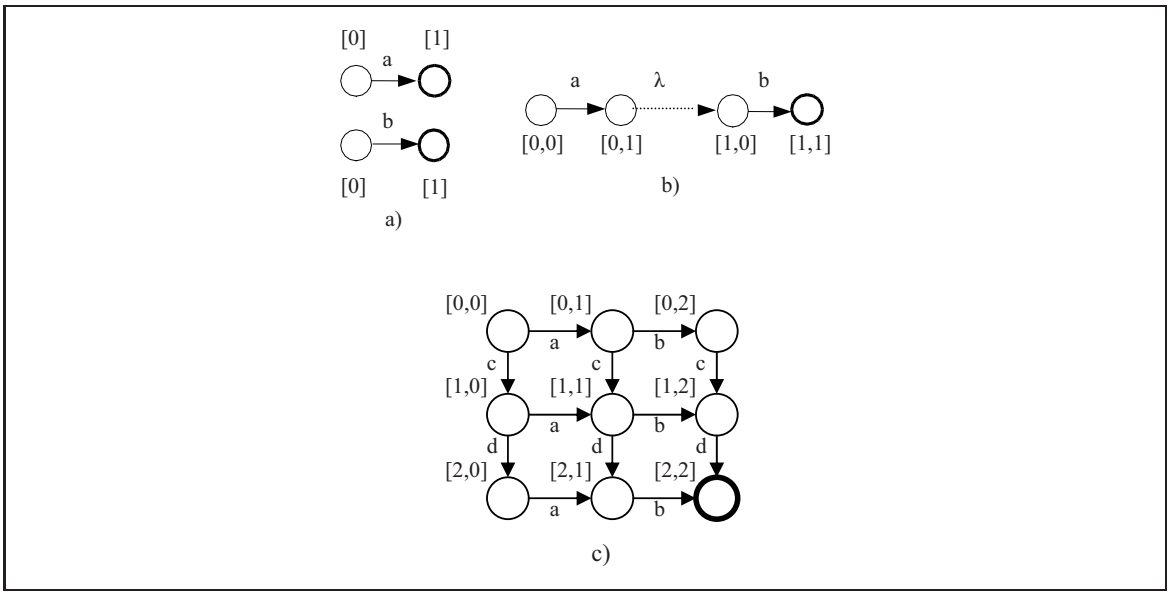


Figure 2.11: a) Primitive automata for the 'a' and 'b' event symbols. b) PTA for '$a;b$'. c) PTA for '$(a;b) \mid (c;d)$'. Legend: A dotted arrow represents an implicit transition λ. State identifiers are in brackets (simplified).

Multinodes. The idea of multinodes is to collapse the nodes of PT featuring the same operator into a single node. For example, in Fig. 2.12 collapsing means representing only a single node for the sequence operator '**;**' (associated with a list of PT subtrees a, b, c, d).

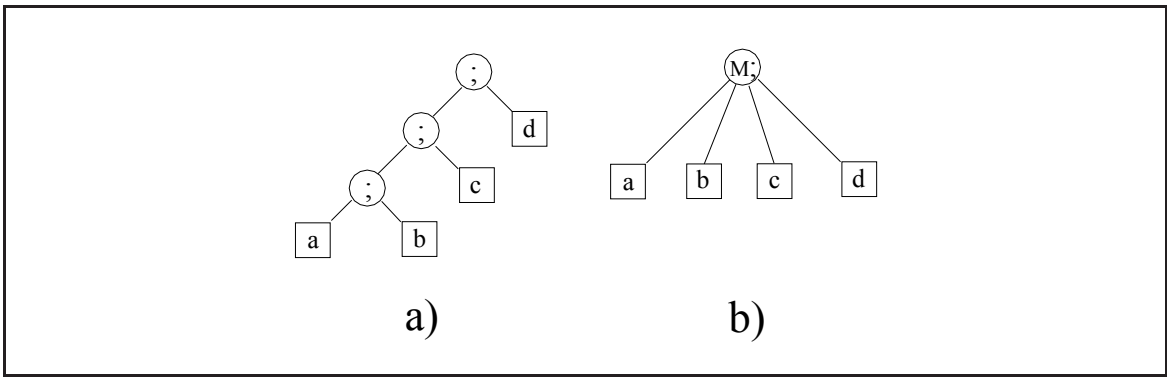


Figure 2.12: a) Original parse tree. b) Parse tree with multinodes for the protocol a;b;c;d.

This way, access time is greatly improved since less computation is required.

Forward cutting (of primitive automata). Removal of the transitions from the state space, which are discarded by a restriction operator, can be easily achieved by removing the affected event symbols nodes from PT.

Again, such optimization can produce PTs with a smaller number of nodes what results in a smaller state identifiers' space and improved access time.

Explicit subtrees. Since performance of explicit representation is very good for state spaces of "reasonable" size, it can be advantageous to combine both the PTA and explicit representations techniques. It is feasible to select those PT subtrees that imply a small state space (typically not featuring "many" parallel operators) and the states of which are generated more than once (e.g. forced by a parallel operator in a higher level of PT) and represent them via explicit automata embedded in PTA.

We implemented two verifiers based on the PTA representation technique ("Python verifier" and "Java verifier"). These implementations provide a flexible framework that allows simple addition of new parsers, optimizations, and verification backend alternatives.

As the architecture of both Python and Java verifiers is almost the same and we achieved better results using the Java verifier, we will omit description of the Python version, but we include a brief comparison of performance of both implementations. Additionally, we also provide comparison of both implementation with the original implementation not using PTAs but explicit in-memory deterministic finite automata as the state space representation.

The Java verifier uses the approach and techniques employed in the former Python verifier, but it introduces new optimizations and backend features. By these optimizations, both time and space requirements decreased and, therefore, the complexity of the protocols that can be checked was pushed a bit further.

Optimizations. Besides the optimizations included in the Python verifier (explicit subtrees and forward cutting), the multinodes optimization (Fig. 2.12) was implemented and found very beneficial. This optimization is performed during the construction of a parse tree in a straightforward, efficient way.

Backend alternatives. In the Java verifier we implemented only two backends: compliance checking and visualization, since these two had been identified as the most frequently needed.

For visualization, we decided to use the dot tool of the Graphviz package [69], since it is freely distributed and its features greatly suffice for our purposes. The visualization backend is able to provide both protocol parse tree and graph of the PTA state space. Since the dot tool supports, among other types of output, the Virtual Reality Modeling Language (VRML), this format can be advantageously used for complex protocols both to get the whole picture of the automaton and zoom into its specific parts.

Implementation details. Because of the differences between Python and Java, we had to cope with a lot of specific problems when rewriting the verifier from Python to Java. A main problem was the state identifiers in Java (handled internally by Python): As implied by the argumentation above, we needed state identifiers that could be computed fast and consume as small amount of memory as possible. We could not use Java references, because of the on-the-fly state generation (potentially repeated for a particular state).

Therefore, each state is represented by a *state tree*, where its leaves represent the states of primitive automata, while inner nodes represent the state of the composed automata

corresponding to the nodes' subtree. The state identifier of a primitive automaton indicates its active state (0 or 1) (Fig. 2.11a). The state identifier of a composed automaton is created as concatenation of its children's identifiers. Thus, the resulting state identifier reflects the structure of PT, uniquely denotes a state within the state space, and its length is linear in the size of PT. Obviously, the state identifier of the main automaton is determined at the root of the state tree. The state identifiers are computed in a lazy way (only when actually needed) and are stored in a cache. Traversal of the state space employs frequent comparison of the identifiers (that is quite fast). Even though the computation of state identifiers was optimized for speed, it is still the most time consuming operation in the checking process (since it is performed for each state visit).

Benchmarks. We employed two types of benchmarks: the first type was focused on the benefits of particular optimizations in the Java verifier and the second one on a comparison of performance of the three verifier versions: the original verifier (written in Java), and Python and Java PTA verifiers. Always we used protocols of various complexity; both real-life and "academic" protocols inducing large state spaces were checked.

The real-life protocols included a set of database server protocols, while the "academic" protocols involved only the parallel operator (such as $a \mid b$, $a \mid b \mid c$, ...), which causes the exponential growth of the state space. Using this enabled us both to generate large state spaces and easily compute their sizes.

The optimization benchmarks have shown that disabling the forward cutting optimization results in a very poor performance. This is caused by the complement operator expanding the state space to an enormous size. Hence, forward cutting is used in each of the benchmarks below. The benefits of the other types of optimization depend on the concrete structure of the protocols being checked (Tab. 2.1). For example, in the case of "academic" protocols using the parallel operator, the most worthwhile optimization are multinodes; the explicit subtrees optimization cannot be used here, because the states of the automaton represented by the only (multi-)node in the parse tree are used only once. While checking the real-life protocols, the explicit subtrees optimization is most beneficial.

Since the most important parameter of the protocol verifier is the state processing speed, in Tab. 2.2 we present the comparison of all verifiers based on checking the "academic" protocols (the results of checking the real-life protocols are not so interesting).

A comparison of memory requirements is not involved, however, the PTA representation requires a much smaller amount of memory than a corresponding explicit representation. In all benchmarks considered below, all optimizations were applied.

In the case of "academic" protocols, the Java verifier is faster than the Python verifier even if we turn off the multinodes optimization; the state processing is about two times faster in the Java verifier, which is probably caused by the fact that the Java Virtual Machine outperforms the Python Psyco compiler. On the other hand, the construction of the explicit subtrees is much slower in Java because of the evaluation of the state identifiers; the Python verifier is also able to keep more states (and larger explicit subautomata) in

	Forward cutting only	All optimizations	No multinodes	No explicit subtrees
Academic (parallel)	100%	76.2%	100.8%	75.7%
Real-life	100%	50.5%	67.7%	81.4%

Table 2.1: Average relative time the Java verifier spent by checking with various optimizations enabled.

Number of parallel operators used	Original verifier	Python verifier	Java verifier
6	100%	38.3%	22.3%
7	100%	16.5%	7.7%
8	100%	6.9%	2.3%
9	100%	2.7%	0.7%

Table 2.2: Relative time spent by checking the "academic" (parallel) protocols by all verifiers.

memory, because its state identifiers are shorter. In any case, the PTA approach beats the original explicit state representation.

BPC was used in [1] to verify compliance of communicating components of a real-life application aimed at providing WiFi access to the Internet at airports. As a part of this project, other two versions of BPC were developed—*Runtime checker* and *Checker for code analysis*. The former version focuses on runtime monitoring of events (i.e., method calls and returns) appearing at component frames and comparing it with behavior protocol associated with this component. The latter one is used, together with Java PathFinder [59], to statically compare the implementation of a component in Java with the associated frame protocol [53]. Compliance of both cooperating components protocols and implementation of each primitive components and a corresponding behavior protocol were evaluated using a decent PC[4] requiring several hours.

2.4 Problem elaborated

Apart from the CoCoME project [64], which is running at the time of writing this thesis, according to our knowledge, none of the formalisms mentioned in this chapter has been used for specification of a real-life-sized case study except for Behavior Protocols in [1]. Moreover, except for BP, majority of the aforementioned formalism used on even simple examples demonstrate their substantial computational complexity, which is most probably the reason why they have not been used for modeling of a larger application. On the

[4]Benchmarking PC configuration: 3 GHz Intel P4 processor with 2 GiB RAM running MS Windows Server 2003 and Sun JDK 1.4.2_b10.

other hand, it turned out that BP provide a reasonable level of abstraction and enable verification of component compliance in the context of a real-life application.

Nonetheless, several issues making the specification hard to design were discovered [1]; we present some details within the following paragraphs.

The behavior specification of a component containing no notion of data often overspecifies the actual behavior thus rising artificial incompatibilities. In other words, to avoid too coarse overspecification and to model the behavior of a component, which is often dependent on data values, at a reasonable level of abstraction, the data have to be present also within the specification.

Data are used in a specification for two main purposes: (1) to pass method-specific information to the "implementation" of a method, and (2) to store information across multiple method calls to model the current component state (mode in [29]). The introduction of data to a modeling language must be done in a careful way, because they may be a major cause of state space explosion.

Behavior protocols provide a natural mechanism for synchronizing events of two behavior protocols, i.e., assuring that both components reach a certain computation point at the same time. However, in case of hierarchical component initialization, this mechanism is not sufficient—more than two components have to be synchronized. In ACP, such synchronization can be achieved by defining the following rules: $\gamma(s_1, s_2) = s_4, \gamma(s_3, s_4) = s_5$. This way, components C_1, C_2, C_3 having in their alphabets s_1, s_2, s_3, respectively, are synchronized on these events resulting in the event s_5.

Behavior protocols allow for checking for absence of composition errors and compliance. In some cases, however, it might be beneficial to check for other properties. In [1], such properties are eventual logout of each previously logged user and eventual destroying of each Token component. Such properties are in classical model checking expressed in a temporal logic, like LTL or CTL.

Last, but not least, BPChecker [42] used for verifying behavior protocol properties is limited to state spaces of the size of 10^8 states, which turned out to be not sufficient in some cases.

