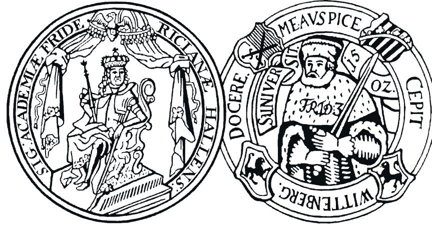


Protocol Conformance Checking of Component-based Systems and Service-oriented Architectures

Dissertation
zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)

vorgelegt der
Naturwissenschaftlichen Fakultät III
(Institut für Informatik)
der Martin-Luther-Universität Halle-Wittenberg

von Herrn
Dipl.-Inform. Andreas Both
geboren am 29.06.1979 in Halle (Saale)

Gutachter:

1. Prof. Dr. Wolf Zimmermann, Martin-Luther-Universität Halle-Wittenberg, Deutschland
2. Prof. Dr. Heinrich (Heinz) W. Schmidt, Melbourne Institute of Technology, Australien

Halle (Saale), 27.11.2009 Datum der Verteidigung: 12.03.2010

Für Sannie, Annelie + 2.

Acknowledgments

Finally, I am in the position to thank all the people who supported me during the last years while I have been working on my thesis.

For giving me the chance to write this thesis I am grateful to my supervisor Professor Dr. Wolf Zimmermann. Moreover, I thank Professor Dr. Heinz Schmidt for his very useful suggestions and his willingness to write an expertise.

I am grateful to my colleagues of the Institute for Computer Science at the Martin Luther University of Halle-Wittenberg for their help and discussions. Fortunately, I was able to motivate many students to work together hardly on implementing the defined verification framework. The "P2"-team and Stephan Prätsch had the most impact on the development and thus deserve a big thank you.

Special thanks go to Anna Rambow, Dirk Richter, Karoline Makosch, Michael Hanke, Henning Thielemann, Hendrik Bugdoll, Martin Gleditzsch, Jacob Dölle, Stefan Noke, Jan Heichler, Alexander Hinneburg, Sandro Wefel, Tobias Habermann, Christian Bodamer, Martin Müller, Norman Richter, and Thomas Piskol for reading (parts of) the thesis and their suggestions for improvements.

I thank the companies ComServ Ingenieurgesellschaft mbH and GISA GmbH for allowing me to implement case studies. I would like to express my sincere gratitude to OR Soft Jänicke GmbH. Their long-term support was a big cornerstone of the success of my work.

Last but not least, I would like to express my gratefulness to my family for their help and my wife Susanne for her love and patience. Cheering me up together with my little sunshine Annelie was a big support during the time of writing this thesis.

Abstract

In this thesis, an approach was developed that allows the fully automatic verification of non-functional properties of component-based software and Service-oriented Architecture (SOA). These properties are considered basing on the performed interactions between software components. Precisely, the contracts – named *component protocols* – state, which interaction sequences (e. g., orders, repetitions) are allowed, while calling a provided interfaces of a component. Protocols can be used to represent properties like safety and reliability. For the checking of protocol conformance a verification process is used, whose unique feature is the usage of an abstract behavior representation that is capable of capturing unbounded parallelism and unbounded recursion.

The verification process is divided into five steps that are performed automatically. This ensures the applicability in the scope of component-based software development, i. e., the component characteristics are obeyed even within the verification process. During the verification process, it is ensured that the behavior of the considered components is captured conservatively. Thus, each contained error will be discovered with respect to the given component protocols.

The definition and implementation of the verification process focuses on the (practical) applicability of the suggested approach in an industrial context. In particular, performance and precision of the model checking algorithm, as well as protection of business secrets and intellectual properties, are considered.

Our approach makes it possible to check a given system for errors before the deployment (static verification). This way, an important aspect of reliability is guaranteed. Besides other scopes of applicability, it is possible to verify component-based software where components are bound dynamically. Moreover, a generalized approach was developed with respect to the programming languages used for implementing components. Furthermore, we show that an integration into a (formal) software specifications process is possible, as well as the integration into an (iterative) software development process.

The suggested approach was checked in (industrial) case studies considering several programming languages. The results of the case studies show that verification is possible to consider practical problems, even for actual projects containing several thousands lines of source code. This is achieved by defining and implementing an efficient model checking algorithm and further optimizations.

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, das die vollautomatische Verifikation von nicht-funktionalen Eigenschaften von aus Komponenten bestehenden Programmen (insbesondere Service-orientierten Architekturen) erlaubt.

Die hier betrachteten nicht-funktionalen Eigenschaften sind die Interaktionen zwischen verschiedenen Softwarekomponenten; genauer: in den sogenannten *Protokollen* wird festgelegt, welche Aufrufe der angebotenen Schnittstellen (z. B.Reihenfolge, Wiederholungen) erlaubt sind. Protokolle können genutzt werden, um Eigenschaften wie zum Beispiel Sicherheit und Zuverlässigkeit sicherzustellen.

Zur Überprüfung der Komponenten (Protokollprüfung) werden deren Abstraktionen herangezogen. Hierbei ist es erstmals gelungen, eine Repräsentation zu nutzen, die sowohl unbeschränkte Parallelität, als auch unbeschränkte Rekursion darstellen kann.

Zur Verifikation dieser Protokolle wurden fünf automatische Verarbeitungsschritte definiert. Diese stellen die Anwendbarkeit in Komponenten-orientierter Programmierung bzw. Entwicklung sicher, d. h. auch bei der Verifikation bleibt der Komponenten-Charakter erhalten. Es ist sichergestellt, dass das Verhalten der Komponenten und damit des Komponenten-basierten Systems konservativ erfasst wird. Somit ist während der Modellprüfung garantiert, dass jeder Fehler gefunden wird.

Während der Entwicklung wurde besonderer Wert auf die Anwendbarkeit des Modellprüfungsansatzes gelegt, so dass eine Anwendung im industriellen Kontext möglich wird. Dies betrifft sowhl die Performance und Genauigkeit des Modellprüfungsalgorithmuses, als auch den Schutz von Geschäftsgeheimnissen.

Mit unserem Ansatz ist es möglich, einen Beitrag zur Verifikation von Komponenten-basierten Systemen zur Erstellungszeit zu leisten, da Systeme bereits vor der Auslieferung auf Fehler überprüft werden können. Dies stellt einen wichtigen Beitrag zur Erstellung von zuverlässigen Software dar. Neben anderen Anwendungsbereichen ist es darüber hinaus möglich, Systeme zu verifzieren, die Komponenten dynamisch einbinden und deren Komponenten in verschiedenen Programmiersprachen entwickelt wurden. Darüber hinaus wird gezeigt, dass eine Integration dieses Ansatzes sowohl in einen (formalen) Spezifikationsprozess als auch in einen (iterativen) Entwicklungsprozess möglich ist.

Das vorgeschlagene Verfahren wurde in umfassenden (industriellen) Fallstudien überprüft, in denen verschiedene Programmiersprachen betrachtet wurden. Die Ergebnisse der Fallstudien zeigen, dass mit einer effizienten Implementierung des Modellprüfungsalgorithmuses und weiteren Optimierungen praktische Probleme selbst dann zu lösen sind, wenn diese vollständige, reale Projekte mit vielen tausend Quelltext-Zeilen betreffen.

Contents

List of Figures **xv**

List of Examples **xvii**

List of Tables **xxi**

1. Introduction **1**

1.1. Basic Problem 2
1.2. Scientific Problem 3
1.3. Properties of Components 4
1.4. Test vs. Verification 5
1.5. Verification of Component-based Software 6
1.6. Representations of Component Behavior 8
1.6.1. Sequential vs. Parallel Behavior 9
1.6.2. Specification vs. Source Code Behavior 9
1.7. Problem Definition 10
1.8. Approach to a Solution 11
1.8.1. Correct Composition of Components 11
1.8.2. Representing the Behavior of Components 12
1.8.3. Ensuring that the Behavior of the Components matches the Specification 12
1.8.4. Developed Verification Process 12
1.9. Structure of this Work 14

2. Related Work **15**

2.1. Classification of Approaches 16
2.2. Detailed Discussion of Approaches 18
2.2.1. Behavior Representations 19
2.2.2. Constraints and Composability 22
2.2.3. Model Checking 23
2.3. Conclusions 24

3. Foundations **27**

3.1. Components and Component Systems 27
3.1.1. Components 27
3.1.2. Component Systems and Composition 33
3.1.3. Classification of Component Behavior 37
3.1.4. Cactus Stack 39
3.2. Formal Descriptions of Behaviors 40
3.2.1. Traditional Representations 40
3.2.2. Process Rewrite Systems (PRS) 44

3.2.3. Hierarchy of Formal Representations 47
3.2.4. Model Checking 49
3.3. Summary 51

4. Protocol Conformance and Abstractions 53
4.1. Component Protocols (short: Protocol) 53
4.2. Protocol Conformance 56
4.3. Abstractions of Source Code 61
4.4. Use Process Rewrite Systems as Behavioral Representation 63
4.4.1. PA-processes and Process Algebra Nets 66
4.4.2. Example Language 66
4.4.3. Capturing Behavior with Process Rewrite Systems 66
4.5. Summary 70

5. Verification Process 73
5.1. Step 1: Creating Single Component Abstractions 75
5.1.1. General Approach 77
5.1.2. Creating Stripped Process Algebra Nets from a Single BPEL Process . 81
5.1.2.1. Introduction of the Web Services Business Process Execution Language 81
5.1.2.2. Example 82
5.1.2.3. Generating Abstractions of BPEL Web Services 84
5.1.3. Implemented Translations to Process Rewrite Systems 86
5.1.3.1. Creating Stripped Process Algebra Nets from a Single Python Component 86
5.1.3.2. Creating Stripped Process Algebra Nets from a Single BPEL Webservices 87
5.1.3.3. Creating Stripped Process Algebra Nets From a Single PHP Component 87
5.1.4. Summary 87
5.2. Step 2: Creating System Abstractions 88
5.2.1. Process 89
5.2.2. Restrictions 94
5.2.3. Optimizations 94
5.2.3.1. Contracting λ-rules 95
5.2.4. Summary 99
5.3. Step 3: Creating Combined Abstractions 100
5.3.1. Process 101
5.3.1.1. Discussion of the Model Checking Problem 101
5.3.1.2. Construction of the Combined Abstraction 102
5.3.2. Optimizations of the Combined Abstractions 107

5.3.2.1. Optimizations during Creation 107
5.3.2.2. Remove Unresolvable Transition Rules 108
5.3.3. Summary 108
5.4. Step 4: Performing Protocol Conformance Checking 109
5.4.1. False Negatives 110
5.4.2. Reducing the Number of False Negatives 113
5.4.2.1. Basic Idea 113
5.4.2.2. The Round-Robin Reachability Algorithm 115
5.4.2.3. Summary 121
5.4.3. Improving Runtime of Model Checking Using PRS Properties 122
5.4.3.1. Construction of Π^{PA} 124
5.4.3.2. Discovering Spurious Counterexamples 126
5.4.4. Summary 127
5.5. Step 5: Evaluating Counterexamples 127
5.5.1. Extending the Counterexamples 128
5.5.2. Evaluating Extended Counterexamples 131
5.5.3. Summary 131

6. Implemented Framework and Case Study 133
6.1. Implemented Framework 133
6.1.1. Abstractions (short: WSA) 134
6.1.1.1. Translation of Python Statements into PRS Transition Rules . 134
6.1.1.2. Translation of PHP Statements into PRS Transition Rules . . 134
6.1.1.3. Translation of BPEL Activities into PRS Transition Rules . . 135
6.1.1.4. Representation of PRS Transition Rules 136
6.1.1.5. Summary 137
6.1.2. PRS Operations (short: WSO) 137
6.1.3. Model Checker (short: WSB) 138
6.1.4. User Interfaces (short: WSC+P2) 139
6.1.5. Summary 140
6.2. Considered Case Studies 142
6.2.1. OR Soft Workbench 142
6.2.2. EMenue.net 144
6.2.3. Fail2Ban 146
6.2.4. BPEL workflows 146
6.2.5. Summary 149
6.3. Verifying Combined Abstractions 149
6.3.1. The Experimental Setting 150
6.3.2. The Results 150
6.3.3. Summary 151
6.4. Discussion 151

7. Method for Using Protocol Conformance Checking in Iterative Component System Integration 155
7.1. Motivation . . . 155
7.2. Iterative Verification . . . 156
7.3. Verification Process for Iterative Development . . . 160
7.4. Evaluating the Result of the Model Checker . . . 161
7.5. Verification Contexts (No Callbacks Allowed) . . . 162
7.5.1. Components with no Unbounded Required Interfaces . . . 162
7.5.2. Components with Unbounded Provided and Unbounded Required Interfaces . . . 165
7.6. Verification Context (Allowing Callbacks) . . . 166
7.7. Discussion . . . 170

8. Conclusions and Future Work 171
8.1. Component Properties . . . 172
8.2. Comparison with other Approaches . . . 173
8.3. Implementation and Practical Applicability . . . 175
8.4. Future Work . . . 175

Index of Definitions 180

Bibliography 181

A. Appendix 197
A.1. Notations Used in this Thesis . . . 197
A.2. Syntax of Used Programming Language . . . 197
A.3. More Figures, Tables and Listings . . . 199
A.4. Extended Consideration of Combined Abstraction . . . 215
A.5. Used Tools . . . 217

List of Figures

1.1. Preparation and verification process (Overview). 13

3.1. Component model. 28
3.2. Interface of initial components. 33
3.3. Hierarchy of basic PRS operators. 44
3.4. Hierarchy of Process Rewrite Systems. 49

5.1. Detailed preparation and verification process. 74
5.2. Implementation of a component C, its abstraction Π_C and protocol P_C. 87
5.3. Schematic representation of the construction of the system abstraction. 88
5.4. Calculation of optimized transition sets. 98
5.5. Directives for construction of transition rules of a Combined Abstractions Π_S^C. . 104
5.6. Concept of a posteriori verification algorithm. 122

