Towards Thread Aware Component Behavior Specifications

Tomáš Poch

Department of Distributed and Dependable Systems
Advisor: Prof. František Plášil
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 received from my advisor Prof. František Plášil. For the various help they provided me, I also thank to my colleagues; a particular thank goes to Ondřej Šerý and Jan Kofroň.

This work and the related research was partially supported by the Grant Agency of the Czech Republic projects SVV-2010-261312, GD201/05/H014, 201/06/0770 and 201/08/0266, by the Czech Academy of Sciences project 1ET400300504, by the Ministry of Education of the Czech Republic grant MSM0021620838, by the ITEA/EUREKA project OSIRIS, and by the Q-ImPrESS research project by the European Union under the ICT priority of the 7th Research Framework Programme.

Last but not least, I am in debt to my family. Their support and patience made this work possible.
Abstract

The component based development is a well established methodology of software development. The industry, however does not take the advantage of component behavior modeling. Although the analyses of models guarantee notion of correctness in form of behavioral compatibility of component composition, the application in practice is limited by the expressiveness of the modeling languages as well as by the fact that the manual preparation of models is demanding and error prone task. To ease the application of behavioral modeling in practice, we propose Threaded Behavior Protocols (TBP) — a modeling language aiming at the gap between the modeling and imperative languages and the imperative languages. By providing the developers with the concepts known from the imperative languages at the model level, we enable easier preparation of component models. The theoretical framework of TBP provides the notion of correctness based on absence of communication errors and the refinement relation preserving the correctness in arbitrary environment. Thus, the analyses supported by the framework provide similar benefits as the legacy modeling languages, however considering also the imperative language concepts.

Keywords

Behavior modeling, component systems, threads, composition correctness, refinement
Název  Specifikace chování komponent zohledňující vlákna
Autor  Tomáš Poch
       poch@dst.mff.cuni.cz
       (+420) 221 914 285
Školitel  Prof. František Plášil
         plasil@dst.mff.cuni.cz
       (+420) 221 914 266
Katedra  Katedra distribuovaných a spolehlivých systémů
         Matematicko-fyzikální fakulta
         Univerzita Karlova v Praze
         Malostranské nám. 25, 118 00 Praha, ČR

Abstrakt
Komponentový přístup je již poměrně zavedenou metodologií používanou při vývoji software. Při komerčním vývoji aplikací, se však ještě nevyužívají modely chování komponent a jejich následná analýza, ačkoliv by to zaručilo, že komunikace mezi složenými komponentami nebude obsahovat chyby. Reálnému použití v praxi brání jak relativně omezené výrazové prostředky modelovacích jazyků tak i náročnost psaní modelů.
Abychom usnadnili použití modelů chování, navrhujeme modelovací jazyk Threaded Behavior Protocols (TBP), který se snaží překlonout rozdíly mezi modelovacími a imperativními programovacími jazyky. Tím, že umožníme programátorům používat koncepty z imperativních jazyků, na které jsou zvyklí, usnadníme přípravu modelů. Teorie TBP definuje pojem správnosti kompozice komponent jako absenci dvou pevně daných komunikačních chyb a poskytuje relaci zjemňování modelu, která zachovává správnost vzhledem k libovolnému prostředí. Díky tomu, přináší analýza TBP podobné výhody jako starší modelovací jazyky, přičemž bere v úvahu i koncepty z imperativních jazyků.

Klíčová slova
Modely chování, komponentové systémy, vlákna, správnost kompozice, zjemňování
## 1 Introduction
- 1.1 Correct SW ........................................... 1
- 1.2 Software Models ..................................... 2
- 1.3 Component Based Development ...................... 3
  - 1.3.1 Application Architecture .......................... 3
  - 1.3.2 Development of Components ...................... 4
  - 1.3.3 Benefits and Issues of CBD ....................... 5
- 1.4 Software Verification .................................. 7
- 1.5 Problem Statement .................................... 9
- 1.6 Research Goal and Objective ......................... 10

## 2 State of the Art in the Behavior Modeling .......... 11
- 2.1 Formal Methods in CBD ............................... 11
- 2.2 Notion of Correctness ................................ 13
  - 2.2.1 Composition Errors ............................... 13
  - 2.2.2 User-defined Properties ........................... 14
  - 2.2.3 Low Level Errors in the Implementation ........... 14
- 2.3 Formalisms Available ................................. 15
  - 2.3.1 Process Algebras .................................. 15
  - 2.3.2 Automata Based .................................... 22
  - 2.3.3 Other ............................................... 25
  - 2.3.4 BP .................................................. 26
  - 2.3.5 EBP ............................................... 31
- 2.4 Goals Revisited ........................................ 32
- 2.5 Overview of Publications ............................. 34
- 2.6 Overview of Contribution ............................. 35

## 3 Threaded Behavior Protocols ......................... 37
- 3.1 Syntax .................................................. 37
  - 3.1.1 Relation to the Component Model .................. 38
  - 3.1.2 Type Declarations .................................. 38
  - 3.1.3 State Variables .................................... 39
  - 3.1.4 Reactions .......................................... 39
  - 3.1.5 Threads ............................................ 42
CONTENTS

3.1.6 Provisions ................................. 43
3.2 Semantics .................................. 47
  3.2.1 TBP Model ................................. 47
  3.2.2 Analysis of Closed Models ................. 53
  3.2.3 Analysis of Open Models .................. 60
3.3 Discussion .................................. 76
  3.3.1 Relation to Imperative Languages ............ 76
  3.3.2 Relation to Process Algebras ................ 78
  3.3.3 Employing TBP in CBD ...................... 83

4 Validation .................................. 85
  4.1 Session Manager .............................. 85
  4.2 TBP Checker ................................. 89

5 Conclusion and Future Work .................. 91

References .................................. 93
Introduction

1.1 Correct SW

Computers and information technologies are influencing many areas of human activity. Computers become an integral part of many products (cars, telephones), improve effectiveness of business processes and reduce costs in many industries. Whole new industries and services emerged.

However, usage of computers and information technologies is advantageous as long as the software involved is correct. Incorrect software may cause lost of large amounts of money or even human lives. Thus growing demand on correct software is a driving force behind changes in the way software is being developed.

Software engineering is a discipline studying the methodologies for software development. It ranges from project management, through definition of development cycle and its individual phases, to techniques and tools aiming at specific details of particular tasks involved in the development cycle.

At the dawn of information technologies, software was built more or less in an ad-hoc manner. Software being developed those days used to be relatively simple due to hardware limitations. Moreover the amount of software produced was relatively small and there were enough highly qualified specialists who were able to keep those whole programs in mind and steer the development from the initial idea to the final product. It is worth to mention, that many challenges the developers had to face in those days there were caused by hardware limitations. Nowadays, most of these limitations disappeared, since hardware is unbelievably cheap. Instead, the developers face other challenges ranging from integration with other systems to user friendliness.
CHAPTER 1. INTRODUCTION

At certain point, the ad-hoc development became insufficient, which lead to beginnings of software engineering. The rest of this section presents several techniques used by software engineers to write better maintainable and correct code.

1.2 Software Models

Software models are used in software engineering in all phases of development. Since each model emphasizes certain information while abstracting from other, there are different kinds of models used for various purposes. Models bring several important benefits. On one hand, models serve for documentation purposes and as a basis for communication among members of the team. The right model allows capturing the important information in a straight-forward manner. On the other hand, models can be automatically analyzed to identify inconsistencies which could lead to bugs in the software. The sooner a bug is discovered the easier (and cheaper) is to fix it.

Unified Modeling Language (UML) [13] is a general purpose graphical modeling language (in fact a set of languages) which provides several kinds of models. They include diagrams for specification of the software use-cases and business processes in the customer’s company. These models capture just the interaction of the user with software and utterly abstracts from internals. Component diagrams capture overall structure of software at the design phase. These diagrams identify relatively independent parts of the application and the way the parts interact with each other. Finer structure of the implementation is depicted in class diagrams. Class diagrams capture individual classes and relations among them. Behavior models capture behavior of individual parts of software. To capture behavior, there are activity diagrams, sequence diagrams and state machine diagrams in UML. Since they are closely related to the topic of this thesis, they are discussed in the following chapter together with the other approaches to behavioral modeling.

UML Class diagrams and Sequence diagrams become quite popular in industry since they are illustrative and allowing to quickly grasp the idea. In addition to UML, there is a large number of other modeling approaches, used for special purposes (e.g. performance modeling).
1.3 Component Based Development

The basic idea of component based development (CBD) [69, 49] is to compose complex software from well defined artifacts denoted as components. Individual components are isolated, developed independently (possibly by different vendors) and communicate with each other through well defined interface. The isolation encourages reuse of single component in different applications requiring the same functionality.

There is a wide range of component systems supporting component-based software life cycle (e.g. EJB [67, 23], COM/DCOM [36], Fractal [18], SOFA [21], CCM [57]). On one hand a component system is based upon its component model - set of concepts for describing structure and properties of the application as well as properties of individual components. On the other hand, component systems also provide runtime support for the application—an environment providing the components computational resources, means for communication, as well as other features like transactions or persistence. Runtime also provides means for monitoring and management of applications.

1.3.1 Application Architecture

The application architecture plays a key role in the development process [69]. It identifies individual components the application consist from, their functionality, additional parameters (e.g. performance) and also the way the components communicate with each other—interfaces.

The Fig. 1.1 contains an example of such architecture. Each component defines the set of provided interfaces (set of methods implementing functionality provided to other components), as well as the set of required interfaces (functionality the component requires from other components) Each interface has its type and types of the connected interfaces must correspond to each other (in the meaning common to object oriented languages).

The example contains a fragment of a web-based information system. The application logic is implemented in the BusinesLogic component and the UserInterface component mediates the user input entering the system in form of http requests. The commands from the user, however, do not go directly to the business logic. The communication is intercepted by the SessionManager component that takes care about authentication of commands. The SessionManager requires some additional functionality provided by Log
CHAPTER 1. INTRODUCTION

Figure 1.1: Architecture employing the SessionManager component and Database components. The latter one is shared with the BusinessLogic component.

1.3.2 Development of Components

The application architecture is created as one of the first steps taken in each software project. When possible, the architect should reuse existing components available to the development team to reduce the design and implementation effort. The remaining components, typically those implementing the specific business logic of the application, have to be implemented. The newly developed components become part of component library to be used in future applications.

The new components are either implemented in a programming language (primitive components) or by composition of existing components (composed components) [29]. The programming language used to implement primitive components has to be supported by a component system—at least there has to be a mapping of the component model concepts to the concepts of the implementation language, and also runtime support. Component systems supporting composed components are referred as hierarchical component systems. A composed component is defined by its architecture. Thus, the top level architecture containing composed architectures can be seen as hierarchy of architectures.
1.3. COMPONENT BASED DEVELOPMENT

The Fig 1.2 contains the architecture of the SessionManager component implemented by composition of other components. The main functionality is implemented by the AuthLogic component. It is using functionality provided by HashFunction and RndGenerator. The Timer component from time to time triggers cleanup of inactive sessions. Other interfaces of the AuthLogic component directly correspond to the interfaces of the SessionManager component and are delegated to the higher level.

1.3.3 Benefits and Issues of CBD

Reduction of Complexity CBD enforces the developers to explicitly specify the architecture of the application at the very beginning. There is a limited set of means for capturing the architecture, so that the information is compatible for all applications developed for a particular component model.

By explicitly specifying individual components, their provided interfaces and required interfaces, the development of the whole application is clearly divided into a number of simpler tasks. The individual components serve as a natural boundary when assigning responsibilities and tasks to individual members of team. Moreover, there is always someone in charge of the overall architecture who, on the other hand, does not have to keep in mind particular implementation details of individual components.

Obviously, it is possible to split complex software into modules and libraries, even without a component system. It, however, requires certain
CHAPTER 1. INTRODUCTION

discipline from developers, which is enforced when a component approach is used.

**Isolation**  Components run in isolation—they do not influence each other by other means than communication through interfaces. Thus, any change of a component implementation that does not change the behavior observable on the component’s interfaces does not influence functionality of other components. Such degree of isolation allows designing, implementing, and also testing components independently of the others.

**Reuse**  The explicit specification of provisions and requirements of a component by standardized means, as well as the isolation allows reusing the component in different applications. It suffices to check that the component fits the role expected in the architecture.

In the ideal case, there is a library of components which can be reused in a new application—components of the shelf—COTS. If the library is rich enough, the new application is created by composition of previously existing components with a few newly created components implementing specific business logic of the application. The library can contain components from different sources — components originated in previous projects implemented by different developers, or even components bought from other companies.

One can see certain similarity of CBD with well established methods in other industries. When designing electronic devices, electrical engineers often reuse integrated circuits as well as civil engineers use panels to build buildings. Software development, however, significantly differs from the mentioned areas by the structure of costs. Most of the costs are spent on development, while production of media as well as distribution is almost for free.

**Additional Level of Abstraction**  Components can be seen as another level of abstraction positioned between the abstractions provided by the application to the user and abstractions provided by operating system to programmer. The individual components can be perceived from the administration and maintenance point of view. It is possible to monitor their behavior at runtime and refer the components in system logs. Component runtime often provides middleware functionality, so that developers do not have to pay special attention to distribution of the application.

**Other Side of the Black Box**  The benefits mentioned so far are closely related to the fact that component internals are hidden to people who use them. This is referred to as black box approach to components. In particular,
only the person who implements a primitive component has access to the source code of the component. Other persons are expected to deal with the component via its interfaces and rely on its correctness.

In particular, when a component is used in an architecture, it must be ensured that it meets the expectations of the architecture. In practice, the developers rely on syntactic compatibility of interfaces (at the implementation language level) and a precise documentation. This, obviously, does not guarantee a correct result. A developer can oversight subtle details in the documentation which can lead to an erroneous behavior. Also, the documentation does not necessarily reflect the actual behavior of the component implementation. The errors caused by inconsistency of components can be revealed by testing, or appears in production. This weak point of CBD is aimed by many efforts in academia.

1.4 Software Verification

The goal of software verification is to provide techniques and tools which prevent developers from making errors in software. The ultimate solution of this task would be a tool which takes as an input a software project and decides whether the project is correct or not. In the former case, it reports the reason why the software is incorrect. Existence of such a tool is, however, prevented by severe obstacles.

First, such a tool would require a precise definition of correctness. Since the purpose of each software is different, the specification of correct behavior should be part of the input to the tool. However, even if we limit the notion of correctness to termination (i.e. the program after accepting arbitrary input terminates or is ready to accept next input in a finite time), such tool cannot exist. Still termination is a general property which should hold in vast majority of software. The tool cannot exist because termination is undecidable for Turing-complete programming languages [71]. At the same time, the programming languages used in practice (e.g. Java) are Turing-complete. Other useful general properties include absence of low level errors, like division by zero, wrong accesses to memory (e.g. NullPointerException in Java) and deadlocks. Unfortunately, all of these are also undecidable.

Thus, there is no way to provide a tool that would verify arbitrary software. On the other hand, there are classes of programs for which many properties are decidable (e.g. it is easy to say that a Hello World example terminates). Thus, a realistic goal of software verifications can be formulated as an effort to improve verification tools and techniques to make feasible their
application on realistic programs important for practice.\footnote{Important point is that such programs often do not contain self-reference (do not process themselves) which is an important step in the proof of undecidability} To further widen the set of feasible programs, the user can be asked to provide some additional information (e.g. invariants).

Verification tools either directly process the source code, possibly annotated with additional information or analyze abstract models. For instance, Java Pathfinder [38] traverses the state space of Java program (including different scheduling) to detect assertion violations as well as deadlocks and race conditions. Blast [12], on the other hand is using Counter Example Guided Abstraction Refinement to analyze C code with smaller memory requirements. Other approaches allow checking validity of user-specified formulas (method pre-conditions, post-conditions, loop invariants) [28, 20]. The method is based on construction of a huge formula representing the computation of the program. The formula involves meaning of individual statements as well as the user specified formulas. Satisfiability of the huge formula is then checked by a third party theorem prover.

Abstract software models focusing on modeling of software behavior require the user to describe the software in a special purpose modeling language (e.g. Promela [41], LTSA [52]) While the analysis of abstract models is more feasible than analysis of source codes (the generated state space is smaller) there is always a problem with ensuring consistency of the model with the real implementation. On the other hand, design flaws can be revealed before the implementation even exists.

Even the current state of the art software verification methods already provide useful results [74]. Their usage in practice is, however, limited by two facts. First, usage of tools is not trivial and typically requires a skilled specialist to add the additional information required by the tool and also to interpret results. Second, the tools typically do not scale well and fail to analyze large projects. All in all, application of current verification methods yields tangible results only in special cases. Since the application requires additional effort, it pays of only in cases when correctness is a crucial factor.

Software verification and CBD can benefit from each other [2]. On one hand, CBD splits a complex program into smaller fragments, which enables compositional verification. In particular, correctness of each component can be verified separately. To provide a sound result, also the correctness of composition has to be checked. In this context, each component is equipped with a formal model specifying its behavior. In particular, when composing components it is checked that their models fit together.

The requirements posed to a formal model to make it successfully used
1.5. PROBLEM STATEMENT

in the component based development, are formulated as interface theories in [3]. The requirements include support of incremental design and independent implementability. A selective overview of several diverse approaches related to component behavior evaluation, which ranges from LTS based behavior models, through contracts, analysis of implementation, to summaries is in [27].

When aiming at a modeling language for description of component behavior, the list of candidates includes first of all UML, Promela, and process algebras. While UML is quite popular in industry, it does not provide any notion of correctness, composition, nor refinement. It is used mainly for documentation purposes. Promela on the other hand, is designed as input language of the SPIN model checker and models in Promela are intended only for verification. Promela provides communication through message channels to allow composition of models. Moreover, it allows specifying assertions in the code, as well as state properties of the modeled system in form of LTL formulas (a correct model avoids assertions and fulfills the LTL formulas). On the other hand, there is no support for refinement. Moreover, since Promela does not directly aim at component systems, abstractions of the component model are not first-class entities of the language. This is also the case for process algebras (e.g. CCS[55], CSP[40]). At the same time, while providing a rich theory including various kinds of simulation and refinement, process algebras are too complex to be used in day-to-day development. Their semantics is too different from common programming languages. Practical usability is also a key problem of automata based theoretical frameworks. (e.g. interface automata [3, 10], CIA [26]).

1.5 Problem Statement

The component based development has found its way to industry. However, the component systems used in practice (e.g. EJB [53], DCOM [36], CCM [57], Koala [72]) hardly take an advantage of behavior modeling and subsequent analysis.

We believe, the problem is that there is no suitable specification language for behavioral modeling aiming at component systems in a comprehensive way. In particular, no specification language allows writing behavior specification in a straight-forward way, while providing features important for analysis of a component-based system such as behavior composition and transitive refinement.
1.6 Research Goal and Objective

The main objective of this thesis is to make the application of behavioral modeling in component based development more suitable for day-to-day practice. In particular, we propose the specification language Threaded Behavior Protocols (TBP) aiming at fulfilling the following two goals. On one hand, writing specifications in TBP should resemble programming in an imperative programming language. Since programmers are used to the concepts provided by imperative languages, this aspect should significantly decrease the effort needed to prepare specifications of individual components. On the other hand, the formal framework should provide means for its successful application in CBD. In particular, the formal framework should provide the composition operator, as well as refinement relation, both designed with respect to precisely defined notion of correctness and imperative concepts of the language. This way, we want to combine the strong aspects of specification languages aiming at practical usability and existing theoretical frameworks focused at component behavior.

Based on the overview of related work in the following chapter, we will suggest particular features of the language as well as properties of its theoretical framework in the Sect. 2.4.
State of the Art in the Behavior Modeling

2.1 Formal Methods in CBD

Rigorous application of formal methods in software development always requires some additional effort. The additional tasks range from running analysis tools and interpretation of results to preparing models in various languages and formalisms as, well as maintaining the consistency and cooperation with other tasks in the development process.

To lower the overall effort and maximize formal methods benefits, it is necessary to clearly state their position in the development process. In this section we discuss several options how to employ formal methods in context of component based development.

The component based development has the advantage that it already forces developers to explicitly define the architecture of an application. Thus, there is a solid ground the formal models can rely on. Then, behavior models of individual components allow to check whether the components composed together fulfill its respective assumptions on the environment as well as to check whether the implementation, provided either as composition of other components or in an imperative language, such as Java, conforms to the specification provided in a form of a behavior model.

Top-down Approach  When the development follows the classical waterfall development model, the developer is expected to provide design as soon as possible. When formal methods are applied, certain aspects of the design are expressed formally. For instance, behavior model is should be provided
right after the application architecture is available. When the architecture is hierarchical, the behavior models can be created level by level—the behavior models of the top level components can be created prior the architecture of the lower levels of the hierarchy. At each level, the composition of individual models is checked for correctness and also whether it fulfills correctly the specification on the higher level. The relation of the composition to the specification on the higher level is refereed as refinement.

When the architecture is ready, the next step is to provide an implementation of primitive components. Since the behavior model of primitive components is already available at that moment, it is convenient to generate a skeleton of the implementation from the model. Then, the developers are expected to implement manually the aspects of the behavior which are not captured by the model (e.g. by means of inheritance). Alternatively to the skeleton generation, the developers can provide the whole implementation manually. In such case, it is important to gain confidence that the implementation conforms to the model of primitive components. Otherwise the whole process is not sound.