These issues can be summarized as follows:

1. Absence of data, namely method parameters and variables for storing component states caused the specification to be imprecise and two coarse in several cases (artificial communication errors appeared),

2. synchronization of event from more than two behavior protocols became difficult and resulted in incorporating of artificial BPs,

3. the properties were limited to the absence of the communication errors, i.e., no user specific properties could be verified, and

4. the complexity and size of the specification of the Airport Internet Access application [1] reached the limits of BPChecker; more complex protocol would not be verifiable.

2.5 Goals revisited

Due to the reasons given in the previous section, we decided to use BP as a basis for a formalism aimed at component behavior specification and to extend the formalism with means solving the issues mentioned. In particular, our goals follow:

(G1) Extend the formalism of Behavior Protocols with the following constructs:

- (G1a) Method call parameters—the behavior of a software component, i.e., the way it process a specific method call, often depends on the parameters of the method call issued to the component. Even if the behavior can be usually modeled regardless of parameters, such behavior is usually nondeterministic and thus may cause behavior incompatibility during composition with specification of other components.
- (G1b) Local variables—even though behavior of a stateful component (i.e., a component "remembering" a piece of information across several method calls) can be modeled via BP, due to the absence of data, the BP has to store the information about the state implicitly as a position within the expression. This causes both parts to be duplicated within the expression and legibility of such expression to be bad.
- (G1c) Multisynchronization events—in some cases of hierarchical component initialization, it is necessary to first complete the initialization part of all BPs and start the other parts of BP afterwards. This cannot be achieved in a simple way in BP when more than two components are involved.

(G2) Enhance the readability of specification written in BP.

(G3) Provide a way of checking general properties described in a standard formalism (LTL, CTL, etc).

(G4) Provide a more efficient (in the sense less time/memory demanding) way of verification of extended BP.

With respect to the nature of the goals (G1) – (G4), the part of this thesis describing the contribution is divided into two parts. First, we describe the extensions of Behavior Protocols addressing the goals (G1) and (G2), which is followed by rather technical part aiming at description of transformation of the extended Behavior Protocols into Promela thus solving the goals (G3) and (G4).

Chapter 3

Proposed specification language (EBP)

In this chapter, we describe a new specification language aiming at description of software component behavior. The language is based on Behavior Protocols (BP) [54]. It extends the original language with new features to make the specification more precise and readable. We refer to the proposed language as *Extended Behavior Protocols* or shortly *EBP*.

EBP take the form of expressions describing sequences of events appearing on component frames and as well as BP, each EBP corresponds to a nondeterministic finite automaton accepting the same language (i.e. set of traces) as the EBP generates.

To illustrate the basic idea, consider the example of EBP in Fig. 3.1. This specification describes behavior of the *FlyticketClassifier* component from [1]. It consists of three parts: (i) type definitions, (ii) local variables definition, and (iii) definition of behavior.

Within the part (i), the specification contains definition of enumeration types *AIRLINES* containing two possible values—*AIRFRANCE* and *CZECHAIRLINES*, and *CONTROL* containing the values *EXEC* and *STOP*.

The part (ii) defines a local variable *ctrl* of the type *CONTROL*; the variable is initialized to the *EXEC* value.

The third part (iii) defines the behavior as a set of traces, i.e., sequences of events appearing on the frame of the component with which this EBP is associated. In particular, the component behaving according to this specification is able to accept the *CreateToken* method call on its provided interface *IFlyTicketAuth*. According to the value of the *al* parameter, the *GetFlyTicketValidity* method is called either on the *IAfFlyTicketDb* or *ICsaFlyTicketDb* interface. Similarly with the *IsEconomyFlyTicket* method—this call is optional (the reason is not visible in the specification), which is denoted by the "*+ NULL*" part. The *CreateToken* method call is accepted several times, as long as the value of the *ctrl* local variable is equal to *EXEC*. The value of the *ctrl* variable is changed after accepting the *stop* method on the *ICtrl* interface. This method call can be accepted in parallel with (during processing of) a *CreateToken* method call.

As apparent from the example, EBP extends the original language of BP with several new constructs; in particular:

```
component FlyTicketClassifier {
  types {                                                    ↑
    AIRLINES = { AIRFRANCE, CZECHAIRLINES }                  (i)
    CONTROL = { EXEC, STOP }                                 ↓
  }
  vars {                                                     ↑
    CONTROL ctrl = EXEC                                      (ii)
  }                                                          ↓
  behavior {
    ( while (ctrl == EXEC) {                                 ↑
        ?IFlyTicketAuth.CreateToken(AIRLINES al) {           (iii)
          switch (al) {                                      ↓
            AIRFRANCE: {
              !IAfFlyTicketDb.GetFlyTicketValidity ;
              (!IAfFlyTicketDb.IsEconomyFlyTicket + NULL)
            }
            CZECHAIRLINES: {
              !ICsaFlyTicketDb.GetFlyTicketValidity ;
              (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL)
            }
            default: { NULL }
          }
        }
      }
    )
    | ?ICtrl.stop { ctrl <- STOP }
}}
```

Figure 3.1: Example of EBP specification

- local variables of components of enumeration types,
- parameters of method call requests,
- *switch* statements on values of local variables and parameters to define multiple behaviors,
- repetition of a specification part controlled by a value of a local variable, and
- synchronization of events from more than two EBP at a time.

In the following sections, we first present a motivation for each particular extension and its informal description, which will later be made exact by a formal definition.

3.1 State variables and method parameters

Data can be used in a software specification for two main purposes: (1) to pass method-specific information to the method, and (2) to store information across multiple method calls to model component states, e.g. component modes [29]. Introduction of data to a modeling language must be done in a careful way, as data may be a major cause of the state space explosion in model checking of the specification. To illustrate this claim, consider the behavior protocol $?i.m1 \mid ?i.m2$. A component that behaves according to this behavior protocol is able to accept the $m1$ and $m2$ methods on the i interface in parallel. The state space generated by this protocol consists of nine states[1]. Adding a parameter of the integer type to each method that is used in the potential method bodies blows up the state space to $9 \times 2^{sizeof(integer)} \times 2^{sizeof(integer)2}$. Moreover, in almost all cases a data type of a much smaller domain, e.g. *byte*, would be sufficient.

As an example consider a part of the BP specification of the *FlyTicketClassifier* component from [1] in Fig. 3.2. It corresponds to the the specification in Fig. 3.1. After accepting the *CreateToken* method request, the *GetFlyTicketValidity* method call request is emitted either on the *IAfFlyTicketDb* or *ICsaFlyTicketDb* interface depending on the parameter passed to the *CreateToken* method. In the case of a malformed or wrong parameter, no event is emitted (expressed by the *NULL* subprotocol).

$$
\begin{array}{l}
?IFlyTicketAuth.CreateToken\ \{ \\
\quad (\\
\quad\quad !IAfFlyTicketDb.GetFlyTicketValidity; \\
\quad\quad (!IAfFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\quad) \\
\quad + \\
\quad (\\
\quad\quad !ICsaFlyTicketDb.GetFlyTicketValidity; \\
\quad\quad (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\quad) \\
\quad + \\
\quad NULL \\
\}
\end{array}
$$

Figure 3.2: A Part of the FlyTicketClassifier behavior protocol.

Because of the absence of parameters, the behavior protocol specifies a superset of the potential (in the future) implemented component behavior. To illustrate this claim, consider a situation in the potential implementation when the *CreateToken* method is called

[1]The protocol $?i.m1$ is a syntactic abbreviation for $?i.m1\hat{}\,;!i.m1\$$, which generates the state space with three states. Hence, the parallel composition yields the state space of nine states.

[2]Assuming the integer size to be 4 bytes, the resulting state space consists of more than 10^{20} states.

with a parameter denoting an AirFrance fly ticket. In this situation, the *GetFlyTicketValidity* method on the *IAfFlyTicketDb* interface should be called. However, the specification also allows the *GetFlyTicketValidity* method on the *ICsaFlyTicketDb* interface to be called. Even if such imprecisions usually do not extend the state space size, a behavior incompatibility among communicating components may be missed or an artificial one may appear.

As argued above, the behavior specification of a component containing no notion of data often overspecifies the desired behavior—to avoid this, some notion of data has to be present within the specification.

3.2 Multisynchronization

In the case of original Behavior Protocols, two protocols can be synchronized on an event. This is achieved via application of the consent composition operator on two behavior protocols which results in transformation of two complementary events into a single τ-event. There are two issues regarding such a synchronization:

1. The synchronization is done in an one-directional way so in states when the accepting side is not able to accept a particular event the bad-activity error appears.

2. Events from no more than two BPs may be synchronized in this way[3].

Both the aforementioned issues become a problem in cases when synchronization of events from more than two BPs is needed. As an example, consider the following situation occurring also in [1]: There are three components A, B, and C, whose interfaces are bound as shown in Fig. 3.3. The (simplified) corresponding behavior protocols for components B and C are listed in Fig. 3.4.

The communication scenario is as follows: First, the A component creates the data necessary for initialization of the B and C components and pass it to them. After the B and C components complete their initialization, they are ready to start their business functionality. A problem lies in the construction of the protocol for the component A. The first option is to initialize the components in the order of B, C. In this case, however, after initialization of the B component through its $BPI1$ interface, the component may emit a request to the A component (using the $BRI1 \rightarrow API1$ binding) before the C component has finished its initialization. This would result in the bad-activity error—the A component is not ready to accept the request at this point. Similarly, using the initialization order of C, B may result in a call to the non-initialized component B using the $CRI1 \rightarrow BPI1$ binding. Generally, there are two ways to cope with this problem in this particular situation:

[3]Of course, as the synchronization of events of a fixed number of BPs can be modeled by a finite automaton, it can be also described by BP. However, in such a case, the protocol is quite complex and its length grows with the number of components involved.

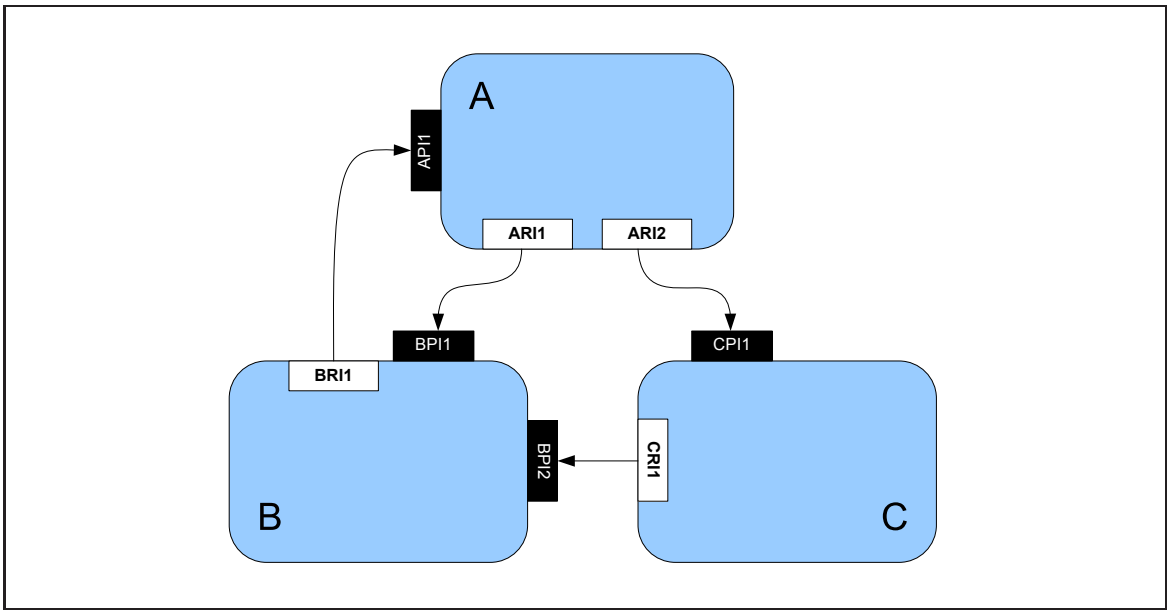


Figure 3.3: Synchronization of three components.

$B : ?BPI1.init;\ (!BRI1.m1 \mid ?BPI2.m2)$
$C : ?CPI1.init;\ !CRI1.m2$

Figure 3.4: Behavior protocols corresponding to the B and C components in Fig. 3.3.

(i) to use the parallel operator; such a protocol models more than the desired behavior and thus may cause problems with encapsulation of the components into a supercomponent[4], or
(ii) to use a software connector/helper component for synchronization.

$!ARI1.init;\ (?API1.m1 \mid !ARI2.init)$

Figure 3.5: Behavior protocol corresponding to the A component in Fig.3.3.

A solution via taking the first option is listed in Fig. 3.5, while a behavior protocol of a software connector intercepting the communication among all components is listed in Fig. 3.6. In the case of the option (ii), we suppose that the name of the interface of the connector bound to an interface A is denoted as C_A at the connector side. Furthermore, we suppose that the order of the initialization is not important, while the order of business-logic method calls is important, thus the connector respects and preserves the order of these method calls.