6.1. Overview of implemented architecture. 134
6.2. Implemented workflow. 135
6.3. Definition of Web Service interface “Abstractions”. 136
6.4. Definition of Web Service interface “PRS Operations”. 137
6.5. Definition of Web Service interface “Model checker”. 138
6.6. Definition of Web Service “user interface”. 139
6.7. Screenshot of implemented frontend “P2”. 140
6.8. Correlation of verification process and implemented components. 141
6.9. Case study: OR Soft Workbench. 143
6.10. Case study: EMenue.net. 145
6.11. Case study: Fail2Ban. 147
6.12. Case study: Web Services Business Process Execution Language. 148

7.1. Possible scenarios during development. 159
7.2. Verification processes for iterative development. 161

8.1. Overview: Verification process. 172

A.1. A Saguaro cactus (Carnegiea gigantea), `wikimedia.org`. 199
A.2. Limits of the decidability of linear-time logics. 200
A.3. Limits of the decidability of branching-time logics. 201
A.4. The complexity of reachability of Process Rewrite Systems. 201

List of Examples

3.1. Set of interfaces (implementations in Example 3.2 on page 31). 30

3.2. Set of components (corresponding interfaces in Example 3.1 on page 30). We reuse these components partly in the thesis. 31

3.3. Set of blackbox components (corresponding to Example 3.2). 32

3.4. Composite component C_3 consisting of the component C_1 and C_2. 34

3.5. Composed blackbox components (corresponding to Example 3.2 on page 31). . 36

3.6. Growing and shrinking of a cactus stack. 41

3.7. Process-algebraic expression and corresponding cactus stack. 44

3.8. Growing of cactus stack and corresponding process-algebraic expressions. . . . 45

3.9. Transformations of process-algebraic expressions using a given set of transition rules. 48

4.1. SSO-component and corresponding protocol. 55

4.2. Sequence diagrams (drawn using showing Π_{S,C_2}, use of component C_2. 57

4.3. Glassbox components, where the execution results in an error. 59

4.4. Cactus stack representing erroneous execution trace (counterexample). 60

4.5. System abstraction of S and use of component as PRS. 65

4.6. Behavior not represented exactly by a Process Rewrite System. 71

5.1. $\Pi_{C_{\text{start}}}$, Π_{C_0}, Π_{C_1} and Π_{C_2} as Stripped Process Rewrite Systems. 76

5.2. Constructed system abstraction Π_S of a component-based software. 77

5.3. Constructed system abstraction Π_{S,C_2} under consideration of only the interactions of component C_2 in Example 5.2. 78

5.4. Erroneous execution trace (derivation). 79

5.5. Computing an abstraction of a component C. 80

5.6. Bank system implemented using BPEL. 83

5.7. Process rewrite rules of the Stripped Process Algebra Nets Π_C, Π_B and Π_L. . . 84

5.8. Rewrite rules of system abstraction $\Pi_{S,C}$ of Example 5.7 according to the service protocol of the Web Service C. 90

5.9. Component-based software with labeled program points. 93

5.10. Rewrite rules of abstraction Π_{S,C_2} according to the protocol of the component C_2. 94

5.11. Contracting λ-rules. 95

5.12. Erroneous execution trace (derivation) considering component C_2 only. 100

5.13. Visualization of the encoded state of the Combined Abstraction. 103

5.14. Problematic situation while dealing with a Combined Abstraction Π_S^C. 105

5.15. Trace constructed by the reachability algorithm. 110

5.16. Abstraction resulting in a real false negative. 111

5.17. Abstraction containing a spurious false negative. 114

5.18. Spurious false negatives. 115

5.19. Adaption of Example 5.18 on page 115: language inclusion results in an impractical output. . . . 116
5.20. Situation described by Property 5.3. . . . 116
5.21. Complete information of Example 5.18 on page 115. . . . 120
5.22. Example with four interfaces and components. . . . 123
5.23. Constructed Π^{PA} from Π^{PAN} ($\Pi_{S,\mathtt{C3}}$). . . . 125
5.24. A computable counterexample in Π^{PA}. . . . 125
5.25. Inflating number of counterexamples. . . . 129
5.26. Extended counterexample of Example 5.25a on page 129. . . . 130

7.1. Application assembled from three component. . . . 157
7.2. Abstractions of the components in Example 7.1 on page 157. . . . 158
7.3. Construction of the verification driver. . . . 163
7.4. Verification context, we use C2 and C3 (and $\mathcal{P}_{\mathtt{C2}}$) from Example 3.2 on page 31 and 7.2 on page 158. . . . 165
7.5. $\Pi_S \hat{=} C_A^{\circlearrowleft} \oplus \Pi_{\mathtt{C2}} \oplus \Pi_{\mathtt{C3}} \oplus C_A^{\circlearrowleft}$ (cf. Example 7.2 on page 158) with callbacks. . . . 167

A.1. Service for registration and confirmation of service items based on time and material. The example is discussed intensely in [BZ09b]. . . . 199
A.3. Program returning thread identifier as return value. . . . 202
A.2. A more specific protocol of component C_2 of Example 4.3 on page 59. . . . 202

List of Algorithms

5.1. Construction of $\rightarrow_S$ (needed in Definition 5.3 on page 89) 91
5.2. Contraction of λ-rules. 97
5.3. Lazy construction of the Combined Abstraction 107
5.4. Mayr's Algorithm 112
5.5. Round-robin reachability algorithm in pseudo code. 118
5.6. Verification algorithm in pseudo code. 124

List of Tables

2.1. Overview of related work. 19

3.1. Model checking results considering Process Rewrite Systems. 50

5.1. Considered BPEL activities. 82

6.1. Results of model checking. 150
6.2. Timeouts while using $\mathcal{M}^{\text{PAN}}$ in comparison to $\mathcal{M}^{\text{PA}} + \mathcal{M}^{\text{PAN}}$. 151

8.1. Comparison with related work (extended version of Table 2.1 on page 19). . . . 174

A.1. Overview of notations. 197
A.2. Meaning of the acronyms of the PRS-hierarchy. 200

1 Introduction

Software plays a very big role in business, industrial and and scientific environments. Nowadays, almost no business can be kept running and almost no research can be done without using software. Thus, software development (and software engineering in general) plays an indisputably important role. It supports future trends of many parts of the information society. Permanent development and rigorous research on software is needed to allow the integration of software in our daily life.

Currently, several problems exist during software development. The most important ones are: First, the development is expensive in time and money. Second, the developed software is often contaminated with bugs which result in inappropriate costs and delays the time to market or reduces the customer satisfaction.

One main reason for both problems was already described in 1968 by Bauer [NR69]. The so-called "software crisis" is caused by the problem that the effort for developing software increases rapidly in computer power and the complexity of the problems. Consequently we have to tackle the problem, how a correct, understandable and verifiable computer program can be written [DDH72]. Because of these fundamental problems the special field considering software development – named "software engineering" – was established. This (research) field describes and improves the technologies, processes and methods for requirements analysis, design, construction, testing, maintenance, configuration management, engineering management, engineering, tools and development methods, quality of the software and systems [ABD+04]. The main goal is always to provide a product (application) which ensures requested properties. These could be in any combination: safety, security, price, performance and many more.

One current trend to fight the rising complexity of software is the use of component technologies. The goal is to create software while composing software like parts in the automotive industry. Keywords or visions in this context are "market of components" and "software factory" [Gri93, GS03b]. One topic in component-based software engineering (CBSE) is to ensure the functionality of a component-based software. It is assumed that all parts work as expected, thus an error can be triggered by interactions only. Considering the composability of components is part of current research, as for example it is unacceptable if an externally developed B2B-component crashes, so that the business case cannot be performed. Thus, only with reliable composability it is possible to apply component technologies for all the purposes, where they are needed.

Currently, composition is based on technical properties only (interfaces). Thus, non-functional requirements (e. g., expressed as expected interaction sequences) are not considered. Consequently, the component developer has to ensure that any interaction with a component is handled in an acceptable way. This method needs large effort and will never be error-prone. Moreover, this approach has the disadvantage that a user of a component has only textual definitions of the purpose and expected usage of a component.

In [PTDL07] "composability analysis for replaceability, compatibility, and process conformance" are entitled as one of the "most notable research challenges" for service composition in

the near future.

To tackle this problem, component constraints (named “component protocols”) are introduced describing the non-functional requirements for the usage of a component in a formal way. We will extend the current state of the research, so that the practical applicability is raised.

1.1 Basic Problem

In this work we tackle one fundamental problem of software development:

> *“How can be guaranteed that a developed application works in the way it is expected (e. g., by the customer or user)?”*

Although there is no silver bullet to solve this problem (cf. Turing-powerful programming languages), it is worth to invest in software quality. Otherwise the costs for developing new applications and maintaining existing software will get unacceptably high [Boe81].

During the past decades several steps of evolution were performed to raise the quality of software, decrease costs and time to deployment. Some of these improvements with large impact are object-oriented programming languages, descriptive database languages, development processes like the V-model, architecture principles like n-tier architectures or model-driven architecture, integrated development environments, generated tests and many more. We can observe a strong tendency to develop approaches encapsulating (complicated) formal aspects behind an easier representation.

Currently, the following approach is already widely spread and of increasing importance:

- Component-based Architecture and Service-oriented Architecture on an architectural level, dynamical composed applications from preexisting, encapsulated components,
- text-based technologies on a communication layer (e. g., XML-RPC, SOAP, WSDL, . . .) allowing binding of distributed components, implemented in different programming languages.

These ideas are oriented towards the (well defined) behavior of parts in hardware environments (e. g., personal computer hardware). The goal is to allow more independence while developing functionality (e. g., if the interface requirements are ensured, a customer has not to care about using another graphic card). The parts can be connected without having to care about the constraints of the programming languages, the architecture within the component, and so on. The used component system (cf. Section 3.1.2) takes care of the technical requirements. Besides other aspects, this should improve the reusability and hopefully decrease the costs while developing new applications.

This approach solves some problems (e. g., the technical compatibility of components written in any programming language), but also intensifies some others. A main problem is that the parts used to build an application can be maintained, developed and deployed distributively. This is no side effect, but intended. As a result no implementation exists, that can be evaluated globally, as it was possible in former times.

1.2 Scientific Problem

Several schools of thought already exist considering the problem of reliable composition. Using specifications or abstract behavior of components several properties are considered with respect to composition. Details will be discussed in Chapter 2.

The main flaw of existing approaches is that the real behavior of the components is not considered, the possible component's behavior cannot be captured or the architecture is restricted. This leads to less scopes of applicability, because either the result is uncoupled from the actual implementation or only a small set of implementations can be captured. Hence, there is a need for a precise abstract representation of source code with the aim of using it for the evaluation of properties. We represent this demand by the following question:

> *"Is it possible to capture the abstract behavior of component's implementations closely?"*

This question has to be discussed in a more detailed way. The current state of research is to use finite, only concurrent or only sequential behavior representations. Moreover, this leads to the claim for a capturing of infinite behavior including the fundamental concepts of programming languages. Precisely:

> *"Is it possible to apply a representation that represents parallel as well as sequential behavior while allowing an infinite number of states?"*

Having such a powerful representation is only half the battle won. Many authors use verification techniques to proof demanded properties (comparison to test techniques in Section 1.4). So we will do, too. However, verification is a very difficult task. Often, verification tools and processes need a specific knowledge about the interns of a method. This reduces the applicability in industrial contexts. From our point of view a user of a verification should only formulate the provable question or property (without bothering about the technical details). Therefore, we formulate the question:

> *"Is it possible to establish an automatic verification process?"*

Last but not least, the applicability of an approach will stand or fall with the time requirements an approach will have. This leads to the final question, which should be considered not in a theoretic way, but in a practical way:

> *"It is possible to formulate the problem in such a compact way, that it is still solvable in adequate time?"*

These scientific questions have to be evaluated and answered with respect to the parts that should be put together. In the context of this thesis, these parts are called components or services (cf. Section 3.1.1). Their properties are discussed in the next section.

1.3 Properties of Components

If one intends to deal with applications assembled from components, which are possible in an industrial environment, the following issues have to be respected (in accordance with [SGM02, JC00, HC01, Ver01]):

- Components can be implemented using all kinds of programming paradigms. Thus, parallel behavior (e. g., threads) as well as sequential behavior (e. g., recursion) could be hidden within the behavior of a component (described below). They can communicate by interactions or messages. In general, they do not share global data with other components[1]. Several kinds of interactions are possible, messages could be exchanged synchronously (blocking, cf. Section 3.1.2), asynchronously (non-blocking) or while synchronizing (e. g., in Web Services Business Process Execution Language, cf. Section 5.1.2).
- The main task of component-based software engineering is to put components (cf. Section 3.1.1) together to a larger part. Thus, applications can be implemented by putting parts together. This composition (cf. Section 3.1.2) is the main task during the implementation using components. Components can be implemented using several component systems (cf. Section 3.1.2) and programming languages. Most of the industrial middleware allow signature-based composition[2] only.
- Components are developed context-insensitive. This means, if the technical requirements are fulfilled, a component can be used in completely distinguished applications.
- A component has no global view on the Component-based Architecture. This means that the component just knows the (required and provided) peers within the application. Nevertheless, errors can be triggered by interaction sequences containing several components (callbacks). Thus, an interaction with another component might lead to an interaction with the primary component. This behavior cannot be recognized by a single component [ZS06]. Component developers mainly just know their “own” component, the component they have developed by themselves[3]. Hence, they can describe the supposed functionality of this component.
- A component can behave stateless or stateful:
  - Stateless components provide specific behavior at any time. They can be considered as libraries (as known from operating systems), e. g., for mathematical functions. Thus, every call to such a component is allowed. There exists one condition only: The component has to work in the defined manner.
  - Stateful components act like instances of classes in object-oriented programming. It is possible to manipulate attributes of a component, while using its interfaces.

[1] A database accessible by different components can be considered as global data storage. But it can also be considered as component which other components communicate with.

[2] Only pieces of information about the interfaces of a component are used.

[3] This is even true for object-oriented programming.

- Components are often developed, delivered and deployed by external parties. The source code of a component might not be accessible (blackbox components, cf. Section 3.1.1). Moreover, if dynamic binding of components is allowed, it is not clear which components will be used.

Thus, the two main techniques for checking the properties of (traditional) software – test and verification – are difficult to apply.

1.4 Test vs. Verification

One fundamental concept of checking the functionality of software is testing:

> Testing is the process of executing a program with the intent of finding errors [Mye79, MBTS04].