The top-down approach allows continuous verification as the model is getting more precise. Thus, potential errors at higher levels of hierarchy are detected as soon as possible and can be fixed with small impact on the lower levels.

**Bottom-up Approach** The top-down approach is not always applicable; especially in case of legacy applications being originally developed without formal methods. In such cases, the architecture of the system is available at the best. Moreover, in some methodologies which do not follow the waterfall principle, a rapid prototyping comes first. Then the code is refined and improved. In all those cases, there is a code first and then a behavior model comes.

Thus, the first step is to provide behavior models for all primitive components. Here comes again the moment for a skilled specialist. Alternatively, the models can be generated from the implementation by means of static analysis or monitoring. Once the models for all primitive components are available the architecture can be flattened and checked for correctness (e.g. absence of deadlock). This may, however, shown to be an unfeasible task for current analysis tools (e.g. exponential growth of state space). In such case, the hierarchy can help to separate the task to smaller subtasks and check each composed component individually. In particular, when the specification of composite components is provided, refinement ensures that the specification corresponds to the composition of primitive components and in
the next step, correctness of composition of components at higher level is checked.

**COTS** Orthogonally to the approaches discussed in the previous paragraphs stays the concept of *components-of-the-shelf*. In such case, a third party is responsible for the behavioral correctness of the component. Formal methods play again an important role when composing components from different vendors. In particular, each component is equipped with its formal specification with clear semantics agreed on by the component vendor and the component user. The developer of the component takes care of ensuring that the implementation fulfills the specification. The developer of the application, on the other hand, checks that the component’s specification fits into the whole application.

From the perspective of top-bottom/bottom-top approaches, the problem of third-party components can be formulated as passing of the responsibility for the correctness of the part of component hierarchy to someone else.

### 2.2 Notion of Correctness

A key benefit of formal methods is the guarantee that the created software is in certain sense correct. (i.e. satisfies specific properties) The notion of correctness is given by the particular formalism used. Obviously, the formalism cannot give a guarantee related to the aspects of software it abstracts from.

#### 2.2.1 Composition Errors

Behavior specification of each component describes how the component communicates with its environment. In particular what it provides to the environment under what assumptions and what it requires from the environment. As the specifications are being composed in the way prescribed by the architecture, violations of these specifications can emerge (composition errors). Obviously, to support this scenario, the formalism must provide a means for composition—a way to create a more complex models from simpler ones.

There are different kinds of composition errors. When the specification captures the sequences of allowed method calls on the component’s interfaces, the most obvious composition error is an invocation of a method on a component when the component does not expect it. Other composition errors are different variants of deadlock. It is important to distinguish a situation when all components are waiting for a method call but none of them
emits it from the situation where just several components are waiting for each
other (forever) while the rest of the system continues in correct computation.
While the former situation is often referred to as deadlock in theory of pro-
cess algebras, the latter one is a deadlock for practitioners (e.g. operating
system developers).

Each formalism defines its built-in composition errors in a specific way.
This will be discussed in following sections devoted to individual formalisms.

2.2.2 User-defined Properties

In addition to built-in errors, some formalisms provide means to specify ad-
ditional properties required to hold in a specification. This allows specifying
application-specific properties reflecting the actual business logic of the ap-
application. Consider for instance the requirement that all method calls in a
component should be preceded by an initialization and eventually followed by
a special method call which frees resources (working with a file, transaction-
based processing).

User-specified properties are typically provided in the form of a temporal
logic (e.g. LTL, CTL).

2.2.3 Low Level Errors in the Implementation

By providing the behavior specification for each component one can also
improve capabilities of other verification approaches. Assuming a complete
application (not-necessarily a component based) it is possible to use a state-
of-the-art code model checker (e.g. JPF) to traverse the state space of the
application and search for errors (e.g. uncaught exceptions). In most cases,
however, the analysis of the whole application is out of the capabilities of
current tools—the size of state space is exponential the size of source due to
non-determinism of the user input and thread scheduling. Having a formal
specification for each component available, one can take the implementation
of each component and analyze it separately under the assumption that the
rest of the application is using it in a way compliant with its specification.
2.3 Formalisms Available

During past decades, several languages and formalisms for behavior modeling of software systems have been proposed. They range from those very generic ones (e.g., process algebras) to those specific to components (e.g., Darwin [51], Wright [4], Behavior Protocols [60]). In this section, we focus on those based on labeled transition systems (LTS), as they are well studied, meaning that their properties (e.g., decidability of model checking of particular temporal logic formulas) as well as algorithms for formal reasoning are well established.

An LTS is a very natural way to capture control flow—the labels represent atomic computational actions and a sequence of transitions models a behavior. In particular, to model externally observable behavior of components, the transition labels have to be chosen as the names of input and output actions; in the case of modeling a black-box component, this implies involving a naming scheme encoding the names and kind (provided/required) of interfaces and methods. For instance, ?interface_name.method_name can express that the modeled component is able to accept the method_name method of the interface_name provided interface (as an input action) and !interface_name.method_name to express issuing such call on a required interface (as an output action).

Semantics of the formalism given in form of LTS is used to define model properties and relations among models. The properties range from generic ones (e.g., deadlock, livelock) to user specified in a temporal logic. The relations include various kinds of behavioral equivalence and refinement.

Evaluation of these properties as well as determining whether two specifications are in certain relation often faces huge size of LTS. Basically, modeling behavior of a software component involving parallelism can cause exponential growth of LTS size in the number of parallel activities. Moreover, number of formalisms induces infinite LTSes.

2.3.1 Process Algebras

Classical process algebras (e.g., CSP [40] and CCS [55]) describe a behavior as a set of cooperating processes syntactically defined by a set of recursive equations; each of them associates a process name with an expression determining the process behavior. Formally, the semantics of an expression is given via derivation rules an LTS. A number of operators are typically defined to combine action names (they determine the transition labels of
the corresponding LTS) and process names. The operators include sequencing, alternatives, parallel composition typically with the option to create a synchronous product, event hiding, renaming, etc.

2.3.1.1 CCS

In Calculus of Communicating Systems (CCS) [55], the behavior is defined in form of processes. Syntax of a process is defined as follows. Let \( \mathcal{A} \) is a set of complementary actions (\( \alpha \in \mathcal{A} \iff \bar{\alpha} \in \mathcal{A} \), \( \overline{\overline{\alpha}} = \alpha \)) and \( \text{Id} \) is a set of process identifiers. Then the basic CCS calculus allows to form processes as prescribed by the following BNF grammar

\[
P :: \emptyset | \alpha . P | A | P + P | P | P | P[f] | P \setminus L \text{ where } \\
\begin{align*}
\alpha & \in \mathcal{A}, \ A \in \text{Id}, \ L \subseteq \mathcal{A} \text{ and } f : \mathcal{A} \to \mathcal{A}.
\end{align*}
\]

Figure 2.1: CCS grammar

Elementary CCS process emitting single action \( \alpha \) is then represented by expression \( \alpha . \emptyset \), often simplified to \( \alpha \). More complex processes are created using operators for alternative (+), parallelism (|) restriction (\( \setminus L \)) and renaming ([f]). The renaming function \( f \) respects complementarity (\( f(\alpha) = b \iff f(\bar{\alpha}) = \bar{b} \)). The function \( f \) is often specified in form \( \alpha_1/\alpha_1, \ldots, \alpha_n/\alpha_n \). Then \( f(\alpha_i) = \alpha'_i \) for \( i = 1, \ldots, n \) and \( f(\alpha) = \alpha \) otherwise. Named processes (\( A \overset{\text{def}}{=} P \)) can be used in other process definition. The recursion is used to capture repetitive behavior. The full CCS calculus introduces actions with value passing. This allows to model method parameters. The data are, basically, encoded into labels.

\[
\begin{align*}
\text{Act} & : \alpha . E \xrightarrow{\alpha} E \\
\text{Alt} & : E \xrightarrow{\alpha} E' + E \xrightarrow{\alpha} E' \quad \text{(symetric)} \\
\text{Com} \_1 & : E \xrightarrow{\alpha} E' F \xrightarrow{\alpha} F' \quad \text{(symetric)} \\
\text{Com} \_2 & : E \xrightarrow{\alpha} E' F \xrightarrow{\alpha} F' E' \xrightarrow{\alpha} E' \xrightarrow{\alpha} E' \\
\text{Res} & : E \xrightarrow{\alpha} E' \quad (\alpha, \bar{\alpha} \notin L) \\
\text{Ren} & : E \xrightarrow{f} E' \quad \text{where } f : \mathcal{A} \to \mathcal{A} \\
\text{Con} & : P \xrightarrow{\alpha} P' \quad (A \overset{\text{def}}{=} P)
\end{align*}
\]

Figure 2.2: CCS transition rules

Semantics of a CCS process is given by the LTS created by application of structural operational semantics rules depicted in Fig. 1.2. Each state of
2.3. FORMALISMS AVAILABLE

Figure 2.3: LTS induced by parallel composition of CCS processes

the LTS corresponds to a CCS process and transitions are labeled by actions from \( \mathcal{A} \cup \{ \tau \} \). If there is a transition labeled by the action \( \alpha \) leading from the state \( P \) to the state \( P' \), then the process \( P \) after emitting action \( \alpha \) is equivalent to the process \( P' \). Each rule \( \frac{P \xrightarrow{\alpha} P'}{Q \xrightarrow{\beta} Q'} \) is interpreted as follows. If there is an transition labeled by \( \alpha \) connecting states \( P \) and \( P' \), then there is also the transition labeled by \( \beta \) connecting \( Q \) and \( Q' \). In particular, the rule \( \text{Act} \), which does not state any precondition above line generates the transitions of single action processes (\( \alpha, \emptyset \)).

When it comes to modeling of two communicating systems represented by CCS processes \( P \) and \( Q \), the parallel operator \( | \) is used. As prescribed by rules \( \text{Com} \), the resulting LTS consist of cartesian product of LTSe representing the original processes \( P \) and \( Q \). Moreover there are \( \tau \) transitions representing atomic synchronization of two complementary actions (rule \( \text{Com}_2 \)). The atomic synchronization is used to model communication among processes. To achieve one-to-one communication among processes, restriction operator is used to eliminate transitions representing actions that were not matched to a complementary action. In particular, the process \( P|Q \) in Fig. 2.3 contains both, the \( \tau \) transition representing communication among \( P \) and \( Q \) as well as transitions labeled by \( c \) resp. \( \tau \) so that the process can further communicate with other processes producing \( c \) resp. \( \tau \) actions. When the restriction operator is applied, \( \left( (P|Q)\setminus\{c\} \right) \) only the \( \tau \) transitions representing communication among \( P \) and \( Q \) remain.

When applied in the component context, individual components are rep-
represented by CCS processes. Complementary actions represent issuing of a method call on a required interface resp. accepting of method call on provided interface. Names of actions correspond to method names found in the architecture (e.g. `session.createSession`). Method parameters and data can be captured in the full calculus by value passing. Composition of two components is represented by combination of renaming, parallel and restriction operators. Actions representing method calls on interfaces bound to each other are renamed to be complementary, then, the parallel operator is applied and finally, the restriction removes the actions of on the bounded interfaces that were not matched to complementary actions so that other components can no longer communicate via these interfaces.

For instance, composition of components SessionManager (represented by process SM) and BussinesLogic (represented by processes BL) could look like this:

```plaintext
(SM[logic_cmd.invokeStatement/logic.invokeStatement]  
|  
BM[logic_cmd.invokeStatement/cmd.invokeStatement]  
)(logic_cmd.invokeStatement)
```

In the SM process, all the occurrences of the `logic.invokeStatement` method in SM are replaced by `logic_cmd.invokeStatement` as well as occurrences of `cmd.invokeStatement` in BM. Thus, when SM and BM are composed together by the parallel operator, the names of methods on bounded interfaces match.

The CCS behavior model of an architecture, can be analyzed for deadlocks or livelocks. There are also various notions of equivalence (bisimulation), and refinement used to answer the question whether one model can replace another. In particular, the refinement relation proposed in [33] preserves stuck-freedom property.

2.3.1.2 CSP

Communicating Sequential Processes (CSP) [40] takes different approach to communication than CCS. There are no complementary actions and the communication among two processes running in parallel is represented by matching equal actions. Moreover, CSP provides two alternative operators. Their differs in who makes the choice whether it is the process itself (internal choice and external choice.

Let \( A \) is a set of actions. The basic CSP syntax is given by the grammar from Fig. 2.4
2.3. FORMALISMS AVAILABLE

P ::= STOP | SKIP | \alpha \rightarrow P | A |
P \square P | P \triangleleft P | P \parallel P | P[f] | P\setminus L

where \alpha \in \mathcal{A}, A \in \text{Id}, L \subseteq \mathcal{A} and f : \mathcal{A} \rightarrow \mathcal{A}.

Figure 2.4: CSP grammar

The grammar contains special purpose processes SKIP resp. STOP to represent successful termination resp. deadlock. \alpha \rightarrow P is a prefix operator, \square is alternative operator with external choice and \triangleleft is alternative operator with internal choice. Finally, \parallel is a parallel operator. Full version of the calculus provides timeout and interrupt operators as well as value passing. Apart from the recursion employing named processes (A), nameless processes can be used (\mu p.\text{tick} – > p represents a process producing unlimited number of ticks) to represent loops.

\begin{align*}
\text{Alt}^{\text{Int}}_{1} & \quad \frac{E \rightarrow E' \quad F \rightarrow F'}{E \parallel F \rightarrow E' \parallel F'} \\
\text{Alt}^{\text{Ext}}_{1} & \quad \frac{E \rightarrow E' \quad E' \parallel F \rightarrow F'}{E \square F \rightarrow E' \square F} \\
\text{Alt}^{\text{Ext}}_{2} & \quad \frac{E \rightarrow E' \quad \alpha \rightarrow F' \quad \alpha \rightarrow E'}{E \parallel F \rightarrow E' \parallel F'} \\
\text{Com}_{1} & \quad \frac{E \rightarrow E' \quad F \rightarrow F'}{E \parallel F \rightarrow E' \parallel F'} \\
\text{Com}_{2} & \quad \frac{E \rightarrow E' \quad F \rightarrow F'}{E \parallel F \rightarrow E' \parallel F'} \\
\text{Com}_{3} & \quad \frac{E \rightarrow E' \quad F \rightarrow F'}{E \parallel F \rightarrow E' \parallel F'} \\
\end{align*}

Figure 2.5: Selected CSP transition rules. All rules are symetric.

The Fig. 2.5 contains CSP transition rules that differs from the similar rules in CCS. The internal choice chooses one of its operands, moves to it by \tau, so that when the result LTS (depicted in Fig. 2.6) is composed in parallel

\begin{center}
\begin{tikzpicture}
\node (a) at (0,0) {a}; \node (b) at (1,0) {b}; \node (c) at (2,0) {c};
\draw[->] (a) -- (b); \draw[->] (b) -- (c);
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
\node (a) at (0,0) {a}; \node (b) at (1,0) {b}; \node (c) at (2,0) {c};
\draw[->] (a) -- (b); \draw[->] (b) -- (c);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
\node (a) at (0,0) {a}; \node (b) at (1,0) {b}; \node (c) at (2,0) {c};
\draw[->] (a) -- (b); \draw[->] (b) -- (c);
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
\node (a) at (0,0) {a}; \node (b) at (1,0) {b}; \node (c) at (2,0) {c};
\draw[->] (a) -- (b); \draw[->] (b) -- (c);
\end{tikzpicture}
\end{center}

Figure 2.6: LTS induced by CSP choice rules
The parallel composition is similarly to CCS used to represent communication among processes. In CSP, however, the parallel operator is parameterized by the set of actions to synchronize \( X \). In the result LTS (Fig. 2.7), there are only such transitions labeled by \( c \in X \) that represent synchronous execution of the \( c \) action in both processes. In contrast to CCS restriction, hiding transforms the \( c \) action to internal action \( \tau \).

Apparently, although CSP provides concepts known from CCS, there are slight differences in their semantics users must be aware of when writing models.

Application of CSP in the component context is similar to application of CCS. There are no complementary events, but the direction can be encoded into the name of the action during renaming. All in all, CSP is richer than CCS and provides special processes (STOP for deadlock or div for divergence) to explicitly model error states.

Apart from the structural operational semantics, there are also alternative semantics based on sets of traces or failures. Each provides different notion of equivalence and refinement.

**Wright** Particular example of CSP usage in the component context is the Wright specification language. Wright [4] is an ADL for defining a component-based architecture enriched by behavior specification. The key abstractions of the component model include component and its ports (interfaces), and connector and its roles (interfaces). An assembly is created by binary bindings of ports to roles. Each of the key abstractions is associated with its behavior specification in the form of a process in a subset of CSP.

The process describing behavior of a component on its ports is called computation, while the process specifying behavior of a connector on its roles is called glue. Having the behavior of ports, roles, glues and computations...
specified in CSP, automated checking of composability (based on refinement and deadlock-free testing) is possible. Here, the authors transform Wright specifications into plain CSP and use the FDR tool [66].

2.3.1.3 $\pi$-calculus

The $\pi$-calculus [56] stems from CCS by adding notion of mobility. In particular, the value passing principle introduced in the full CCS was extended to support also passing of channel identifiers. In other words, while CCS distinguishes channel names from variable names and constants, in $\pi$-calculus, a variable can be used to specify a communication channel. Thus the change of the variable content changes also the communicating party.

Apart from the operators known from CCS, there is replication operator $!P$ and creation of a new name ($\nu x)P$. As an example, consider the following process:

$$!(x)target(x).data(y).\tau\langle y\rangle$$

The process first accepts target and data and then sends the data to the target. Since there is the replication operator, it can process arbitrary number of requests in parallel.

Darwin [51] and $\pi$-ADL [58] are examples of application of $\pi$-calculus in the component context. Both use $\pi$-calculus to model evolving architectures.

2.3.1.4 Other Process Algebras

So far, we have discussed CCS, CSP and $\pi$-calculus in details sufficient to explain main ideas behind process algebras in general and to demonstrate different options their authors taken. However, there are also other process algebras.

One of them is LOTOS [16] (Language of temporal ordering specification), a specification language designed to specify parallel distributed systems. LOTOS supports all concepts present in CCS or CSP. In particular there are both synchronous and non-synchronizing parallel products; the former is based on blocking semantics, i.e., a process has to wait until its counterpart is ready to proceed with a complementary action. Moreover, LOTOS allows users to explicitly express structure of specification. In particular, LOTOS distinguishes process type and instance. Process type is not just a named expression. It also specifies communication gates used to synchronize the
process with the environment. Process type specifies formal gates while actual gates are determined by particular process instance. Process instances are used to construct more complex process types.

Moreover, Full LOTOS [16], supports also data modeling. In the component context, bindings and composition are modeled via gates and composition operators. Similarly to other process algebras, LOTOS can also serve as a target language for translation of a component specific formalism (e.g., [6], [24]). It is worth to mention that LOTOS is an international standard (ISO 8807).

By introducing structured specifications and process instances, LOTOS gets closer to the programming languages that operate with similar concepts. In other words, LOTOS made a step from the world of process algebras which prefers mathematical simplicity of concepts (to make proofs easier) in the direction of practically usable specification language that prefers readable and maintainable specifications. The step, however does not seem to be big enough to ensure large number of users from practice.

The Algebra of Communicating Processes (ACP) [11] on the other hand emphasizes the algebraic aspect [7]. There are several semantical models. ACP also allows specifying the way the processes running in parallel communicate with each other. In particular, the parallel operator is parameterized by a function that prescribes what pairs of actions are to be synchronized and what is the corresponding action in the result. The function allows specifying both CCS and CSP communication style.

Then, there are other process algebra aiming at specific area. Several formalisms originated within the Sensoria project [73] to model software services (SCC [14], SOCK [37] and COWS [31]). They provide service oriented concepts (e.g. session) as a first class entity. Performance Evaluation Process Algebra (PEPA) [39] is a stochastic process algebra. Alternative operator, as well as prefixes are equipped with probabilistic distribution which allows to study properties like throughput, utilization, response time, resource consumption and so on.

2.3.2 Automata Based

Automata based languages define the LTSes of individual communicating systems graphically. Such definition is straight-forward since it does not require deep knowledge of the semantics of the given formalism. On the other hand drawing complex systems can show to be tedious and time consuming work.

Individual formalisms differ in the supported actions (labels), composition
operator and by studied properties and relations over models.

2.3.2.1 Interface Automata

Interface automata introduced by Alfaro and Henzinger [30] distinguishes input actions (?name), output actions (!name) and internal actions (name). The parallel composition is used to form more complex systems from simple ones.

Generally, the composition is defined on pairs of composable (having fitting sets of input and output actions) automata $A_1$ and $A_2$. A synchronous product automaton $P_A = A_1 \otimes A_2$ is created; $A_1$ and $A_2$ synchronize on complementary actions. The result contains error states if an automaton emits an output action and the counterpart automaton is not able to accept it (there is no complementary input action).