[4]In cases the A component reacts to requests from B and C by interacting with other components or the B component communicates with another component.

$$
\begin{array}{l}
(\\
\quad (((?C\_ARI1.init\ \{!C\_BPI1.init\}\ |\ ?C\_ARI2.init\ \{!C\_CPI1.init\})\ | \\
\qquad (?C\_BRI1.m1\hat{}; ?C\_CRI1.m2\hat{})); !C\_API1.m1\hat{}; !C\_BPI2.m2\hat{}) \\
+ \\
\quad (((?C\_ARI1.init\ \{!C\_BPI1.init\}\ |\ ?C\_ARI2.init\ \{!C\_CPI1.init\})\ | \\
\qquad (?C\_BRI1.m1\hat{})); !C\_API1.m1\hat{}; ?C\_CRI1.m2\hat{}; !C\_BPI2.m2\hat{}) \\
+ \\
\quad (((?C\_ARI1.init\ \{!C\_BPI1.init\}\ |\ ?C\_ARI2.init\ \{!C\_CPI1.init\})\ | \\
\qquad (?C\_CRI1.m2\hat{}; ?C\_BRI1.m1\hat{})); !C\_BPI2.m2\hat{}; !C\_API1.m1\hat{}) \\
+ \\
\quad (((?C\_ARI1.init\ \{!C\_BPI1.init\}\ |\ ?C\_ARI2.init\ \{!C\_CPI1.init\})\ | \\
\qquad (?C\_CRI1.m2\hat{})); !C\_BPI2.m2\hat{}; ?C\_BRI1.m1\hat{}; !C\_API1.m1\hat{};) \\
) \\
| \\
(\\
\quad ?C\_API1.m1\$; ?C\_BPI2.m2\$; !C\_BRI1.m1\$; !C\_CRI2.m2\$ \\
+ \\
\quad ?C\_API1.m1\$; !C\_BRI1.m1\$; ?C\_BPI2.m2\$; !C\_CRI2.m2\$ \\
+ \\
\quad ?C\_BPI2.m2\$; ?C\_API1.m1\$; !C\_CRI1.m2\$; !C\_BRI1.m1\$ \\
+ \\
\quad ?C\_BPI2.m2\$; !C\_CRI1.m2\$; ?C\_API1.m1\$; !C\_BRI1.m1\$ \\
)
\end{array}
$$

Figure 3.6: Behavior protocol of a connector avoiding the bad activity error in the architecture in Fig.3.3.

As sketched above, there is no possible correct sequential order of initialization of components B and C and even introducing artificial parallelism does not solve the problem satisfactorily.

Therefore, an extension of BP is necessary aiming at a mechanism that would allow for synchronization of events from more than two BP. In [1], this problem was addressed by extending the semantics of Behavior Protocols by *atomic actions* [35]. An atomic action is a special event consisting of other ordinary events (enclosed by '[' and ']'). All the events inside an atomic action are composed with their respective counterparts and then appear at once as a single event. For illustration, consider the behavior protocols in Fig. 3.7 addressing the situation in Fig. 3.3. Although this enabled behavior specification of all components involved in the demo application [1], several issues arose. Violation of associativity of the composition operator and much more complex formal system were the most important ones. In each proposed solution of these issues, at least one of the aforementioned problems persisted. Therefore, we do not consider the use of atomic actions for synchronization of more BPs as a suitable approach.

$A: (!ARI1.init\hat{} \mid !ARI2.init\hat{}); [?ARI1.init\$, ?ARI2.init\$]); ?API1.m1$
$B: ?BPI1.init; (!BRI1.m1 \mid ?BPI2.m2)$
$C: ?CPI1.init; !CRI1.m2$

Figure 3.7: Behavior protocols corresponding to the situation in Fig. 3.3 using atomic actions.

To cope with multisynchronization, we propose an extension of BP with special (neither emit nor accept) events that are shared by the EBPs involved in a synchronization. These events are blocking. Each of the multisynchronization events is allowed to occur if and only if both EBPs being composed allow it to happen in the current state. After the consent composition, both events appear as a single event of the same form in the resulting trace, i.e., they are not converted into a τ-event. This way, another EBP can be synchronized on the same event after it is composed with the composition of those two.

3.3 While loops

Another issue identified when using BP as a specification platform is the lack of expensiveness when specifying loops. A loop can be specified using the repetition operator, which generates any arbitrary number of repetitions of its operand. This may become insufficient if stopping the associated component is desirable under specific conditions.

As an example, consider the following situation: There is a component providing a service via a provided interface (and in particular via two methods of this interface). Behavior of such a component can be modeled by a behavior protocol in Fig. 3.8. In this case, the component provides the *svc1* and *svc2* methods on the interface *if*. We cannot stop the component in the sense that it would reject future method call requests. A way to achieve interruption of the repetition is to use ';' and to append the protocol $?if.stop()$ to it. This way, however, the *stop* method call can be accepted only at the end of the loop, which is definitely not sufficient in most cases.

$((?if.service1\{!r\_if.svc1\}) + (?if.service2\{!r\_if.svc2\}))*$

Figure 3.8: Behavior protocol of a component providing the *svc1* and *svc2* methods.

A behavior protocol specifying behavior of a component that can be explicitly stopped is listed in Fig. 3.9. The protocol expresses the fact that the $if.stop$ method request may be accepted any time; after accepting this event, the other method requests are no more accepted. Although precisely describing the desired behavior, this protocol is not well readable and hard to understand. Moreover, in many cases, it is desired that the loop is controlled by a value of a local variable expressing the state of the component, which is even more difficult to specify without a special construct. Therefore, a better way of controlling loops than using the repetition operator would be beneficial.

$((?if.service1\{!r\_if.svc1\}) + (?if.service2\{!r\_if.svc2\}))*;$
$(?if.stop +$
$(($
$(?if.service1\{?if.stop\hat{}; !r\_if.svc1\}) +$
$(?if.service2\{?if.stop\hat{}; !r\_if.svc2\}) +$
$(?if.service1\{!r\_if.svc1\hat{}; ?if.stop\hat{}; ?r\_if.svc1\$\}) +$
$(?if.service2\{!r\_if.svc2\hat{}; ?if.stop\hat{}; ?r\_if.svc2\$\}) +$
$);\ !if.stop\$)$
$)$

Figure 3.9: Behavior protocol of a component providing the *svc1* and *svc2* that can be stopped.

3.4 Syntax and semantics

In this section, we present a definition of the syntax and semantics of Extended Behavior Protocols. Before coming up to the syntax, let us mention some basic facts regarding EBP specifications. Each EBP specification yields an abstraction of the associated component behavior in the form of a set of traces. Each trace consists of events—method call events, assignment events, and synchronization events. The traces generated by an EBP specification are finite, however, an infinite number of traces may be generated by a single EBP as a consequence of using the repetition operator '$*$' or the *while* expression. Further, protocols of communicating components can be composed using the *extended consent operator*; this way, communicating errors (bad activity and no activity) can be detected similarly to original BP.

3.4.1 Syntax of EBP

The syntax of EBP defined in the Extended Backus–Naur form (EBNF) [57] can be found in Appendix A; in this section, we provide a description along with explanation of basic concepts of the EBP syntax.

An EBP specification starts with the *component* keyword followed by three main parts: (i) type definitions, (ii) local variable definition, and (iii) behavior definition (Fig. 3.10).

Within the *types* section, enumeration types of local variables and parameters are defined. The definition of a type takes the form:

type_name = { *value1, value2, ...* }

This defines a type *type_name*; variables of this type can hold only defined values—*value1, value2, ...* There can be any arbitrary (but finite) number of values of a type and any arbitrary but finite number of types defined within an EBP specification.

component *component_name* {

  types {
    types_definition
  }

  vars {
    variable_definition
  }

  behavior {
    behavior_definition
  }

}

Figure 3.10: Structure of the EBP specification

The *vars* section contains definition of local variables. These variables are visible and accessible only from the *behavior_specification* part of this EBP. The variable definition is of the following form:

type_name var_name = initial_value

This way, a local variable *var_name* of the type *type_name* is defined; the initial value of this variable is set to *initial_value*. Again, there can be any finite number of local variables defined within the *vars* section.

The behavior itself is defined inside the *behavior* section. This section contains an expression defining the set of traces, i.e., sequences of events, occurring on the component frame (the set of component provided and required interfaces). The basic entities of this expression are *event tokens*. There exist seven event tokens with the following meaning:

1. !*if.method*(*val1*, *val2* ...) ^ — emitting a method call request with parameters *val1*, *val2*,... ,
2. !*if.method*$ — emitting a method call response,
3. ?*if.method*(*type1 par1*, *type2 par2*, ...) ^ — accepting a method call request and assigning the values the caller used to *par1*, *par2*,...,
4. ?*if.method*(*val1*, *val2*, ...) ^ — accepting a method call request only if the caller used values *val1*, *val2*, ... as parameters,
5. ?*if.method*$ — accepting a method call response,
6. *var* <– *val* — assigning the value *val* to the local variable *var*,
7. @*synchro* — synchronization event.

Each of the event tokens (1)-(7) represents an event. An event is supposed to be atomic, i.e, events are interleaved, but do not overlap.

The events are combined using the following operators: '+' (alternative), ';' (sequence), '|' (parallel composition), and '$*$' (repetition). Furthermore, there are two special constructs: *while* and *switch*. The *while* expression is of the following form:

while ($var == val$) {
  protocol
}

This expression denotes repetition of the *protocol* as long as the value of the *var* local variable is equal to *val*. The value of the variable *var* is tested each time before the *protocol* takes place.

The *switch* expression takes the following syntax:

switch (*var*) {
  val1: { *protocol_1* }
  val2: { *protocol_2* }
  ...
  default: { *protocol_d* }
}

According to the value of variable *var* (either a local variable or a method parameter), the *protocol_i* takes place. The protocol *protocol_d* in the *default* branch, which is optional, takes place if and only if the value of the *var* variable is distinct from all *val1, val2, ...* If the *default* branch is not present, the meaning of such expression is the same as if there would be a *default* branch of the form:

default: { *NULL* }

Further, a syntactic abbreviation referred to as *method call processing* is defined:

$?if.method$ (*type*1 *par*1, *type*2 *par*2, ...) { *protocol* }
stands for:
$?if.method$ (*type*1 *par*1, *type*2 *par*2, ...)^; *protocol*; $!if.method$\$.

Finally, we define two syntactic rules:

1. For each trace generated by the EBP and for each method call request event:
   $!if.method(...)\hat{}$, resp. $?if.method(...)\hat{}$
   there must be a method call response event:
   $?if.method$\$, resp. $!if.method$\$.

2. The method calls has to be well structured, i.e., they are allowed to be nested but they cannot syntactically overlap. For instance, protocols like:
   $!if.method1(\dots)\hat{}$; $!if.method2(\dots)\hat{}$; $?if.method1$\$; $?if.method2$\$
   are not allowed.

3.4.2 Semantics of EBP

The semantics of the Extended Behavior Protocols will be defined using a formalism similar to the Symbolic Transition Graphs with Assignment (STGA) [40]. We refer to our formalism as *Nondeterministic Finite Automata with Assignment* or shortly *NFAA*.

Definition 3.4.1 *Let V be a finite set of variables and H be a finite set of values. Let Comp be a set of expressions of one of the following four forms:* true, false, *$v = h$, and $v \neq h$ where $v \in V, h \in H$. The BoolExp set is defined in the following way: Each $a \in$ Comp is in BoolExp. If $a, b \in$ BoolExp, then $a \wedge b \in$ BoolExp and $a \vee b \in$ BoolExp.*

Informally, the *BoolExp* is the set of finite boolean expressions over members of the sets V and H, where the variables can be only tested for equality or inequality to a concrete value. These comparisons can be combined using logical *and* and logical *or*. We use the standard propositional logic for evaluation of boolean expressions.

Definition 3.4.2 (NFAA) *Let Σ be a finite set of event tokens, V be a finite set of variables with initial values defined, and H be a finite set of values. The* Nondeterministic Finite Automaton with Assignment *NFAA A is a 7-tuple $(S, \Sigma, T, s_0, A, V, H)$ where*

- *S is a finite set of states,*
- *Σ is a finite set of symbols (event tokens),*
- *V is the finite set of variables, and*
- *H is the finite set of values.*
- *T is a transition function: $T : S \times BoolExp \times \Sigma \rightarrow 2^S$,*
- *$s_0 \in S$ is an initial state,*
- *$A \subseteq S$ is a set of accepting (final) states,*

Definition 3.4.3 *A* Computation *of the NFAA A is a finite sequence $p_0, l_0, p_1, l_1, \ldots, l_{n-1}, p_n$ such that $l_0 \ldots l_{n-1} \in \Sigma$, $p_0 \ldots p_n \in S$, $p_0 = s_0$, and $\forall i = 1 \ldots n-1$ holds that $\exists b \in BoolExp : T(p_i, b, l_i) = Q_i$ such that $p_{i+1} \in Q_i$.*

The transition function T defines transitions among states. Each transition is labeled by a *guard* $g \in BoolExp$ and a *label* $l \in \Sigma$, written (g, l).