Tests follow a quite simple method: Values are input and the behavior of the application as well as the output are observed and evaluated.

Tests have the advantage that it is possible to search for errors, like taking a stab in the dark without additional information. Thus, an error has not to be anticipated, one has just to try. E. g., a used null pointer reference can be discovered this way.

Nevertheless, it is not easy to write appropriate tests [Whi00]. In the ideal world testing will check every state of an application and check the computed output values against the expected values. In this case the full program state is evaluated. If no error appears, the application works as defined. However, this ideal process has at least two lacks:

1. The exponential rise of test cases – in both number of methods (permutations) and bit length of the data types – leads to a number of test cases that are practically impossible to check completely.

2. The tester needs an intention, he has to know which results are expected and which errors could appear.

Several approaches are represented to deal with these problems. For component testing the blackbox semantics have to be considered, which is done in [BW96, SK00], for example. The two main goals of research in this area are to deal with the unknown behavior of components and to recognize an acceptable cover of all test cases. But even if approaches work well for single components (often with a given specification), there is still the problem that the components have to be checked within a complete system. At least, after allowing dynamic binding of components the state space explosion is a very hard problem.

Thus, tests cannot be used to prove the absence of errors [Dij69, Dij70, Mye79, MBTS04]. We have to accept errors which are left undiscovered (false positives). However, tests are a good approach to check an implementation for errors, where it is unknown which problems can appear. In contrast, checking if certain (known) constraints are always fulfilled, model checking is the appropriate method [CGP99].

The verification of software behavior was developed to ensure several properties summarized as contracts. Verification gives the developer safety that contracts are (always) fulfilled. If a conservative consideration is preconditioned, then every error can be discovered. This is very important, particularly in a context with dangers for life and limbs. As software pervades every part of nowadays industry and products, verification should be used for critical software systems (e. g., part of nuclear plants, airplanes, medicine, ...). However, also in business critical settings, a need for ensuring of software behavior (e. g., for legal reasons) exists. Thus, a contract can prevent exceptions, ensures business cases, raises safety and security and many more issues.

In contrast to testing, no false positives will appear while model checking a conservative representation. However, constraint violations might be announced that do not appear in the actual implementation (false negatives). It is an important task to reduce the false negatives to an acceptable level.

Furthermore, because a large share of costs in the software life cycle is caused by maintenance (cf. [Boe81]) it is also important to invest in software quality from an early stage of the development process. At least before the deployment of a software, several properties should be ensured (static verification). This is not state of the art, but topic of current research [PTDL07], especially as the currently propagated software architectures (Service-oriented Architecture, Component-based Architecture) focus on the reuse of software components. Especially in these architectures, where programs are composed (dynamically) from components (classes, modules, Web Services, ...), the developer might have no clue what happens inside the imported components or which components are imported. Often only a brief textual documentation is given. This statement might be weakened, if we use whitebox or glassbox components (where the source code is visible and thus can be evaluated, cf. Section 3.1.1), but it is still difficult for a developer to ensure that the usage does not initiate some unexpected side effects, especially in the presence of stateful components.

At least with the rise of Service-oriented Architectures since the millennium, there is a lack of options to check the behavior of software. Service-oriented Architecture and the ancestor Component-based Architecture can and will omit often the behavior of the integrated software components. This blackbox semantics reduces the options to verify the behavior directly, thus a new approach for verification is needed.

For these reasons we will focus in this work on developing a verification approach which is applicable for component-based systems.

1.5 Verification of Component-based Software

The composition of components is relatively simple, actually. If a required interface of a component matches the provided interface of another component (depending on the component system) then they can be bound[4]. This can be done statically (at compile or deploy time) or dynamically

[4] Several work [BR04, YS97, BBC05] is done to allow a binding, even if the interfaces do not match directly. Adapters are used for this purpose.

(at run time using e. g., service repositories). The composition is signature based and does ensure technical properties only (e. g., matching of datatypes).

As mentioned above a constraint is needed for verification. In this work, we consider the behavior of components by their specified task. Thus, we assume that a component is created to fulfill a specific purpose. Examples for this purpose are components which implement:

- A file manager: open, write, close file
- An account manager: registration, login, change data, logout
- An automated teller machine (ATM): insert card, input pin, choose money value, return card
- A time recording system: save arrival time, save idle time, save departure time
- A flight booking system: choose flight, choose class, choose seat, choose meal
- An online shop: create shopping cart, add product, insert voucher code if available, add shipping costs, finish order
- A point of sale: register good, insert zip code if wished, print receipt
- A warehouse manager: check stock, allocate product, send product to customer, reorder product if stock is low

These examples can be extended easily[5]. We can conclude an importance for developing component-based software.

As we can see, there is a strong coherence to the consideration of workflows. They are often represented by Petri nets [Pet73, Pet77, Rei85] (or representations based on Petri nets like workflow-nets [vdA97], colored Petri nets [Jen91] and many more). The research field workflow analysis and verification considers many properties of workflows [AAH98, Aal00, VBvdA01, KMZB02, LL02, DDO08]. Some of these concepts are applicable, but in general the behavior of components cannot be considered like workflows, because of the expressiveness of programming languages.

If the behavior of components can be ensured in respect to the allowed input values (e. g., no null pointer exception, checked using test technologies), then we call them reliable components. If reliable components are assembled to an application, then it is not guaranteed that the application works in the correct manner. Problems could be caused by the interactions of the composed components [NM95].

Motivated by the observation that a composition of reliable components does not always result coercively in a working application, several research groups consider the relevant problems. One solution for this problem is a dynamic verification. There, every (incoming) interaction is checked whether it is allowed in the current state (of the component) or not. If not, the execution is aborted (and a specific exception is thrown). This approach is similar to the concept

[5] They could be applied while using stateless as well as stateful components.

of pre- and postconditions ("Design by Contract" [Mey92b]), which can be checked dynamically, too. A model of the components behavior is used to decide which interaction is permitted currently. Here, this model is named *component protocol* (cf. Definition 4.1 on page 54). This method results in the problem that the user of the application is still confronted with an unexpected exception preventing the execution which was desired primarily by the user. Thus, the allowed behavior is checked during run time. Hence, the deployed component is not checked for reliability.

To check the constraints of an application before the deployment, a static model checking approach is needed. Static verification means the check of the possible behaviors without executing the application. Thus, the application can be checked before the deployment at the user space. Hence, the user will not be confronted with an error triggered by an uncertain composition.

To our knowledge all research groups use an abstracted behavior of the components behavior, currently. This behavior is specified by hand or is given. This can be seen as a remarkable lack, as the behavior of the components has to be the same as the behavior of the specification[6]. The mentioned behavior is represented using a formal approach, like finite state machines, push-down automata, Petri nets and other formalism. Hence, a formal proof of the presence or absence of violations of the given constraint is possible. The chosen formalism has to represent the behavior of the components conservatively. This means that every possible execution trace is contained in the abstraction, too. This property ensures that every error can be found. Moreover, it is worth to discuss, which formalism is capable to deal with the fundamental concept of programming languages: parallelism and recursion[7].

1.6 Representations of Component Behavior

The representation of the behavior is a cornerstone of each approach. A less powerful representation would not be capable to represent the components behavior conservatively with a few restrictions only. The more powerful a representation is, the more complex behavior of real applications can be represented. Thus, it is important to choose an appropriate representation. Some of the well known representations are finite state machines (FSM), push-down automata (PDA) [BS57, Cho57, Sch63, AU72, ABB97], Petri nets [Pet73, Pet77, Rei85] and process algebras like Calculus of Communicating Systems (CCS) [Mil89] or Communicating Sequential Processes (CSP) [BHR84, Hoa04].

Another requirement is the theoretical background of the representation. Since the abstraction has to be model checked in order to prove the wished properties, at least the method for this purpose has to exist. Considering this property, finite state representations have a significant advantage as several model checking techniques exist that are applicable.

However, finite state representation cannot capture the behavior if (unbounded) recursion and (unbounded) parallelism (e. g., unknown number of threads) is present. Thus, they can only be

[6] Ensuring the correspondence of real component behavior and abstracted behavior is hard.

[7] E. g., in [ZS06] it is shown, that the behavior representation has to be capable to deal with context-free languages if recursion is allowed within the components implementation to ensure that all possible errors can be found.

used for applications which do not contain these behaviors. There exists a rule-of-thumb that the more powerful the representation is, the less options are available to perform formal checks. Consequently, the most accurate representation is the source code, which is Turing-powerful (in most programming languages). But a Turing-powerful representation cannot be model checked, as even the *reachability problem* is undecidable [BL74].

1.6.1 Sequential vs. Parallel Behavior

The focus during the extension of the behavior representation goes into two dimensions. First, parallel behavior captures the behavior of at least two processes, which can work independently (cf. Definition 3.14). The well known threads are an example for this concept. Parallel behavior is often represented using Petri nets (e. g., in [VdAvHvdT02], Definition 3.26), because they have a well known theory. However, there exist several other representations which allow to formulate parallel behavior, e. g., process algebras could be used (e. g., used in [Ada06]) and CSP (e. g., used in [AG97]).

In the industrial practice parallel behavior is very important. At least after the spread of multi-core computers, a representation capturing no parallel behavior will be usable for a few scopes of applicability only.

On the other dimension sequential behavior takes place. This describes simply the need for representing dependencies between the execution of tasks. This behavior will lead to recursion – a fundamental concept of computer science – if a process is called in a cycle (in the simplest way by itself). Recursion cannot be represented using finite state machines nor Petri nets nor process algebras (like CCS or CSP), in general. A suitable representation is known as push-down automata (cf. Definition 3.25). This representation disposes of a stack, which can be used to store pieces of information [HU79]. Push-down automata (and relatives) are used for representations of behavior for example in [Reu02b, Sud05, ZS06].

Mayr [May00] states, that the parallel behavior and the sequential behavior are orthogonally. I. e., it is not possible to use a representation describing sequential behavior to describe parallel behavior in general and vice versa.

Parallelism and recursion are two fundamental concepts of programming languages and computer science in general. Hence, they have a high significance for component-based software. Thus, we do not want to resign to represent these concepts within the component behavior.

1.6.2 Specification vs. Source Code Behavior

To our knowledge, other research groups use (human defined) specifications while evaluating the problem of compositionality. From our point of view this might be a problem itself. If a model checking algorithm states, that the considered behavior is obeyed, than it is still possible that an error within the real implementation is not discovered. It is not clear, where the specification is generated. Consequently, a given specification has to be checked, if the component's behavior is captured conservatively. To our knowledge this is not done. These approaches leave it to the developers to ensure that the contract is conservative with respect to the implementation. They

have to understand and evaluate the formal specification of the component behavior and match it against the actual implementation to determine, which situation leads to the discovered problem.

1.7 Problem Definition

"Are components composed correctly?"

We want to develop an approach, which enables to deal with many applications in industrial environments. Thereby, we will focus here on static verification, i. e., verification before the execution of the application.

Derived from properties of components in industrial systems, middleware and applications, we see the request for an approach, which eliminates the lacks of current technologies. The following requirements are derived from the previous discussion and the context of component-based software.

- First we have to answer the question: What means correct composition? Thus, we have to use or define a certain formalism allowing to describe the constraint that has to be model checked. A constraint should be signature-based, because only the interface definition of a component is published.
- The definition of a constraint should only request little knowledge of the user. It would be better, if the component developer has only to care about the main idea of his component. Moreover, we assume that a task sharing might be fruitful, where another person defines a constraint independently. E. g., a software architect or business analyst can define constraints without knowing the implementation.
- The approach should be capable of dealing with many (or all) component systems (cf. Section 3.1.2). Hence, the different advantages and disadvantages should be no obstacle for the applicability. E. g., it should be capable to deal with synchronous and asynchronous interactions as it is allowed in different component systems.
- The approach should be able to deal with the component character (blackbox). Without loss of generality we assume that a single developer can access the source code of a single component only.
- Static as well as dynamic binding should be possible. The fact should be represented that it is not predictable which (matching) component is chosen.
- Less restrictions on the representation of the component behavior are demanded. Ideally, parallel behavior as well as sequential behavior will be possible to capture. Therewith a larger part of possible component behavior is ascertainable leading to a higher practical applicability.
- The model checking approach should result in a qualified answer. Thus, it is possible to determine if and how a situation (violating the constraint) can be reached.

Hence, we search for an approach for static verification ensuring the components are composed correctly. The approach should be capable to deal with as many component systems and behaviors as possible. This will lead to a better practical applicability.

1.8 Approach to a Solution

The main contribution of this work is providing a verification process, that is capable to deal with an expressive representation for component's behavior. Moreover, the characteristics of components and component systems are preserved. It needs an easily to understand input, defined by the user of the verification process. The remaining tasks are performed automatically. In doing so, a representation is created, processed and evaluated, that is capable to represent unbounded parallel behavior and unbounded sequential behavior. If the component's behavior does not match the required properties, then meaningful counterexamples are computed using model checking techniques.

1.8.1 Correct Composition of Components

As mentioned before, model checking requires the definition of a formal property. The application is checked, whether a given property is fulfilled or not.

As Nierstrasz and Meijler [NM95] already have requested, a verification process (of components) should be used to ensure the correct usage of interfaces. To our knowledge Nierstrasz uses the term *protocol* firstly [Nie93, Nie95], while considering the interactions between components (there are mostly objects).

Thus, a protocol P_C of a component C in our sense describes the possible interaction sequences with the component C (cf. Section 4.1). Like other approaches [AG97, Reu02b, FLNT98, PV02] we use finite state machines to represent the constraints.

The protocol of a component can be interpreted as the workflow, the component has implemented (cf. examples in Section 1.5). A component or system designer can define this protocol, as no implementation details are contained. It is assumed that this workflow is checked internally by the component developer. In the case all local checks of the component behavior are passed, the component is called reliable[8]. A composition is called correct, if the given protocols are not violated.

If reliable components can be composed to an application, we will check whether the protocols are obeyed. This is the main focus in this work. The problem is called "protocol conformance" (cf. Section 4.2). If dynamical composition is possible, we only have to find a combination of components fulfilling the requirements and obeying the protocols of the composed components. In this case other components are not conform with the protocols of this combination.

Summarized, we consider the composition of components to ensure reliability and safety of component-based software. Other criteria like liveness and safety are not considered.