The existence of an error state in the product $P_A$, however does not mean that $A_1$ and $A_2$ cannot work together. If $A_1$ and $A_2$ forms an open system (i.e. there are still some input and output actions) there can be an environment $E$ that avoids the error state. If such automaton $E$ exists, $A_1$ and $A_2$ are considered compatible. From this point, the most ‘helpful’ environment (i.e. environment that does not cause any error on its own and does not force $P_A$ to do anything) is the one which accepts all output actions of $P_A$ and does not supply any input actions to $P_A$. Following this reasoning, the composition $C$ of $A_1$ and $A_2$ ($C=A_1 || A_2$) is derived from $P_A$ by omitting each transition for which an error state is reachable from its target state via sequence of output or internal actions. Such approach is optimistic—‘being compatible’ means, that $A_1 || A_2$ does at least some (nonempty) action. In other words, the components modeled by compatible automata $A_1$ and $A_2$ makes sense to make subject of assembly since with the right environment there won’t be any error.

The refinement relation ($\geq$) supported by the formalism of interface automata preserves the compatibility. In particular, if $P$ and $R$ are compatible and $P \geq Q$ then also $Q$ and $R$ are compatible. Such notion of refinement can be directly used in the component context to verify hierarchical architectures.

The refinement relation is based on alternation simulation which directly aims the error supported by interface automata (missing complementary input action). In particular for $P \geq Q$, the alternation simulation is a simulation relation among states of $P$ and $Q$ such that it requires states from $P$ to emit more output actions than the corresponding states in $Q$ and on the other hand it requires states from $Q$ to accept more input actions than the corresponding states in $P$. 
2.3.2.2 Modal I/O automata

I/O automata [50] are syntactically similar to the interface automata, however the notion of compatibility and refinement is different. In particular, while considering the same kind of error as interface automata, the I/O automata takes pessimistic approach to the composition. Once the composition operation yields an error, the result is considered erroneous even though there is an environment where it could work correctly. Implications of the pessimistic approach on refinement are discussed in [30]. As refinement, simulation relation is used for I/O automata.

Modal automata [47] involve two transition relations—“must” and “may” transitions. In contrast to interface automata, there are only external and internal actions (external actions does not divide into input and output actions). While the “must” transition represents obligation of the model to perform the action, the “may” transition represents just option to perform the action. The definition of compatibility reflects the distinction in an intuitive way (a ‘may’ transition must be matched with ‘must’ transition in the counterpart specification) as well as the definition of refinement (observational modal refinement).

It appears, that interface automata are a special case of modal automata. There is a simple transformation function that transforms an interface automaton into modal automaton. The transformation function preserves the refinement relation.

Modal I/O automata [47] combine I/O automata with modal automata. In particular, there are “may” and “must” transition relation as well as input, output and internal actions. In [48], theory of modal I/O automata is further developed and four different notions of model consistency (i.e. existence of implementation) and refinement are presented.

Finally, in [9] weak modal compatibility for modal I/O automata is presented together with the proof that it is preserved by one of the refinements presented in [48]. There is also overview of refinement and compatibility notions.

2.3.2.3 CIA

Component Interaction Automata (CIA) are another example of a formalism directly based on automata. A component-interaction automaton is a LTS, where each label is a triple identifying the sender component, the action label, and the receiver component, respectively. Either the sender or the receiver can be a wildcard representing an unbound interface. The formalism is designed for composition of hierarchical components and features also a
refinement (substitutability) relation based on equivalences [26]. This relation is, however, specific to a particular environment. Thus, it does not solve the problem of component refinement in general. Moreover, we believe that equivalence is a too restricting relation which forbids useful scenarios (e.g., substitution of a nondeterministic component by a deterministic one). The formalism is very general and other formalisms can be transformed into it and further analyzed using the DiVinE model checker [8] (e.g., for reasoning on execution paths).

2.3.3 Other

Unified Modeling Language (UML) [13] addresses the problem of behavior modeling as well. However, the semantics of particular diagrams aiming at behavior modeling (activity, state machine, communication, interaction overview, sequence, and timing diagrams) is not defined precisely enough to allow formal verification. Also the formal notion of composition and refinement is missing. Although UML supports profiles précising semantics for special cases, none of them has been generally accepted yet. UML, however, is often used in industry to visualize software design and even for prototype code generation.

Another generic specification language is Promela, the input language of the Spin model checker [41]. Although, Promela has an excellent support for modeling control flow, data flow, and modes, there is no easily way to handle composition, component hierarchy, and no notion of refinement. Nonetheless, since the Spin model checker is a mature tool having been developed for many years, transformations from other specification languages into Promela are frequently introduced, e.g., in [70] and [63].

In addition to the aforementioned languages and formalisms, several other approaches have been developed, mostly in academia, but not widely accepted. STSLib [32] is an attempt for employing Symbolic Transition Systems (STS)[62] as a behavior specification formalism for software components. Composition of components can be analyzed using the CADP [35] tool after translation of the STS specifications into LOTOS. Unfortunately, no refinement relation is supported by STSLib. The focus of STSLib is also generation of Java code from an STS specification.

The pNET [25] formalism is another example of LTS enhanced with symbolic representation of actions on the transition labels. It is used for specification of distributed objects in the ProActive library and the Grid Component Model (GCM) and it supports hierarchical composition. The CADP tool is used for analysis of implicit composition properties, e.g., absence of dead-
lock, as well as user supplied properties in $\mu$-calculus. Again, no refinement relation is defined for pNET.

In AutoFocus, [17] a basic concept is a (possibly infinite) stream of messages. A component accepts such streams on its input channels and generates streams on its output channels. The behavior is conveniently specified by a number of graphical diagrams (e.g., State Transition Diagram, Message Sequence Chart). User specified properties in LTL and CTL are analyzed (reasoning on execution paths) using either NuSMV or Cadence SMV backend. Composition is defined, but composition verification is not explicitly considered. No refinement relation is studied either.

Java/A [10], [43] is an approach which applies an automata based formalism in context of particular component system. Java/A extends the Java language to capture architecture directly in source codes which prevents the architectural erosion.

The architectural elements (ports and components) are equipped with behavior specification in form of interface automata. Correctness of an assembly is understood as deadlock-freeness and it is checked in several steps involving product of automata, reduct of automata to particular port and interface automata refinement. By checking the deadlock-freeness in several steps (by pairs of components) involving the interface automata refinement, Java/A resembles the Wright approach.

### 2.3.4 BP

The formalism of behavior protocols (BP) [60] [1], was developed for behavior specification of software components. From the beginning, it aimed at the SOFA component model [21]. It was also applied in the Fractal component model [18].

Basically, behavior protocols are a simple process algebra (since they satisfy the 7 basic laws mentioned in [7]) designed to describe the externally observable behavior of a component. The semantics is trace-based. The behavior protocol specifying behavior of a particular component on all of its external interfaces (frame) is called frame protocol.

The naming scheme of atomic actions corresponds to the basic input and output actions as follows: $?a \uparrow$ stands for accepting an invocation of a method $a$, $!a \uparrow$ for issuing an invocation, $?a \downarrow$ means accepting the response (end) of a method execution and finally $!a \downarrow$ means issuing the response. Syntactically, a behavior protocol is an expression composed of actions, operators, and parenthesis () and {}.

The basic operators are: ';' sequencing, '+' alternative, '*' repetition, and
2.3. FORMALISMS AVAILABLE

'\|' parallel composition with no synchronization. Moreover, there are constructs to easily encode method calls and functionality (reaction) of methods; ?a stands for ?a↑; !a↓, further more ?a{P} stands for ?a↑;P;!a↓, while !a stands for !a↑;?a↓ and !a{P} for !a↑;P;?a↓, where P is a reaction of the method a. The interpretation of an alternative operator (+) in terms of whether the choice is done internally or externally is specified implicitly: If the traces generated by both operands start with an input action (such as ?a↑) the choice is made externally, while if they start with an output action (such as !a↑), the choice is done internally. However, the alternative traces starting with the same action (no matter whether input or output) are handled differently: Due to the trace semantics is the underlying automaton transformed to deterministic one and then the previous rule applies to the end of the common prefix. For instance, ?login{!RegisterPrivate} + ?login {!RegisterPublic} is handled as ?login{!RegisterPrivate + !RegisterPublic}, i.e. the choice is internal. Mixing of input and output actions at the beginning of the alternative traces cannot be generally interpreted in either way and needs to be avoided.

For purposes of component composition, BP introduces the consent operator. The consent operator ∇m (parameterized by a the set of actions m) produces all interleavings of the traces generated by its operands; however, in any interleaving, there is the obligation to merge the complementary output '!' and input '?' actions from m into an internal action prefixed by τ (e.g., !a↑ and ?a↑ are merged into τa↑) similarly to CCS. Moreover, original unsynchronized actions from m are removed.

The set m of actions names given as a parameter of the composition determines communicating actions. On the other hand, non-communicating actions are unmodified in the interleaving (they can participate in further compositions with other protocols). Basically, input actions are blocking (the process waits until a complementary output action is emitted by the counterpart process, while internal and output actions are non-blocking. In addition to actions reflecting desired behavior, the consent operator produces also error actions to express following errors: (a) bad activity (εa) occurs when a communicating output action cannot be matched (i.e. there is no complementary input action ready) (b) no activity (deadlock; ε⊘ ) all of the communicating actions ready to be composed are input actions so that no process can proceed.

Composition of components AuthLogic, Timer, RndGenerator and HashFunction from the Fig. 1.2 (represented respectively by processes AL,T,RG and HF) uses the consent operator and renaming.

Arch=(

27
AL[hash_prov.invokeStatement/hash.invokeStatement,  
idGen_prov.generate/idGen.generate,  
req_timer.invokeStatement/timer.invokeStatement]\n∇{req_timer.invokeStatement}\nT[req_timer.invokeStatement/req.invokeStatement] \n∇{idGen_prov.generate}\nRG[idGen_prov.generate/prov.generate] \n∇{hash_prov.invokeStatement}\nHF[hashProv.invokeStatement/prov.invokeStatement]

Since this composition forms an open system, even when the resulting behavior contains an erroneous trace, there can be an environment where the result of composition would work correctly.

In general, composition in SOFA is either used to compose the top level components or to form a composite component. In the former case, the result of the composition is a closed system. It is directly checked for correctness. In the latter case, however the result is open. Each composite component, however, has its own behavior specification - the frame protocol. The frame can be seen from two perspectives. On one hand, it describes how the composite component interacts with its environment. On the other hand it also specifies how the frame interacts with the component’s internals. In particular, to create a protocol of the desired environment from a frame protocol $F$, it suffices to reverse the direction of actions, i.e., to swap ‘!’ and ‘?’ in $F$. The resulting protocol is called inverted frame protocol (denoted $F^{-1}$).

In the example above, the closed system behavior would take the form $Arch \, \n\nD \, SM^{-1}$, where SM is the behavior protocol of the SessionManager component and D is a set of methods delegated to upper level.

Refinement relation in BP is defined between an architecture protocol and the corresponding frame protocol. An architecture protocol $A$ refines a frame protocol $F$ if the composition $F^{-1}$ yields no communication errors. This means that the architecture protocol $A$ of an assembly can be used instead of the frame protocol of the encapsulating component. This way, the refinement checking of two open systems (frame and architecture) is transformed to checking of correctness composition of two protocols forming a closed system.

For Java implementation of SOFA components, this correctness is verified via a modified Java PathFinder and an environment code generator [59].

**Component Reliability Extensions for Fractal**  BP were employed in a project funded by France Telecom R&D [42, 44] aiming at extension of
the Fractal component model by behavior description. A part of the project was a case study that modeled a system controlling passenger access to the internet at an airport; the case study consisted of roughly twenty non-trivial components. Modeling behavior of a component system using BP turned out to be beneficial in the following:

Scaling. The textual notation of BP scales much better than a graphical one that might be considered better for simple cases. Also, BP notation is closer to a programming language than ordinary process algebra because recursive equations are replaced by the repetition operator.

BP as an abstraction of implementation. Correspondence of the behavior model and actual implementation in Java can be verified - consequently, this allows verification of trace-related properties on the model level which is usually much easier than on the code level.

Compositional verification. BP come with tools for composability and refinement verification - first, hierarchy of components allows verification of communication correctness separately for each particular level of component hierarchy, and second, inverted frame protocol allows verification of component correctness under the “maximal” correct usage of the component (i.e., all declared functionality is verified).

On the other hand, the case study revealed several issues. The important ones include:

Absence of explicit component state. Component state is encoded only implicitly, by the history of method calls which leads in nontrivial cases to clumsy specifications (e.g. to overuse of alternatives and introduction of “artificial” methods).

Absence of method parameters. Data-dependent alternatives have to be encoded with an over-abstraction such as in \( \text{?login} \{\text{!RegisterPrivate + !RegisterPublic}\} \), where the internal choice between the registering calls is to be in reality driven by the value received as an actual parameter of the login method. Overall, this decreases specification accuracy in terms of both comprehension and correspondence with code in primitive components.

Lack of multiparty synchronization. The concept of complementary input and output actions (inspired by CCS) appeared to be limiting in capturing situations when multiple components had to wait for an event, e.g., for the end of initialization phase.

Guaranteeing refinement transitivity. When combining input and output actions on the beginning of alternative traces, the refinement relation looses transitivity in general. To illustrate this issue, consider three frame protocols: \( \text{A!x} \), \( \text{B!x+?y} \), and \( \text{C?y} \). Here \( \text{C} \) refines \( \text{B} \) and \( \text{B} \) refines \( \text{A} \), however, \( \text{C} \) does not refine \( \text{A} \). The problem is in the protocol \( \text{B} \) — it puts together input and output action using the alternative operator. To cope with this problem,
either refinement has to be addressed in a different way, e.g., a similar one described in [30], or the syntax of the BP language has to be modified to avoid such situations and guarantee refinement transitivity.

Another issue regarding the refinement realized by the consent operator is illustrated by the following example. Consider again three protocols: A: \(!x;!x\), B: \(!x+!x;!x\) and C: \(!x\). Here, again, C refines B and B refines (incorrectly) A, but C does not refine A. This is caused by a subtle flaw in original consent definition which does not address the agreement on final states properly (i.e., B should not refine A). This issue can be solved by an “update” of the consent operator to treat final states differently, so that in the example B would not refine A anymore.

**Lack of call event pairing in parallel activities.** A method call is modeled by a pair of events representing begin and end of method processing. When parallel activities contain the same method calls, it is not ensured that pairs of events forming single method calls comes from the same process. When these parallel processes are used to model threads, unnatural situation when the result of method call invoked by one thread is accepted by another thread.

To illustrate the problem, consider the following fragments of three frame protocols (methods with the same name are indexed for the purpose of explanation):

A: (?!a;?!y;?!x) | (?!y;?!c)
B: (?!a;?!x;?!x) | (?!x;?!c)
C: ?x1{?!y} | ?x2{?!y}

Consent composition of these three frame protocols yields a bad activity with the error trace (the first number in the index represents output action while the other one the input action) \(\tau x_{2-2} \uparrow; \tau y_{2-2} \uparrow; \tau y_{2} - 2 \downarrow; \tau a \uparrow; \tau a \downarrow; \tau x_{1-1} \uparrow; \tau x_{2-1} \downarrow; \epsilon b \uparrow\). In this case, the bad activity is caused by the calling of the b method after an incorrect match of the return from the x method. As another example consider the following pair of frame protocols:

A: ?x1{?!y} | ?x2 {?!z}
B: !x1{?!y} | !x2 {?!z}

On contrary, composition of these protocols does not yield an error although the model is misleading - obviously the calls \(\tau x_{1-2} \uparrow\) and \(\tau x_{2-1} \uparrow\) should yield an error. This issue is however a consequence of the trace semantics of BP, not of absence of missing pairing information.

The case study served as source of inspiration for BP improvements that lead to a specification language closer to the imperative programming languages - EBP.
2.3. FORMALISMS AVAILABLE

2.3.5 EBP

The formalism of EBP shares the goal with BP specification of externally observable behavior of software components. In addition to that, it supports data to capture component state and method parameters. Also, it supports the multiparty synchronization. The structure of an EBP frame protocol takes the following form:

\[
\text{component Component_name} \{ \\
\quad \text{types} \{ \text{Definition of enumeration types} \} \\
\quad \text{vars} \{ \text{Definition of state variables} \} \\
\quad \text{behavior} \{ \text{protocol} \} \\
\}
\]

The definition of types and variables helps to capture the component state. The defined variables are referenced in the behavior part (protocol). The syntax of the protocol stems from the syntax of BP. It contains an expression composed of atomic events, operators, and abbreviations. The atomic actions are enhanced by introducing (i) assignment to a state variable (\(\text{variable} \leftarrow \text{value}\)), (ii) accepting a method call (\(\text{?if.method(type}_1\text{ par}_1, \text{type}_2\text{ par}_2, \ldots, \text{type}_n\text{ par}_n)\)) where \(\text{par}_i\) is a formal parameter of type \(\text{type}_i\), (iii) issuing a method call with parameters (\(\!\text{if.method(val}_1, \text{val}_2, \ldots, \text{val}_n)\)), where \(\text{val}_i\) is an actual parameter), and (iv) multiparty synchronization action (@action).

Since the variables are of finite enumeration types, each EBP frame protocol, as well as in the case of BP, is represented by a finite LTS. Technically, the basic idea is that each combination of a particular n-tuple of variable values and the position within the protocol already processed (history of method calls and variable assignments) determines the LTS state. In a way similar to BP, syntactic abbreviations expressing method call reactions are defined. In addition to the operators defined in BP, the formalism of EBP introduces two new operators: while and switch. Both contain conditions involving state variables, method parameters and constants. Intuitively, while the while operator denotes repetition of its operand as long as the condition holds, the switch operator denotes a choice determined by the content of particular variable.

CoCoME When it comes to applicability of different modeling approaches in practice, the CoCoME contest [65] has to be mentioned. The goal of the contest was to assess strengths and weaknesses of various approaches to modeling software components. The common assignment, modeling a trading
system for a chain of stores, formed a common platform for the evaluation by the jury committee.

Based on the comparison of the BP [19] and EBP [22] specifications of the CoCoME assignment we learned that still being very simple, EBP can capture more complex behavior in a comprehensive way closer to an imperative programming language than BP can. In particular, by having component states explicitly represented by a state variable, the EBP frame protocol is less error-prone and more readable. Also introduction of parameters helps avoiding mangling parameters into the method names.

Overall, the following issues/flaws of BP have been addressed: absence of component state and method parameters (even though only of enumeration types) and lack of multiparty synchronization. Guaranteeing refinement transitivity is still not fully addressed in general, but the switch operator syntactically helps avoiding mixing of input and output actions, which is the main source of the issue. Also, lack of call event pairing in parallel activities concludes the list of issues resolved later by TBP.

2.4 Goals Revisited

As apparent from the description of the formalisms above and also from the case studies, crafting of behavioral models is time consuming and error prone task. One of the reasons is that the modeling languages of the formal frameworks are too different from imperative programming languages used by developers at on daily basis. In order to push behavioral modeling closer to the day-to-day practice and lower the effort needed to craft a behavioral model, we propose TBP - a behavioral specification language fulfilling the following goals:

**Goal 1**  TBP will feature a straight-forward syntax and semantics to capture the behavior of a component, as well as the assumptions on the behavior of its environment.

(a) We would like the parts of TBP describing (specifying) the behavior of component to resemble an imperative programming language in the following:

i. The specification will be structured in a similar way as declaration of classes in object-oriented languages. There will be dedicated constructs for declaration of state variables, provided methods, as well as threads.
2.4. GOALS REVISITED

ii. There will be limited support for data modeling. In particular TBP will support variables of enumeration types used the same way as variables in imperative languages.

iii. TBP control flow statements will take the form known from Java. Moreover, to support abstraction, there will be a special construct representing a nondeterministic choice.

iv. TBP will provide the concept of threads as known from programming languages including synchronization

(b) TBP will support specification of assumptions on an environment based on process algebras.

i. The assumptions will capture the allowed sequences of provided method calls.

ii. Each assumption will be related to a subset of methods provided by the specification.

iii. Overall approach to the environment behavior will be permissive—behavior not explicitly prohibited will be allowed.

Goal 2 Theoretical framework of TBP will provide guarantees of correctness in the following sense:

(a) The notion of correctness will be based on the comparison of actual behavior of a closed system to the expected behavior expressed in the form of assumptions. There will be two kinds of assumption violations: bad activity and no activity.

(b) Composition operator will reflect the concept of threads.

(c) There will be a preorder refinement relation considering both, bad activity and no activity to support hierarchical architectures. The refinement should work with respect to unlimited number of threads entering the specification from the environment.

(d) Although getting closer to the imperative languages, the analyses of correctness should be decidable.

To achieve this goal, the theoretical framework of TBP has to precisely define the semantics of concurrent thread execution in a form suitable for definition of correctness and of the refinement relation.
2.5 Overview of Publications

This section contains a commented list of publications related to the topic of this thesis I participated on. First, there is the CoCoME case study. It helped to identify weak points of BP and EBP already discussed in the respective sections. We employed two different approaches to model the assignment. While the Fractal model was equipped with behavior specification in BP [19], the SOFA [22] model was equipped with EBP specification.