Definition 3.4.4 *For a given computation, the* value of a variable v in the last state of the computation *is determined (v keeps the value h) by the last transition of the computation with the label $v <- h$. If there is not such a transition in the computation, the value of v is its initial value.*

Definition 3.4.5 *A transition is* enabled *if and only if the guard of the transition evaluates in the starting state of the transition to* true.

Definition 3.4.6 *A computation of the NFAA A is* enabled *if and only if all transitions of the computation are enabled.*

Definition 3.4.7 *The language accepted by the NFAA A is the set of finite sequences $l_0; l_1; \ldots l_n$ such that $l_0, l_1, \ldots l_n \in \Sigma$ and there exists an enabled computation $p_0, l_0, p_1, \ldots l_n$, p_{n+1} such that $p_0 = s_0$ and $p_{n+1} \in A$.*

Before definition of the EBP semantics, we suppose the EBP specification to take a specific form; to achieve this, we transform the input behavior definition in the following way:

Definition 3.4.8 (Parameter unwinding) *Let all method call processings in the EBP E be expressed as the syntactic abbreviation defined above. Each method call processing in E of the form:*

$$?if.method\ (type^1\ par_1,\ type^2\ par_2, \ldots, type^n\ par_n)\ \{\ protocol\ \}$$

is then replaced by:

$$\begin{array}{l} ?if.method\ (type^1_{val_1},\ type^2_{val_1}, \ldots, type^n_{val_1})\ \{\ protocol_{11\ldots1}\ \}\ + \\ ?if.method\ (type^1_{val_2},\ type^2_{val_1}, \ldots, type^n_{val_1})\ \{\ protocol_{21\ldots1}\ \}\ + \\ \vdots \\ ?if.method\ (type^1_{val_{n_1}},\ type^2_{val_1}, \ldots, type^n_{val_1})\ \{\ protocol_{n_1 1\ldots1}\ \}\ + \\ ?if.method\ (type^1_{val_1},\ type^2_{val_2}, \ldots, type^n_{val_1})\ \{\ protocol_{n_1 1\ldots1}\ \}\ + \\ \vdots \\ ?if.method\ (type^1_{val_{n_1}},\ type^2_{val_{n_2}}, \ldots, type^n_{val_{n_n}})\ \{\ protocol_{n_1 n_2 \ldots n_n}\ \} \end{array}$$

where $type^i_{val_1} \ldots type^i_{val_{n_1}}$ are all values of the type $type^i$ for all i and $protocol_{i_1 i_2 \ldots i_n}$ are modifications of the expression protocol such that each switch expression inside the protocol of the form:

$$\begin{array}{l} switch\ (var)\ \{ \\ \quad case_1 :\ \{\ prot_1\ \} \\ \quad case_2 :\ \{\ prot_2\ \} \\ \quad \vdots \\ \quad case_m :\ \{\ prot_m\ \} \\ \quad default :\ \{\ prot_{default}\ \} \\ \} \end{array}$$

is replaced in the $protocol_{i_1 i_2 \ldots i_n}$ by the $prot_l$ such that $case_l = type^j_{val_{i_j}}$ where var is one of par_j for some j. If there is not such l, the default branch is used.

Informally, we first replace each subexpression of method processing containing parameters with the alternative of method processings where the parameters are replaced with all possible combinations of values of corresponding types. Then, each *switch* expression, whose variable is one of the original parameters, is replaced with the branch that corresponds to the value of the parameter accepted in the particular alternative branch.

We are now ready to define the language generated by an EBP specification; we show how the EBP specification corresponds to a NFAA A. Recall that an EBP specification

consists of (i) definition of types, (ii) definition of local variables, and (iii) definition of behavior. The parts (i) and (ii) are referenced in the part (iii)—they do not define any behavior on their own. The language generated by an EBP is the same as the language accepted by the NFAA A, whose construction is described in the following paragraphs.

First, each event e of the form of event tokens (1)-(7) is represented by a primitive automaton having two states—an initial state and a final state. The transition from the initial state to the final state is labeled by $(true, e)$.

Having NFAA A and B for expressions P_A and P_B, respectively, the automaton representing $A + B$ is created by joining the initial state of A with the initial state of B.

The automaton representing $A; B$ is created by adding edges from each accepting state of the automaton A to the initial state of the automaton B. These transitions are labeled by $(true, \lambda)$ denoting a silent (invisible) transition.

In the case of automaton representing $A*$, for each transition t leading from the initial state, a new transition for each final state is added from the final state to the target state of t with the same label as t. As we want the automaton representing $A*$ to accept the empty word, the initial state is added to the set of accepting states.

The automaton representing $A \mid B$ is created as a cartesian product of the automata representing A and B. The automaton representing $A \mid B$ accepts the language containing any arbitrary interleaving of the traces generated by A with the traces generated by B.

In the case of the automaton representing the expression *while* $(var == value)$ { *expr* }, let us denote the automaton representing the *expr* expression as A. Now, in A, the label (b, l) of each transition leading from the initial state is modified to $((var == value) \wedge b, l)$; let us denote the modified automaton A'. Then, the automaton representing the entire *while* expression is the automaton representing $A'*$.

The last construct, for which we have to provide a representation, is the *switch* expression. The general form of this expression follows:

switch (var) {
  $value_1 : \{prot_1\}$
  $value_2 : \{prot_2\}$
  $\vdots$
  $value_n : \{prot_n\}$
  $default : \{prot_{default}\}$
}

Let us denote the automata for $prot_1$, $prot_2$,...,$prot_n$, and $prot_{default}$ by A_1, A_2,...,A_n, $A_{default}$, respectively. For each $i = 1 \ldots n$, let us denote A'_i the automaton that is created from A_i by modifying the guard g of each transition (g, l) leading from the initial state to $(var == value_i) \wedge g$. Next, let us denote $A'_{default}$ the automaton created from $A_{default}$ by modifying the guard of each transition leading from the initial state to $(var \neq value_1) \wedge (var \neq value_2) \wedge \cdots \wedge (var \neq value_n) \wedge g$. Now, the automaton representing the entire *switch* expression is the automaton representing $A'_1 + A'_2 + \cdots + A'_n + A'_{default}$.

3.4.3 Consent composition of EBP

In this section, we describe the semantics of the composition of two EBP specifications via the extended consent composition operator. Since for each EBP specification S there is a NFAA A_S accepting the same language as S generates, we define the consent composition of two EBPs in terms of composition of two NFAAs.

Definition 3.4.9 (Extended consent operator) *Given two EBP specifications X and Y, let us denote $N_X = (S_X, \Sigma, T_X, s_{0_X}, A_X, V_X, H_X)$ an NFAA accepting the same language as X generates, and $N_Y = (S_Y, \Sigma, T_Y, s_{0_Y}, A_Y, V_Y, H_Y)$ an NFAA accepting the same language as Y generates. Let N_X and N_Y are minimal in the number of states. Let us denote ϵe and $\epsilon\oslash$ for each e new events that are not in Σ. Let us refer to the ϵ-events as the error events. Let* error *be a new state not present in $S_X \cup S_Y$. Let a transition t from a state* a *to a state* b *being labeled by (g, l) where g is a guard and l is a label be denoted as $(a, (g, l), b)$. Let us denote by $?e(vars)\hat{}$ an event token of the form $?it.m(val_1, val_2, \ldots, val_n)\hat{}$ where $vars$ represents a parameter list $val_1, val_2, \ldots, val_n$. Further, let us denote by $!e(vars)\hat{}$ an event token of the form $!it.m(val_1, val_2, \ldots, val_n)\hat{}$ where $vars$ represents the same parameter list $val_1, val_2, \ldots, val_n$. Similarly, let $?e\$$ and $!e\$$ denote the event tokens $?it.m\$$ and $!it.m\$$, respectively. Let us denote $\tau e\hat{}$ and $\tau e\$$ the event created by the composition of complementary events, i.e., either $?e\hat{}$ and $!e\hat{}$ or $?e\$$ and $!e\$$. The $\tau e\hat{}$ and $\tau e\$$ events can be denoted as τe. Next, let us denote e and $\overline{e}$ two complementary event tokens, i.e., event tokens differing only in the emitting/accepting the event e—$\overline{?e(vars)\hat{}} = !e(vars)\hat{}$ and vice versa and similarly for the other event tokens representing method call events. Finally, let $@s$ denote an event token $@synchro$. Let $sig(e)$ denotes the method signature of the event token e, i.e., $sig(?it.method(vars)\hat{}) = it.method(vars)$ and similarly for the other event tokens associated with method call events.*

Let J be the set of events of the form $@s$ used for synchronization of N_X and N_Y. Let S be the set of events used for communication between N_X and N_Y except for the events in J.

The NFAA $N_C = (S_C, \Sigma, T_C, s_{0_C}, A_C, V_C, H_C)$ accepting the language of the composition $N_X \nabla_{S,J} N_Y$ using the extended consent operator is defined in the following way:

$$
\begin{aligned}
&S_C = S_X \times S_Y \cup \text{error}\\
&s_{0_C} = (s_{0_X}, s_{0_Y})\\
&A_C = A_X \times A_Y \cup \text{error}\\
&V_C = V_X \cup V_Y\\
&H_C = H_X \cup H_Y
\end{aligned}
$$

A transition is an element of T_C if and only if it is deduced by one of the following rules, in which $f, g \in BoolExp$:

$$((q_1, q_2), (f, ?e(vars)\hat{}), (q_1', q_2)) \in T_C \text{ if } (q_1, (f, ?e(vars)\hat{}), q_1') \in T_X \text{ and } e(vars) \notin S$$

$$((q_1, q_2), (f, !e(vars)\hat{}), (q_1', q_2)) \in T_C \text{ if } (q_1, (f, !e(vars)\hat{}), q_1') \in T_X \text{ and } e(vars) \notin S$$

$$((q_1, q_2), (f, ?e(vars)\$), (q_1', q_2)) \in T_C \text{ if } (q_1, (f, ?e(vars)\$), q_1') \in T_X \text{ and } e(vars) \notin S$$

$((q_1, q_2), (f, !e(vars)\$), (q_1', q_2)) \in T_C$ *if* $(q_1, (f, !e(vars)\$), q_1') \in T_X$ *and* $e(vars) \notin S$

$((q_1, q_2), (f, @s), (q_1', q_2)) \in T_C$ *if* $(q_1, (f, @s), q_1') \in T_X$ *and* $@s \notin J$

$((q_1, q_2), (f, ?e(vars)\hat{}), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, ?e(vars)\hat{}), q_2') \in T_Y$ *and* $e(vars) \notin S$

$((q_1, q_2), (f, !e(vars)\hat{}), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, !e(vars)\hat{}), q_2') \in T_Y$ *and* $e(vars) \notin S$

$((q_1, q_2), (f, ?e(vars)\$), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, ?e(vars)\$), q_2') \in T_Y$ *and* $e(vars) \notin S$

$((q_1, q_2), (f, !e(vars)\$), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, !e(vars)\$), q_2') \in T_Y$ *and* $e(vars) \notin S$

$((q_1, q_2), (f, @s), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, @s), q_2') \in T_Y$ *and* $@s \notin J$

$((q_1, q_2), (f, t), (q_1', q_2)) \in T_C$ *if* $(q_1, (f, t), q_1') \in T_X$, *where* t *is either* τe *or an error token*

$((q_1, q_2), (f, t), (q_1, q_2')) \in T_C$ *if* $(q_2, (f, t), q_2') \in T_Y$, *where* t *is either* τe *or an error token*

$((q_1, q_2), (f \wedge g, \tau e), (q_1', q_2')) \in T_C$ *if* $(q_1, (f, e), q_1') \in T_X$ *and* $(q_2, (g, \overline{e}), q_2') \in T_Y$ *and* $sig(e) \in S$

$((q_1, q_2), (f, \epsilon e), error) \in T_C$ *if* $(q_1, (f, !e(vars)\hat{}), q_1') \in T_X$ *and there is no* $q_2' \in S_Y$ *s.t.* $(q_2, (g, ?e(vars)\hat{}), q_2') \in T_Y$ *and* $e(vars) \in S$

$((q_1, q_2), (f, \epsilon e), error) \in T_C$ *if* $(q_1, (f, !e(vars)\$), q_1') \in T_X$ *and there is no* $q_2' \in S_Y$ *s.t.* $(q_2, (g, ?e(vars)\$), q_2') \in T_Y$ *and* $e(vars) \in S$

$((q_1, q_2), (f, \epsilon e), error) \in T_C$ *if* $(q_2, (f, !e(vars)\hat{}), q_2') \in T_Y$ *and there is no* $q_1' \in S_X$ *s.t.* $(q_1, (g, ?e(vars)\hat{}), q_1') \in T_X$ *and* $e(vars) \in S$

$((q_1, q_2), (f, \epsilon e), error) \in T_C$ *if* $(q_2, (f, !e(vars)\$), q_2') \in T_Y$ *and there is no* $q_1' \in S_X$ *s.t.* $(q_1, (g, ?e(vars)\$), q_1') \in T_X$ *and* $e(vars) \in S$

$((q_1, q_2), (true, \epsilon\oslash), error) \in T_C$ *if*
(*there is no* $(q_1, (f, t), q_1') \notin T_X$ *s.t.* t *is* $!e(vars)\hat{}$, $!e(vars)\$$, $?e(vars)\hat{}$, $?e(vars)\$$, $@s$ *where* $@s \notin J$, *or an error token) and*
(*there is no* $(q_2, (g, t), q_2') \notin T_Y$ *s.t.* t *is* $!e(vars)\hat{}$, $!e(vars)\$$, $?e(vars)\hat{}$, $?e(vars)\$$, $@s$ *where* $@s \notin J$, *or an error token) and*
($q_1 \notin A_X$ *or* $q_2 \notin A_Y$)

$((q_1, q_2), (f \wedge g, @s), (q_1', q_2')) \in T_C$ *if* $(q_1, (f, @s), q_1') \in T_X$ *and* $(q_2, (g, @s), q_2') \in T_Y$ *and* $@s \in J$

3.4.4 EBP inversion

As well as in the case of BP, the extended consent operator is used for verification of both horizontal and vertical compliances. Since the principle of the vertical compliance verification is the again based on composing the inverted frame protocol of the component with its architecture protocol, we need to define the *inversion* of an EBP specification.