[8] In this work, we assume that each available component is reliable.

1.8.2 Representing the Behavior of Components

A big challenge is that parallel as well as sequential behavior should be captured. We use the slightly common and less used representation named Process Rewrite Systems [May98, May00] for this purpose. They represent a unification of the well known push-down automata and Petri nets (cf. Section 3.2.2).

Therefore, we can represent unbounded recursion (like in push-down automata) and unbounded parallelism (like in Petri nets) to represent the behavior (cf. Section 5.1.1). While parallel and sequential behavior can be captured, it is also possible to represent synchronous and asynchronous interactions.

We will represent the behavior of each component individual, by an independent component abstraction. This ensures the practical applicability in a context, where components are developed distributively. Moreover, we separate the contract from the component behavior, thus they could be created independently.

The behavior of the complete component-based software is represented while combining the component abstractions to a system abstraction. In this step the composition directives of the considered component system are taken into account and are imitated.

1.8.3 Ensuring that the Behavior of the Components matches the Specification

Other approaches use a specification to represent the behavior of a component (cf. Chapter 2). It is assumed implicitly that this specification represents the behavior conservatively (“abstract behavior”). From our point of view this method reduces the practical applicability as long as no mechanism is presented for computing the specifications.

For this reason we define a process for computing component abstractions from many programming languages. We use compiler construction technologies for this purpose (cf. Section 4.3). The process works automatically. Thus, the correlation between the implementation and the behavior is ensured. We assume that this method leads to a lower burden, while working with our approach.

Here, we call the components behavior representation “component abstraction”.

1.8.4 Developed Verification Process

To ensure the desired properties the theory of Process Rewrite Systems is extended and adapted to match the requirements of component-based software verification. Several enhancements are performed to ensure the practical applicability.

The verification process represents the connecting method, which allows to fulfill the requested requirements. It is shown in Figure 1.1.

The process is divided into five independent steps.

Step 1: An abstraction is generated for a single component. This abstraction represents the components with all required and provided interfaces as well as the abstract behavior (cf. Section 5.1). Only these abstractions are needed for the next step.

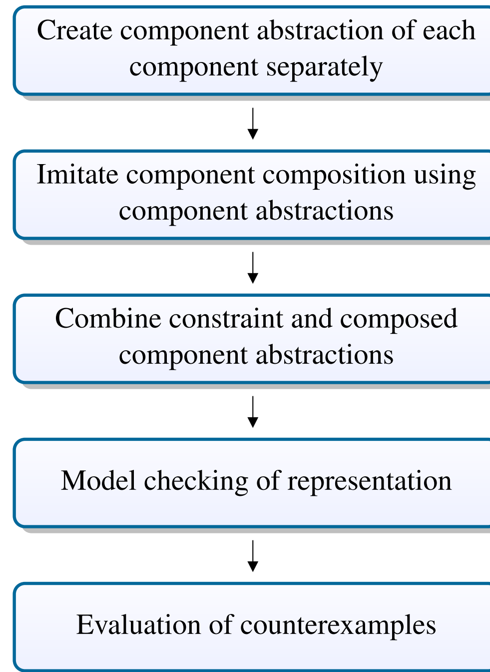


Figure 1.1.: Preparation and verification process.

Step 2: By obeying the directives and technical requirements of the considered component system the single component abstraction is composed. Thereby, the composition of the user application is imitated on the abstraction level (cf. Section 5.2).

Step 3: At this step we generate a representation which is in a form that can be model checked. A protocol is combined with the system abstraction of the considered application (cf. Section 5.3). It results in a formal representation where each derivation from the initial state to the final state might describe a violation of the constraint encapsulated in the protocol.

Step 4: In this step the model checking is performed finally (cf. Section 5.4). The result is either the statement that the components are composed to an application, which ensures the protocol conformance, or execution traces are returned, which describe a protocol violation. Protocol violations are called counterexamples.

Step 5: In the last step the counterexamples are evaluated and represented to the user of the verification process (cf. Section 5.5). He can then evaluate how this problem can appear.

The full process works automatically. Only the component protocol has to be defined by a responsible human. Using this verification process we can ensure that the components are composed correctly. Otherwise a qualified answer is returned, pointing to the concrete situation and participating program points of each component. The definition of correctness ensures that the interactions are performed as expected in any possible execution trace[9].

[9]An error-free implementation of the components is assumed.

1.9 Structure of this Work

This thesis is organized as follows:

The first chapter contains the introduction, motivation and problem definition (you have almost finished reading). In the following chapter we will evaluate the related work.

In the third chapter the foundations and terms of this work are described. This includes our component model (sections 3.1.1, 3.1.2). The actual problem of this work is considered in Chapter 4. Furthermore, we will discuss the limitations of currently used abstractions and how we overcome some of their problems using another representation called Process Rewrite Systems.

Chapter 5 contains the main contribution of this work: the new verification process defined over Process Rewrite Systems. First, a brief overview of the five steps of the verification process is given. Later, each step is discussed in detail separately (sections 5.1, 5.2, 5.3, 5.4, 5.5). In these sections several applications and optimizations are included.

In Chapter 6 we evaluate our verification process in an (semi) industrial case study. For this purpose we developed a component-based software implementing the verification process that is suggest in this thesis. Using this implementation we consider source code from two industrial partners (a powerful program implements functional extensions for SAP® ERP® systems and an online shop), open source application and an academic case study. We will show that the suggested verification process results in an acceptable problem size. Moreover, we discuss the model checking problem and present results of improvements.

In Chapter 7 we extend our approach to application to widely spread iterative software development processes. The thesis finishes with Chapter 8 containing a summary of our approach, a consideration of the contribution and possible future work.

2 Related Work

The task of this work is to develop a mechanism for verifying the composition of components to a component-based software. This contains the main problems: behavior representation and model checking. Several problems are derived from them and the requirements of component systems are used in practice. The following research fields are affected:

- The idea to verify the composability of parts is scope of research for a long time. Before the rise of component technologies, there already existed parts which can be composed to applications. E. g., libraries, modules and objects (e. g., [NTD+95]). Several approaches exist to ensure the composability.
  - Meyer [Mey88] enriches the interfaces of components with additional constraints. The constraints are named pre- and postconditions as well as class invariants. They are used to express the expectations and promises of each partner in a contract (used in the programming language Eiffel [Mey92a]). Some of these constraints are used as extension of the type system and proved statically, others are checked dynamically.
  - Other approaches extend the expressiveness of the type system of a programming language. Without needing an extended definition of a constraint the polymorphic types in functional programming languages are used to raise the level of safety in contrast to imperative programming languages.
- The use of models for representing behavior is a quite common technique for formulating specifications. Examples are the programming languages Haskell [JHA+99, PJHA+99, Jon03] and Metalanguage (ML) [HMM86, MTMH97, Ull98]. Moreover, models allow to define model checking problems and to prove properties (constraints). Well known examples are finite state machines, *abstract state machines* [BS03] or the different models provided by the Unified Modeling Language (UML [RJB04]). In general, the model decides about the expressiveness of an approach. Properties that could not be represented by the chosen model are not provable.
- As mentioned before, components can be developed, sold, and deployed distributively. The person who wants to use a component has only the information, which interfaces the component is implementing and (hopefully) a documentation giving an intention about the components purpose (blackbox concept). This concept has to be represented while verifying component-based software.
- The model checking approach is another main topic. It is a common observation that a model with higher expressiveness has less options for the applicability of model checking technologies. E. g., for the analysis of finite state machines many logics (like LTL, CTL or operations on the derived language) are applicable. Derived from the component implementations possible in the wild, it should be ensured that sequential behavior as well

as parallel behavior can be represented, otherwise the scope of applicability will be reduced strongly. In contrast, the models representing CH-0 languages (Turing-powerful languages) of the Chomsky hierarchy [Cho56, Cho57] have no available model checking technique. Thus, we have to migrate our problem to a representation, where model checking techniques can be applied.

2.1 Classification of Approaches

In the next section, we will describe related approaches. There we present the main properties being important for component-based software verification.

We name the approaches by the model that is used to formulate the behavior (or the constraint). Although not all representations are easy to compare, this allows a raw classification about the expressiveness and the properties of the approach. We assume that a more powerful representation is better, as the component behavior can be represented more precisely. This would raise the practical applicability.

We classify the approaches by the following properties formulated as questions:

I. **Can recursion be modeled using the considered approach?**

Recursion is a main concept of computer science. Thus, it is also important while implementing component-based software. If recursion cannot be modeled the behavior has to be restricted in that way, to prevent false positives. They can appear while a component C_1 calls another one C_2 and then C_2 calls C_1 directly back. This problem of recursive callbacks was described in [ZS06].

II. **Can concurrent behavior be modeled using the considered approach?**

Like recursion parallelism is one of the main concepts of programming languages. Concurrent execution has an indisputable importance in current component-based systems. It is used to raise the performance of applications, while taking advantage of the widely spread multi-core technology and the distributed deployment of components. E. g., implementing threads is possible in many programming languages.

III. **Can synchronous interactions (blocking) be represented?**

Interactions are very important for component-based software. Sequential behavior is used to transfer a message to another component and waits until the calculation of the result is finished. The execution of the callee (sender of the message) is suspended (cf. Definition 3.17 on page 39).

IV. **Can asynchronous interactions (non-blocking) be represented?**

Asynchronous interactions are used for performing an interaction with another component, without suspending the current execution trace. Thus, the calculations of the original component can be performed while the result of the message is computed. In general, there is no option to receive the calculated result directly. The caller has to call back the callee

to deliver the result. If the component system allows callbacks using synchronization (cf. Definition 3.19 on page 39), then the callee can deliver the result to the same execution trace of the caller, that has performed the initial interaction. Moreover, if asynchronous interactions and synchronization by interactions are possible, they can be used together to imitate synchronous interactions if a procedure can be called only once[1].

V. **Is the contract separated from the behavior?**

The contract (constraint) identifies the properties, that should be verified. If the contract is separated from the representation of the behavior, it is possible to exchange the behavior or the component, while the constraint is retained. This could be used representing the situation during the component development, where a component designer decides the rough purpose of the component, while the actual implementation is created later by the component developer.

VI. **Is the generated specification using source code?**

If the specification of the behavior is generated by using the source code, a proveable correlation exists between the implementation and the model. Otherwise, there exists the problem that an answer by the model checker cannot be mapped back to the source code. Moreover, a worse scenario can appear, when the specification captures not all possible behavior of the source code, which could lead to false positives (applications that are classified as correct, but actually are not; cf. Definition 3.34 on page 50).

VII. **Is the complete application behavior taken into account?**

If just local checks are performed, it is possible that errors are not discovered where an indirect interaction (involving at least three components) is performed. In component-based software a single component has no survey about the influence an interaction might have on any other component. The verification should balance this flaw to prevent false positives resulting from indirect interactions.

VIII. **Are correct interactions between two components verifiable?**

If the interaction of just two (or a fixed number of) components can be considered, it is possible to verify the behavior independently. In this case interactions with other (unknown) components are omitted. This might be useful in the common case of component-based software engineering, where components are composed to a larger component (aggregated component), which represents a specific behavior. Aggregated components (cf. Definition 3.9 on page 37) are easier to handle because less constraints have to be evaluated in the next composition step.

IX. **Uses the approach only provided interfaces to define the constraints?**

The provided interfaces (cf. Definition 3.2 on page 29) define the signatures of the procedures that have to be implemented by the considered component. They are part of the

[1] In this case an asynchronous call is followed directly by a synchronization by interaction, waiting for the result.

component specification and defined already in the design phase, in general. Hence, we assume that the definition of a contract should be based on the provided interfaces only. Otherwise, the constraint contains already parts of the implementation, as it defines interactions performed after calling the considered component. But this is a double-edged sword as it can be desired to restrict the implementation by the contract. Moreover, it seems to be easier to define or derive terms for substitutability if the calls to required interfaces are taken into account within the constraints. Thus, we cannot use this property for a rating, it depends on the intention of the approach.

X. **Is it decidable whether a component can be exchanged by another, showing the same behavior?**

Substitutability is the counterpart of composability (which we consider in this work). While composability checks if the considered component matches the constraints of the application, substitutability does evaluate whether a component can be exchanged by another component. For this evaluation the environment has not be known. The restriction is that the behavior is compatible. Thus, the new component should not trigger errors, if the old component does not. The compatibility is fixed on the interactions only.

XI. **Does the approach allow the formulation of non-deterministic constraints or behavior?**

Non-deterministic models are often more compact (e. g., finite state machines) or more powerful (e. g., push-down automata). Thus, it could be gainful if the approach allows to model the constraint or the behavior using non-deterministic rules.

XII. **Does the approach allow more powerful constraints than finite state machines?**

Many works [AG97, Reu02b, FLNT98, PV02, ZS06] use finite state machines as representation of the constraints. Finite state machines are well known models and can be handeled easily. But they can only represent behavior which language is contained in the class CH-3 (of the Chomsky hierarchy [Cho56, Cho57]). This restricts the expressiveness of the constraints. E. g., a constraint demands that as many tokens have to be returned (via interactions) as are reserved (via interactions) cannot be expressed using finite state machines. Thus, it might be gainful to allow more powerful representations.

2.2 Detailed Discussion of Approaches

The classifications of the following approaches are summarized in Table 2.1. There, the sign "**Y**" is used if the approaches fulfill the property. If not, "**N**" is used. If the property is only fulfilled partly, "**O**" is used. Some attributes have no meaning for the considered approach, in this case "**/**" is used.