For the purpose of checking EBP models crafted within the CoCoME contest, translation of EBP specifications into Promela language was proposed and published in [45]. The tool is available as part of the Sofa2 component model [68].


Inspired by the weaknesses of BP and EBP, first ideas and requirements posed on the new formalism were published in [46].


One of the issues identified in the case studies was that crafting behavior specifications for individual components is a demanding task. This issue is especially burning in the case of legacy applications when the code is already available. It still makes sense to create models of these applications to ease maintenance and modifications of the application. In the case of legacy applications, the information about component behavior is already included
in the project sources. Thus, extracting the behavior specification from the
sources saves a lot of effort. In particular, while [61] aims at extraction
of behavior specification of primitive components implemented in Java, [5]
considers semi-automatic extraction of architecture.

[61] Tomáš Poch and Frantisek Plasil. Extracting Behavior Specification of
Components in Legacy Applications. In Grace A. Lewis, Iman Poernomo,
and Christine Hofmeister, editors, CBSE, volume 5582 of Lecture Notes in

[5] Nicolas Anquetil, Jean-Claude Royer, Pascal Andre, Gilles Ardourel, Petr
Hnetyka, Tomas Poch, Dragos Petrascu, and Vladiela Petrascu. JavaCom-
pExt: Extracting Architectural Elements from Java Source Code. In WCRE
’09: Proceedings of the 2009 16th Working Conference on Reverse Engineer-

2.6 Overview of Contribution

Based on the experience from the papers presented in the previous section,
we propose the specification language of Threaded Behavior Protocols. The
proposed language would suit better to the case studies discussed so far
[42, 44, 19, 22]. TBP would be also a better means for behavior extraction
presented in [61]. This section briefly overviews the key contributions of this
thesis by listing the novel features of TBP compared to the other modeling
languages.

Regarding specifications in TBP, the main contribution of TBP is that its
syntax and semantics resemble an imperative programming language (Goal
1.a). Thus, the user, typically a programmer, operates with the well known
categories. For instance, threads in TBP use the same concepts as threads
in Java, including the principle of synchronization. Apart from the imper-
ative aspects of the component behavior, user provides also assumptions on
the environment behavior (Goal 1.b). Although these assumptions are inte-
gral part of the specification, they are clearly separated from the imperative
part. Thus, the user is always aware whether he/she specifies the behavior
in an imperative manner or an assumption. The assumptions are specified
independently on each other, and the overall approach to the environment is
permissive.

The semantics of TBP is defined in two stages. Since the TBP model
keeps all information about threads, it is straightforward to support thread
aware composition at this stage (Goal 2.a). By simulation of TBP model,
a computation in the form of a finite LTS is obtained. For LTS, we define
the notion of correctness and the refinement relation. Notion of correctness includes inherent deadlock in the imperative part as well as the violation of the assumptions (Goal 2.b). Assuming assumptions formulated as sets of allowed traces, we show that the bad activity and no activity cover all possible violations. The notion of refinement builds upon the alternation simulation introduced in [30]. However, the relation is extended to consider also the no activity error (Goal 2.c). Moreover, the definition of refinement is modular, which allows seamless modification of key aspects (e.g. dealing with nondeterminism). Unfortunately, the delivered notion of refinement is limited by the maximum number of threads entering the specification from the environment in parallel. Apart from the prohibited recursion, the limit on the number of threads is one of the limitations taken to keep the analyses decidable (Goal 2.d). For the same reason, we also do not support dynamic thread creation. Still, we believe that we did not harm the practical expressiveness of the language by these limitations.

Finally, to demonstrate the features of TBP we provide an illustrative example.
Threaded Behavior Protocols

3.1 Syntax

This section presents the formalism of Threaded Behavior Protocols (TBP) from the user point of view. In particular, intuitive notion of component execution is used to explain meaning of individual concepts. Moreover, since specification of individual components is a typical usage scenario, this chapter, unless explicitly stated, considers just specification of a single component. Note, however, that the formalism of TBP allows expressing syntactically also the result of a composition.

The basic structure of a TBP specification is formed by five parts—declarations of types, declarations of state variables, reactions on method calls, threads and provisions. While the reactions and threads specify the behavior exercised by the component itself in the imperative manner, provisions specify the behavior of an environment assumed by the component. The assumptions are stated over sequences of provided methods.

```plaintext
component ComponentName {
    types {
        ...
    }
    vars {
        ...
    }
    provisions {
        ...
    }
    reactions {
        ...
    }
}
```
3.1.1 Relation to the Component Model

TBP specification describes the behavior a component on the level of method calls observable at the component boundaries. In particular, the description distinguishes provided, required, and internal methods. Provided methods are provided by the component to the environment. The environment is expected to call them with respect to certain assumptions as explained later. On the other hand, required methods are methods invoked by the specified component. The methods must be provided by another component in the environment. The internal methods are invoked from threads running in the context of the component.

The concept of provided and required methods (often formulated using interfaces - groups of methods) is already present in a typical component model. Since TBP is designed to be an extension of such model, there is no need to provide explicit syntax construction to determine which methods are provided and which methods are required. Later, in the semantics section (Section 3.2), we assume $\Sigma_{req}$, resp. $\Sigma_{prov}$, resp. $\Sigma_{int}$ to denote the set of required, resp. provided, resp. internal methods of the component. Their content is taken from the component model.

Moreover, the component models define compositions of components via bindings among the interfaces. Thus, since the information about bindings can be taken from the component model, there is no need to specify syntax for composition of TBP specifications.

3.1.2 Type Declarations

The types section defines enumeration types used to declare variables and method parameters. The types may be used for declaration of method parameters, state variables and local variables.

```plaintext
types {
  result = {OK, FAILED};
  mode = {INIT, RUNNING, MAINTENANCE, SHUTDOWN}
}
```
3.1. SYNTAX

The fragment contains declaration of two enumeration types. While the `result` type consists of two enumeration values OK and FAILED, the `mode` type consists of four enumeration values.

### 3.1.3 State Variables

The `vars` section defines state variables important for behavior of the component. The variables can be accessed only from the component (i.e., reaction or thread). A state variable declaration consists of a name, type, and initial value. Later on, within the threads and reactions sections, the variables are referenced by assignments and conditions.

```
vars {
    mode actualMode = INIT;
}
```

The fragment contains declaration of the `actualMode` state variable. The variable’s type is `mode` and the initial value is `INIT`.

There is a special type of variable—`mutex`. A mutex variable serves as a synchronization object, upon which threads can synchronize, e.g., to achieve mutual exclusion.

### 3.1.4 Reactions

The `reactions` section contains description of the actual behavior performed by the component in reaction on a method call (either provided or internal method). Each reaction specified in the section consists of the method name, declaration of arguments and body. The body begins with declaration of local variables followed by the actual behavior.

```
reactions {
    interface.methodName(ArgType1 argName1, ArgType2 argName2, ...):ReturnType {
        LocalVarType localVar = FAILED;
        ...
    }
}
```

#### 3.1.4.1 Local Variables and Arguments

Each local variable declaration specifies a name, a type, and an initial value. Arguments, on the other hand, consist just from the type and name. Local variables and arguments may be accessed only from the reaction body.
Moreover, the value is not shared by parallel executions of the body - each execution has its own copy.

3.1.4.2 Elementary Actions

The actual behavior consists from elementary actions composed together by control flow operators. There are three kinds of elementary actions.

- **Method call** — \texttt{i.a(argValue1, argValue2, ... )}
  The execution of the current reaction is postponed and the \texttt{a} method on the \texttt{i} interface is invoked with parameters \texttt{argValue1}, \texttt{argValue2}, etc. The \texttt{a} method is either required or internal. The execution of the current reaction is resumed when the method \texttt{a} is finished.

- **Return** — \texttt{return value}
  Explicit termination of reaction. The returned value may be assigned to a variable at the calling site.

- **Variable assignment** — \texttt{var=val}
  The content of the \texttt{var} variable is changed to the \texttt{val} value. The variable is either local variable or state variable and the value is either constant (enumeration value), another variable or a method call. In the last case, the return value of the invoked method is assigned to the variable.

3.1.4.3 Control Flow Operators

Control flow operators correspond to the control flow statements known from imperative languages. Additionally, explicit notion of internal non-determinism is supported in the form of expression \texttt{?}.

The condition used in control flow operators is a boolean expression which takes one of following forms

- **equality** — \texttt{var == val}
  Compare value of variable \texttt{var} with value \texttt{val} of corresponding types. The variable is either local variable or state variable and the value is either enumeration value, local variable or state variable. The result is given by the actual content of variables during execution.

- **non-deterministic value** — \texttt{?}
  The result is non-deterministic — either \texttt{true} or \texttt{false}. The choice is internal and it is not influenced by an environment.
3.1. SYNTAX

- application of a boolean operator — || && !
  Allows composing more complex conditions in straightforward manner. Since neither the comparison nor the non-deterministic value does have side effects, it does not matter whether short-circuit evaluation is used or not. It is valid to apply the operator to a non-deterministic value.

Following operators are available to construct more complex behavior descriptions

- Sequence operator

```plaintext
i.m();i.n()
```

The operands are executed sequentially. In the example, the n method is invoked immediately after m.

- Alternative operator

```plaintext
if(condition){
    thenBranch
} else {
    elseBranch
}
```

Depending on the actual value of the condition, either thenBranch is executed or elseBranch is executed.

- Switch operator

```plaintext
switch(var){
    case constA: aBranch
    case constB: bBranch
    ...
    default: defaultBranch
}
```

Depending on the actual value of the var variable, at most one branch is executed. All constants (constA, constB) must be of the same type as the variable. Either the branch which corresponds to the actual value of the variable is executed or the default branch is executed. If there is neither the default branch nor the corresponding branch, nothing is executed. If the non-deterministic expression ? is used instead of var a random value of the type determined by constants is used to choose a branch.

- Repetition operator
The block is being executed repetitively as long as the condition holds. If the non-deterministic value \( ? \) is used as the condition, the loop may be executed any number of times (i.e. non-deterministic decision whether to continue or not is taken before each iteration).

- **Synchronized operator**

```plaintext
sync(m){
  block
}
```

The block cannot be executed by more than one thread in one moment. In particular, \( m \) must be a state variable of the built-in type `Mutex`. Technically, the `Mutex` is a two value type and mutual exclusion is achieved by atomic text-and-set operation over the synchronized variable.

- **NULL operator**

```plaintext
NULL
```

Represents an empty action

### 3.1.5 Threads

The *threads* section contains description of the autonomous behavior performed by the component’s internal threads. Each thread is declared by a name and a body, which consists of local variable declarations followed by description of behavior. The constructs used to describe reaction bodies are used also to describe thread behavior. Each thread may invoke required methods as well as internal methods. It may change a state variable or a local variable. The number of threads is constant, each thread starts its execution immediately at the beginning of the model execution and once a thread reaches its end it does not perform any action.
3.1.6 Provisions

Previous sections specify the behavior exercised by the component itself in the imperative manner. In contrast, the purpose of the provisions section is to declare the allowed usage of the component — assumptions posed on the environment (i.e. limit the set of environments the component is supposed to cooperate with). Key distinction from the imperative part is that the assumptions posed on the environment should be weak. While the imperative parts specify the behavior as precisely as possible, the assumptions must not prohibit too many environments to enable reuse of the component in different contexts.

Each assumption is specified as a set of allowed sequences of component’s provided method calls and returns. The set of allowed environments then includes all environments that do not violate the assumption.

Each provision consists of an expression and a set of methods it constrains. The expression defines a language in similar manner as a regular expression. An environment fulfills the provision if all traces it generates, when restricted to the methods from the set belong to the language.

```
provisions {
    { expr1 } for {i.m, i.n}
    { expr2 } for {i.n, i.q}
}
```

In the example, `expr1` is an expression related to the methods `m` and `n` on the interface `i` while `expr2` is related to methods `m` and `q`.

3.1.6.1 Elementary expression

An elementary expression consists of two actions—a provided method call and the corresponding return. Both actions can contain additional data—either the method call parameters or a return value. Each method used in the expression must be member of $\Sigma_{Prov}$.

```
{ i.m(val):retVal } for {i.m}
{ i.n() } for {i.n, i.q}
```

The first provision states, that the `m` method on interface `i` is expected to be invoked exactly once by the environment. Moreover, the argument value must be equal to `val` and the method returns `retVal`. The second provision states that the environment calls the method `n` exactly once. The arguments used and the return value may be arbitrary, however. Moreover, the `q` method cannot be invoked by the environment at all.
3.1.6.2 Operators

To construct more complex provisions, the following operators are available.

- **Sequence operator** — 

\[
\{ \text{i.m(val);i.n()} \} \text{ for } \{\text{i.m, i.n}\}
\]

Sequence of operands. In the example, the environment must call the method `m` with argument `val` and later it must call the method `n`. In between, before and after method calls, however, any number of other methods than `i.m` or `i.n` is allowed, since the other methods are not guarded by the provision.

- **Alternative operator** — 

\[
\{ \text{i.m()+i.n()} \} \text{ for } \{\text{i.m, i.n}\}
\]

Alternative of operands. In the example, the environment is expected to call either the method `m` on the interface `i` or the method `n` on the interface `i`. As opposed to the internal choice of imperative parts (the if operator), the choice is not to be done by the provision itself—the decision is taken either by environment (method calls) or by imperative parts of the specification (method returns). The following example, the alternative operator is used to influence the behavior according to a return value.

\[
\{ \{\text{i.login():ACCESS_GRANTED;i.n}\} \\
\text{+} \\
\{\text{i.login():ACCESS_DENIED}\} \\
\} \text{ for } \{\text{i.m, i.n}\}
\]

The provision in the example claims that if the method `login` returns the value `ACCESS_GRANTED`, then the method `n` must be called. Alternatively, when the method returns the `ACCESS_DENIED` value, other calls of neither `login` nor `n` are allowed. Here, the alternative helps to specify two traces - one for the case when the method `login` returned `ACCESS_GRANTED` and one for `ACCESS_DENIED`.

- **Repetition operator** — 

\[
\{ \text{i.m()+i.n()} \} \text{ for } \{\text{i.m, i.n}\}
\]
The arbitrary number of repetitions of the operand. In the example, the environment is expected to call the method $m$ an arbitrary number of times (including zero). Then it is expected to call the method $n$. The decision whether to perform the next iteration is again done either by the environment or by the imperative parts of the specification. The provision does not constrain it in any way.

```plaintext
{ i.login():ACCESS_DENIED*;  
i.login():ACCESS_GRANTED; i.n()} for {i.login, i.n}
```

The provision in the example claims that the environment is supposed to keep calling method $i\text{.login}$ until it returns $\text{ACCESS\_GRANTED}$. Then, it must call the method $n$.

- The (and-)parallel operator — $|$  

```plaintext
{ i.m|i.n() } for {i.m, i.n}
```

Parallel composition stands for alternative of all possible interleavings of operands. For instance, $(a;b)|(c;d)$ is equivalent to $(a;b;c;d)+(a;c;b;d)+(c;d;a;b)+(c;d;a;b)+(c;a;d;b)+(c;a;b;d)$.

The provision in the example states that the environment must call both methods $m$ and $n$. The order, however is not determined and one of methods may be even invoked during processing of the other.

- The or-parallel operator — $||$  

$A || B$ is equivalent to $A + B + (A \parallel B)$. Thus, alternatively, the environment may follow just one of the operands.

- The limited reentrancy operator — $|n$ for $n \in \mathbb{N}$  

$A |n$ is equivalent to $A \parallel A \parallel \ldots \parallel A$ where there are $n$ occurrences of $A$ within the expression. Thus, $A$ may be followed by the environment at most $n$ times in parallel.

- The (full) reentrancy operator — $A |*$  

$A |*$ stands for $A \parallel A \parallel A \parallel \ldots$. Thus, the environment may proceed according to $A$ as many times in parallel as it needs.

```plaintext
{ {i.login():ACCESS_DENIED*;  
i.login():ACCESS_GRANTED; i.n()}|*  
} for {i.login, i.n}
```
In the example, the environment may attempt to login in parallel. Each attempt, however, must end by a successful login and invocation of the method \( n \).
3.2 Semantics

The syntax presented so far accompanied by the information about interfaces from an underlying component model form a TBP specification. Its semantics, as depicted in Fig 3.1, is defined in two stages.

In model stage, the TBP model is defined. The TBP model is a 5-tuple capturing by mathematical means (e.g. Labeled Transition System) the essential information from the TBP specification. Composition is defined at the model stage, so that composition of two TBP models $(\oplus)$ is also a TBP model.

Notion of correctness (i.e. absence of communication errors) and refinement is defined at the LTS stage. The computation of the TBP model is represented by an LTS, either finite or infinite. Communication errors are defined for a closed model (i.e. model which does not exercise any externally observable activity - $\Sigma_{prov}$ and $\Sigma_{req}$ are empty) as a property over computation states.

For an open system, refinement is defined as a relation over observation projections. Observation projection is an LTS modeling only externally observable activity (e.g. observation projection of a closed system is a single state with no edges) of the computation. In the special case, when the number of external threads expected to use the provided methods of the component is limited to $k$, the LTS is finite. Finally, notion of refinement is defined as a relation over observation projections.

The motivation behind those stages is to bridge the gap between the TBP specification, which provides relatively rich concepts to the user, and mathematical structures used to clearly define notions of composition, communication error and refinement.

3.2.1 TBP Model

The TBP model precisely defines meaning of the syntax presented so far by mathematical means. As already indicated, provisions differ from the imperative parts of the specification (threads and reactions). Not surprisingly, in the TBP model, those are captured by different means as well. The provisions are represented by a set of important events and a set of allowed traces. To formally capture threads and reactions a variant of LTS enhanced with variables, guards, and assignments—Labeled Transition System with Assignments (LTSA)—is used.

In the following definitions, let $E$ be a set of enumeration types and $V$ be a set of variables of types from $E$. Each variable $v \in V$ determines its type
CHAPTER 3. THREADED BEHAVIOR PROTOCOLS

Figure 3.1: TBP semantic stages

and initial value. $Dom_e$ is a set of values of type $e \in E$, $Dom_v$ is a domain of the variable $v$, $Dom_E = \bigcup_{e \in E} Dom_e$ and $Dom_V = \bigcup_{v \in V} Dom_v$.

**Definition 1 (Parameterized labels)** Let $\Sigma$ be a set of labels and $Par$ is a set of parameters. Then, we define the set of parameterized labels $\Sigma_{Par} = \{(\alpha, \langle v_1, v_2, \ldots, v_n \rangle) : \alpha \in \Sigma, n \in N^0, v_i \in Par\}$.

The function $param_i : \Sigma_{Par} \rightarrow Par$ returns the $i$-th parameter of the parameterized label and the function $name : \Sigma_{Par} \rightarrow \Sigma$ returns the original label without parameter.

Labels are later used to model method calls. In particular, parameterized labels allow encoding method parameters as well as return values into labels. Thus, in the following definitions, the set of labels is often parameterized by variables and constants—$\Sigma_{V \cup \Dom E}$.

**Definition 2 (Valuation function)** The valuation function $\gamma_V : V \cup \Dom E \rightarrow \Dom_E$ assigns a value to each variable from $V$. Moreover, it is identity for constants ($\gamma_V(e) = e$ for $e \in \Dom_E$). The initial valuation function $\gamma_0$ assigns initial values to all variables. Modification of the valuation function
is denoted as $\gamma_V[v \mapsto e]$.

$$
\gamma_V[v \mapsto e](a) = \begin{cases} 
\gamma_V(a) & : a \neq v \\
e & : a = v \land v \in V \\
\text{undefined} & : a = v \land v \notin V
\end{cases}
$$

**Definition 3 (Guards)** A guard over $V$, is a finite expression derived using the following rules:

- $\text{true}$ is a guard,
- $v == l$, where $v \in \text{Var}$, $l \in \text{Dom}_v$ is a guard,
- if $X$ and $Y$ are guards, then $X \land Y$, $X \lor Y$ and $\neg X$ are also guards.

Actual value of the guard $g$ for valuation $\gamma_V$ is denoted as $\gamma_V(g)$ and the set of guards over $V$ is denoted as $G_V$.

Note that a mutex $m$ is considered to be a special case of a variable with domain $m = \{\text{LOCKED}, \text{UNLOCKED}\}$ and init $m = \text{UNLOCKED}$.

**Definition 4 (Assignment)** Assignment over $V$ is a label in the form

- $v = c$ where $v \in V, c \in \text{Dom}_v$
  
  assigning constant of corresponding type to $v$
- $v = w$ where $v, w \in V, \text{Dom}_v = \text{Dom}_w$

  assigning the actual value of another variable to $v$

Let $A_V$ be a set of assignments over $V$.

**Definition 5 (Labeled Transition System with Assignments)** A Labeled Transition System with Assignments (LTSA) is a tuple $(S, s_0, F, \delta, \Sigma, V)$, where $S$ is a finite set of states, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, $\Sigma$ a set of labels, $V$ a set of variables and $\delta \subseteq S \times G_V \times (\Sigma \cup A_V) \times S$ is a transition relation.

**Definition 6 (LTSA computation state)** Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA. We call the tuple $(s, \gamma_V)$ computation state of $l$ if $s \in S$ and $\gamma_V$ is valuation function for the set $V$. The tuple $(s_0, \gamma^0_V)$ is denoted as initial computation state.