Definition 3.4.10 (Inverted EBP) *Let A be an EBP specification and A_b its behavior part. The* inverted EBP A^{-1} *is created from A by modification of the behavior part A_b in the following way:*

1. *A_b is transformed according to Def. 3.4.8.*

2. *Each event of the form $!e(vars)\hat{}$ is replaced by $?e(vars)\hat{}$ and vice versa. Similarly, $!e(vars)\$$ is replaced by $?e(vars)\$$ and vice versa.*

Chapter 4

Transformation of EBP into Promela

The Behavior Protocol Checker described in Sect. 2.3.4 is our proprietary tool used for checking compliance and absence of composition errors. It is able to handle state spaces of the size in the order of magnitude of 10^8 states, which is not sufficient in all cases.

The Spin model checker [32] is a state-of-the-art explicit model checker featuring LTL checking abilities, bit-state hashing, and quite friendly user interface. It allows traversal of state spaces of several orders of magnitude higher sizes than behavior protocol checker does. Additionally, there are a number of extensions of Spin, e.g. dSpin [18] extending the Promela language by functions, exceptions, etc. Therefore, we decided to translate the specification in EBP into Promela—the input language of the Spin model checker.

4.1 Basic approach

When choosing the basic approach for translation of the extended behavior protocols into Promela, we had to face, above all, the problem of the different handling of nondeterminism in EBP and Promela. In the case of nondeterminism in EBP, i.e., there are more alternative branches with the same prefix allowed, their semantics allows each time all possible suffices. Conversely, in Promela, in case of this kind of nondeterminism in a process, one executable statement is selected and the statements following this one are sequentially executed. The decision about the suffix is done at the beginning of the alternative branch. Mapping of EBP into Promela cannot thus be done in a straightforward way.

A way to avoid the problem with nondeterminism described above is to use deterministic EBP equivalent to original EBP, i.e., generating the same set of traces. Disallowing nondeterministic EBP at the side of user input is not a good idea as it restricts the language in an inconvenient way; moreover, in cases of need for parallel operator, some behavior could not have been specified.

According to these facts, we decided to:

1. Transform each EBP of the specification into a NFAA,

2. transform this NFAA into an equivalent deterministic finite automaton with assignments (DFAA),

3. transform the DFAA into a Promela model, and

4. use the Spin model checker to verify the absence of the communication errors within the Promela model.

Even if the transformation of NFAA (built up according to EBP) to DFAA usually does not raise the number of states in a significant way, application of a minimization algorithm to DFAA can further improve performance of the consequent checking process.

4.2 Modeling composition

When modeling the composition of two behavior protocols (using the consent operator in our case), we need to synchronize execution of complementary events to form a single internal τ-event. Using the aforementioned approach where each EBP is represented by a Promela process, this implies a kind of interprocess communication.

In Promela, the basic mean for interprocess communication are message channels. Message channels are bidirectional; unless declared as exclusive, several processes may send data to a channel and several processes may read the data sent earlier. There are two modes of the message channels in Promela—in the first mode, the send-message command gets blocked in the case it cannot be sent immediately (either there is no process waiting for this message, or the message buffer associated with this channel is full), while in the other mode, the message gets lost in such a case. In behavior protocols, on the contrary, we want each emit event that cannot be accepted immediately to cause the bad activity error. Thus, modeling such behavior would require incorporation of an algorithmic mechanism (e.g. message counting) to detect bad activity errors. Therefore we decided to use shared variables for communication between processes.

Each method of an exported interface is associated with two boolean variables representing the events of method call request and response. Initially, all variables corresponding to these events are set to zero. Additionally, there is a shared variable *lock* used for mutual exclusion of particular method calls, because each internal event requires execution of at least one statement in each of the processes involved in the communication.

To emit an event, the *lock* variable has to be set to zero. Each time a component emits an event, it first set the value of the *lock* variable to one, then it sets the variable corresponding to the event to one. Because the *lock* is acquired, only accept events may be now performed to form, together with the event just being emitted, a τ-event. The process accepting the emit event resets the value of the variable corresponding to this event as well as the *lock* variable to zero. If there is no component able to accept the emitted event, deadlock occurs. Since the *lock* is acquired at the time of the deadlock, bad activity error is detected. If the lock is not acquired at the moment of a deadlock, either no activity has

occurred or the execution has been successfully accomplished and the behavior specification is thus communication error free.

The communication errors are detected as a Promela deadlock; then a variable representing the current event being performed is checked and according to the value either successful accomplishing, bad activity, or no activity is reported. If a communication error is discovered during verification of the Promela model via the Spin model checker, a trail through the state space is stored within a file (a feature of Spin) and Spin is run again in the simulation mode to obtain the corresponding EBP error trace.

4.3 Modeling data

There are two kinds of data in EBP—the state variables and method parameters. As variables of both of these data kinds may hold symbolic values, they have a lot in common regarding the translation into Promela. The main difference between state variables and method parameters from the semantics point of view is that the value of a method parameter is assigned at a moment of accepting a method call (accepting a request event) and cannot change, while the value of a state variable may change anytime (even "asynchronously" as a consequence of an event in a parallel branch). In the following paragraphs, we describe the solution of data treatment for both method parameters and state variables.

4.3.1 State variables

State variables can be seen as method parameters that may change their value during method call processing by a parallel process. Or, similarly, the method parameters can be seen as state variables that cannot change during method call processing. Since a state variable is accessible only within the component, i.e., only from a single Promela process, we decided to represent it as a variable.

4.3.2 Method parameters

Method parameters can be viewed as a simpler case of state variables. As we can argue the same way as in the previous case above and to keep the data treatment in the model unified, we decided to use the global variables for transfer of parameter values as well.

4.4 Modeling multisynchronization

The multisynchronization events are modeled again using shared variables. This approach is suitable here as it is, unlike message channels, symmetric.

Technically, there is a byte variable *syncvar* associated with each multisynchronization event *@sync*; the *syncvar* is initialized with the number of processes containing this multisynchronization event. All the processes representing the EBP of components using *@sync* are ordered (in the order of composition). If a process is able to execute the *@sync* event,

it tries to do so via decreasing the variable *syncvar* after all the smaller (in the ordering) processes have performed the event and waiting until all the other processes greater than this process execute this event. If a process is not able to perform the *@sync* event, the *syncvar* is increased again by each process that has previously decreased *syncvar* to return the state before multisynchronization (kind of a "rollback"). Conversely, after all other processes perform the *@sync* event, the variable is reset by the smallest process (in the ordering) to the initial number. A simple decreasing and a test of the associated variable (*var == value*) is not sufficient, as the *@sync* event may appear in a cycle within a single specification.

4.5 Example

In this section, to illustrate the translation process, we present a translation of a simple EBP specification into Promela. The specification under consideration is listed in Fig. 4.1 and describes behavior of the *LightDisplay* component from the CoCoME project [64].

The protocol specifies that the associated component is able to accept two instances of the *onEvent* method—one instance with the *EMEnabled* parameter value and the other with the *EMDisabled* value. Each time, it assigns the corresponding value to the *state* local variable.

```
component LightDisplay {

    types {
        states = {LIGHT_ENABLED, LIGHT_DISABLED}
    }

    vars {
        states state = LIGHT_ENABLED
    }

    behavior {
        ?LightDisplayControllerEventHandlerIf.onEvent(EMEnabled) {
            state <- LIGHT_ENABLED
        }*
        |
        ?LightDisplayControllerEventHandlerIf.onEvent(EMDisabled) {
            state <- LIGHT_DISABLED
        }*
    }
```

Figure 4.1: EBP specification of the *LightDisplay* component.

This specification corresponds to the FSM with nine states depicted in Fig 4.2. For the sake of readability, the labels are present only on the topmost and leftmost branches (the missing labels are the same as the corresponding labels that are present).

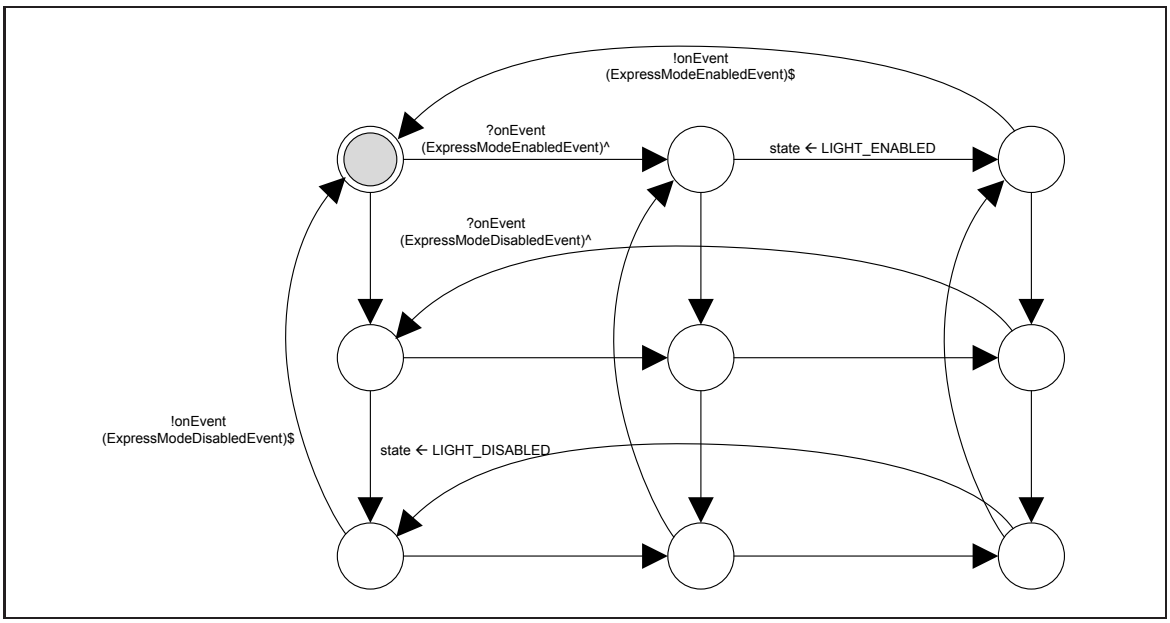


Figure 4.2: FSM corresponding to the *LightDisplay* EBP specification.

This automaton is then translated into the Promela language; it takes the form listed in Fig. 4.3. In addition to this code fragment, there are data structures containing the information about transitions that are stored in an array (not listed here). Furthermore, the fragment take advantage of several macros (*emit*, *accept*, ...) simplifying the Promela code.

```
proctype LightDisplay()
{
int st = 8; /* the initial state */

do
::emit(st,tr0[Pc_0->st*6+0],gr0[Pc_0->st*6+0],S0);
::accept(st,tr0[Pc_0->st*6+1],gr0[Pc_0->st*6+1],E0,
      "LightDisplayControllerEventHandlerIf.onEvent(EMEnabled)$");
::assign(st,tr0[Pc_0->st*6+2],gr0[Pc_0->st*6+2],V0_state,LIGHT_ENABLED);
::emit(st,tr0[Pc_0->st*6+3],gr0[Pc_0->st*6+3],S1);
::accept(st,tr0[Pc_0->st*6+4],gr0[Pc_0->st*6+4],E1,
      "LightDisplayControllerEventHandlerIf.onEvent(EMDisabled)$");
::assign(st,tr0[Pc_0->st*6+5],gr0[Pc_0->st*6+5],V0_state,LIGHT_DISABLED);
::final(fn0[Pc_0->st])
od;
DONE:
skip;
}
```

Figure 4.3: Promela fragment of the *LightDisplay* component specification

Chapter 5

Evaluation

5.1 BP vs. EBP comparison

As stated in Sect. 2.4, to our knowledge, except for BP, no formal method has been successfully used for behavior modeling and verification of a real-life-sized component architecture (composed of e.g. 20 components). However, there is an ongoing project CoCoME [64] still running at the time of writing this thesis which aims at comparison of various modeling approaches. Since the results of the project are not yet available, we are not aware of how many of the participants aim at not only modeling but also verification of component behavior; hence, in this chapter, EBP is thoroughly compared with the original Behavior Protocols only.