In the following we will discuss the important properties of different approaches which are related to this work.

approach is using	I. capture recursive behavior	II. capture parallel behavior	III. capture synchronous calls	IV. capture asynchronous calls	V. contract and behavior separated	VI. use source code as behavior	VII. verify complete system behavior	VIII. verify peer to peer behavior	IX. use provided interface, only	X. consider substitutability	XI. allow non-determinismus	XII. more powerful constraints
1. finite state machines [YS94, YS97]	/	N	Y	N	N	N	N	Y	N	Y	N	N
2. CSP^- [AG97]	Y	O	N	O	Y	N	N	Y	N	N	Y	Y
3. counter automata [Reu02b, Reu02a]	O	N	/	/	Y	N	N	Y	N	Y	Y	Y
4. FSM^+ [PV02]	/	Y	O	Y	N	O	N	Y	N	Y	Y	Y
5. workflow nets [VdAvHvdT02]	N	Y	O	Y	N	N	Y	N	N	Y	Y	Y
6. DFSM [SKPR04]	/	Y	Y	N	N	N	/	Y	N	Y	O	Y
7. non-regular protocols [Sud05]	/	N	N	Y	N	N	N	Y	N	Y	N	Y
8. STS^+ [PNPR05b]	/	N	Y	Y	N	O	N	Y	N	N	/	Y
9. context free grammars [ZS03, ZS06]	Y	N	Y	N	Y	Y	Y	N	Y	N	Y	N
10. eLTS [AAA05, AAA06, AAA07]	/	N	Y	Y	N	N	N	Y	N	/	Y	Y

Table 2.1.: Overview of related work.

2.2.1 Behavior Representations

As mentioned before, the chosen representation has a strong influence on the expressiveness of the approach. The research about component composability was influenced strongly by Nierstrasz et al. and his work about the composability of object-oriented programs [Nie93, NM95, Nie95]. The authors state the need for an explicit object interaction representation. They assume that specifications of the observable behavior of objects would answer the purpose best. The protocol contains the interactions which are receivable and also the interactions performed by the object. The protocol is a finite state machine where transitions take place upon communications with other objects [Nie95]. Thus, it is possible to prove that a specification is compatible to another.

Yellin and Strom [YS94, YS97] adapt the approach of Nierstrasz. They also use finite state

machines to represent a specification of the behavior. Three kinds of procedure invocation are allowed: asynchronous (non-blocking), synchronous (blocking) and asynchronous, where the caller waits until the message is passed to the callee. The authors state that a main reason for their approach was the simplicity and the avoidance of a hard or computationally hard problem.

Allen and Garlan [AG97] describe an approach similar to [YS97]. But as representation of the behavioral specification a subset of CSP [Hoa78] is used. Thus, parallel composition is possible. Parallel processes may interact by synchronously engaging interactions.

In [Reu02b, Reu02a] counter-constraint finite state machines are used as behavioral specification. This allows to count the number of specific interactions with the components (cf. Section 2.2.2). Here, just the compatibility of the outgoing and incoming interactions is checked (therefore, no distinction of synchronous and asynchronous interactions is needed).

Plasil and Visnovsky use another extension of finite state machines to represent the behavior specification [PV02], a similar approach is used in other works (e. g., [BHJ09]), too. The representation is called *behavioral protocols*. This extension allows the use of two different parallel operators. The first operator "|" defines the and-parallelism, which results in the shuffle language of both participating processes. The second operator "||" describes the or-parallelism, where either both processes are performed via the and-parallelism or just exactly one of the participants performs its interactions. Moreover, synchronization during interaction is possible. Adamek describes in [Ada06] an extension of this approach for allowing an unbounded number of components. Each component is represented by a finite state machine. But an arbitrary number of components can communicate. The author suggests to allow to create a behavior template at component-design time. After the actual architecture is known (the complete component-based software can be considered), the number of components is chosen based on the level of parallelism in the concrete architecture. Thus, the number of parallel processes is bounded at the verification time. [PP09] is an extension of our work [BZ08b] towards behavior protocols. However, this approach is not capable to deal with (unbounded) recursion.

In [VdAvHvdT02] a subset of Petri nets [Pet73, Pet77, Rei85] named *component nets* (C-nets) is used. C-nets are workflow nets (also defined by van der Aalst et al. [vdA97, vdA98]) where a unique source place and a unique sink place are existing. Moreover, for every node of the C-net it is valid that for every two places a firing sequence exists allowing a connection[2]. Component nets inherit the strong and well known theory from Petri nets. They are very suitable for representing parallel behavior and asynchronous interactions including synchronization. The authors forbid that a component is bounded to more than one required interface. Moreover, a component net has to be well formed, i. e., the call graph has a root and is strong connected. It is not considered whether synchronous interactions can be modeled. As mentioned before, the asynchronous interaction and synchronization can be used to simulate synchronous interactions if a procedure can be called only at one time, which is the case here.

Schmidt et al. use a representation named Dependent Finite State Machines (DFSMs) [SKPR04]. DFSMs allow the formulation of a dependency between processes. Thus, regular languages are

[2]For the sound definition a transition from the source place to the sink place is added.

extended to parallel trace languages [DR95]. They can be interpreted as bounded Petri nets. Thus, parallel behavior – in the sense of shuffle languages – can be represented as specification. Recursion is forbidden (therefore, the complete system has not be considered, as recursive callbacks are impossible).

Südholt [Sud05] uses "*non-regular protocols*" as representation. They are based on "*non-regular process types*" introduced by Puntigam [Pun99]. They specify a set of messages and constraints on acceptable sequences of these messages. The specification can contain nested expressions, which lead to non-regular expressions. Only deterministic specifications are allowed. Asynchronous interactions are allowed only, the components can be synchronized (by coordination). Non-regular protocols are a superset of context-free grammars.

In [PNPR05a, PNPR05b] Symbolic Transition Systems (STS) [CMS02, IL01] are used to represent the specification of a component. The purpose is here to specify the allowed behavior and bind a generated protocol implementation on the actual implementation of the component. Synchronous interactions as well as asynchronous interactions are allowed.

Zimmermann and Schaarschmidt [ZS03, ZS06] use a unique approach. They define no specification, but create a conservative abstraction of the component behavior. This has the advantage that an error contained in the actual source code can be found. The behavioral abstraction of the component and the protocol are separated. Moreover, the authors suggest an approach for dealing with reference parameters, hence (recursive) reference callback can be represented. The abstraction is represented by context-free grammars. Thus, sequential behavior and synchronous procedure calls can be captured.

In [AAA05, AAA06, AAA07] a new component model "Kmelia" is introduced. The specification of the behavior of the components is represented using extended labeled transition systems (eLTS)[3]. There services (not messages) are units of interactions, but the approach is compatible to [PV02] and [Sud05].

Other approaches are predicate-based [LW94, ZW97] describing behavioral protocols with arbitrary complexity. But this results in an uncomputable protocol conformance check.

Review

The different approaches show a clear trend. Starting with the regular expressions Nierstrasz has used, several extensions are suggested to extend the behavior. The aim is to allow a more concrete representation leading to a better practical applicability. Several approaches target parallel behavior (e. g., [VdAvHvdT02, SKPR04, PV02]) to fulfill this main requirement for dealing with components in real component systems. Others introduce sequential behavior to allow a counting of interactions (e. g., [Reu02b, Sud05, ZS06]). But there exists no approach allowing unbounded parallel behavior and unbounded sequential behavior in one model.

We consider the following properties worth mentioning:

- All approaches represent the control flow within the behavior only. This is a common procedure to reduce the possibility of state space explosion.

[3] Extended labeled transition systems are compatible to regular labeled transition system, while using unfolding.

- Many approaches consider only peer to peer interactions [PV02, Reu02b, Nie95]. Only Zimmermann and Schaarschmidt [ZS03, ZS06] allow the composition of the component abstractions to a complete representation of the component-based software.
- Another unique feature of this work is the generation of a conservative abstraction. This eliminates the open question of the other approaches if the real implementation matches the behavioral specification. Thus, the danger of passing an error and accepting an incorrect application as working is not present.

To our knowledge no representation is used for behavioral representation, which is capable to represent unbounded parallelism (like in Petri nets) and unbounded recursion (like push-down automata).

2.2.2 Constraints and Composability

Most of the approaches represent the constraint within the specification (behavioral protocol). Verifying the composability is possible using language inclusion (e. g., for regular languages) [PV02, Reu02b, SKPR04, Ada06, ZS06], reducing the problem to subtyping [Sud05], or checking of representation-depended properties like deadlocks freeness [YS97, VdAvHvdT02]. For example, Schmidt et al. use parametrized contracts [SKPR04]. They are named this way, as the postcondition is parametrized with the precondition of the component and vice versa, thus only the interactions between given components are considered. So, the protocol conformance can be formulated as inclusion problem of regular languages (described by the incoming and outgoing interactions) based on the behavioral protocol.

Although a kind of finite state machines is used, it is possible that a stronger constraint can be formulated (e. g., in [Reu02b] the need for stronger constraints is claimed). For this purpose the author introduces constraints counting the number of interactions, which is similar to context-free languages (but orthogonally). While sequences of the different interactions and the number of these interactions are captured by this constraint, it is possible to define a constraint for a component implementing a stack or queue (where one interaction has to be performed as often as another before). A lack of the approach is, that no distinction between a stack and a queue is possible.

Again a unique feature is represented by Zimmermann and Schaarschmidt [ZS03, ZS06]. They use a separated finite state machine for each component as contract, which is defined over the signatures of the provided interfaces only. The behavior of the component is defined separately as context-free grammars (described in the previous section). Thus, it is possible to generate for each component a statement separately, whether the component constraint is obeyed by the complete component-based software (using the intersection operation of context-free grammars and regular languages). Solely the complete application is taken into account, reference parameters and recursive callbacks leading to protocol violations can be discovered.

Van der Aalst [VdAvHvdT02] also considers the complete application. The representation of the application is checked using deadlock analysis. Thus, the components are composable, if

component interactions cannot result in a deadlock. But the author requests an acyclic callgraph, which weakens the practical applicability.

Review

The constraint is very important. It can be used in a formal way to decide, whether the components are composable in the sense of the contract. Thus, the contract should identify the main property, that should be checked. If the constraint is inappropriate or difficult to implement we will be unable to use an approach in an industrial environment.

Moreover, it has to be balanced between the requirements of the constraint, the understandability and expressiveness. E. g., the behavioral protocols in the sense of [YS97, PV02, SKPR04, Sud05, Ada06] are contained in the component behavior definition. Thus, a change of the implemented behavior of the component has to be checked whether it is still captured by the specification or not. Because the constraint is defined implicitly within the component specification, an adaption can change the protocol, too. A similar approach is used in [VdAvHvdT02]. The constraints are defined as global properties of a composed system (composed from component specifications). They can be validated easier as they are reducible to Petri nets problems (e. g., deadlock freeness of Petri nets used in [VdAvHvdT02]).

The approach of separating component contraints from the component behavior seems to be promising. This ensures that the behavior can be changed as the component implementation is changed. The protocol will not be affected. Moreover, the protocol can be defined for example by a designer, without knowing the implementation. The method is formulated very clear in the approach of Zimmermann and Schaarschmidt [ZS03, ZS06].

On the other hand stronger protocols seem to be promising to define more accurate protocols. The idea of Reussner [Reu02b] (to allow counters) is well suited to capture scopes of applicability, like the representation of stacks, queues and trust management (example in [Sud05]).

2.2.3 Model Checking

While using finite state representations one is enabled to use several model checking techniques, e. g., temporal logics like computation tree logic (CTL), linear temporal logic (LTL), linear time μ-calculus and many more [CGP99]. Model checking for finite state systems is well tool supported, e. g., SPIN [Hol91], PROD [Val92], JavaPathFinder [HP00, BHPV00], and many more.

In contrast, analyzing software applications can often result in a possible infinite state space. If a representation is capable to capture a more expressive behavior (e. g., Petri nets or pushdown automata), the number of model checking possibilities shrinks and the rank of the corresponding complexity classes is raising. For example, LTL model checking of finite state systems is PSPACE-complete, while LTL for Petri nets is EXPSPACE-hard.

The verification of components should contain both important concepts: recursion and parallelism. Chaki et al. described in [CCK$^+$06] a method to verify communicating recursive C programs. This problem seems to be similar to the verification of component-based software, although they considered synchronous procedure calls only. In contrast to the common approach

of component-based software verification they consider even the data manipulation and synchronization statements. The problem can be reduced to the intersection of – by C programs described – context-free languages, which were calculated approximately by a CEGAR-loop. There are other works [QR05, TMP08, LTKR08] which consider the verification of concurrent programs, but these reduce the problem with bounded context switching, which results in a bounded parallelism.

The verification of Process Rewrite Systems (a representation which is capable to represent unbounded parallelism and unbounded recursion) is discussed in [May98, May00]. Only reachability is decidable for Process Rewrite Systems. Moreover, no implementation is available.

Other works discuss the model checking of Process Rewrite Systems. While in [PST07] overapproximations of the execution paths are generated, underapproximations of the reachable configurations are computed in [LTKR07] (bounded model checking). In contrast to these works we focus only on the interpretation of the information included in our representation model.

Review

The model checking problems and logics for finite state representation are well known and well tool supported. Model checking infinite state systems is more complicated (measured in correspondence to the model size). For this reason several works consider improvements to reduce the size of models (e. g., [MT00, YG, Ric08]). Currently, the model checking of infinite-state representations containing parallelism as well as recursion is considered poorly.

The more expressiveness a representation is capable to represent, the less model checking options are available.

2.3 Conclusions

A main criterion for verification of component-based software should be the expressiveness of the model. To represent a component behavior, we need a conservative approximation. Currently, no approach is known, which captures unbounded parallel behavior (e. g., partly in [VdAvHvdT02]) and unbounded recursion (e. g., [ZS06]). If these behaviors cannot be captured the verification might miss an error which is not representable within the model. This case is not acceptable for model checking. Thus, a less powerful representation leads to a smaller scope of applicability, as all implementations have to be excluded, which cannot be captured conservatively.

On the other hand, a concrete behavior representation is needed, otherwise the behavior will not be close enough to the component behavior. This might lead to the problem that a large number of reported errors cannot be reproduced within the source code. This will frustrate the user of the verification process and hence reduce the applicability. The same argument can be used to motivate the correlation between the source code and the modeled behavior.

The criteria shown in Table 2.1 on page 19 represent the main properties. The best approach will fulfill each criterion. Thus, all "good" criteria should be integrated.

The work of Zimmermann and Schaarschmidt [ZS03, ZS06] recognize the need for stronger formalism in comparison with finite state machines. The authors use push-down automata as representation of the behavior. It is generated from the source code. Moreover, the constraint definition is very simple and allows non-determinism.

Thus, we assume that this work provides a good idea, which we will use and extend, so as many criteria will be fulfilled as possible.