**Definition 7 (Enabled LTSA transition)** Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA and $cs = (s, \gamma_V)$ is its computation state. We call the transition $t = (s, g, \alpha, s'), t \in \delta$ enabled in $cs$ if $\gamma_V(g)$ is true.
In the TBP model, LTSAs are used to capture control flow of imperative parts (e.g., reactions and threads). In this context, the set of labels contains parameterized labels representing issuing of a method call (for all required and internal methods).

Let \( \Sigma \) contain method names \( (m \in \Sigma) \), \( \Sigma^\uparrow \) contains events for issuing a method call \( (m^\uparrow \in \Sigma^\uparrow) \) and \( \Sigma^\downarrow \) contains events for accepting a method call result \( (m^\downarrow \in \Sigma^\downarrow) \). Moreover, \( \Sigma^{\uparrow/\downarrow} = \Sigma^\uparrow \cup \Sigma^\downarrow \). Then, \( \Sigma^\uparrow_{\text{V,Dom_E}} \) contains the events for issuing a method call parameterized by all possible combinations of variables from \( V \) and constants from \( \text{Dom_E} \).

For purposes of TBP model definition, we define \( \text{LTSA}^{\Sigma^\uparrow_{\text{V,Dom_E}}} \) to be a set of all LTSAs using labels from \( \Sigma^\uparrow_{\text{V,Dom_E}} \) and variables from \( V \). In such case, each transition in \( \delta \) is thus guarded by \( g \in G_V \) and labeled by \( l \in \Sigma^\uparrow_{\text{V,Dom_E}} \cup A_V \). Thus, \( l \) represents either issuing of a method call (including parameters), or an assignment involving a variable from \( V \). The return value of a method call is assigned to the special purpose \( \text{Ret} \) variable. Thus, to assign the return value to an user specified variable, it suffices to assign the content of \( \text{Ret} \) variable.

Informally, the TBP model definition says that a model is defined by an alphabet of methods names, a set of variables, a set of provisions (each captured as a set of allowable traces), a set of reactions (represented as LTSAs), and a set of active threads (given also as LTSAs).

**Definition 8 (TBP model)** Let \( E \) is a set of enumeration types used in a TBP specification. A TBP model is a five-tuple \( (\Sigma, P, R, T, G) \), where:

a) \( \Sigma = (\Sigma_{\text{prov}}, \Sigma_{\text{req}}, \Sigma_{\text{int}}) \) denotes disjunct sets of provided, required and internal method names used in the model.

b) \( G \) is a set of state variables.

c) \( P \) is a set of provisions \( \{P_1, P_2, \ldots, P_n\} \) taking the form \( P_i = (\text{filter}^R_i, \text{traces}^R_i) \), where \( \text{filter}^R_i \subseteq (\Sigma_{\text{prov}}) \) specifies methods observed by the provision and \( \text{traces}^R_i \) specifies a set of allowed finite sequences of events from \( (\text{filter}^R_i)^{\uparrow/\downarrow}_{\text{Dom_E}} \).

d) \( R \) is a partial function: \( (\Sigma_{\text{int}} \cup \Sigma_{\text{prov}}) \rightarrow (L, N \rightarrow L, \text{LTSA}^{\Sigma_{\text{int}}}_{\text{G,Dom,E}}) \) representing a mapping of method names to their local variables \( (L) \), a parameter mapping function and reactions in form of LTSA. The \( L \) set contains always at least the variable \( \text{Ret} \).

e) \( T \) is a set of threads \( T_1, T_2, \ldots, T_m \), where \( T_i \in (L, \text{LTSA}^{\Sigma_{\text{int}}}_{\text{G,Dom,E}}) \) is a tuple specifying a set of local variables and the behavior of the \( i \)-th thread in the form of LTSA.
3.2. SEMANTICS

In addition to a) where convenient, we use $\Sigma_{ext} = \Sigma_{prov} \cup \Sigma_{req}$ to denote the set of all externally visible method names, $\Sigma_{all} = \Sigma_{ext} \cup \Sigma_{int}$ for all names and $\Sigma_{imp} = \Sigma_{int} \cup \Sigma_{req}$ for names of methods which can be actively invoked in imperative parts.

Each provision $P_i$ in c) represents a set of finite sequences over events representing issuing of a method call and accepting the response. The methods are either provided or internal methods and the events are parameterized by constants from $Dom_E$. TBP model directly representing a TBP specification states the provisions just over the provided methods. However, after composition, some provided methods become internal methods. The parameters of response events represent return values. This allows reflecting return values in sequences and, in particular, describing an environment which behaves with respect to the value obtained from the method call. Notice, that events are not parameterized by variables as in other cases, but just by constants from $Dom_E$ (actual values of method calls).

The labels in the LTSA for reactions in d) contains calls of methods from $\Sigma_{imp}$ and assignments over local variables and state variables. In addition, constants from $Dom_E$ are allowed in method calls. The Ret variable is a special purpose variable used to store results from method calls. The parameter mapping function is used to assign parameters from a parameterized method call event $\alpha$ to variables from $L$.

### 3.2.1.1 From TBP Specification to TBP Model

Method sets ($\Sigma$), global memory ($G$), and filters ($filter^{P_i}$) directly correspond to the sets from TBP specifications. Provisions ($traces^{P_i}$) and construction of LTSA s ($R,T$) , however, deserve precise definitions. Formally, we define the function $model(TBP\ SPEC,k)$ taking as an argument a TBP specification and the number of threads from the environment allowed to enter the specification in parallel.

**Provisions** Every set $traces^{P_i}$ is represented by a finite state machine (FSM). Its construction directly follows the algorithm for construction of FSM from a regular expression. In particular, method calls are translated into a sequence of two events representing issuing a method call and return from a method call. Both events are parameterized - either by parameters or by return values. The parallel operator results in a interleaving of FSMs induced by its operands. Finally, the reentrancy operator is treated as a sequence of or-parallel operators. The number of parallel operators is given by the parameter $k$ of the $model$ function.
Imperative Parts  The structure of $\text{LTSA}_{\Sigma V,E}$ is constructed in a bottom-up fashion. The basic building blocks are method calls and variable assignments. The LTSA representing a method call contains three states sequentially connected by two transitions labeled by a method call and, optionally, a Ret value assignment. A state variable assignment is represented by an LTSA with two states connected by a single transition labeled by the assignment.

LTSA of a more complex expression is constructed from the LTSA of its subexpressions. It is also similar to construction of a nondeterministic finite automaton from a regular expression. The sequence operator (‘;’) corresponds to the concatenation of LTSA, if switch correspond to alternative, and the while statement is related to repetition. The difference inheres, however, in guards. If the if statement contains a deterministic condition (not ‘?’), the corresponding transitions are equipped with the guards derived from the condition and its negation. Similarly, the edges coming from the final states of the while statement may contain a guard.

Finally, a block synchronized by a mutex $\text{sync}(m)\{\ldots\}$ adds a new initial state to the LTSA connected by a transition to the original initial state. The new transition is labeled by the guard ensuring that the connected mutex is unlocked $m == \text{UNLOCKED}$ and by the assignment $m <- \text{LOCKED}$, which locks the mutex $m$. The resulting LTSA has only one (newly added) final state with a transition targeting it from each original final state and labeled by the assignment $m <- \text{UNLOCKED}$.

3.2.1.2 Composition

Before defining the composition itself, we first make a simple observation. The names from the sets $\Sigma_{\text{int}}$ and $G$ are not visible to the outer world and thus should not influence the result of the composition. In other words, a protocol defines the same behavior under any arbitrary renaming of $\Sigma_{\text{int}}$ and $M$. Therefore, without loss of generality, we assume that there are no name clashes in these internal names$^1$.

Moreover, when two models are being composed, they should not provide a method with the same name as this would yield a binding of a single required interface to multiple provided interfaces, which is not supported (semantics of such a broadcast method call is unclear). Thus, to capture these requirements, we define notion of composable models.

Definition 9 (Composable models) Let $A = (\Sigma', P', R', T', G')$ and $B = (\Sigma'', P'', R'', T'', G'')$ be TBP models. We say, that $A$ and $B$ are composable

$^1$Formally, this could be also handled by introducing a name substitution. However, this would obfuscate the otherwise simple definition.
Composition of two composable TBP models is again a TBP model. The composition unifies the corresponding sets of provisions, reactions, threads and state variables.

**Definition 10 (TBP Composition)** Let \( A = (\Sigma', P', R', T', G') \) and \( B = (\Sigma'', P'', R'', T'', G'') \) be TBP composable models. Then:

\[
A \oplus B = ((\Sigma_{prov} \cup \Sigma''_{prov}), P' \cup P'', R' \cup R'', T' \cup T'', M' \cup M'')
\]

where

\[
\Sigma_{prov} = \Sigma'_{prov} \cup \Sigma''_{prov}, \\
\Sigma_{req} = (\Sigma'_{req} \cup \Sigma''_{req}) \setminus (\Sigma'_{prov} \cup \Sigma''_{prov}) \quad \text{and} \\
\Sigma_{int} = \Sigma'_{int} \cup \Sigma''_{int}
\]

Notice, that the set of methods provided by composition is union of methods provided by original models. This way, a method provided by the input model which is at the same time required by the other input model stays in the set of provided methods in the composition. Thus a provided method can be required (thus, invoked) by several components composed together by sequential application of the composition operator.

**Definition 11 (Closed TBP model)** Let \( M = (\Sigma, P, R, T, G) \) be a TBP model. We call \( M \) closed if \( \Sigma_{req} = \emptyset \)

The essence of a closed model is that it does not communicate with the environment. In particular, there are no methods required from the environment. The closed computation introduced in the following text considers only the threads from \( T \) as source of activity and does not expect environment to invoke any method.

### 3.2.2 Analysis of Closed Models

In this section, we define the computation of a closed TBP model as a finite LTS — *closed computation*. Intuitively, the closed computation is created by composition of LTSAs of individual threads and reactions. In particular, the way individual LTSAs are put together is inspired by a typical stack-based execution model of imperative languages.
The number of threads is fixed in the closed TBP model. There is a single stack for each thread. The top of a stack refers to the actual position in the LTSA being currently executed by the thread. Also the local variables and parameters referenced by LTSA guards and assignments are on the stack. Thus, each computation state is represented by a number of stacks and valuation of state variables.

**Definition 12 (Computation state)** A computation state of the TBP model \((\Sigma, P, R, T, G)\) is a tuple \((\text{Stacks}, \gamma_G)\), where \text{Stacks} is a set of stacks—sequences of tuples \((s, \gamma_L)\). The size of Stacks correspond to the number of threads \(|\text{Stacks}| = |T|\), \gamma_G and \gamma_L are valuation functions and \(s\) is a state of the LTSA \(l\). The \(l\) either captures behavior of a reaction or a thread from the model.

The computation transition represents atomic change of the computation state. In particular, the change is either a modification of a stack or modification of a state variable. The change of the stack size corresponds either to issuing a method call or accepting a method call response. Those transitions are labeled by corresponding parameterized labels. The data in those labels are the actual values used in the particular method calls and returns. Thus, if the method names are from \(\Sigma\), then labels are from \(\Sigma^\uparrow_{\text{Dom}_E}\). Since the labels in LTSA may be also parameterized by variables, parameter valuation function is defined to get the actual values of these variables.

**Definition 13 (Parameter valuation function)** Let \(\gamma_V : V \cup \text{Dom}_E \rightarrow \text{Dom}_E\) be a valuation function. Then, we define the parameter valuation function \(\gamma^V : \Sigma V \cup \text{Dom}_E \rightarrow \Sigma_{\text{Dom}_E}\) in following way:

\[
\gamma^V((\alpha, <p_1, p_2, \ldots, p_n>)) = (\alpha, <p'_1, p'_2, \ldots, p'_n>),
\]

where \(p'_i = \gamma_V(p_i)\).

In the following definition, a stack is treated as the pair \((\text{top}, \text{tail})\) or \text{null}, where \text{top} is the item on the top of the stack, \text{tail} is the rest of the sequence and \text{null} represents the empty stack.

**Definition 14 (Computation transition)** Let \(CS = (\text{Stacks}, \gamma_G)\) be a computation state of the TBP model \((\Sigma, P, R, T, G)\), \(St \in \text{Stacks}\) is a stack, \(St = ((s, \gamma_L), \text{tail}_{St})\). Let \(l = (S, s_0, F, \delta, \Sigma^\uparrow_{V \cup \text{Dom}_E}, V)\) be the LTSA corresponding to the top of \(St\) \((s \in S, V = G \cup L)\). Let \(t = (s, g, \alpha, s_o)\), \(t \in \delta\) is a transition enabled in the computation state of \(l = (s, \gamma_LG)\). Then, we call the triple \((CS, \alpha', CS')\) computation transition, where \(CS' = (\text{Stacks}\setminus\{St\} \cup \{St'\}, \gamma_G')\) and \(\alpha', St' \) and \(\gamma'_G\) are defined depending on the \(\alpha\) and \(s\) in the following way:
3.2. SEMANTICS

a) $\alpha$ is a required method call action — $\alpha \in (\Sigma_{req})^\uparrow_{\Sigma_{req} \cup \text{Dom}_E}$
$$\alpha' = \gamma^\uparrow_{\Sigma_{req} \cup \text{Dom}_E}(\alpha)$$
$$\text{St}' = ((s_0, \gamma_L), \text{tail}_\text{St})$$
$$\gamma'_G = \gamma_G$$

$\alpha'$ is created by assigning actual values to variable parameters of $\alpha$ and the top of the stack is altered so that it contains target state of $t$.

b) $\alpha$ is an assignment $v = e$ — $\alpha \in A_V$
$$\alpha' = \tau$$
$$\text{St}' = ((s_0, \gamma_L), \text{tail}_\text{St})$$, where $\gamma'_L = \gamma_L[v \mapsto \gamma_L \cup G(e)]$,
$$\gamma'_G = \gamma_G[v \mapsto \gamma_L \cup G(e)]$$

Since the assignment to the local variable is not observable communication, special event for internal action $\tau$ is assigned to $\alpha'$. The top of the stack is altered so that it contains target state of $t$. Moreover, either the valuation $v_L$ at the top of the stack or $v_G$ is changed — depending on whether $v$ is local or state— $v \in L$ resp. $v \in G$.

2

"\c) $\alpha \in (\Sigma_{prov} \cup \Sigma_{int})^\uparrow_{\Sigma_{prov} \cup \text{Dom}_E}$ — $\alpha$ is a provided or internal method call event.

Let $R(\text{name}(\alpha)) = (L', p, l')$, where $L'$ a set of local variables for the method reaction, $p : N \to L$ is a parameter mapping function and $l' = (S', s_0', F', \delta', \Sigma^\uparrow_{\Sigma_{prov} \cup \text{Dom}_E} \cup G \cup L'), l' \in \text{LTSA}_{\Sigma_{prov} \cup \text{Dom}_E}$ is the LTSA capturing the control flow of the method reaction.

$$\alpha' = \gamma^\uparrow_{\Sigma_{prov} \cup \text{Dom}_E}(\alpha)$$
$$\text{St}' = ((s'_0, \gamma'_L), ((s_0, \gamma_L), \text{tail}_\text{St})),$$
where $\gamma'_L = \gamma^0_L[p(n) \mapsto \text{param}_n(\alpha')$ for $n \in \text{Dom}(p)],$

$$\gamma'_G = \gamma_G$$

Also here, the top of the stack is altered so that it contains target state of $t$. Moreover, the new computation state of $l'$ $(s'_0, \gamma'_L)$ is pushed. It represents that the thread moved to the $l'$—LTSA determined by reaction mapping function $R$ and name of the method from $\alpha$.

The valuation of local variables in $l'$ stems from the initial values of $L'$ — $\gamma^0_L$, however actual values of parameters used in $\alpha$ are copied into variables determined by the parameter mapping function $p$. In particular, if the $p$ is defined for $i \in N$, $p(i) = v, v \in L$, then $v$ is assigned the actual value of $i$-th parameter from $\alpha$.

Notice that $a \notin A \Rightarrow \gamma_A[a \mapsto \text{const}] = \gamma_A$.\"
d) \( s \) is a final state of \( l \) and \( \text{tail}_{St} = ((s', \gamma'_L), \text{tail}'_{St}) \)
Let \( m \) is the name of the method from \( \Sigma_{\text{int}} \) such that \( R(m) = (L, p, l) \).
Then,
\[
\alpha' = (m^\downarrow, (\gamma_L(Ret))) \quad \text{St}' = ((s', \gamma'_L[Ret \mapsto \gamma_L(Ret)]), \text{tail}'_{St}),
\]
\( \gamma'_G = \gamma_G \)

\( \alpha \) is an event representing return from the method \( m \) whose control flow is captured by \( l \). Thus, it consist from the name of the method name \( m \), response flag \( \downarrow \) and a single parameter keeping the value of the \( \text{Ret} \) variable. Such method \( m \) must exist, since the \( St \) has at least two items and LTSAs of threads are always at the deepest level of a stack.

The \((s, \gamma_L)\) item is popped from the stack and the current valuation on the top of the stack \((s', \gamma'_L)\) is altered so that \( \gamma'_L(Ret) = \gamma_L(Ret) \).

Put together, computation states and transitions form a closed computation:

**Definition 15 (Closed computation)** The closed computation of a closed TBP model \( M = (\Sigma, P, R, T, G) \) is the tuple \( C(M) = ((\Sigma_{\text{imp}})_{\text{Dom}_E}^\uparrow/\downarrow \cup \{\tau\}, S, s_0, \delta, F) \), where:

- Label from \((\Sigma_{\text{imp}})_{\text{Dom}_E}^\uparrow/\downarrow \cup \{\tau\}\) is either an event representing issuing of method calls or acceptance of method calls parameterized by constants from \( \text{Dom}_E \).
- \( s_0 = (\text{Stacks}_{\text{init}}, v^0_G) \) is initial computation state of the model. The set \( \text{Stacks}_{\text{init}} \) contains a stack for each thread \( t = (L, \text{LTSA}_t) \) from \( T \) containing single item \((s^0_t, \gamma^0_L)\), where \( s^0_t \) is the initial state of \( \text{LTSA}_t \) and \( \gamma^0_L \) is initial valuation of \( L \).
- \( \delta \subseteq S \times ((\Sigma_{\text{imp}})_{\text{Dom}_E}^\uparrow/\downarrow \cup \{\tau\} \times S \) is a transition relation corresponding to the computation transition
- \( S \) is a set of computation states of reachable by \( \delta \) from \( s_0 \)
- \( F \subseteq S \) is a set of final states. The state \( s \) is final if for each thread \( t \) its stack contains just single item \((s, L)\) such that \( s \) is a final state of \( \text{LTSA}_t \).

The closed computation of a closed TBP model is finite as long as the model does not contain (even indirect) recursive calls in reactions. Moreover, since there are no required methods, there are no transitions of type a) (Definition 14)
The definitions presented so far provide us with the precise meaning of a set of TBP specifications composed together such that they form a closed TBP model. The semantic in this case is given in the form of finite LTS, which provide straightforward definition of communication errors in the following sections.

3.2.2.1 Communication Error

The formalism of TBP distinguishes two kinds of communication errors. There are inherent errors, which are apparent from the closed computation, and errors with respect to provisions.

**Definition 16** Let $C(M) = ((\Sigma_{imp})^{↑/↓} \cup \{\tau\}, S, s_0, \delta, F)$ is a closed computation of a closed TBP model $M$.

There is inherent no activity in a state $s \in S$ if $s \notin F$ and there are no output transitions from the state $s$. The set of all states with inherent no activity is denoted as $F^{\ominus}$.

There is infinite activity in state $s \in S$ if $s \notin F$, there is no inherent no activity in state $s$ and there is no path leading to a final state. The set of all states with infinite activity is denoted as $F^{\infty}$.

There is internal infinite activity error in state $s \in S$ if there is infinite activity in $s$ and all paths leaving the state $s$ contain only $\tau$ events. The set of all states internal infinite activity error is denoted as $F^{\infty}_{int}$. Apparently, $F^{\infty}_{int} \subseteq F^{\infty}$.

No activity denotes the situation when a thread stuck in a state waiting for a guard which never starts to hold (e.g. to enter a critical section created by `sync` keyword or waiting for a certain value in a variable). In such case, the thread is waiting for an action to be performed by another thread (leave the critical section, set the variable), which unfortunately does not occur. Since such a situation is clearly undesirable, such state is considered to be erroneous.

On the other hand, the infinite activity is desirable in some cases—especially those emphasizing reactive nature of a system while not modeling the shutdown at all. The special case of infinite activity—internal infinite activity, however, is undesirable, since there the computation performs only internal actions with no effect observable by other components. For instance, internal infinite activity captures the situation, when a thread is actively waiting (in a loop) for a guard.

All in all, we consider as erroneous the states from $F^{\ominus}$ and $F^{\infty}_{int}$ while $F^{\infty}$ can be desirable in some models. Moreover, there is other class of errors induced by the provisions.
Provisions of individual components express assumptions posed on the environment in the form of traces. As the models are being composed together, the particular environment of the component is materializing. Once the system is closed, it is checked whether all provisions are obeyed.

Since provisions are based on traces, we define traces generated by closed TBP model first.