The application being the subject of the CoCoME contest aims at providing infrastructure for an enterprise company including a central stock, an items database, and several stores with cash desks and customers. The application structure is depicted in Fig. 5.1. For comparison, we have chosen the *CashDeskApplication* component, whose specification is, when using BP or EBP, the most complex one. The complete specification of this component using BP is listed in Appendix C, while the EBP specification can be found in Appendix D.

As clear from Fig. 5.1, the *CashDeskApplication* component is a part of each *CashDesk* component; it is responsible for controlling particular sales via communication with other parts of the *CashDesk* component using buses. In the form of method calls, it receives information about events reflecting various phases of a sale (bar code scanning, finishing of the sale, opening/closing the cash box, etc.) as well as the switching between normal and express modes (when the customers may use the associated cash desk only when buying few items and they are required to pay cash). Next, as a reaction on the incoming events, it notifies the *StoreServer* about sold items. Finally, in case of a payment using a credit card, it is responsible for communication with the bank.

We compare the specifications with respect to the following criteria:

(C1) Format of the specification, i.e., its length, readability, and complexity of error fixing.

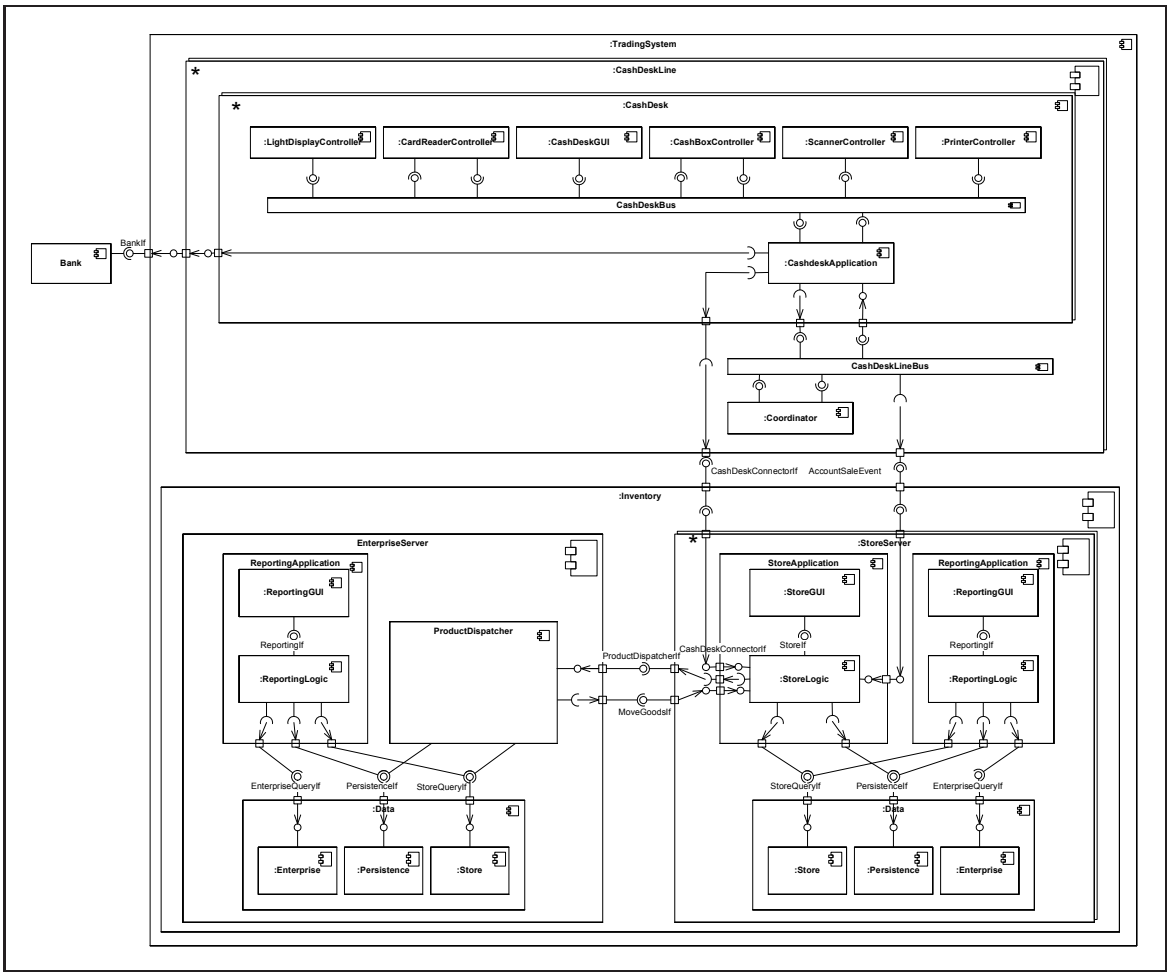


Figure 5.1: Architecture of the CoCoME application

(C2) Preciseness of the specification, i.e., how much detailed with respect to the real (implementation) behavior the specification is.

(C3) Verification efficiency, i.e., the time and memory requirements of the verification.

(C4) Verifiable properties, i.e., what kind of properties can be checked when using a given modeling language.

Specification format (C1)

BP The description of the component behavior using the original BP is, excluding comments, about 160 lines long (7kB). This is caused above all by the fact that several parts are repeating within the specification; because of this, the error fixing is hard and often several places of the BP have to be modified to fix a single bug. However, a specification of that size can be still managed (debugged, updated) with a reasonable effort.

EBP The EBP specification of the *CashDeskApplication* component is slightly shorter (about 150 lines, 5.6kB). However, it is more readable due to the following facts: (i) it is structured in a better way, and (ii) repetition of specification parts is significantly reduced—in fact, it can be entirely eliminated via macros, which are directly supported in EBP[1]. This enables easier and faster specification management.

The length on the complete CoCoME application specification is in both cases about 35kB, however, the EBP version is, as apparent from comparison of Appendix C and D, in many cases more readable and easier to comprehend.

Preciseness of the specification (C2)

BP As described in Sect. 1, Behavior Protocols entirely abstract from data, i.e., no notion of values and variables are present within the specification. However, as argued in Sect. 2.4, the real behavior of a component is often data dependent; in particular, processing of a method call often depends on the parameters passed to the method "implementation". Therefore, as a consequence, the specification of component behavior in BP often introduces nondeterminism, which causes communication errors when composed with specification of other components. In the case of the *CashDeskApplication*, to be able to model the behavior at a reasonable level of abstraction, we decided to modify its method names to reflect acceptance of various events through the bus. Although this seems to be a straightforward and correct solution, we run into problems when comparing the specification with an implementation [53] where the method names within the specification have to conform to the implementation. Then, we had to maintain two versions of a specification.

After receiving the information that the express mode should be enabled, the *CashDeskApplication* component sends this information using *CashDeskBus* to several other subcomponents of the *CashDesk* component. If the *CashDesk* is already in the express mode, the information is accepted by the *CashDeskApplication*, but it is ignored. Since the components being notified about a change of the mode only accept this information without any (external) reaction, the mode itself is not modeled in the specification. Conversely, after receiving the information about enabling the express mode, the specification of the *CashDeskApplication* component models nondeterministic choice between forwarding and ignoring this information.

Except for the aforementioned issues, the component behavior in the sense of BP is modeled correctly.

EBP Unlike in case of BP, an EBP specification, taking advantage of method parameters, allows modeling of the messages accepted by the *CashDeskApplication* component with correct (with respect to the implementation) method names. Hence, the specification can be directly reused for code-against-specification verification.

As to modeling the express mode, it is modeled in the same way as described in the previous paragraph. Even though the mode switching could be modeled precisely in EBP,

[1] We have not used macros in the specification to keep it easier to comprehend for the sake of this thesis.

it would not enable capturing any property of interest. Therefore we decided to model at this level of abstraction.

Similarly to BP, the EBP specification of the *CashDeskApplication* component is up to omitting the mode modeling discussed above precise.

Verification efficiency (C3)

BP Since the BPChecker tool ran out of memory while verifying some parts of the CoCoME application, we used *dChecker* for verification of the application modeled in the Fractal component model instead. The *dChecker* tool is a successor of BPChecker; it is written in Java and supports distribution of the task among several computers. It has been developed in parallel with the *ebp2promela* tool, but it supports the original BP only. It is about an order of magnitude faster than BPChecker while requiring less memory. Verification of the composition correctness was done using a decent PC[2] and took slightly more than 3 minutes; the memory consumption of the most resource demanding verification were about 1.2GB. In Fig. 5.2, the time consumption and state space size for the verification of (vertical) compliance of the specific composite components as well as the state space sizes are listed.

Component	**Time [s]**	**States**
CashDesk	9.2	483,797
CashDeskLine	24.5	1,562
StoreApplication	6.9	63,900
Data	45.9	124,416
ReportingApplication	0.2	17
StoreServer	40.1	297,024
EnterpriseServer	39.5	512
Inventory	0.2	121
TradingSystem	18.0	51,558
Total time	184.5	1,022,907

Figure 5.2: Durations and state space sizes of compliance verification the CoCoME composite components when using original BP.

EBP For verification of EBP, we designed and implemented a tool *ebp2prom* [66] translating EBP specifications into Promela. Then, the Promela model is verified using the Spin model checker [32]. We used the hardware and software configuration as in the BP case for running the tests. The duration of transformation (T. time), verification of (vertical)

[2]PC 2x Intel Core2 Duo (dual core) processor with 4MiB L2 cache and 4GiB operational memory running the Gentoo Linux version 2006.1 and Spin version 4.2.9

compliance (V. time)[3] and entire verification (Time)[4] of particular composite components again together with the state space sizes of the corresponding Promela models are listed in Fig. 5.3.

Component	T. time [s]	V. time [s]	Time [s]	States
CashDesk	41.5	46.1	97.1	3,335,950
CashDeskLine	59.6	1.0	71.2	3,912
StoreApplication	37.5	3.2	50.1	378,466
Data	159.8	9.8	203.6	1,119,740
ReportingApplication	0.3	1.0	1.6	39
StoreServer	154.3	15.3	198.9	2,064,870
EnterpriseServer	151.9	1.0	178.3	2,241
Inventory	0.5	1.0	1.9	386
TradingSystem	54.0	1.4	66.4	71,279
Total time	659.4	79.8	869.1	6,781,743

Figure 5.3: Durations and state space sizes of compliance verification the CoCoME composite components when using EBP.

Compared to the values in Fig. 5.2, the growth of the state space is caused by data; however, apparently, it is not very significant and the verification times are still acceptable for an application of this complexity. On the other hand, the duration of the actual verification is significantly shorter. Also, since the EBP specifications of components are transformed one after another (no behavior composition is involved at this stage), the state space explosion is not an issue in the transformation. Moreover, Spin is able to handle much larger state spaces than BPChecker and dChecker. In fact, when bitstate hashing method is used in Spin, there is no state space size limit; then, however, verification reliability decreases with the growth of the state space.

Verifiable properties (C4)

BP Behavior protocols enable verification of absence of communication errors (bad-activity, no-activity, divergence, and unbound-requires error) at each particular level of component nesting and verification of (vertical) compliance. Verification of other properties (e.g. LTL) is generally possible, but it would require extension of the tools.

[3]We used the bitstate-hashing mode with the lowest hashfactor value of 1287 (in the case of the CashDesk component); hashfactor values greater than 100 are considered as denoting very reliable results.

[4]The transformation time is the time of running of the transformation tool *ebp2prom*; note that the resulting Promela model is subsequently transformed by Spin to a model in the C language, and the C model is compiled using gcc and finally run. The time requirements for the last two transformations (i.e., the generation of the C code and its compilation) are omitted here.

EBP As well as in the case of BP, using EBP as the specification platform enables verification of composition correctness in the sense of absence of the composition errors; as argued in Sect. 4, detection of divergence is not supported by our tool. Furthermore, since the Spin model checker is used, model checking of an arbitrary LTL_{-X} property is possible (in addition to checking for absence of communication errors).

Similar to BP, EBP do not focus on direct support for dynamic reconfiguration—this aspect is indirectly supported via re-verification of the specification parts affected by the change. This way, of course, component behavior compliance only before and after the reconfiguration can be checked; verification of the reconfiguration process itself is not supported.

There is also no support for reasoning about performance and reliability aspects of the behavior of components in the sense of e.g. parametric contracts [55]. However, such a general specification platform would probably, in consequence of state explosion, exceed the abilities of today's computers, and thus become practically hard to use.

5.2 Comparison to other approaches

There are several formalisms aiming at specification and verification of component behavior; they focus on verification of various properties.