3 Foundations

This chapter describes the foundations of this thesis. It describes the used component model (Section 3.1) and discusses the matter of component systems in Section 3.1.2. It contains also a description of the properties components have. To be applicable in an industrial context, they have to be obeyed while developing a new verification approach.

In Section 3.1.3 we describe the behavior, components may have implemented. This contains also the kinds of interactions. Moreover, we discuss the execution model named *cactus stack*.

Formal representations of the component's behavior are represented in Section 3.2. It includes the most important representation for this work "Process Rewrite Systems", with an overview of their hierarchy and model checking issues.

3.1 Components and Component Systems

Components are for composition.

(Clemens Szyperski et al. [SGM02])

3.1.1 Components

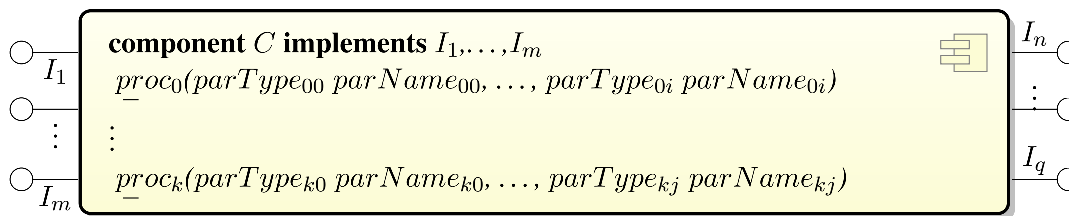
Components are a relatively new programming approach extending existing ones like object-oriented development. Component-oriented programming is often classified as "beyond oo-programming" [SGM02, HC01, JJ01].

The reasons for the introduction of component-oriented programming and component-based software engineering were summarized in [SGM02] (here discussed in a simplified way):

> Object-oriented programming too often concentrates on individual objects, instead of whole collections of objects, i. e., components. Component-oriented programming requires more stringent information hiding, a more dynamic approach, and better safety properties than object-oriented programming.

Thus, the main goals of the component concept are:

- intensify the information hiding, which is often done by introducing a blackbox concept,
- aggregates other programming structures (e. g., objects) and hides them behind a well defined interface,
- allows using components dynamically, often described as import or choose components dynamically during runtime,
- the components should work in the forseen manner, if it can be integrated in a software.



Description:

$proc_x$	represents a callable procedure
$parName_{xy}$	represents a parameter of the procedure $proc_x$
$parType_{xy}$	identifies the type of the parameter $parName_{xy}$, moreover the modifier $inout$ is allowed
I_x	identifies an interface
	where $i, k, j, x, y \in \mathbb{N}$

Figure 3.1.: Component model.

In summary, component-based software engineering (CBSE) treats the process of building software from pieces, while these pieces working together in dynamic combinations (loosly coupling). The expectation is that using components as bricks to assemble an application leads to higher quality, lower costs and better productivity.

Remark

In [Jos08] it is discussed, that components are not often reused in the manner of many users, currently. However, for our research it makes no difference whether a component is reused by one or many more users, although we support the reusability.

Moreover, this software should fulfill the given requirements. But the main focus is on the composition of components (as shown in the introductory quotation).

Analogously to [SGM02], we will describe components as units of deployment, which are reusable assets that can be (and are) sold to different customers. Thus, the main properties of components are:

- can be deployed at distributed locations,
- multiple-use, i. e., components can be used by many other components at the same time, thus the state of a component can be influenced by an unknown number of peers,
- non-context-specific usage, i. e., can be used in any environment providing minimal technical requirements, thus a component has to be implemented obeying that any usage is possible,
- composable with other components (through a well defined interface), i. e., as components can be implemented in completely different programming languages. Any behavior is possible,
- a unit of independent deployment and versioning and encapsulated functionality (i. e., non-investigable through its interfaces), i. e., we have to assume a blackbox behavior,

On a deployment level there exist three general types of components:

- whitebox components: The source code of the component is available, modifiable and executable.
- glassbox components: The source code is accessible, but cannot be modified.
- blackbox components: The source code is invisible. An intensification of this property is represented by components where even the binary code is not accessible (e. g., Web Services).

In this work we assume that all components are blackbox components and the source code might not be accessible. About blackbox components is known only, how they can be called. These callable operations are described using an interface. It contains signatures of procedures. Signatures are defined as usual:

Definition 3.1 (Signature)
A signature *is defined by a pair* sig $= (N, O)$, *where*

$N \in$ *Ident*	*is the name* ,
$O \subseteq$ *Data* $\times$ *Ident*	*is the list of parameters of a signature, where two distinct parameters have to have different names* i, *so that* $\forall (d_k, i_k), (d_m, i_m) \in O\ \forall\, k, m \in \mathbb{N}:$ *if* $k \neq m \Rightarrow i_k \neq i_m$,
	where Ident $\subseteq \Sigma^+$ *and Data is a finite set of datatypes.*

The interfaces subsume signatures.

Definition 3.2 (Interface)
An interface I is a non-empty set of signatures.

Convention 3.1 (Available Interfaces)
The set $\mathbb{I}$ contains all available interfaces.

The signatures of the interfaces have to be implemented by procedures (e. g., Example 3.1). The components in this work are represented using a model similar to the UML component model. On a technical level we assume that a component is an implementation of each provided interface [Szy97]. The provided interfaces are predefined in an interface description (e. g., WSDL). It is possible, that a component uses provided interfaces, thus has required interfaces. Our component model is shown in Figure 3.1. As the required and the provided interfaces are visible, we can determine the signatures of the procedures, that are available.

Definition 3.3 (Component)

Example 3.1: Set of interfaces (implementations in Example 3.2 on page 31).

```
interface I_start
begin
  | sync main()
end

interface I_m
begin
  | sync m(int)
end

interface I_1
begin
  | sync a(int i)
end

interface I_2
begin
  | sync b(int k)
  | sync d()
end

interface I_3
begin
  | sync e(int i)
  | sync f()
end

interface I_5
begin
  | async q(int y)
end

interface I_6
begin
  | sync z(out int r, int x)
end

interface I_7
begin
  | sync set(int n)
  | sync calc(out int r,int m)
  | sync reset()
end
```

A component is a triple $C \hat{=} (\mathbb{P}, \mathbb{R}, \textsc{Imp})$, *where*

$\mathbb{P} \subseteq \mathbb{I}$	*is the finite set of provided interfaces, i. e., interfaces of* C*, that could be called by other components,*
$\mathbb{R} \subseteq \mathbb{I}$	*is the finite set of required interfaces, i. e., interfaces, that could be called by* C*,*
Imp	*the implementation is a set of procedures containing at least the signatures of* $\mathbb{P}$*.*

Remark (Implemented procedures of a component)
The set $\mathbb{P}_C$ *contains all procedures that are defined by the interfaces* I *of the component* $C \hat{=} (\mathbb{P}, \mathbb{R}, \textsc{Imp})$*, where* $I \in \mathbb{P}$.

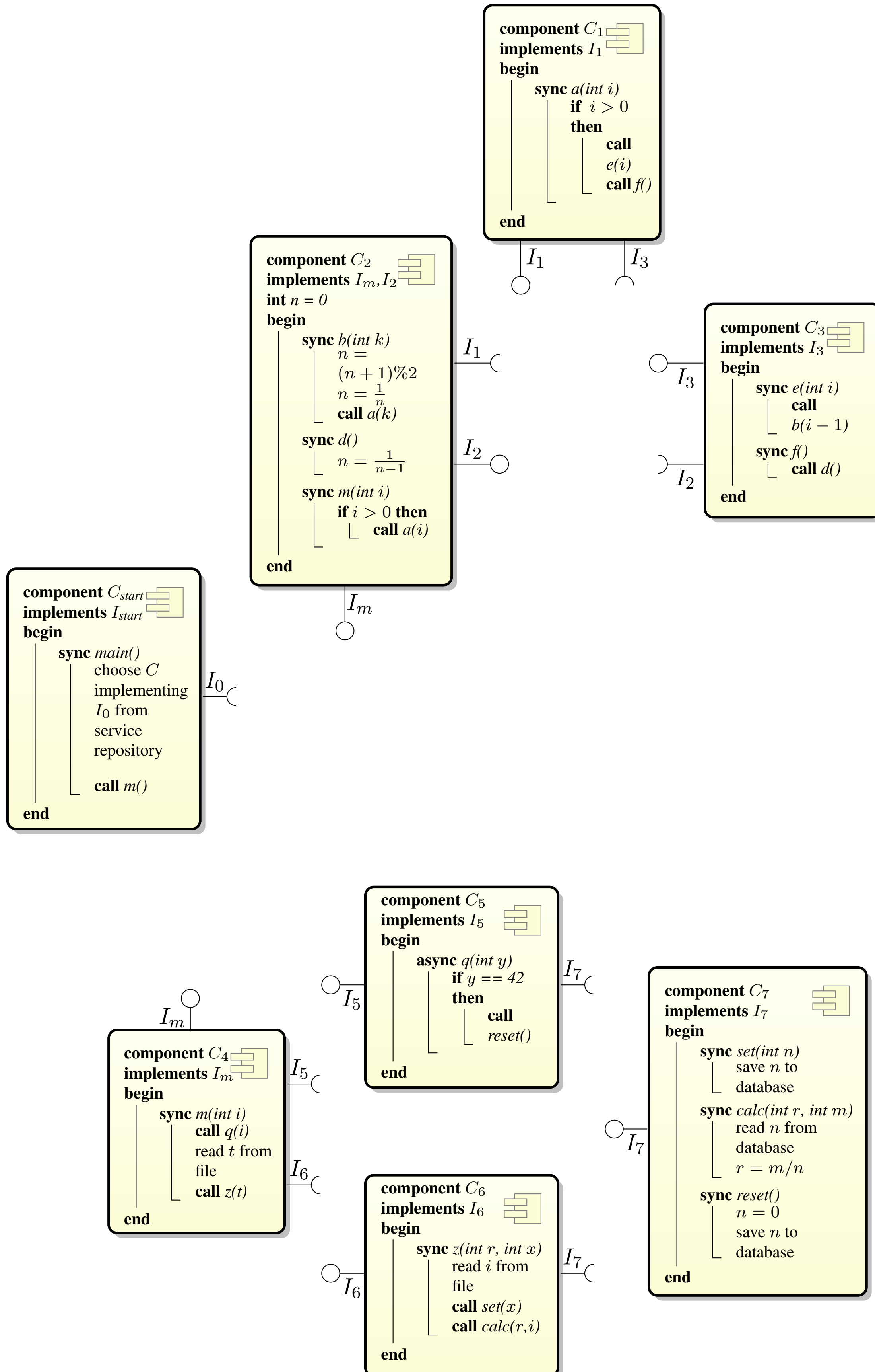
Definition 3.4 (Blackbox component)
Blackbox components can be specified as tuple $C \hat{=} (\mathbb{P}, \mathbb{R})$*, because nothing is known about the implementation (the internal procedures and the implementation of the procedures).*

We make no restrictions on the location where a component is deployed. We assume here, that components may be implemented in any imperative or object-oriented programming language. Of course the implementation could be inaccessible (e. g., Web Service).

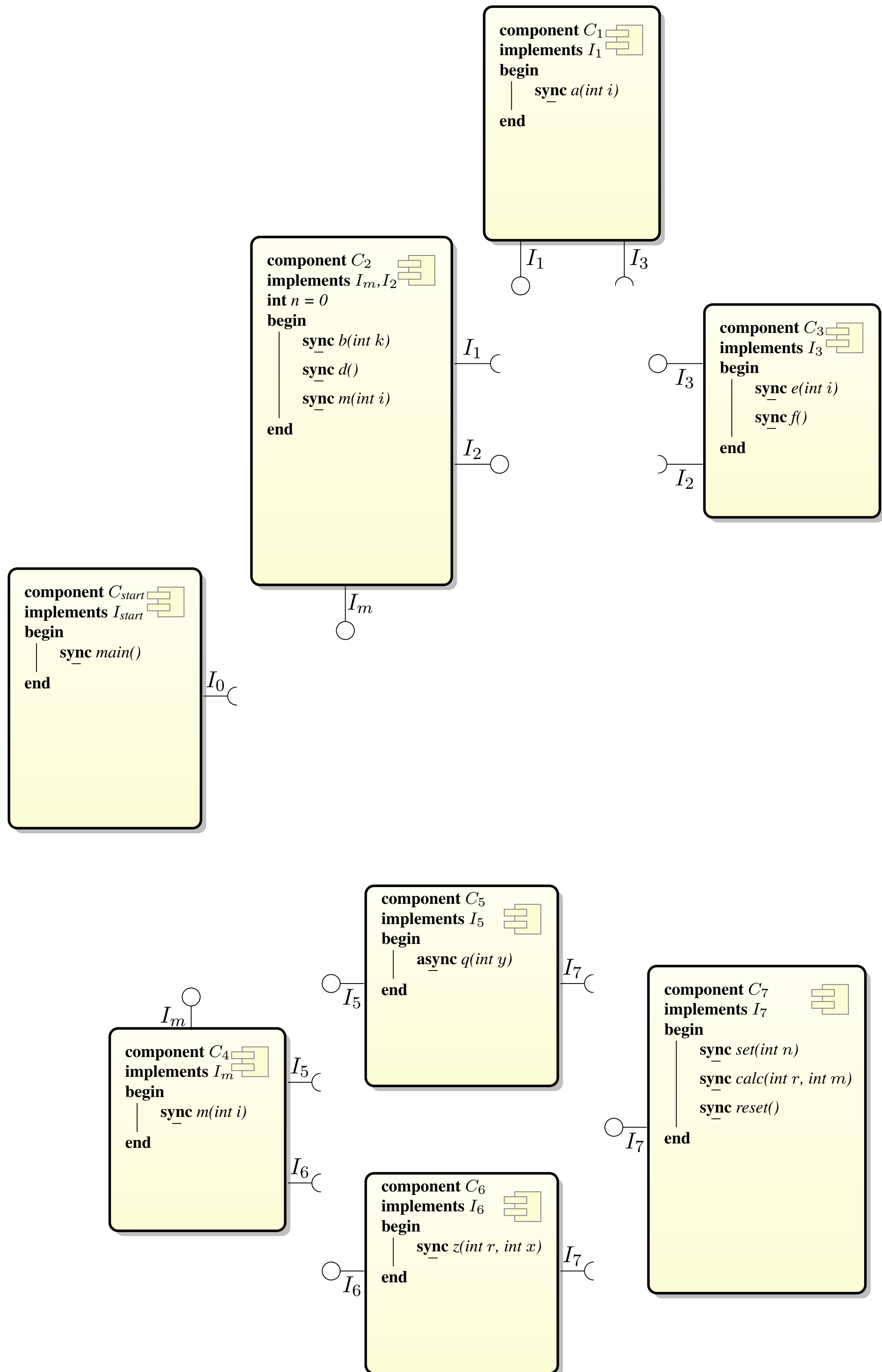
Example 3.2 shows a repository of eight components. In this example each component implements only one interface (Example 3.1). This restriction is made for the purpose of clarity only. There, the implementation is visible, representing the view of each component developer. In Example 3.3 on page 32 the properties of the blackbox semantics are fulfilled, thus only the required and provided interfaces are visible. The application designer comes up with this scenario, while (re)using the components to assemble a new application.