**Definition 17 (Computation trace)** Let \( C = ((\Sigma_{\text{imp}})^{\uparrow/\downarrow}_{\text{Dom}_E} \cup \{\tau\}, S, s_0, \delta, F) \) is a closed computation of a closed TBP model. Then, we call the finite sequence \( \alpha_0, \alpha_1, \ldots, \alpha_n \), \( \alpha_i \in (\Sigma_{\text{imp}})^{\uparrow/\downarrow}_{\text{Dom}_E} \cup \{\tau\} \) computation trace, if there is a sequence of computation states \( s_0, s_1, \ldots, s_{n+1} \) such that \( \forall 0 \leq i < n : s_{i+1} \in S \land (s_i, \alpha_i, s_{i+1}) \in \delta \).

Moreover, we call the computation trace terminating when \( s_{n+1} \in F \), stuck when \( s_{n+1} \in F^\circ \), diverging when \( s_{n+1} \in F^\infty \) and internally diverging if \( s_{n+1} \in F^\infty_{\text{int}} \).

The set of terminating computation traces is denoted as \( L(C)^\vee \) and stuck computation traces as \( L(C)^\circ \). \( L(C)^\infty \) contains the set of representatives of diverging computation traces—all diverging traces sharing the same (already diverging) prefix are represented just by the prefix. The set of representatives of internally diverging computation traces is denoted as \( L(C)^\infty_{\text{int}} \). \( L(C) = L(C)^\vee \cup L(C)^\circ \cup L(C)^\infty \).

The set \( L(C) \) characterizes the closed computation. It contains all traces leading to successful termination as well as traces leading to no activity and traces leading to diverging states. Having such characterizing set of computation traces, we can define the provision violation.

**Definition 18 (Trace restriction)** Let \( t = \alpha_0, \ldots, \alpha_n \) be a computation trace consisting of parametrized labels \( \alpha_i \in (\Sigma_{\text{imp}})^{\uparrow/\downarrow}_{\text{Dom}_E} \cup \{\tau\} \) and \( f \subseteq \Sigma_{\text{imp}} \) be a set of labels referred as filter. Then, the trace restricted by \( f \), \( t \upharpoonright f = \alpha'_0, \ldots, \alpha'_m \) is a computation trace consisting of \( \alpha'_i \in f^{\uparrow/\downarrow}_{\text{Dom}_E} \) such that

\[
 t \upharpoonright f = \begin{cases} 
 \alpha_0.(t_1 \upharpoonright f) : \text{name}(\alpha_0) \in f \\
 t_1 \upharpoonright f : \text{otherwise}
\end{cases}
\]

where \( t_1 = \alpha_1, \ldots, \alpha_n \) is the trace \( t \) without the first parametrized label and \( \cdot \) is the concatenation operator. Moreover, restriction of an empty trace is empty trace.

Informally, the trace restriction operator removes from the trace the parametrized labels that are not based on a label belonging to the filter.
3.2. SEMANTICS

Definition 19 (Provision violation) Let \((\Sigma, P, R, T, M)\) be a closed TBP model and \(C\) its closed computation. We say, that the provision \(P_i \in P\), \(P_i = (\text{filter}^{P_i}, \text{traces}^{P_i})\) is adhered by a trace \(t \in L(C)\) iff \(t \upharpoonright \text{filter}^{P_i} \in \text{traces}^{P_i}\). The provision is adhered in the TBP model if it is adhered by all computation traces from \(L(C)\). If the provision is not adhered in the model, it is violated.

With violation of provision defined in general, let us discuss specific kinds of violations. Violation of a provision occurs, if there is a trace in \(L(C)\) such that its restriction \(t \notin \text{traces}^{P_i}\). In such case, let \(t' \in \text{traces}^{P_i}\) is a trace, sharing the longest prefix \(\alpha\) with \(t\).

Fig. 3.2 illustrates the case when \(t = \alpha\), which is denoted as no activity. In such case, the restricted trace \(t\) follows the provision up to the certain point \((\alpha)\) and then immediately terminates without following the rest of the trace \(t'\). It corresponds to the situation when the model is expected to perform some action (e.g. close a session), however the model does not perform any action and just terminates.

Definition 20 (No activity) A TBP model with a closed computation \(C\) generating computation traces \(L(C)\) and containing a provision \((\text{filter}, \text{traces})\) contains no activity error if there is \(t \in L(C)\) \(\upharpoonright \text{filter}, t \notin \text{traces}\) such that it is a prefix of a trace \(t' \in \text{traces}\).

The Fig. 3.3 reflects the remaining two situations. Either \(t' = \alpha\) or \(\alpha\) is shorter than both \(t\) and \(t'\). In the former case the trace \(t\) is a witness of a situation when a source model attempts to invoke a provided method of a target model, which is already in a final state and does not expect any further method calls. The latter case, on the other hand, reflects the situation.
when the target component does not expect the invoked method, but expects another method call.

**Definition 21 (Bad activity)** The TBP model with closed computation \( C \) generating computation traces \( L(C) \) and containing provision \((\text{filter}, \text{traces})\) contains bad activity error if there is \( t \in L(C) \upharpoonright \text{filter}, t \notin \text{traces} \) such that it is not a prefix of any trace \( t' \in \text{traces} \).

For the purposes of the following text, we define the predicate \( \text{ErrFree} \) over the set of all closed TBP models.

**Definition 22 (ErrFree)** Let \( t \) is a closed TBP model. Then we define

- \( \text{ErrFree}_{\text{BA}}(t) \iff \) there is no bad activity in the model.
- \( \text{ErrFree}_{\text{⊘}}(t) \iff \) there is neither inherent no activity nor no activity in the model.
- \( \text{ErrFree}_{\text{∞ int}}(t) \iff \) there is no internal infinit activity in the model.
- \( \text{Errfree}(t) \iff \text{ErrFree}_{\text{BA}}(t) \land \text{ErrFree}_{\text{⊘}}(t) \land \text{ErrFree}_{\text{∞ int}}(t) \)

### 3.2.3 Analysis of Open Models

So far, notion of correctness for a closed TBP model was presented. In the context of hierarchical component models, however, developers often deal with open systems. Thus, it is necessary to extend the notion to the realm of open TBP models.

Even if there is an error (in the sense of previous paragraphs) in the open system, it often depends on the particular environment whether the error becomes evident or not. The environment may steer the open system away from the error so that the error is never reached in the closed system.

Thus, instead of mere identification of errors in an open system, comparison of open systems with respect to an environment is preferred. In particular, the question asked is whether a TBP model \( I \) behaves correctly in all environments where a TBP model \( S \) behaves correctly. Such relation is referred to as refinement in further text.

In the typical scenario, \( I \) represents a complex model (I stands for the implementation, which is typically a composition of other models—there are all details on the internal communication) while \( S \) is a relatively simple model describing important aspects of behavior directly communicating with the environment.
Having the refinement relation, the hierarchical system verification is divided into two subtasks. First task is done when an open system is being created. The developer of a composite component provides the model $S$ and ensures that the actual component behavior corresponds to $S$—by means of refinement. The second task is done when the open system is put into a particular environment to create either a closed system or an open system on the higher level. In former case, the developer uses the specification $S$ to verify correctness of the closed model by means of ErrFree predicate. In the latter case, the specification $S$ is put together with other components to form the implementation of the composed component. Then, the refinement is checked again on the higher level.

Formally, the refinement is defined as follows:

**Definition 23 (Refinement)** Let $I$ and $S$ are TBP models and $E$ be a set of TBP models such that $\forall e \in E \ e \oplus I$ be a closed TBP model.

- We say that $I$ refines $S$ with respect to bad activity iff $\forall e \in E : \text{ErrFree}_{BA}(e \oplus S) \Rightarrow \text{ErrFree}_{BA}(e \oplus I)$

- We say that $I$ refines $S$ with respect to no activity iff $\forall e \in E : \text{ErrFree}_{\emptyset}(e \oplus S) \Rightarrow \text{ErrFree}_{\emptyset}(e \oplus I)$

- Finally, we say that $I$ refines $S$ iff $\forall e \in E : \text{ErrFree}(e \oplus S) \Rightarrow \text{ErrFree}(e \oplus I)$

Definition of the refinement is based on the errors considered. Moreover, the implication in definitions ensures transitivity of all three refinement relations as stated in the following lemma.

**Lemma 1 (Transitivity)** Let $A$, $B$ and $C$ are TBP models such that $A$ refines $B$ and $B$ refines $C$. Then, $A$ refines $C$.

The rest of this section provides means for deciding whether $I$ refines $S$. This is done in several steps depicted in Fig 3.4.

First, an open TBP model is transformed into provision-driven computation. It is an LTS similar to the closed computation, however it also contains the transitions labeled by input actions representing the actions the environment is allowed (or expected) to perform. Those input actions are distinguished from the actions actively performed by the model (output actions).

In the next step, observation projection is created from the provision-driven computation. Purpose of the observation projection is to resolve the non-determinism in the model. It is a pessimistic approximation of the
provision-driven computation representing the behavior of the model as observed by an environment. In particular, the environment is not able to distinguish states of the model reached by the same sequence of observable actions. All these states are represented by a single state (super-state) in the observation projection. Moreover, the super state allows the environment to perform only the actions allowed by all states of provision-driven computation it represents. On the other hand, the observation projection requires the environment to be ready for all actions that may occur in any state the super state represents.

Final step when deciding whether $I$ refines $S$ is the parameterized alternation simulation. Basically, the alternation simulation [30] identifies pairs of states which must fulfill a property in order to ensure refinement of specifications. The property $P$ parametrizing the particular variant of refinement is designed to preserve the corresponding error.

Currently, the refinement requires that the number of threads originated in the environment is limited. The limit $k$ goes through all the following definitions and theorems. In this sense, we define a weaker notion of refinement as follows:

**Definition 24 (Refinement up to $k$ threads)** Let $I$ and $S$ are TBP models and $E$ is a set of TBP models such that $\forall e \in E e \oplus I$ is a closed TBP model and $e$ does not invoke more than $k$ provided methods of $I$ in parallel.
3.2. SEMANTICS

- We say that $I$ refines $S$ with respect to bad activity up to $k$ threads iff
  \[ \forall e \in E : \text{ErrFree}_{\text{BA}}(e \oplus S) \Rightarrow \text{ErrFree}_{\text{BA}}(e \oplus I) \]

- We say that $I$ refines $S$ with respect to no activity up to $k$ threads iff
  \[ \forall e \in E : \text{ErrFree}_{\text{c}}(e \oplus S) \Rightarrow \text{ErrFree}_{\text{c}}(e \oplus I) \]

- Finally, we say that $I$ refines $S$ up to $k$ threads iff
  \[ \forall e \in E : \text{ErrFree}(e \oplus S) \Rightarrow \text{ErrFree}(e \oplus I) \]

3.2.3.1 Provision-driven Computation

The provision-driven computation is a counterpart of closed computation for open specifications. In contrast to the closed specification, the open specification, besides its internal threads allows threads from the environment to invoke provided methods. The way the external threads invoke the provided methods is, however, limited by provisions. Thus, in the provision-driven computation (which is an LTS) occurs also externally triggered activity (i.e. transitions representing invocation of a provided method by an external thread).

Formally, provision-driven computation is a tuple $C_{PD} = (\Sigma_{\text{all}}^{\uparrow/\downarrow}_{\text{Dom}_E} \cup \{\tau\}, S, s_0, \delta, F)$. Notice, that it differs from the computation signature by the set of labels appearing on transitions — there can be also input actions representing provided methods. In the further text, we use $?m$ to denote $m \in \Sigma_{\text{prov}}$, $!m$ to denote $m \in \Sigma_{\text{req}}$ and finally $\tau m$ to denote $m \in \Sigma_{\text{int}}$.

In contrast to the closed computation, the number of stacks in the individual states changes to reflect the external threads allowed by provisions to call the provided methods. Moreover, individual states also contain the information about the actual state of provisions.

The first step to create a provision-driven computation is to combine individual provisions of the specification to form single LTS. This is done in two phases. First, each provision is modified to guard all methods from $\Sigma_{\text{prov}}$. This is done by adding new transitions to each state which. These new transitions allow accepting all methods that are not constrained by the provision in parallel. Then, an LTS that accepts intersection of traces accepted by all automaton is constructed.

**Definition 25 (Combined provisions for $k$ threads)** Let $P = \{P_1, P_2, \ldots, P_n\}$ be a set of provisions of the TBP model $M$, where $P_i = (\text{filter}^{R_i}, \text{traces}^{P_i})$ and $k$ is the maximum number of threads allowed in the environment. Moreover, $P_i$ does not contain the reentrancy operator $\ast$ and the maximum reentrancy allowed in the limited reentrancy operator is $k$. 
CHAPTER 3. THREADED BEHAVIOR PROTOCOLS

Let FSM$_{P_i}$ be a deterministic finite state machine accepting the sequences from traces$^{P_i}$ and $\Sigma_{prov}$ is the set of all methods provided by $M$. Then, let FSM$_{P_i}$ be an automaton created by parallel composition with automaton induced by the provision $(m_1+m_2+\ldots+m_n)|k$ where $m_1, m_2, \ldots, m_n \notin \text{filter}^{P_i}$. Then, composed provisions of $M$ for $k$ threads, Prov$_M^k$, is FSM accepting the intersection of languages accepted by all FSM$_{P_i}$.

As long as the environment features less then $k$ threads, the composed provisions allow the environment to perform exactly the same behavior as was allowed by the original set of provisions. The fact is expressed by the following lemma:

**Lemma 2** Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model and $N = (\Sigma, \{\Sigma_{prov}, \text{Prov}_M^k\}, R, T, G)$ is the same model where the set of provisions $P$ was replaced by composed provisions of $M$ for $k$ threads where $k$ is the number of threads in $M$ ($|T|$). Then, ErrFree($M$) $\Leftrightarrow$ ErrFree($N$).

Once a composed provisions representing the behavior the model $M$ expects from the environment is available, provision-driven computation can be constructed. The following definitions are based on definitions of closed computation. For instance, the provision-driven computation state is a computation state enriched by a position in the combined provisions.

**Definition 26 (Provision-driven state for $k$ threads)** Let $M = (\Sigma, P, R, T, G)$ is a TBP model and $\text{Prov}_M^k = (S, s_0, \delta, F)$ its composed provisions for $k$ threads. A state of Provision-driven computation of $M$ for $k$ threads is a tuple $(\text{Stacks}, \gamma_G, s)$, such that the tuple $(\text{Stacks}, \gamma_G)$ forms an computation state and $s$ identifies a state from $\text{Prov}_M^k$ $(s \in S)$.

In addition to the transition of a closed computation, the provision-driven computation transition is changing the state of combined provisions as a method is invoked. Moreover, there are transitions representing invocation of provided methods by threads from the environment.

**Definition 27 (Provision-driven transition for $k$ threads)** Let $CS = (\text{Stacks}, \gamma_G, s_{prov})$ be a provision-driven computation state of the TBP model $M = (\Sigma, P, R, T, G)$, $St \in \text{Stacks}$ be a stack, $St = ((s, \gamma_L), \text{tail}_{St})$ and $s_{prov}$ be a state of $\text{Prov}_M^k = (S_{prov}, s_0_{prov}, \delta_{prov}, F_{prov})$. Let $l = (S, s_0, F, \delta, \Sigma^\uparrow_{V \cup \text{Dom}_P}, V)$ be the LTSA corresponding to the top of $St$ $(s \in S, V = G \cup L)$. Let $t = (s, g, \alpha, s_0), t \in \delta$ is an enabled transition in the computation state of $l$—$(s, \gamma_L, \gamma_G)$ and $t_{prov}$ is a transition leaving $s_{prov}, (s_{prov}, \alpha_{prov}, u_{prov}) \in \delta_{prov}$. Then, we call the triple $(CS, \alpha', CS')$ provision driven computation transition, where $CS' = (\text{Stacks}\backslash\{St\} \cup \{St'\}, \gamma_G', s'_{prov})$ and $\alpha'$, $St'$ and $\gamma_G'$ are defined depending on the $\alpha, \alpha_{prov}, s$ and $s_{prov}$ in following way:
3.2. SEMANTICS

a) \( \alpha \) is a required method call event or an assignment

\( s'_\text{Prov} = s_{\text{Prov}} \) state of provision remains unchanged since required method calls are not guarded by provisions. \( \alpha' \), \( St' \) and \( \gamma'_G \) are defined as for computation transition.

b) \( \alpha \in (\Sigma_{\text{int}} \cup \Sigma_{\text{prov}})^+_{\cup \text{Dom}_E} \) - \( \alpha \) is a provided or internal method call event.

If there is a state \( u_{\text{Prov}} \in S_{\text{Prov}} \) such that \( (s_{\text{Prov}}, \alpha, u_{\text{Prov}}) \in \delta_{\text{Prov}} \), then \( \alpha' = \tau \) and \( St' \) and \( \gamma'_G \) are defined as for computation transition and \( s'_\text{Prov} = u_{\text{Prov}} \).

Otherwise, CS is a state causing the bad activity error and the transition is undefined.

c) \( s \) is a final state of \( \ell \) and \( \text{tail}_{St} = ((s', \gamma_L'), \text{tail}_{St}') \)

If there is a state \( u_{\text{Prov}} \in S_{\text{Prov}} \) such that \( (s_{\text{Prov}}, \alpha', u_{\text{Prov}}) \in \delta_{\text{Prov}} \), then \( \alpha' = \tau \) and \( St' \) and \( \gamma'_G \) are defined as for computation transition. Moreover \( s'_{\text{Prov}} = u_{\text{Prov}} \).

If there is no such transition in \( \delta_{\text{Prov}} \), then CS is a state causing the bad activity error and the transition is undefined.

On the other hand, if \( \text{tail}_{St} \) is empty, it is not returned to Stacks in \( CS' \) and \( \alpha' = (m^l, (\gamma_L(\text{Ret}))) \) as for computation transition.

d) \( \alpha_{\text{Prov}} \in (\Sigma_{\text{prov}})^+_{\cup \text{Dom}_E} \) - Provision allows to call a provided method

No stack is modified. Instead, new stack is added to Stacks in \( CS' \) reflecting the newly started processing of the provided method.

Let \( R(\text{name}(\alpha)) = (L', p, l') \), where \( L' \) a set of local variables for the method reaction, \( p : N \rightarrow L \) is a parameter mapping function and \( l' = (S', s'_0, F', \delta', \Sigma'_{\text{Dom}_E} \cup \Sigma_{\text{imp}}', G \cup L') \) \( l' \in \text{LTSA}_{G,L',E}^{\Sigma_{\text{imp}}'} \) is the LTSA capturing the control flow of the method reaction.

\( \alpha' = \alpha_{\text{Prov}} \)

\( St' = ((s'_0, \gamma_L'), \bot), \)

where \( \gamma_L = \gamma^{0}_{L'}[p(n)\mapsto \text{param}_n(\alpha') \text{ for } n \in \text{Dom}(p)] \)

\( CS' = (\text{Stacks} \cup \{St'\}, \gamma_G) \)

It is worth to notice that while the transitions defined in a) b) and c) represent an activity performed actively by the model (!m), the transitions defined in d) state that the model is willing to perform the activity if asked by the environment (?m).

**Definition 28 (Provision-driven computation for k threads)** The provision-driven computation of a TBP model \( M = (\Sigma, P, R, T, G) \) for \( k \) threads is the tuple \( C^k_{PD}(M) = ((\Sigma_{\text{all}})^+_{\cup \text{Dom}_E} \cup \{\tau\}, S, s_0, \delta, !F, !F) \), where:
• Label from $\{\Sigma_{all}\}^{\uparrow/\downarrow}_{\text{Dom}_E}$ is either an event representing issuing of method calls or acceptance of method calls parameterized by constants from $\text{Dom}_E$.

• $s_0 = (\text{Stacks}_\text{init}, \gamma_0^G, s^0_{\text{Prov}})$ is initial provision-driven state. The set $\text{Stacks}_\text{init}$ contains a stack for each thread $t = (L, \text{LTSA}_t)$ from $T$ containing single item $(s^0_t, \gamma^0_L)$, where $s^0_t$ is the initial state of $\text{LTSA}_t$ and $\gamma^0_L$ is initial valuation of $L$. $s^0_{\text{Prov}}$ is an initial state of $\text{Prov}^k_M$.

• $\delta \subseteq S \times (\Sigma_{\text{imp}})^{\uparrow/\downarrow}_{\text{Dom}_E} \cup \{\tau\} \times S$ is a transition relation corresponding to the provision-driven transition for $k$ threads.

• $S$ is a set of provision-driven computation states reachable by $\delta$ from $s_0$.

• $!F \subseteq S$ is a set of imperative final states. The state $s = (\text{Stacks}, \gamma_G, s^{\text{Prov}})$ is a final imperative state if for each model’s thread $t$ there is a stack $\in \text{Stacks}$ containing just single item $(s^t, L)$ such that $s^t$ is a final state of $\text{LTSA}_t$. There are no other stacks in $\text{Stacks}$ representing the threads originated in the environment.

• $F \subseteq S$ is a set of final states. The state $s = (\text{Stacks}, \gamma_G, s^{\text{Prov}})$ is a final state if it is imperative final state and $s^{\text{Prov}}$ is a final state of $\text{Prov}^k_M$.