Parametric contracts

Parametric contracts [55] and their extensions present in the Palladio component model [8] aim at prediction of performance and reliability of components, in particular performance and reliability of the services (methods) provided by the component. The Service-Effect specification defines the possible set of reactions in the sense of calling external (required) methods of other components as a reaction to a call of a provided service (method). The allowed sequences of method calls on each provided interface are specified by a provides-FSM. The interplay of all interfaces of the component frame is not modeled, since it is not necessary for performance prediction. However, we believe that it is necessary for evaluation of the compliance relation.

Component interaction automata

Component interaction (CI) automata [10] provide a general framework upon which a more specific theory can be elaborated. The CI automata allow general composition of particular automata that can be fine-tuned for a concrete theory. They do not directly support specification of data values (parameters and variables). The checking of component compliance can be modeled by defining a custom composition. The evaluation would require implementation of a tool performing the composition. The CI automata do not support any reliability nor performance reasoning.

Promela

Promela [32] as a general specification language for verification of properties of parallel processes can be used also for component behavior specification and compliance verification. However, to model the same semantics (the trace semantics) as in EBP, the model is very hard to write by hand and read. However, using Promela as the output language turns out to be beneficial.

Darwin (Tracta)

In Darwin [43], the monadic form of the π-calculus is used for specification of component behavior. Using this approach, it is possible to reason about dynamic architectures. The behavior of only primitive components is specified, while the behavior of composite components is modeled by a composition of behavior of their subcomponents. Therefore, Darwin supports verification of deadlock freedom only—it does not support the verification of (vertical) compliance. Conversely, the verification framework Tracta provides a developer with a tool able to check various properties of the specification of the entire component application (being the parallel composition of particular specifications). It has been proven, that the monadic form of the π-calculus is strong enough to model any arbitrary data structure, however, since the direct support for data within Tracta is missing, it will be probably hard for an application designer to include them in the specification. There is also no direct support for reliability and performance reasoning in Tracta.

Wright

Wright [3] is not tied to a specific component model, it is rather a modeling language, which includes modeling of component behavior in CSP, for component based systems. The behavior is modeled using *interacting protocols* (being a subset of CSP). Similar to Tracta [43], behavior of only basic entities (component interfaces, roles, and connectors—glues) is provided by an application designer. The behavior of composite components is not provided by an application designer, but it is modeled as a parallel composition of behavior of its subcomponents; here, only deadlock freedom may be verified. Hence, again, the verification of the compliance relation is not supported. Wright provides a direct support for reasoning about data in a symbolic way, which may be used for modeling both method parameters and return values. Again, no direct support for reliability and performance reasoning is present.

LOTOS

LOTOS [70] is a specification language used e.g. in [6]. As well as in majority of other approaches, an application designer is responsible for providing behavior specification of only primitive components. The behavior of composite components is modeled as a composition of subcomponent specifications—vertical compliance cannot be verified. The Enhanced LOTOS (E-LOTOS) adds a support for reasoning about time (i.e. performance),

and modifies the approach to data and type specification to make it easier to use. Although originally targeted to protocol and service specifications, it can be advantageously used for specification of component behavior [6]. Since a specification in E-LOTOS is quite detailed, to our knowledge, most models of real-life-sized applications suffer from the state space explosion problem.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, we presented Extended Behavior Protocols—a new language for specification of component behavior. EBP are based on BP and, similarly to BP, they enable verification of behavior compliance of communicating components.

To our knowledge, no other formalism focused on component behavior specification and compliance verification was used in the scope of a real-life-sized application except for BP. However, when using BP for behavior specification of the component application (aimed at providing wireless Internet access at airports) [1], several issues arose in terms of its expressive power. Therefore, we decided to extend BP to address them. The resulting EBP language is based on expressions determining again finite automata; moreover, the size of automata of an EBP specification is still comparable with the automata determined by the corresponding BP specification. As a positive consequence, verification of the behavior compliance in EBP can be still evaluated in a reasonable time and space. For this purpose, a compliance verification algorithm based on transformation of the EBP specification into Promela was designed. As a proof of the concept, the transformation algorithm was implemented in the *bp2prom* tool [66].

When verifying compliance of the CoCoME components, the actual verification of the EBP specification in Spin was significantly faster, however, total time including the transformations increased to 470% when compared to the case of BP. Also, the state space size of the EBP specification was higher than the corresponding BP specification (660%). On the positive side, the expressive power of EBP reduces the size of specification and makes it more accurate (by capturing method parameters and component states, which can be expressed by an enumeration type). Also, large state spaces are efficiently traversed by Spin in a reasonable time. So, in a result, based on the CoCoME case study, EBP turn out to be a better specification platform than BP.

Overall, all goals (G1)–(G4) stated in Chapter 2 were fulfilled; in particular, the original BP were extended to model method parameters, component states, and synchronization of multiple events, while making the specification more readable ((G1) and (G2)), after

transformation to the Promela language, the verifiable properties include LTL_{-X} properties (G3), and the efficiency of the verification has been greatly improved (comparing to BPChecker) via using Spin as a model checker (G4).

6.2 Future work

As a future work, we plan to implement a new tool to check the Java code of primitive components against their EBP specification similar to the verification of the Java code against the original BP [53].

Next, we plan to use EBP for specification of other real-life-sized applications (like CoCoME [64]) to obtain a better evaluation of the benefits of EBP as a new behavior specification language aimed at real-life applications.

Finally, we would like to focus on articulating important properties (expressed in e.g. LTL_{-X}) such that they would be, besides absence of the communication errors already captured by the consent operator, of the designer/developer interest and whose validity could be verified.

Bibliography

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component reliability extensions for Fractal component model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.

[2] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[5] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathmatiques de l'Informatique"*, pages 35–68, 1982.

[6] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)*, August 2006.

[7] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[8] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the Palladio component model. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 54–65, New York, NY, USA, 2007. ACM Press.

[9] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theorethical Computer Science*, 37:77–121, 1985.

[10] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes*, 31(2):4, 2006.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[12] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.

[13] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134, New York, NY, USA, 2004. ACM Press.

[14] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.

[15] I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata modelling language. Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic, October 2006.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[18] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[19] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[20] A. Evans, R. France, K. Lano, and B. Rumpe. Developing the UML as a formal modelling notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 297–307, 1998.

[21] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(1):219–236, 1989.

[22] H. Garavel. Compilation of lotos abstract data types. In *FORTE '89: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 147–162, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[23] H. Garavel. Binary Coded Graphs: Definition of the BCG Format. Technical Report SPECTRE C28, Laboratoire de Génie Informatique — Institute IMAG, Grenoble, January 1991.

[24] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. Technical Report 254, INRIA, Rhone-Alpes, December 2001.

[25] H. Garavel and J. Sifakis. Compilation and verification of Lotos specifications. In Logrippo, R. L. Probert, and H. Ural, editors, *Proc. 10th International Symposium on Protocol Specification, Testing and Verification*, Amsterdam, 1990. Elsevier (North-Holland).

[26] D. Giannakopoulou, J. Kramer, and S. C. Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6(1):7–35, 1999.

[27] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[28] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[29] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.

[30] P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[31] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International (UK) Ltd., 1985.

[32] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[33] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.

[34] P. Jezek, J. Kofron, and F. Plasil. Model checking of component behavior specification: A real life experience. In *Electronic Notes in Theoretical Computer Science*, volume 160, pages 197–210, August 2006.

[35] J. Kofron. Enhancing behavior protocols with atomic actions. Technical Report 2005/8, Dep. of SW Engineering, Charles University in Prague, 2005.

[36] J. Kofron. Extending Behavior protocols with data and multisynchronization. Technical Report 2006/10, Dep. of SW Engineering, Charles University in Prague, October 2006.

[37] J. Kofron. Software component verification: On translating Behavior protocols to Promela. Technical Report 2006/11, Dep. of SW Engineering, Charles University in Prague, October 2006.

[38] J. Kofron. Checking software component behavior using Behavior Protocols and Spin. In *Proceedings of Applied Computing 2007*, pages 1513–1517, Seoul, Korea, March 2007.

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

[40] H. Lin. Symbolic transition graph with assignment. In *International Conference on Concurrency Theory*, pages 50–65, 1996.

[41] M. Mach. Formal verification of behavior protocols. Master's thesis, Department of SW Engineering, Charles University in Prague, Czech Republic, 2003.

[42] M. Mach, F. Plasil, and J. Kofron. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 6(1):22–30, 2005.

[43] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[44] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[45] R. Mateescu and H. Garavel. Xtl: A meta-language and tool for temporal logic model-checking, 1998.

[46] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.*, 46(3):255–281, 2003.

[47] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[48] B. Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[49] R. Milner. *Communication and Concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[50] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[51] R. D. Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Proceedings of the LITP Spring School on Theoretical Computer Science*, pages 407–419, London, UK, 1990. Springer-Verlag.

[52] M. Nivat. Sur la synchronisation des processus. Thomson-CSF II (1979) 899-919, 1979.

[53] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 133–141, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[54] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

[55] R. Reussner, I. Poernomo, and H. W. Schmidt. Reasoning about software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, pages 287–325. Springer, 2003.

[56] V. Roy and R. de Simone. Auto/autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer-Verlag.

[57] R. S. Scowen. Extended BNF — A generic base standard. *Software Engineering Standards Symposium*, 1993.

[58] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *ICALP '94: Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, pages 304–315, London, UK, 1994. Springer-Verlag.

[59] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[60] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications (Monographs on Discrete Mathematics and Applications)*. Society for Industrial & Applied Mathematics, 2000.

[61] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, October 1994.

[62] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th International Computer Aided Verification Conference*, pages 59–70, 1993.

[63] OMG Corba Component Model Specification, `http://www.omg.org/technology/documents/formal/components.htm`.

[64] Modelling Contest: Common Component Modelling Example, `http://agrausch.informatik.uni-kl.de/CoCoME`.

[65] Microsoft COM Technology, `http://www.microsoft.com/com`.

[66] EBP2Prom — A tool translating EBP specifications into Promela, `http://dsrg.mff.cuni.cz/~kofron/phd-thesis/tools.zip`.

[67] Sun Enterprise Java Beans, `http://java.sun.com/products/ejb`.

[68] Failures Divergences Refinement: User Manual and Tutorial, formal systems (europe) limited.

[69] Graphviz — open source graph drawing software, `http://www.graphviz.org/`.

[70] ISO: Information Processing Systems — Open Systems Interconection. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, 1989.

[71] The SOFA project, `http://sofa.objectweb.org`.

[72] Tcl/Tk Tool Command Language — a dynamic programming language.

[73] Object management group: Unified Modeling Language, `http://www.uml.org`, 2005.

Appendix A

Syntax of Extended Behavior Protocols

In this appendix, the syntax of Extended Behavior Protocols is described in the EBNF format.

```
ebp = "component", component_name, "{",
            [ types_def ],
            [ variables_def ],
            behavior_def,
          "}";

component_name = idf;

idf = char, [ { digit | char }  ];

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

char = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
       "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
       "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" |
       "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
       "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_" | "-" | "<" |
       ">";

types_def = "types", "{", type, [ { type } ], "}";

type = idf, "=", "{", idf, [ { "," idf } ], "}";

variables_def = "vars", "{", var, [ { var } ], "}";
```

```
var = idf, idf, "=", idf;

behavior_def = protocol;

protocol = alt;

alt = seq, [ { "+", seq } ];

seq = par, [ { ";", par } ];

par = rep, [ { "|", rep } ];

rep = term, "*";

term = "(", protocol, ")" |
       switch |
       while |
       event;

switch = "switch", "(", idf, ")", "{", switchbody, "}";

switchbody = case, [ { case } ];

case = idf, ":", "{", protocol, "}";

while = "while", "(", idf, "==", idf, ")", "{", protocol, "}";

event = ereq | eres | areq | ares | assign | sync;

ereq = "!", method, "(", parlist, ")", "^";

method = idf, ".", idf;

parlist = [ idf, [ { ",", idf } ] ];

eres = "!", method, "$";

areq = "?", method, "(", ( fparlist | parlist ), ")", "^";

fparlist = [ idf, idf, [ { ",", idf, idf } ] ];

ares = "?", method, "$";
```

```
assign = idf, "<-", idf;

sync = "@", idf;
```