We classify here two kinds of components, that are summarizing specific properties. A com-

Example 3.2: Set of components (corresponding interfaces in Example 3.1 on page 30). We reuse these components partly in the thesis.



Example 3.3: Set of blackbox components (corresponding to Example 3.2).



interface I_{start}
begin
| **sync** *main()*
end

Figure 3.2.: Interface of initial components.

ponent containing an expression identifying an initial program point is called *initial component*. In general, the initial component contains a procedure named “main” or a comparable construct.

Convention 3.2 (Interface of initial component I_{start})
In this work, we request that an initial component has to implement an interface I_{start} containing exactly one signature, named “main” (indecisive number of parameters), e. g., I_{start} (cf. Example 3.1 on page 30).

Definition 3.5 (Initial component)
A component is called initial component C_{start}, if it implements an interface I_{start} (example shown in Figure 3.2). Formally: $C_{\text{start}} \hat{=} (\mathbb{P}, \mathbb{R})$, where $I_{\text{start}} \in \mathbb{P}$. Moreover, we state that the “main” procedure of an initial component is not callable (through other components).

E. g., component C_{start} in Example 3.3 is an initial component.

Remark
As components can implement several interfaces, an initial component can have other provided interfaces than just I_{start}.

Definition 3.6 (Base component)
A base component has no required interfaces. Thus $C_{\text{base}} \hat{=} (\mathbb{P}, \emptyset)$.

Typical base components are configuration, mathematical or output components. Component C_6 in Example 3.3 is a base component.

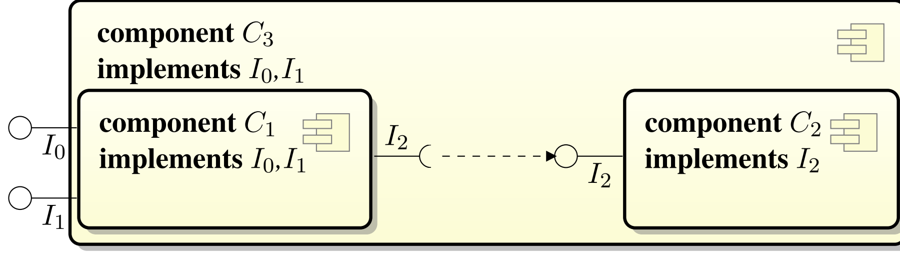
3.1.2 Component Systems and Composition

> A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.
>
> *(Leslie Lamport)*

A component system defines the requirements a component has to fulfill to interact with other components. Thus, it can be used in an application assembled in this component system. These requirements contain technical regulations (e. g., interface definition language), as well as semantics (e. g., synchronization on callbacks).

As a component system defines how a component has to publish its provided (and required)

Example 3.4: Composite component C_3 consisting of the component C_1 and C_2.



interfaces, it can be used to "connect" components. This operation is called "composition". The term stands for the assembly of parts (components) to a component (a "composite") without modifying the parts. Parts have compositional properties if the semantics of the composite can be derived from those of the components. A component C_1 can be composed with a component C_2, if C_1 requires an interface I_2, while C_2 implements the same interface. The result is a composite component. In Example 3.4 a composition is shown resulting in the composite component C_3.

A composition is the connection of two or more components. We represent the composite of components by a graph. The edges of the graph represent the bindings of a graph, where two components can be bound, if the first requires an interface I_1, while the second provides the interface I_1.

Definition 3.7 (Composition)
The composition of components is represented by a graph $G = (\textsc{Comp}, \textsc{Bind})$, where

$\textsc{Comp}$	*is a non-empty set of components* $C_i = (\mathbb{P}_i, \mathbb{R}_i)$,
$\textsc{Bind} \subseteq \textsc{Bind}_{all}$	*are the bindings of a component, where* $\textsc{Bind}_{all} \hat{=} \{(C_1, I, C_2) \mid C_1 \in \textsc{Comp} \wedge C_2 \in \textsc{Comp} \wedge I \in (\mathbb{R}_1 \cap \mathbb{P}_2)\}$.

Remark
The composite needs not to be a runnable application. It is also possible to create an aggregated component (cf. Definition 3.9 on page 37) by composing components. Aggregated components may not be standalone executables. This might have two reasons:

- *No initial component exists within the composition $G = (\textsc{Comp}, \textsc{Bind})$:*

$$\forall C \in \textsc{Comp} : I_{start} \notin \mathbb{P}$$

- *A required interface I (that is not the initial interface I_{start}) was not bound within $G = (\textsc{Comp}, \textsc{Bind})$:*

$$\exists I_q : I_q \notin \{I : (C_1, I, C_2) \in \textsc{Bind}\} \wedge I_q \neq I_{start}$$

Remark

It is possible to bind the required interfaces to the own provided interfaces: $(C, I, C) \in \textsc{Bind}$.

Thus, component systems provide the technical requirements to enable interoperability between components implemented in different programming languages and running on different platforms. One of the main tasks of a component system is to define, how interactions are performed and which kinds of interactions are allowed[1]. Often, they provide a base class library, too. Examples for component systems in an industrial context are Microsoft's COM, DCOM, .NET [DOT], Sun's Enterprise Java Beans (EJB) [BMH06], OMG Corba [COR01], Web Services (WSDL, SOAP) [Web07b, Web07a], and OSGi [All09], while Fractal [BCS03] and SOFA [BHP06] are some of the academic alternatives [LW07]. Moreover, many specialized component systems exist, e. g., Microsoft's Object Linking and Embedding (OLE), Bonobo [Mee01, Fri99] and KParts [KDE]. We will not distinguish between different component systems, but develop a generalized approach.

If components are composed at compile time or deploy time the composition is called *static* (e. g., linking of libraries). Components can also be chosen at run-time. In this case the components only import interfaces. If the set of components implementing the interfaces is finite and known at deploy time we call it a *dynamic composition* (e. g., dynamically class loading). If the set of components implementing the required interface is unknown (e. g., because an unknown service repository is queried) we call the composition *fully dynamic*. The dynamic selection of a component implementing the required interface is done using a (semi-)automatic mechanism (e. g., service repository or UDDI in a Web Service context)[2].

In general, a component C_0 requiring an interface I can be connected to a component C_1 implementing the interface I and afterwards C_0 can perform interactions (remote procedure calls) to C_1.

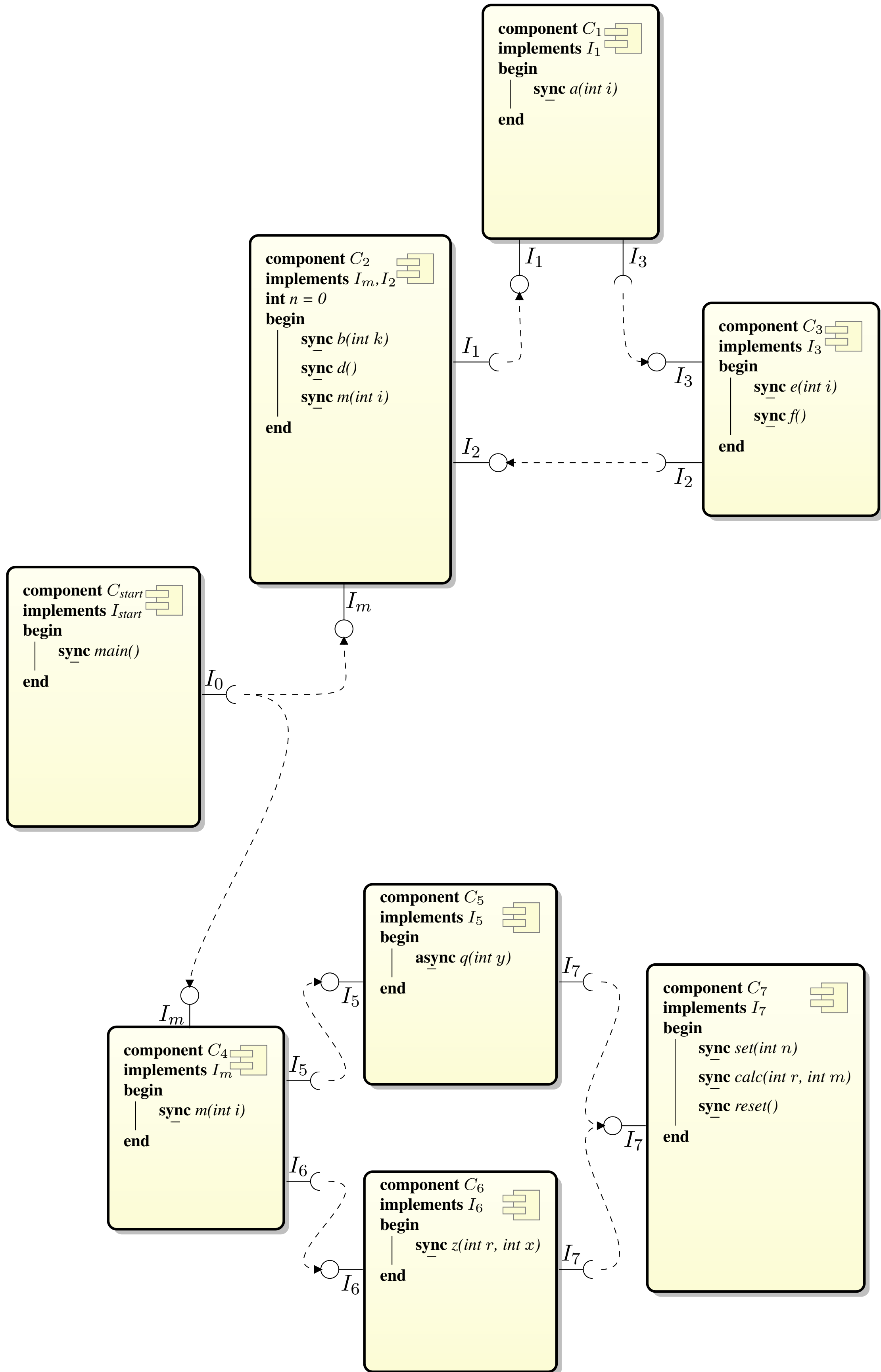
A representation of the composition of the components in Example 3.3 on page 32 is shown in Example 3.5. As component C_{start} requires a component implementing interface I_0, it can either choose C_0 or C_4. The result of this operation is not predictable as the component is querying a service repository (e. g., UDDI) to choose a component (e. g., by the current lower price). In this example the composition is performed dynamically, thus we know that the set of possible components for this operation contains just C_0 and C_4.

We consider two possibilities of interactions: The interaction can be performed in blocking (synchronous) or non-blocking (asynchronous) modes. In this work, we allow that the provided interface definition might contain the information how an interaction is performed. In that case the definition of a remote procedure call is extended with the attribute *sync* for synchronous interaction and *async* for asynchronous interaction. A special case of asynchronous interaction is the synchronization of components via an interaction. In this case a component stops the computation (execution trace) and waits for an incoming message. If the message is received it

[1] For this purpose serves the interface definition language (IDL), too.

[2] It is possible to specify extra-functional properties, which are evaluated during the selection process (e. g., response time or price).

Example 3.5: Composed blackbox components (corresponding to Example 3.2 on page 31).



can be evaluated and the execution is continued.

A composition of components can result in two kinds of results: an executable component-based software or an aggregated component, which is not executable. To formalize these kinds of composition, we define the unbounded required and provided interfaces.

Definition 3.8 (Unbounded interfaces)
The multisets of unbounded required interfaces $\mathbb{R}_{G,\text{unbound}}$ *and unbounded provided interfaces* $\mathbb{P}_{G,\text{unbound}}$ *of a composite* $G = (\textsc{Comp}, \textsc{Bind})$ *are defined as follows:*

$$\begin{aligned}\mathbb{R}_{G,\text{unbound}} =& \{\{q \mid q \in \mathbb{R}_i \wedge C_i = (\mathbb{P}_i, \mathbb{R}_i) \wedge C_i \in \textsc{Comp}\}\} \setminus \{\{q \mid (C_1, q, C_2) \in \textsc{Bind}\}\} \\ \mathbb{P}_{G,\text{unbound}} =& \{\{q \mid q \in \mathbb{P}_i \wedge C_i = (\mathbb{P}_i, \mathbb{R}_i) \wedge C_i \in \textsc{Comp}\}\} \setminus \{\{q \mid (C_1, q, C_2) \in \textsc{Bind}\}\}\end{aligned}$$

An aggregated component (often called composite component) behaves like regular components. It is generated while composing components and binding them tightly.

Definition 3.9 (Aggregated component)
An aggregated component is defined as $C = (\mathbb{P}, \mathbb{R})$ *that is based on a composition* $G = (\textsc{Comp}, \textsc{Bind})$, *where*

$$\begin{aligned}\mathbb{P} &= \mathbb{P}_{G,\text{unbound}}, \\ \mathbb{R} &= \mathbb{R}_{G,\text{unbound}}.\end{aligned}$$

Thus, C *might have no provided interfaces, i. e.,* $\mathbb{P} = \emptyset$, *or no required interfaces* $\mathbb{R} = \emptyset$.

A special case of an aggregated component is a component-based software.

Definition 3.10 (Component-based software)
An application assembled from components is called component-based software S *in this work, where* $S = (\{I_{\mathit{start}}\}, \emptyset)$ *is a component or an aggregated component.*

We do not restrict the component system used to create the component-based software.