In addition to the final states $F$ representing the situation when both internal threads and provisions are in a final state, we also define the set $!F \subseteq S$ to denote the states where all threads are in a final state, but there is no restriction on the provision state ($!F \subseteq F$). Those states reflect the situation when the model is neither obliged to perform any action on its own nor to terminate.

Similarly to the closed computation, using the definition of provision-driven final states, we define the sets $F^\circ, F^\infty$ and $F^\infty_{\text{int}}$. Moreover, we define $F^{BA}$ to be a set of states causing the bad activity error (Definition 27 c).

In contrast to the closed computation, mere existence of an error state (e.g. $s \in F^\circ$) does not automatically mean that the model is useless. It can still work perfectly in a number of environments which use it in a way such that the error is avoided.

The provision-driven computation is well defined even for closed systems. Such computation features no transitions labeled by input actions (?m). Moreover, it can be used for detection of error states.
Lemma 3 Let $M$ be a closed TBP model containing $k$ threads and $C^k_{PD}(M)$ is its provision driven computation for $k$ threads. The set of error states $F^{BA}$ is empty iff there is no bad activity state in $M$.

3.2.3.2 Observation Projection

A key step when deciding whether the $I$ model (implementation) refines the $S$ model (specification) is to compare their ability to work in various enclosing environments. In particular, the model $I$ has to work in all environments where $S$ works. Thus, only the model’s behavior observable by the environment is important. In particular, the environment cannot make any assumptions regarding the internal non-determinism of the model (including internal communication). To include this fact in further reasoning about refinement, we define the observation projection of the provision-driven computation. The observation projection is an LTS which abstracts from the non-determinism of the original computation in a pessimistic way — whenever an error can occur due to the non-determinism, it must be reflected in the observation projection.

Each state of the observation projection (super-state) represents a set of states of the original provision-driven computation. The individual states represented by the same super-state cannot be distinguished by an environment. For instance, states originally connected by $\tau$ transition always belong to the same super-state.

Let $C^k_{PD}(M) = \left( (\Sigma_{all})^{\uparrow/}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F \right)$ is a provision-driven computation of TBP model $M$ for $k$ threads. Then let $A \subseteq S$ is a set of states. Then we define $\tau$-closure$(A)$ as to be a set of states reachable from $s \in A$ by a set of externally invisible transitions — $\tau$-transitions originated as assignments (cases a) and b) in the Definition 28).

Definition 29 (Observation projection for $k$ threads) Let $C^k_{PD}(M) = \left( (\Sigma_{all})^{\uparrow/}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F \right)$ is a provision-driven computation of TBP model $M$ for $k$ threads. Then the observation projection of $M$ for $k$ threads is a tuple $C^k_{OP}(M) = \left( (\Sigma_{ext})^{\uparrow/}_{Dom_E}, S_{OP}, s^0_{OP}, \delta_{OP}, F_{OP}, !F_{OP} \right)$ such that

- $S_{OP} \subseteq 2^S$
- $s^0_{OP} = \tau$-closure$(\{s_0\})$
- $\delta_{OP} : S_{OP} \times (\Sigma_{ext})^{\uparrow/}_{Dom_E} \rightarrow S_{OP}$
  $\delta_{OP}(s_{OP}, ?m) = \tau$-closure$(s'_{OP}) \iff \forall s \in s_{OP} \exists s' \in s'_{OP} : (s, ?m, s') \in \delta$
  $\delta_{OP}(s_{OP}, !m) = \tau$-closure$(s_{OP}) \iff \exists s \in s_{OP} \exists s' \in s'_{OP} : (s, !m, s') \in \delta$
Moreover, we define a set of various error states. In particular

- $E_{BA}^{OP} = \{ s_{OP} : \exists s \in s_{OP} : s \in F^{BA} \}$
- $E_{NA}^{OP} = \{ s_{OP} : \exists s \in s_{OP} : F^{\otimes} \cup F^{\infty}_{int} \}$
- $E_{OP} = E_{BA}^{OP} \cup E_{NA}^{OP}$

The observation projection simplifies the original provision-driven computation by reducing non-determinism. The internal choices are expected to behave as in the worst case w.r.t. bad activity. So if just one of states belonging to a super-state (e.g. several states connected by internal actions) produces an output action it must be produced also by the super-state. On the other hand, an input action leaving the super-state must be present in all states of the super-state. In other words, output action in the observation projection represents option of the model to emit the action while input action represents obligation to accept it. Consequentially, since some input actions disappear, while all output actions remain, the observation projection reduces the non-determinism in a way that tends to cause bad activity.

The Fig. 3.5 contains a fragment of a provision-driven computation. There are states connected by transitions labeled by internal, input and output actions. The leftmost state is the initial one while the rightmost state is the final one. The LTS is illustrated with initial state, final state, and the rest of the LTS.
Figure 3.6: Fragment of observation projection

a final state. The rest of the LTS is represented by the dashed line. The corresponding observation projection (Fig. 3.6) consists of three super-states. The super-state $k$ is the initial state, since one of the states it represents is the initial state of the provision-driven computation. Moreover, there is no transition labeled by an input action leaving $k$ since not all of the states are willing to accept $?p$. On the other hand, there are two output transitions. The one labeled by $!q$ leads to the super-state $l$, while the other one leads to the part of the observation projection which is not depicted. In contrast to $k$, there is an input transition leaving the super-state $l$ since all the states $l$ represent are willing to accept $?p$. The target states of these transitions are not distinguishable by the environment. Thus, they all belong to the super-state $m$.

The definitions provided in this section gradually simplify an open TBP model so that in the end, the observation projection is just a transition system labeled by input and output actions—similarly to interface automata. There are, however, two significant differences. The transition system is deterministic (there are no internal actions and each state contains at most one outgoing transition for each external action. Second, there is also an additional termination information associated with super-states.

Determining whether the observation projection $C_{OP}^k(I)$ refines $C_{OP}^k(S)$ is based on the parameterized alternation simulation.

**Definition 30 (Parameterized alternation simulation)** Let $I$ resp. $S$ be an observation projection of TBP models. Let $S_I$ be the set of states of $I$,......
% $S_S$ be the set of states of $S$, $\delta_I$ and $\delta_S$ be the transition functions of respective observation projections. Moreover, let $P \subseteq S_I \times S_S$ be a relation. Then, we call the relation $\preceq_P \subseteq S_I \times S_S$ parameterized alternation simulation if

- $\forall (s_I, s_S) \in \preceq_P: (s_I, s_S) \in P$
- $\forall (s_I, s_S) \in \preceq_P: \delta_I(s_I, !m) = s_I' \Rightarrow \exists s_S': \delta_S(s_S, !m) = s_S' \land s_I' \preceq_P s_S'$
- $\forall (s_I, s_S) \in \preceq_P: \delta_S(s_S, ?m) = s_S' \Rightarrow \exists s_I': \delta_I(s_I, ?m) = s_I' \land s_I' \preceq_P s_S'$

The relation is extended to observation projections using initial states as follows.

We say that $I$ refines $S$ with respect to the property $P$ ($I \preceq_P S$) iff there is a parameterized alternation simulation $\preceq_P$ such that $s_I^0 \preceq_P s_S^0$, where $s_I^0$ is the initial state of $I$ and $s_S^0$ is the initial state of $S$.

The parameterized alternation simulation stems from the alternation simulation introduced for interface automata in [30]. The main purpose of the alternation simulation is to relate states of the individual observation projections that correspond to each other from the observer's point of view. In particular, let $E$ be an environment enclosing $S$ and the result of composition does not contain any error. Then, when $C_k^{OP}(S)$ exercised by the environment $E$ is in the state $s_i$, then $C_k^{OP}(I)$ exercised by $E$ must be in the state $s_s$ such that $s_i \preceq_P s_s$.

The additional predicate $P$ allows specifying an additional property that must hold to address a specific error. In particular, $I \preceq_{True} S$ preserves bad activity that occurs in communication among $I$ (resp. $S$) and its enclosing environment $E$, however, it considers neither no activity nor bad activity caused by internal communication within $I$.

The Fig. 3.7 contains an example of two observation projections $I$ and $S$ such that $I \preceq_{True} S$. In particular, $c' \preceq_{True} c$ holds trivially, since there are no leaving transitions and $(c', c) \in True$. The implication 2 and $c' \preceq_{True} c$ implies $b' \preceq_{True} b$ and the implication 1 and $b' \preceq_{True} b$ implies $a' \preceq_{True} a$. Since the initial states $a' \preceq_{True} a$ then also $I' \preceq_{True} S$.

**Lemma 4 (Transitivity of $\preceq_P$)** Let $A$, $B$ and $C$ be observation projections of TBP models, and $P$ be a transitive relation over their sets of states $S_A$, $S_B$ and $S_C$. ($\neg(s_A P S_B) \land (s_B P S_C) \Rightarrow (s_A P S_C)$). Let $A \preceq_P B$ and $B \preceq_P C$. Then, $A \preceq_P C$. 

---

70
3.2. SEMANTICS

3.2.3.3 Preserving Bad Activity

Theorem 1 (Refinement w.r.t. BA up to k threads) Let $I_{OP}$ resp. $S_{OP}$ be an observation projection of a TBP model $I$ resp. $S$ for $k$ threads. Let $E_{I}^{BA}$ resp. $E_{S}^{BA}$ be sets of bad activity error states in the observation projections. Let $BA(a,b)$ be a relation over set of states of observation projections such that

$$BA(a,b) \iff (a \in E_{I}^{BA} \Rightarrow b \in E_{S}^{BA}).$$

Then $I_{OP} \preceq_{BA} S_{OP}$ implies that $I$ refines $S$ with respect to bad activity up to $k$ threads.

To proof, the theorem, we define the following lemma first.

Lemma 5 Let $A \oplus B$ form a closed TBP model inducing the provision-driven computation $C_{PD}^{k}(A \oplus B) = ((\Sigma_{inv})_{Dom_{E}}^{\uparrow/\downarrow} \cup \{\tau\}, S, s_0, \delta, F, !F)$ and let $C_{OP}^{k}(B) = ((\Sigma_{ext})_{Dom_{E}}^{\uparrow/\downarrow}, S_{OP}, s_0^{OP}, \delta_{OP}, F_{OP}, !F_{OP})$ be the observation projection of $B$. Let $t = (s_0, \alpha_0, s_1, \alpha_1, \ldots, \alpha_{n-1}, s_n)$ be a sequence of states and labels from $C_{PD}^{k}$ such that $s_0$ is the initial state, and $s_{i+1} \in \delta(s_i, \alpha_i)$.

Let $\Sigma_{prov}$ be labels representing methods provided by $B$, $\Sigma_{req}$ be labels representing methods required by $B$, and $\Sigma_{int}$ be other labels. We denote $\Sigma_{ext} = \Sigma_{req} \cup \Sigma_{prov}$. Let $ext(i)$ be the index of $i$-th label from the sequence $t$ belonging to $\Sigma_{ext}$. ($\forall i < m, ext(i) \in \Sigma_{ext}$ of $B$, where $m$ is the number of labels in $t$ representing the communication among $A$ and $B$). Moreover, let for each $i < ext(m)$, there is no state causing bad activity in $\tau$-closure($s_i$).

Then, there is a unique sequence $(s'_0, s'_1, \ldots, s'_{m+1})$ of super-states from $C_{OP}^{k}(B)$ such that (i) $s'_0 = s_0^{OP}$ and (ii) $s'_{i+1} \in \delta_{OP}(s'_i, \alpha_{ext(i)})$. 

Figure 3.7: Alternation simulation example
Figure 3.8: Example of the state sequences defined in Lemma 5

The Fig. 3.8 contains an example of the sequences defined in the Lemma 5. The sequence of states \((s_0, \alpha_0, \ldots, \alpha_8, s_9)\) induces the unique sequence of super-states \((s'_0, \alpha_1, s'_1, \alpha_3, s'_2, \alpha_4, s'_3, \alpha_7, s'_4)\). Important point is, that while the states \((s_0, \ldots, s_9)\) comes from \(C^k_{PD}(A \oplus B)\), which is a closed model and each state encodes position of threads from A as well as B, the super-states \((s'_0, \ldots, s'_4)\) are from the open \(C^k_{OP}(B)\) and encodes the threads from B and possibly some threads produced by an environment.

The lemma is proven by induction along the size of \(t\). If the size is 0 \((n = 0)\) we are looking for a super-state from \(S_{OP}\) fulfilling the condition (i), which is trivially \(s'_0\) by construction. Such state is unique.

Let \(r = (s'_0, s'_1, \ldots, s'_m)\) be the sequence given by lemma for the sequence \(t = (s_0, \alpha_0, s_1, \alpha_1, \ldots, \alpha_{n-1}, s_n)\) of size \(n\) and \(\alpha_n, s_{n+1}\) is the continuation of \(t\). Then, either \(\alpha_n \in \Sigma_{int}\) or \(\alpha_n \in \Sigma_{prov}\) or \(\alpha_n \in \Sigma_{req}\) of B. In the former case, \(r\) is the sequence we are looking for (both conditions (i) and (ii) hold). In other cases, the super-state \(s'_m\) represents all states of B \(C^k_{PD}(B)\) reachable from its initial state by sequence of labels \(t_{ext} = (\alpha_{ext(0)}, \alpha_{ext(1)}, \ldots, \alpha_{ext(m)})\) interleaved by arbitrary internal actions of B. If \(\alpha_n \in \Sigma_{req}\), one of these states has to have output transition labeled by \(\alpha_n\). If not, there could not be similar transition leaving the state \(s_m\) of \(C^k_{PD}(A \oplus B)\). Since one of states
3.2. SEMANTICS

from $C_{PD}^k(B)$ reachable by $t_{ext}$ has the output transition labeled by required event, the corresponding state in the observation projection has to have the transition too (by construction of observation projection).

On the other hand, if $\alpha_n \in \Sigma_{prov}$ of B, all states reachable by $t_{ext}$ must have the output transition $\alpha_n$. If there is a state which does not have such a transition, it would cause the bad activity error in $C_{PD}^k(A \oplus B)$ reachable by internal communication in A or B or $\tau$. Since the $\alpha_n$ transition is in all states reachable by the sequence, the observation projection contains the same transition.

To proof the theorem, we need to show that for each $E$ such that $E \oplus S$ forms a closed model and features less than $k$ threads

$$ErrFree_{BA}(E \oplus S) \Rightarrow ErrFree_{BA}(E \oplus S)$$

Let $F_{E\oplus I}^{BA}$ is the set of states causing the bad activity error in $C_{PD}^k(E \oplus I)$ and $F_{E\oplus S}^{BA}$ is the set of states causing the bad activity error in $C_{PD}^k(E \oplus S)$. Then the implication is equivalent to the following implication (application of lemma )

$$F_{E\oplus S}^{BA} = \emptyset \Rightarrow F_{E\oplus I}^{BA} = \emptyset$$

which is equivalent to

$$\exists e_I \in F_{E\oplus I}^{BA}, \exists e_S \in F_{E\oplus S}^{BA}$$

We will prove the last statement by contradiction. Let $e_I$ be a state causing the bad activity error from $C_{PD}^k(E \oplus I)$ such that it can be reached from the initial state by the smallest number of transitions labeled by events from $\Sigma_{ext}$. Let the sequence $(s_0,\alpha_0,s_1,\ldots,\alpha_m,e_I)$ be the trace leading from the initial state to $e_I$. The sequence fulfills the assumption from Lemma 5. Let $t_{ext} = (\alpha_{ext(0)},\alpha_{ext(1)},\ldots,\alpha_{ext(m)})$ be the sequence of actions representing communication between E and I. Then, there is a sequence of super-states $(s^e_0,s^e_1,\ldots,s^e_{m+1})$ in the observation projection $C_{OP}^k(E)$ following the sequence $t_{ext}$ and also $(s^i_0,s^i_1,\ldots,s^i_{m+1})$ in the observation projection $C_{OP}^k(I)$ following the sequence $t_{ext}$. The difference is that each label from $t_{ext}$ representing required method from E represents provided method in I and vice versa. We claim, that if there is no bad activity in $C_{PD}^k(E \oplus S)$, then, there is also a sequence of super-states $(s^e_0,s^e_1,\ldots,s^e_{m+1})$ from the observation projection $C_{OP}^k(S)$ following the sequence $t_{ext}$. Moreover, $\forall j: 0 \leq j \leq m: s^i_j \leq BA s^e_j$. The existence of the sequence can be proven by induction using the fact that $C_{PD}^k(E \oplus S)$ does not contain bad activity and $s^e_0 \leq BA s^e_0$. The Fig.
3.9 contains example of sequences used in the proof. The dashed grey lines illustrate the relation between states of $C_{PD}^k(E \oplus I)$ and super-states from $C_{OP}^k(I)$ resp. $C_{OP}^k(E)$.

The bad activity in $e_I$ is either caused by (i) internal communication in $E$ or by (ii) internal communication in $I$ or by (iii) communication among $I$ and $E$. For (i), the same error would be reached in $C_{PD}^k(E \oplus S)$. For (ii), $s_m^i$ is an error state and since $s_m^i \leq_{BA} s_m^s$ also $s_m^s$ is an error state. Finally, for (iii), the error is either caused by method call required by $s_m^e$ or by method call required by $s_m^i$ not accepted by the counterpart. Since the method call required by $s_m^e$ is not accepted by $s_m^i$ it cannot be accepted also by $s_m^s$ which would cause bad activity in a state in $C_{PD}^k(E \oplus S)$. On the other hand, a method call required by $s_m^i$ which is not accepted by $s_m^e$ is also required by $s_m^s$. This again leads to an error state in $C_{PD}^k(E \oplus S)$.

\[\square\]

3.2.3.4 Preserving No Activity

To define a refinement relation preserving the no activity error additional information is needed in the observation projection. Let $A_{OP}$ be an observation projection. The predicate running holds for each state of the observation projection that has to emit an output action. In particular, the predicate running is defined as follows.

**Definition 31 (Running)** Let $s_{op}$ be a super-state of the observation projection $A_{OP}$ and $S$ is a set of states in the provision-driven computation
3.2. SEMANTICS

represented by \( s_{op} \). Then

\[
\text{running}(s_{op}) = s_{op} \notin !F \\
\land \\
\forall s \in S \exists s_\tau, !m, s': s_\tau \in \tau\text{closure}(s) \land \delta(s_\tau, !m) = s'
\]

The definition consists of two properties of the state. If the set of states of the provision-driven computation represented by the super-state contains a state representing termination of all active threads (\( s_{op} \in !F \)), the whole model can terminate on its own and not to emit an output action. The second property states that for each state of the set, there has to be a path consisting of internal actions leading to a state \( s_\tau \) producing an output action.

**Theorem 2 (Refinement w.r.t NA up to k threads)** Let \( I_{OP} \) resp. \( S_{OP} \) be an observation projection of a TBP model \( I \) resp. \( S \) for \( k \) threads. Let \( E_{I}^{NA} \) resp. \( E_{S}^{NA} \) be sets of no activity error states in the observation projections. Let \( NA(i, s) \) be a relation over set of states of observation projections such that.

\[
NA(i, s) = BA(i, s) \land (i \in E_{I}^{NA} \Rightarrow s \in E_{S}^{NA}) \land \\
s \in F \Rightarrow (i \in F \lor \text{running}(i)) \land \\
\text{running}(s) \Rightarrow \text{running}(i)
\]

Then \( I_{OP} \preceq_{NA} S_{OP} \) implies that \( I \) refines \( S \) with respect to no activity up to \( k \) threads.

The proof has the same structure as the proof of preservation of bad activity. There are same sequences of states in observation projections. In particular, there are states \( s^e_i \), \( s^e_m \) and \( s^s_i \) in observation projections representing the states where the error in \( C_{PD}(E \oplus I) \) occurs. Reasoning when the error is caused by internal communication is the same. What remains to be proven is that the no activity caused by \( s^e_m \) waiting for \( s^i_m \) or vice versa implies no activity caused by \( s^e_m \) waiting for \( s^s_i \) or vice versa.

If no activity is caused by \( s^e_m \) waiting for \( s^i_m \), \( s^e_m \) is not in the final state and there is no transition labeled by required method call leaving \( s^i_m \) or \( \neg\text{running}(s^i_m) \) which implies \( \neg\text{running}(s^s_m) \). Thus \( s^e_m \) is waiting for the same event even when composed with \( S \). On the other hand, if the no activity is caused by \( s^i_m \) waiting for \( s^e_m \), \( s^i_m \) is not a final state and \( \neg\text{running}(s^e_m) \). This implies, that \( s^s_m \) is not a final state and if it is also waiting for \( s^e_m \).
3.3 Discussion

One of the objectives of the TBP design was to provide a formalism which will be simple to use by practitioners during day-to-day development. To achieve this goal, both TBP syntax and semantics are designed to be similar enough to the imperative languages commonly used to implement components. At the same time, we wanted to keep the typical usage scenario of process algebra analyses. Important part of the scenario is that verification of built-in properties as well as refinement is decidable so that both can be analyzed by an autonomous tool.