Appendix B

Behavior protocol of the IpAddressManager component

$$
\begin{array}{l}
(\\
\quad ?IDhcpServerLifetimeController.Start\hat{}; \\
\quad !IListenerLifetimeController.Start\hat{}; \\
\quad [?IListenerLifetimeController.Start\$, !IDhcpServerLifetimeController.Start\$] \\
) \\
; \\
(\\
\quad (\\
\quad\quad (\\
\quad\quad\quad (\\
\quad\quad\quad\quad ?IDhcpListenerCallback.RequestNewIpAddress\{ \\
\quad\quad\quad\quad\quad !IIpMacTransientDb.GetIpAddress\_1; \\
\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad !IIpMacTransientDb.Add\_1; \\
\quad\quad\quad\quad\quad\quad\quad !ITimer.SetTimeout \\
\quad\quad\quad\quad\quad\quad) + NULL \\
\quad\quad\quad\quad\quad) \\
\quad\quad\quad\quad \} \\
\quad\quad\quad\quad + \\
\quad\quad\quad\quad ?IDhcpListenerCallback.RenewIpAddress\{ \\
\quad\quad\quad\quad\quad !IIpMacTransientDb.GetIpAddress\_1; (\\
\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad !IIpMacTransientDb.SetExpirationTime\_1; \\
\quad\quad\quad\quad\quad\quad\quad !ITimer.SetTimeout \\
\quad\quad\quad\quad\quad\quad) + NULL \\
\quad\quad\quad\quad\quad) \\
\quad\quad\quad\quad \}
\end{array}
$$

```
      +
     ?IDhcpListenerCallback.ReleaseIpAddress{
      !IIpMacTransientDb.GetIpAddress_1; (
       (
        !IIpMacTransientDb.Remove_1;
        !IDhcpCallback.IpAddressInvalidated_1;
        (!ITimer.CancelTimeouts_1 + NULL)
       ) + NULL
      )
     }
    ) *
    |
    (
     ?ITimerCallback.Timeout{
      (
       !IIpMacTransientDb.GetExpirationTime_2; (
        (
         !IIpMacTransientDb.Remove_2;
         !IDhcpCallback.IpAddressInvalidated_2
        ) + NULL
       )
      ) *
     }
    ) *
   )

)
 +
(
 (
  (
   (
    (
     ?IDhcpListenerCallback.RequestNewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;
      (
       (
        !IIpMacTransientDb.Add_1;
        !ITimer.SetTimeout
       ) + NULL
      )
     }
      +
     ?IDhcpListenerCallback.RenewIpAddress{
```

```
      !IIpMacTransientDb.GetIpAddress_1;(
        (
          !IIpMacTransientDb.SetExpirationTime_1;
          !ITimer.SetTimeout
        ) + NULL
      )
    }
    +
    ?IDhcpListenerCallback.ReleaseIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
        (
          !IIpMacTransientDb.Remove_1;
          !IDhcpCallback.IpAddressInvalidated_1;
          (!ITimer.CancelTimeouts_1 + NULL)
        ) + NULL
      )
    }
  ) *
  |
  (
    ?ITimerCallback.Timeout{
      (
        !IIpMacTransientDb.GetExpirationTime_2;(
          (
            !IIpMacTransientDb.Remove_2;
            !IDhcpCallback.IpAddressInvalidated_2
          ) + NULL
        )
      ) *
    }
  ) *
)
|
?IManagement.UsePermanentIpDatabase^
);!IManagement.UsePermanentIpDatabase$;(
(
  (
    ?IDhcpListenerCallback.RequestNewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
        (
          !IIpMacPermanentDb.GetIpAddress;(
            (
              !IIpMacTransientDb.Add_1;
```

```
          !ITimer.SetTimeout
        ) + NULL
      )
    ) + (
      !IIpMacTransientDb.Add_1;
      !ITimer.SetTimeout
    )
  )
 }
 +
 ?IDhcpListenerCallback.RenewIpAddress{
  !IIpMacTransientDb.GetIpAddress_1; (
    (
      (
       !IIpMacPermanentDb.GetIpAddress; (
         (
          !IIpMacTransientDb.SetExpirationTime_1;
          !ITimer.SetTimeout
         ) + NULL
       )
      ) + (
       !IIpMacTransientDb.SetExpirationTime_1;
       !ITimer.SetTimeout
      )
    ) + NULL
  )
 }
 +
 ?IDhcpListenerCallback.ReleaseIpAddress{
  !IIpMacTransientDb.GetIpAddress_1; (
    (
     !IIpMacTransientDb.Remove_1;
     !IDhcpCallback.IpAddressInvalidated_1;
     (!ITimer.CancelTimeouts_1 + NULL)
    ) + NULL
  )
 }
) *
|
(
 ?ITimerCallback.Timeout{
  (
   !IIpMacTransientDb.GetExpirationTime_2; (
```

```
                (
                 !IIpMacTransientDb.Remove_2;
                 !IDhcpCallback.IpAddressInvalidated_2
                ) + NULL
               )
              ) *
             }
            ) *
           )
           |
           ?IManagement.StopUsingPermanentIpDatabase^
          );!IManagement.StopUsingPermanentIpDatabase$
         )
        )*
```

Appendix C

Specification of the CashDeskApplication component in BP

```
(
  # INITIALISED
  (
    (
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
      ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent +
      ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
      ?CashDeskAppEventHandlerIf.onPINEnteredEvent
    )*;
    ### The important part :
    ?CashDeskAppEventHandlerIf.onSaleStartedEvent
  );

  #SALE_STARTED

  (
    ### The important part :
    ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent{
      # ExpressMode & products.size == 8
      NULL
       +
      (
```

```
        !CashDeskConnectorIf.getProductWithStockItem;

        !CashDeskAppEventDispatcherIf.sendProductBarcodeNotValidEvent
        +
        !CashDeskAppEventDispatcherIf.sendRunningTotalChangedEvent
      )
    ###
    }+
    ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
    ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
    ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
    ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
    ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent +
    ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
    ?CashDeskAppEventHandlerIf.onPINEnteredEvent
  )*;#<---LOOP

  (
    ### The important part :
    ?CashDeskAppEventHandlerIf.onSaleFinishedEvent;
    (
      ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
      ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent +
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
      ?CashDeskAppEventHandlerIf.onPINEnteredEvent +
      ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent
    )*
  );

  #SALE_FINISHED

  (
    ### The important part :
    ?CashDeskAppEventHandlerIf.onPaymentModeEvent;
    ###
    (
      ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent +
      ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
```

```
    ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
    ?CashDeskAppEventHandlerIf.onPINEnteredEvent
  ) *
);

# PAYING_BY_CASH
(
  (
    (
      ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent +
      ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
      ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
      ?CashDeskAppEventHandlerIf.onPINEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
      ### The important part :
      ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent
      ###
    )*;

    # On Enter
    ### The important part :
    ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent{
      !CashDeskAppEventDispatcherIf.sendChangeAmountCalculatedEvent
    };
    ###

    (
      ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
      ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
      ?CashDeskAppEventHandlerIf.onPINEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent
    )*;
    ### The important part :
    ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent{
      !CashDeskAppEventDispatcherIf.sendSaleSuccessEvent;
      !CashDeskEventDispatcherIf.sendAccountSaleEvent;
      !CashDeskEventDispatcherIf.sendSaleRegisteredEvent
```

```
    }
    ###
  )

  +

  # PAYING_BY_CREDITCARD

  (
    (
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent;

      # CREDITCARD_SCANNED

      (
        ?CashDeskAppEventHandlerIf.onPINEnteredEvent{
          !BankLock.lock;
          !BankIf.validateCard;
          (
            !CashDeskAppEventDispatcherIf.sendInvalidCreditCardEvent
            +
            (
              !BankIf.debitCard;
              !CashDeskAppEventDispatcherIf.sendInvalidCreditCardEvent
            )
          );
          !BankLock.unlock
        }
        +
        ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
        ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
        ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
        ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
        ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
        ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
        ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
        ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent
      )*;
      ?CashDeskAppEventHandlerIf.onPINEnteredEvent{
        !BankLock.lock;
        !BankIf.validateCard;
        !BankIf.debitCard;
        !CashDeskAppEventDispatcherIf.sendInvalidCreditCardEvent;
```

```
        !BankLock.unlock
      }
    )*;

    ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent;

    # CREDITCARD_SCANNED
    (
      ?CashDeskAppEventHandlerIf.onSaleStartedEvent +
      ?CashDeskAppEventHandlerIf.onSaleFinishedEvent +
      ?CashDeskAppEventHandlerIf.onPaymentModeEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountEnteredEvent +
      ?CashDeskAppEventHandlerIf.onCashAmountCompletedEvent +
      ?CashDeskAppEventHandlerIf.onProductBarcodeScannedEvent +
      ?CashDeskAppEventHandlerIf.onCreditCardScannedEvent +
      ?CashDeskAppEventHandlerIf.onCashBoxClosedEvent
    )*;

    ### The important part :
    ?CashDeskAppEventHandlerIf.onPINEnteredEvent{
      !BankLock.lock;
      !BankIf.validateCard;
      !BankIf.debitCard;
      !BankLock.unlock;
      !CashDeskAppEventDispatcherIf.sendSaleSuccessEvent;
      !CashDeskEventDispatcherIf.sendAccountSaleEvent;
      !CashDeskEventDispatcherIf.sendSaleRegisteredEvent
    }
    ###
  )
 )
) * | (

 # Enable Express Mode
 ?CashDeskAppEventHandlerIf.onExpressModeEnabledEvent{
   !CashDeskAppEventDispatcherIf.sendExpressModeEnabledEvent + NULL
 }
) * | (

 # Disable Express Mode
 ?CashDeskAppEventHandlerIf.onExpressModeDisabledEvent
)*
```

Appendix D

Specification of the CashDeskApplication component in EBP

```
component CashDeskApplication {

  types {
    states = {
      INITIALIZED, SALE_STARTED, SALE_FINISHED,
      PAYING_BY_CREDITCARD, CREDIT_CARD_SCANNED,
      PAYING_BY_CASH, PAID
    }
  }
  vars {
    states state = INITIALIZED
  }
  behavior {
    (
      ?CDAEventHandlerIf.onEvent(SaleStartedEvent) {
        switch (state) {
          INITIALIZED :
            { state <- SALE_STARTED }
          default :
            { NULL }
        }
      }
      +
      ?CDAEventHandlerIf.onEvent(ProductBarcodeScannedEvent) {
        switch (state) {
          SALE_STARTED :
```

```
        {
          !CashDeskConnectorIf.getProductWithStockItem;
          (
            !CDAEventDispatcherIf.send(ProductBarcodeNotValidEvent)
            +
            !CDAEventDispatcherIf.send(RunningTotalChangedEvent)
          )
        }
      default :
        { NULL }
    }
  }
  +
  ?CDAEventHandlerIf.onEvent(SaleFinishedEvent) {
    switch (state) {
      SALE_STARTED :
        { state <- SALE_FINISHED }
      default :
        { NULL }
    }
  }
  +
  ?CDAEventHandlerIf.onEvent(CashAmountEnteredEvent) {
    switch (state) {
      PAYING_BY_CASH :
        { NULL }
      default :
        { NULL }
    }
  }
  +
  ?CDAEventHandlerIf.onEvent(CashAmountCompletedEvent) {
    switch (state) {
      PAYING_BY_CASH :
        {
          !CDAEventDispatcherIf.send(ChangeAmountCalculatedEvent);
          state <- PAID
        }
      default :
        { NULL }
    }
  }
  +
```

```
?CDAEventHandlerIf.onEvent(CashBoxClosedEvent) {
  switch (state) {
    PAID :
      {
        !CDAEventDispatcherIf.send(SaleSuccessEvent);
        !CashDeskEventDispatcherIf.send(AccountSaleEvent);
        !CashDeskEventDispatcherIf.send(SaleRegisteredEvent);
        state <- INITIALIZED
      }
    default :
      { NULL }
  }
}
+
?CDAEventHandlerIf.onEvent(CreditCardScannedEvent) {
  switch (state) {
    PAYING_BY_CREDITCARD :
      { state <- CREDIT_CARD_SCANNED }
    CREDIT_CARD_SCANNED :
      { state <- CREDIT_CARD_SCANNED }
    default :
      { NULL }
  }
}
+
?CDAEventHandlerIf.onEvent(PaymentModeEvent) {
  switch (state) {
    SALE_FINISHED :
      {
        state <- PAYING_BY_CREDITCARD +
        state <- PAYING_BY_CASH
      }
    default :
      { NULL }
  }
}
+
?CDAEventHandlerIf.onEvent(PINEnteredEvent) {
  switch (state) {
    CREDIT_CARD_SCANNED :
      {
      !BankLock.lock;
      !BankIf.validateCard;
```

```
                    (
                      (!CDAEventDispatcherIf.send(InvalidCreditCardEvent);
                      !BankLock.unlock)
                       +
                      (
                        !BankIf.debitCard;
                        !BankLock.unlock;
                        (
                          !CDAEventDispatcherIf.send(InvalidCreditCardEvent);
                          (
                            (NULL + state <- PAYING_BY_CREDITCARD)
                             +
                            (
                             !CDAEventDispatcherIf.send(SaleSuccessEvent);
                             !CashDeskEventDispatcherIf.send(AccountSaleEvent);
                             !CashDeskEventDispatcherIf.send(SaleRegisteredEvent);
                             state <- INITIALIZED
                            )
                          )
                        )
                      )
                    )
                  }
                default :
                  { NULL }
              }
            }
          )* | (
            # Enable Express Mode
            ?CashDeskAppEventHandlerIf.onEvent(ExpressModeEnabledEvent) {
              !CDAEventDispatcherIf.send(ExpressModeEnabledEvent)
               +
              NULL
            }

          )* | (
            # Disable Express Mode
            ?CDAEventHandlerIf.onEvent(ExpressModeDisabledEvent)
          )*
        }
}
```