Relation of the formalism to both, imperative languages and process algebras is discussed in the following sections

3.3.1 Relation to Imperative Languages

Structure  Structure of TBP specifications is separated to five parts. Each part has its precisely defined purpose and meaning so that the user, when creating one part is not distracted by features related to the other parts. For instance, syntax and semantics of the imperative parts (threads and reactions sections) is intentionally as close to the imperative languages as possible. On the other hand, syntax and semantics of the provisions sections is closer to the process algebras. In particular, we need means to express assumptions on the way the environment is calling the individual methods. This feature is not available in imperative languages.

From the top level perspective, the overall structure of single specification resembles declaration of a class in an object-oriented language (e.g. Java). There are state variables declared (fields in Java) and list of reactions (private and public methods). The structure encourages the specification having the similar structure of control flow (purpose of methods, loops, branches) as the implementation. Moreover, there are explicitly defined threads playing the role of the "main" function.

Imperative Parts  As apparent from Sect. 3.1.4, syntax of TBP constructs for specification of the imperative parts is deeply influenced by the Java language. In particular, there are method calls including the actual parameters and return values, assignment, and control flow statements. The control flow statements differ from their imperative counterparts in the expressiveness of the conditions. The conditions are limited to predicates over enumeration values. Moreover, there is a special condition to model non-
3.3. DISCUSSION

deterministic choice—"?". Its purpose is to capture the decision based on
the data not modeled at all or modeled at a too coarse level. Such a decision
is always an internal choice of a component and it can not be influenced by
the environment.

As already mentioned, TBP supports only enumeration types to be used
to declare a variable or a method parameter. This decision was taken to
prevent users to write too large specifications (in terms of size of state space).
On the one hand, there is some support for data, on the other hand, users
are forced to use data only in an abstract way.

Parallelism Parallel execution is modeled by explicit threads. In compar-
ison to threads in imperative languages, there is no way to create a thread
dynamically in TBP. Moreover, all threads are started at the beginning of the
execution. This decision was taken to keep the formalism relatively simple.

Execution of each thread is driven by a stack, similarly to the imperative
languages. The stack keeps the return positions of reactions being currently
executed by the thread. Important point is that the number of threads
allowed to process single reaction in parallel is not limited. Moreover, to
follow abstractions provided by component models, a thread originated in
one component can execute reaction provided by other component and, after
returning, continue in its original component. Thus, we abstract from the
fact that in the real implementation, there are often more physical threads -
at least one in each participating machine in a distributed environment.

Apart from the return points in calling reactions, the stack contains lo-
cal variables which are not accessible from the other threads. Component
state variables are, on the other hand, shared by all threads executing the
component’s code.

Synchronization primitives provided by TBP are inspired by the concepts
available in Java. In particular, to support mutual exclusion, there is a
special variable of type mutex and the sync keyword which follows the idea
of monitors in Java. However, there are no counterparts for wait and notify
to support passive waiting. Instead, we encourage users to employ active
waiting in the model, since there is almost no difference in the generated
state spaces.

Communication Errors The bad activity error expresses wrong order
of provided method calls on the component boundary. While the order of
method calls is important for logic of application, it does not directly corre-
spend to an error of the execution environment (e.g. division by zero error,
or accessing wrong memory). However, invocation of methods in wrong order
(e.g. skipped initialization) often leads to such errors and should be avoided by defensive checks in programming. By checking absence of bad activity these checks become redundant.

No activity, on the other hand directly corresponds to the state of the execution environment one typically wants to avoid. The state in which no thread performs any action but the computation is not in the final state yet—deadlock.

3.3.2 Relation to Process Algebras

Specification of Provisions Provisions specifies how the environment is expected to call the methods of the component. Since this is not an aspect expressible by imperative languages, we can’t require similarity to any imperative language any longer. Instead, the syntax and semantics is inspired by behavior protocols (BP). There are atomic events (method calls) and operators representing sequence, alternative, repetition and parallelism.

Since provisions capture how the environment uses the component, there are only input actions allowed in the provisions. All in all, provisions specify input actions while imperative parts specify output actions. Following this separation, the alternative operator in the provisions section always represents an external choice - the decision which alternative is taken is done by the environment. Thus, all choices in imperative parts are internal, while the choices in provisions are external. Moreover, there is a repetition operator $|*$ which represents a parallel execution of its operand as many times as the environment requires.

In contrast to BP, the atomic events are equipped with parameters and return values of enumeration types. It allows specifying the assumptions on the environment with finer granularity than mere sequence of method calls.

Another important aspect is that individual assumptions are specified separately. For each model, there can be several provisions guarding different sets of methods. Some methods can be omitted by the provisions section. In such case, the method can be invoked arbitrarily by the environment, even in parallel. On the other hand, a method can be guarded by several provisions and in such a case the environment has to follow all of them. Such approach allows the developer to think about individual assumptions separately.

Communication When process algebras are applied in a component system, each component is typically modeled by a single process. The communication among processes is represented by synchronization of actions representing method calls. The direct consequence is that method calls are
represented by asynchronous communication—a) the method call is not accepted if it is not explicitly awaited by the target process and b) the source process continues in the computation and does not wait for the result. This is far from the semantics of method calls in imperative languages. The issue b) can be resolved by using pairs of actions - the output action at the source process is immediately followed by an input action waiting for the result (and vice versa in the target process).

The issue a) requires further attention. The issue is the most burning when the user wants to model a method which can be invoked as many times in parallel as the environment requires. Such behavior is, by the way, default for all Java classes. Process algebras supports recursion (CSP) which allows unlimited number of processes starting in parallel as required by the environment. This however, makes the formalism more complex and state space of an open system infinite.

TBP approaches the communication among components in a different way. The concept of threads in TBP follows the concept of threads from imperative languages as close as possible. In particular, the thread is a source of activity. The computation starts in threads explicitly listed in the specification. When the thread reaches a method call, its execution moves to the method’s reaction even if the method is provided by another component. In particular, there does not have to be another thread in the target component waiting to accept the method call. Since the thread execution moves to the target’s reaction it does not immediately continue in the execution at the source site. The thread returns there when it finishes the execution of the reaction.

When considering the shape of LTS specified by the model, it captures parallel interleaving of activities executed by individual threads. Individual threads influence each other only by modifications of state variables.

As long as the parallelism is not explicitly limited in provisions, a method can be invoked as many times in parallel as the environment requires.

**Composition** In contrast to the process algebras, composition of specifications is defined at the syntax level as union of the corresponding sets. Semantics define how to create an LTS for an open model (observation projection), however there is no mean for composition of two observation projections. Instead, two open models are composed at the syntax level into single model (possibly a closed one) which is transformed to LTS.

Such notion of composition helps to implement the notion of threads as described in the previous paragraph. In particular, when the closed model already contains all threads and reactions of the communicating components,
the LTS can be easily created by simulation of the threads visiting all reactions while checking whether no provisions are violated. In other words, a part of state space representing particular invocation of a method by a thread is created on demand, as needed when the thread visits the method call.

If such LTS was created separately for each component A (open models) it would be necessary to create parts of the state space representing the provided methods in advance at right places and with sufficient parallelism. Without the information from the other models being composed with A, this would often lead to infinite LTSes (allowing the method to be executed as many times in parallel as the environment requires. Since at the same time, the missing information about the parallelism is in the other models being composed with A it is better to put all information together and create the resulting LTS representing the closed model in a single step.

This way, we avoid creation of LTSes for open models when deciding correctness of closed models. However, such construction can’t be avoided when deciding refinement of open models. This is discussed later in the special paragraph.

Communication Errors  The errors supported by TBP are based on comparison of behavior actively performed by threads in the system and description of behavior passively expected by provisions. Since both can be expressed as a set of traces, the properties can be stated as checking inclusion of these sets. In particular, witness of an error is a trace produced by threads which is not in the set of traces allowed by provisions. As discussed in Sec. 3.2.2.1, such trace is either prefix of a trace allowed by provisions or it differs at i-th position or it is a continuation of a trace allowed by provisions. While in the first case the active threads do less than expected by the provisions (no activity) in the other cases the active threads do more than allowed by the provisions (bad activity).

When comparing actual behavior to expected behavior on the traces level, there are hardly any other errors. However, individual formalisms can differ significantly in the way the traces are constructed. In particular, this involves how they deal with non-determinism and internal/external choices. Related to the non-determinism is the ’degree of optimism’ in the formalism - whether the model is considered to be erroneous if the error can appear or only in the cases when the error is inevitable and must occur during the execution. TBP in particular, is conservative and reports the error if it can occur.

The table 3.3.2 compares the TBP errors with the stuck error introduced in [33] and discussed in [34]. The stuck error resembles both bad activity and no activity. The table presents pairs of label transition systems demon-
3.3. DISCUSSION

Models
Bad Activity
IA Error
No activity
Stuck
A B
a) Yes No No
b) Yes No Yes
c) No Yes Yes
d) No No Yes

Figure 3.10: Error comparison

strating differences between different errors. At each picture the top state is always the initial state while the state with additional circle within represents a final state.

The first pair demonstrates the difference between the bad activity error and stuck error. While in TBP composition of models corresponding to the LTSes is considered to be erroneous because of the \( !b \) action from A which is not accepted by any counterpart in B, the same situation does not mean the stuck error, since there suffices that at least one of actions is accepted by the counterpart (\( !a \) and \( ?a \)). For instance the composition of the second pair produces the stuck error - no action leaving initial states is accepted. The second pair also reflects the difference between the stuck error and no activity. While the definition of no activity considers final states and cannot occur in final states, the stuck error does not consider final states at all. That’s why there is the stuck error while no no activity in the second line. The third line demonstrates that bad activity distinguishes input and output actions (\( ? \) and \( ! \)) while stuck error does not.

The bad activity error has the same meaning as the error introduced in [30] for interface automata.

81
Refinement  Refinement framework is based on the alternation simulation introduced in [30] for interface automata. For TBP, however, the relation is strengthened to preserve absence of no activity. The definition of refinement for TBP is provided in three steps.

First, the provision-driven computation transforms an open TBP model into an LTS similar to the interface automata—there are input, output and internal actions. In the following step, observation projection is created. In contrast, the observation projection is a deterministic LTS.

The final step for the refinement preserving bad activity is to check the parametrized alternation simulation of two observation projections. This means to compare options and obligations of the corresponding states from the implementation and specification. At first, the initial states are compared and then the process continues transitively to the pairs identified by the transitions labeled by the same actions.

By definition of refinement in three steps, the theoretical framework behind gets more modular. In particular, the second step gathers the information from several states of a single provision driven computation indistinguishable by the environment into single state of the observation projection. Most importantly, the second step resolves non-determinism; it determines what events important for preserved errors must or may occur. For instance, in our case the second step deals differently with input and output actions. On the other hand, if the preserved error could occur only in stable processes (like in stuck-freedom [33]) the changes would take place in this step. Thus, the third step, which does actually compare the corresponding states, does not have to consider these differences and can be defined in a straightforward way.

Regarding the third step, it consist of two aspects. The relation is always propagated along the transitions labeled with the same event. The propagation is ensured by the parametrized alternation simulation. Apart from the propagation, there is always a formula P that relates some additional properties of super-states gathered in the third step. The formula serves as the parameter of the parametrized alternation simulation.

In this context, it is worth to mention the special position of the refinement w.r.t bad activity. There, the formula P refers only to potential bad activity caused by composition and no other information is used. In other words, the bad activity error serves as a basis for relating pairs of states. Thus, when two specifications do not refine w.r.t. bad activity, they won’t refine w.r.t. arbitrary property.
3.3.3 Employing TBP in CBD

For successful application of TBP in the component context, several technical issues remains to be solved.

**Interface Names**  In particular, the composition (Definition 10) assumes that the names of interfaces bound together do correspond. Otherwise, it produces formally correct TBP model, however method calls are not linked to provided methods as intuitively expected. Typically, however, the interfaces do not have corresponding names - they share just the interface type. Names differ especially in the cases when the components come from different vendors. To ensure the equality of interface names, renaming driven by the information from the architecture is employed in the same way as suggested for BP in Sect. 2.3.4.

**Compatibility of Enumeration Types**  Similarly, the composition assumes, that type definitions used in composed models are structurally equal. In particular, once a type is used in two communicating parties, its name as well as structure (i.e. number and names of enumeration values) must be equal. Since communicating parties should agree on parameter types in similar way as they agree on interface types (types of parameters belong to signatures of methods) it suffices to check the structural equality when composing the models—no renaming is employed.

Although TBP is a step towards imperative programming languages, there are concepts that are either not supported at all or very simplified. When crafting specifications, developers have to keep these differences in mind and deal with them accordingly.

**Enforced Abstraction**  One of the differences are supported data types. TBP supports only enumeration data types. On the one hand, it makes the formalism simpler. On the other hand, it forces developer to employ abstraction and produce smaller (in terms of state space) specification. To capture more complex data types with the granularity needed for description of control flow, developer should consider conditions used in the implementation and design the enumeration values that reflect the individual conditions. For instance, a list is often abstracted to two value enumeration type representing an empty and non-empty list.

**Dynamic Threads**  Another gap is in the capabilities of threads. First, there is no support for dynamic creation. As long as the number of threads
is limited by a constant $n$, one can create a specification with $n$ threads such that all exist from the beginning, but those that represent the dynamically created threads start with a condition waiting for an associated state variable signaling the dynamic creation. Similar transformation is used in [64] to allow context switch bounded model checking of dynamic concurrent pushdown systems. Unlimited number of threads in closed systems is not allowed and developer has to avoid it. Regarding the open systems, refinement supporting environment with an unlimited number of threads is subject of future work.

**Recursion** Finally, it has to be mentioned, that the tool for analysis of TBP relies on assumption that the closed computation of the input model is a finite LTS. In practice, this means to avoid recursive functions (as mentioned after Definition 15). Actually we avoid this because in presence of concurrency, reachability of recursive programs (modeled often as pushdown systems) is undecidable, although there are partial methods to face it [64]. Formalism for component specification allowing recursion is proposed in [15]. There, the formal framework is backed by the process rewrite systems [54], where the reachability is decidable (In the process rewrite systems communication among threads is not allowed).

To sum it up, we do not support the recursion for two reasons. First, it would make the formal framework of TBP more complex. Moreover, we do not consider recursion among components to be a good practice.
Validation

This chapter presents our experience with behavioral modeling using TBP and subsequent analysis. To demonstrate capabilities of TBP, we present a TBP specification of the SessionManager component.

4.1 Session Manager

The SessionManager component is expected to work in the environment suggested in Fig 1.1. In particular, the SessionManager component intercepts the communication between the (web based) user interface and the application logic to provide the authentication feature.

The basic functionality of the component is to associate commands from individual users with sessions. In particular, in order to invoke commands, the user interface has to acquire a session id (invoke the createSession method provided by SessionManager). Then, if a valid session id is returned, it is used as a parameter for subsequent commands. When SessionManager accepts the command it checks the session id and passes the command to the business logic. Apart from the main functionality, the component implements a maintenance mode. The mode is automatically turned on when the administrator user is logged in. In the maintenance mode, no new sessions can be created (createSession returns INVALID_SESSION) and new requests executed in a context of other than the administrator’s session causes invalidation of the request session. Moreover, the session timeout is implemented, so that SessionManager may decide on its own to invalidate an arbitrary session.

It is important to emphasize the purpose of the model since it determines...
the abstractions to be used. In this case, the goal is to describe dependencies of individual method calls in presence of parallelism. In particular, we want to identify deadlocks and wrong ordering of method calls.

For instance, we do not model the particular authentication algorithm since it is not important. Just the result influences the sequencing of method calls and the implementation must be prepared for the positive result as well as for the negative one. Moreover, we do not distinguish individual users. There are just several types of sessions (ADMIN_SESSION, USER_SESSION and INVALID_SESSION). In general, the data abstractions are chosen to reflect the conditions in the code that influence the control flow.

In the model, there is just one provision (lines 18-28) prescribing how the user interface can call the methods on the session interface. In particular, user interface is allowed to submit a command (invokeCmd), only if the createSession method returns a valid session id (i.e. USER_SESSION or ADMIN_SESSION). The component is able to process requests from several users in parallel (| and |* operators).

Then, there are reactions for the provided methods createStatement() and invokeStatement() and the reaction for the internal method terminate Session() which is invoked from three different places. When the user explicitly issues the CMD_LOGOUT command (line 56) to invalidate an existing normal user session in the maintenance mode (lines 57-59) and finally on timeout (line 86). The timeout is implemented by the single autonomous thread in the specification, Timer.

```plaintext
component SessionManager {
   types {
      DbResult = {DB_GRANTED, DB_REFUSED};
      SessionId = {USER_SESSION, INVALID_SESSION, ADMIN_SESSION};
      UserId = {ADMIN_ID, USER_ID};
      Command = {CMD_LOGOUT, CMD_OTHER};
      OperationMode = {NORMAL_MODE, ADMIN_MODE};
   }
   vars {
      OperationMode opMode = NORMAL_MODE;
      Mutex m;
   }
   provisions {
      OptionalProvisionName {
```
4.1. SESSION MANAGER

```java
19 {session.createSession():USER_SESSION;
20     session.invokeCmd(USER_SESSION,?)*}
21 |
22 {session.createSession():ADMIN_SESSION;
23     session.invokeCmd(ADMIN_SESSION,?)*}
24 |
25 session.createSession():INVALID_SESSION
26 }|
27 for {session.createSession(),session.invokeCmd}
28 }
29
30 reactions {
31     session.createSession(UserId userId):SessionId{
32         DbResult queryResult = DB_REFUSED;
33         queryResult = db.query();
34         sync(m) {
35             if (queryResult == DB_GRANTED) {
36                 log.log();
37                 if (userId==ADMIN_ID){
38                     opMode = ADMIN_MODE;
39                     return ADMIN_SESSION;
40                 }
41                 if (opMode==NORMAL_MODE){
42                     return USER_SESSION;
43                 } else {
44                     log.log();
45                 }
46             return INVALID_SESSION;
47         }
48     }
49     }
50
51     session.invokeCmd(SessionId sessionId,
52                 Command cmd)
53     {
54         if (sessionId == INVALID_SESSION) return
55         if (cmd == CMD_LOGOUT ||
56             (sessionId== USER_SESSION
57                 &&
58                 opMode==ADMIN_MODE))
59             log.log();
60     }
```
As major benefits of the presented specification we consider following points:

- The structure resembles realistic implementation.

- Specification does not assume particular number of threads using its provided methods. The component can be composed with an arbitrary number of components creating sessions in parallel.

- A part of the specification (`intr.terminateSession` method) is reused in different contexts.

  Similar specification in BP, would be either more abstract (i.e. imprecise) or it would contain duplicate fragments of code. EBP specification,
on the other hand would implicitly prescribed particular number of client components allowed to create sessions in parallel.

## 4.2 TBP Checker

The tool for analysis of TBP specifications is under development. Currently, it supports checking of closed specifications for absence of bad activity. Technically, the main idea of the tool is inspired by the SPIN model checker [41]. In particular, the TBP checker generates an executable code traversing the state space of the given model. In contrast to SPIN, we generate Java code instead of C.
Conclusion and Future Work

The proposed specification language, TBP, aims at narrowing the gap between imperative languages used in industrial software development and behavior specification languages.

Key achievements and benefits While TBP remains to be a specification language, it tries to get as close as possible to the syntax and semantics of the mainstream imperative languages (Java in particular). Specifically, both the syntax and semantics is inspired by the imperative object-oriented programming languages the industrial developers are used to (Goal 1.a). Thus, the model can be crafted by similar means as the implementation—developers operate with the same concepts and the specification structure can be followed in the implementation as far as to the individual control flow statements. Important benefit is also that synchronization in TBP is inspired by the approach taken by Java. At the same time, TBP specification allows specifying assumptions on the environment, fulfilling thus the Goal 1.b.

TBP allows analyzing the models of applications in context of component systems (Goal 2). In particular, TBP supports checking of composition correctness, as well as checking of refinement. The notion of correctness is based on verifying the actual behavior of a closed system against the assumptions. We show, that the errors considered (bad activity and no activity) cover all possible violations of the assumptions.

To sum it up, TBP is a step towards narrowing the gap between a specification and implementation language. Strictly speaking, there is neither a feature, nor an idea in TBP that would be utterly new. All ideas already appeared either in the world of behavioral modeling or in an implementation language. TBP, however, puts the ideas together in a novel way.
When compared to previous formalisms from the family of Behavior Protocols, TBP is a step forward in user friendliness and at the same time in the properties of formal framework.

The models in TBP are easier to write and understand when compared to other specification languages as demonstrated on the session manager example originally introduced in Sect. 4.1. The tool for checking TBP correctness is currently under development. Regarding the formal framework, we consider beneficial the refinement relation preserving both bad activity and no activity. Moreover, the refinement relation is modular, which helps enhance it in order to support additional errors.

**Future Work and Open Issues** There are still several features that are worth modeling, but there are no means for that in TBP. Specifically:

- Dynamic thread creation.
- Dynamic object support.
- Unlimited number of threads considered in the refinement.

We consider to address the last bullet: To provide the refinement relation supporting unlimited number of threads triggered by environment, we want to add level of parallelism (based on the reentrancy operator) to the observation projection and define a relation that compares implementation and specification in a way that considers the level of parallelism.

Dynamic thread creation and dynamic object support still remain open issues.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