Another development are Service-oriented Architectures (SOA) [Erl05, MLM$^+$06]. The term is used firstly in a Gartner research note [SN96, Sch96]. SOA can be seen as continuum of component-based software engineering. For this purpose loose coupling, dynamic binding and discoverability are declared as architectural principles[3]. In this work the difference between component-based software and Service-oriented Architectures are insignificant.

3.1.3 Classification of Component Behavior

If components are blackboxes, we can only observe the interactions with other components. The observable behavior of a component $C \hat{=} (\mathbb{P}, \mathbb{R})$ is defined by the translation of the language

[3]The SOA principles are already fulfilled by CORBA, thus the difference marginalized.

defined over the provided interfaces $L(\mathbb{P}) \subseteq \mathbb{P}^*$ into the language $L(\mathbb{R}) \subseteq \mathbb{R}^*$.

Definition 3.11 (Observable component behavior)

$$\begin{aligned} \chi : L(\mathbb{P}) &\rightarrow L(\mathbb{R}), \ \textit{where} \\ L(\mathbb{P}) &\subseteq \mathbb{P}^*, \\ L(\mathbb{R}) &\subseteq \mathbb{R}^*. \end{aligned}$$

While considering the observable behavior of all components C_i of a component-based software S we can recognize the behavior of the component-based software.

Definition 3.12 (Observable behavior of a component-based software or aggregated component)
A language $L(S) \subseteq (\mathbb{P}_1 \cup \mathbb{P}_2 \cup \ldots \cup \mathbb{P}_n)^*$ describes the observable behavior of a component-based software or aggregated component S, where $C_i \hat{=} (\mathbb{P}_i, \mathbb{R}_i)$ are the contained components.

Definition 3.13 (Execution trace)
A word $w \in L(S)$ (cf. Definition 3.12) is called execution trace or execution path.

If two or more components work concurrently, then they show concurrent or parallel behavior.

Definition 3.14 (Parallel behavior)
Parallel behavior *describes concurrency. Thus, there exists at least two processes (execution traces), which can be processed independently.*

I. e., if an aggregated component $C_3 = (\mathbb{P}_3, \mathbb{R}_1 \cup \mathbb{R}_2)$ (containing two components $C_1 = (\mathbb{P}_1, \mathbb{R}_1)$ and $C_2 = (\mathbb{P}_2, \mathbb{R}_2)$, which are working concurrently) is considered, then it is not clear if the next observable interaction was initiated by C_1 or by C_2, while $L(\mathbb{R}_1 \cup \mathbb{R}_2)$ is fixed, in general.

Here, we restrict the parallel behavior within a component-based software considering the interactions. Parallel behavior (Definition 3.14) where every atomic action is processed completely before the next action is processed, is called *interleaving semantics*.

Definition 3.15 (Interleaving semantics)
An interleaving of two execution traces ρ_0, ρ_1 is a topological ordering of ρ_0 and ρ_1.

Definition 3.16 (Sequential behavior)
We speak of sequential behavior *if an execution (sub-)trace ρ_0 has to be completed before the next (sub-)trace ρ_1 can be processed: $\rho_0 \rho_1$. Thus, concurrency of components or within components is not allowed. Sequential behavior is often represented by a stack (cf. Definition 3.25 on page 43).*

I. e., if no parallel behavior within a component-based software is present while considering

an aggregated component $C_3 = (\mathbb{P}_3, \mathbb{R}_1 \cup \mathbb{R}_2)$ (containing two components $C_1 = (\mathbb{P}_1, \mathbb{R}_1)$ and $C_2 = (\mathbb{P}_2, \mathbb{R}_2)$), then it is clear if the next observable interaction based on a fixed $L(\mathbb{P}_1 \cup \mathbb{P}_2)$ will be initiated by C_1 or by C_2. In general, two kinds of interaction invocations between components exist. That lead to different behavior.

Definition 3.17 (Synchronous interactions)
If a component calls a procedure synchronously, then the component waits until the callee returns the computed result. Until this point in time the calling component stops its current calculation, the execution trace is blocked. For this reason synchronous interactions *are often called* blocking interaction. *The resulting execution trace is $\rho \hat{=} \rho_0' \rho_1 \rho_0''$, where ρ_0' is the execution trace before the blocking interaction, ρ_1 is the execution trace of the callee and ρ_0'' the execution trace after the termination of the callee.*

Definition 3.18 (Asynchronous interactions)
An asynchronous interaction *delivers the message from a caller to a callee without stopping the current execution trace of the calling component (*non-blocking interaction*). Thereafter, the called component and the calling component work concurrently.*

If ρ_0' is the execution trace of the caller before the asynchronous interaction (ρ_0'' is the execution trace of the caller after the interaction) and ρ_1 is the execution trace of the callee, then $\rho \hat{=} \rho_0' . \rho_2$, where ρ_2 is an interleaving of ρ_1 and ρ_0''.

There are different kinds of asynchronous interactions. For example the caller can wait until the message is delivered, but not until the result is computed, or the caller can continue without waiting for anything ("fire and forget"). Thus, we speak of a unidirectional message delivery (e. g., in CORBA marked with the keyword *oneway*).

An interaction a initiated by a component C_2 is called *callback to the component* C_1, if an interaction of C_1 initially triggers the execution of interaction a. It is irrelevant if C_1 calls C_2 directly or indirectly.

Definition 3.19 (Callbacks)
A callback is a cycle in the considered composite $G = (\textsc{Comp}, \textsc{Bind})$.

If the participating interactions are performed synchronously, we talk about a *recursive callback* to component C_1, because this behavior is a recursion over component boundaries.

3.1.4 Cactus Stack

As it was mentioned before, sequential behavior (Definition 3.16) can be represented using a stack. The natural execution model for capturing unbounded recursion and unbounded concurrency uses states that are represented as a *cactus stack*.

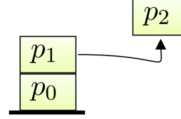
Definition 3.20 (Cactus stack)
A cactus stack (or spaghetti stack) [HD68] is a variant of a stack which can be constructed by attaching a stack as branch at the current top of a stack. Attached stacks are called branches. *The*

top of each branch (stack) can be processed independently. An empty branch can be removed.

A state of program containing parallel processes can be represented using a cactus stack. If a procedure call (in a process) is executed, a stack frame is pushed onto a stack. If a new parallel process (new branch) is started (forked), a new stack grows for this process. Hence, the branches of a cactus stack grow like the branches of a saguaro cactus[4]. Thus, an execution transforms one cactus stack into a cactus stack while performing an action. Hence, a transformation step describes a step of the interleaving semantics. An exemplary process is shown in Example 3.6.

Restriction of Execution Model

Although the execution model of cactus stacks is well known and used for representing parallel behavior, it is not possible to represent a global pool of parallel processes. Forks of a process are still bounded to the parent process. Precisely, the stack frame p_0 placing the frame p_1 that forks the new parallel process p_2 cannot be worked off, before p_1 and p_2 are eliminated. We represent this dependency with the arc (curved arrows) between p_1 and p_2 in the following example:



It is possible to warn the user of the verification process that there might be an unrepresented behavior. This behavior has to be checked in another way. Techniques to uncover situations, where our approach creates no conservative approximation, are not considered here.

3.2 Formal Descriptions of Behaviors

3.2.1 Traditional Representations

A simple and well known formal concept is named finite state machines.

Definition 3.21 (Finite state machine)
A finite state machine A is defined as $A \hat{=} (Q, \Sigma, I, \rightarrow, F)$, where

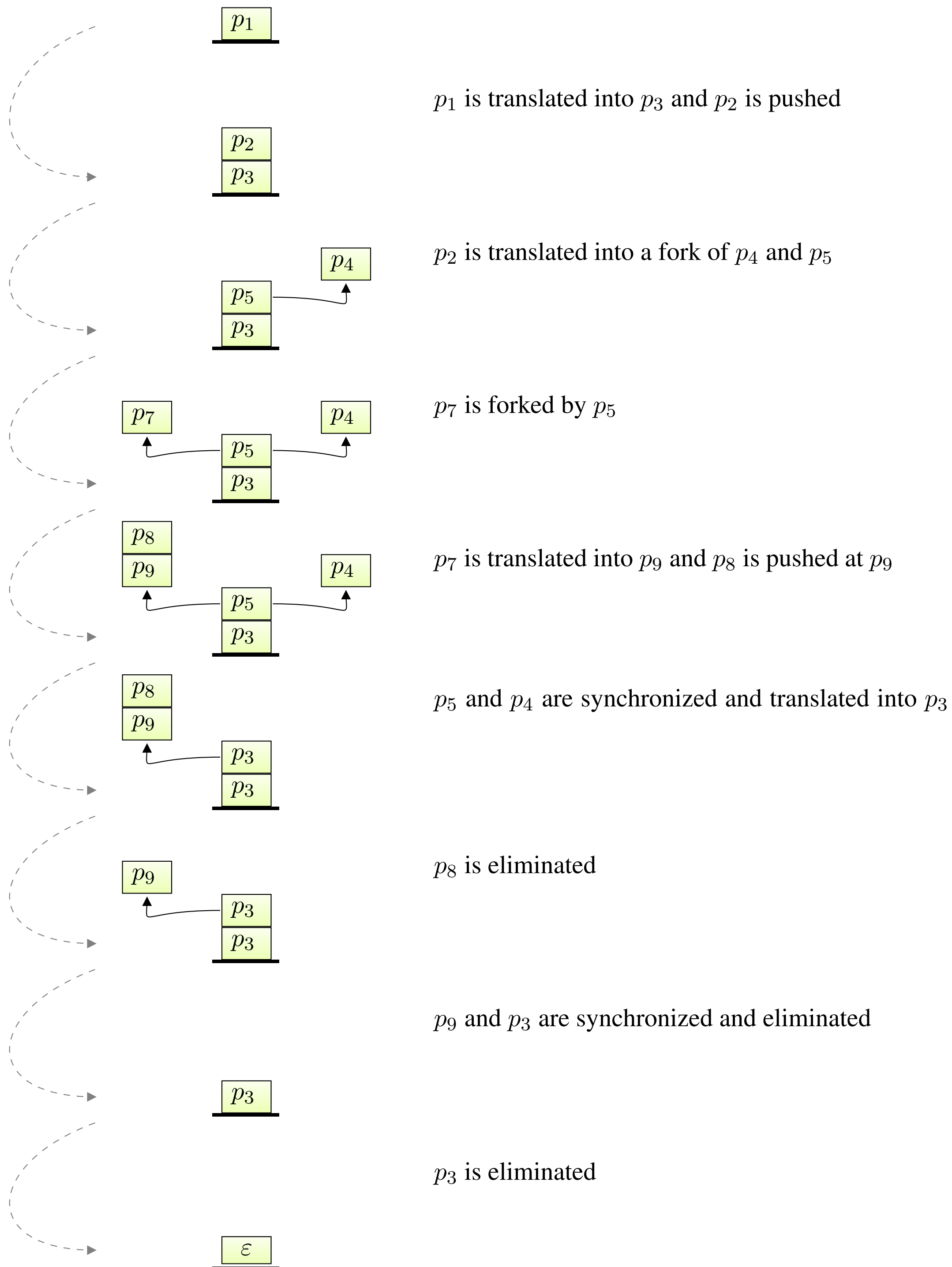
Q	*is a finite set of states,*
Σ	*is a finite set of (atomic) actions,*
$I \in Q$	*is the initial state,*
$\rightarrow \subseteq Q \times \Sigma \times Q$	*is a finite set of transition rules,*
$F \subseteq Q$	*is the set of final states.*

A finite state machine defines a derivation $I \Rightarrow f$ (where $f \in F$) using the following inference rules:

$$\frac{(t_1 \overset{a}{\rightarrow} t_2) \in \rightarrow}{t_1 \overset{a}{\Rightarrow} t_2}, \quad \frac{t_1 \overset{x}{\Rightarrow} t_2 \quad t_2 \overset{a}{\Rightarrow} t_3}{t_1 \overset{x\,a}{\Longrightarrow} t_3},$$

[4]A picture of a saguaro cactus is shown in Figure A.1.

Example 3.6: Growing and shrinking of a cactus stack.



where $t_1, t_2, t_3 \in Q^*$, $a \in \Sigma$ and $x \in \Sigma^*$.

If a finite state machine is used to represent the component behavior, then a derivation $I \overset{w}{\Rightarrow} f$ defines an execution trace w. In [Nie95, PV02, Reu02b] finite state machines are used as representation of the component behavior (cf. Chapter 2).

Definition 3.22 (Regular expression)
A regular expression R is defined as usual. It is a pattern describing every word which is accepted by the regular expression. A regular expression $r \hat{=} r' \maltese r''$ or $r \hat{=} c$, where r' and r'' are also regular expressions, $c \in \Sigma$. The following operators $\maltese$ are allowed:

"$|$" *This operator separates alternatives (or).*

"$*$" *The preceding element of this operator has to appear zero or more times.*

"$+$" *The preceding element of this operator has to appear one or more times.*

Moreover, parentheses are allowed for the purpose of grouping. Grouped elements are considered as atomic.

Definition 3.23 (Regular language)
A regular language $L(A)$ is a language, that can be accepted by a finite state machine $A \hat{=} (Q, \Sigma, I, \rightarrow, F)$. Hence, there exists a derivation within the finite state machine, so that $I \overset{w}{\Rightarrow} f$, where $w \in \Sigma^*$ is the word accepted on the path and $f \in F$, we write $w \in L(A)$. Moreover, a regular expression R can be given, which describes all words contained in the language $L(R)$. Thus, $L(A) = L(R)$.

Regular languages cannot describe parallel or sequential behavior in a general case. E. g., it is only possible to represent the behavior of a limited number of parallel execution traces explicitly.

Definition 3.24 (Inverted finite state machine)
For an inverted finite state machine $\overline{A} \hat{=} (\overline{Q}, \Sigma, \overline{\rightarrow}, I, \overline{F})$ of a finite state machine $A \hat{=} (Q, \Sigma, I, \rightarrow, F)$ it is valid that if and only if $w \in L(A)$, then $w \notin L(\overline{A})$, where $w \in \Sigma^*$.

Remark (Inverted regular language)
The definition follows directly from Definition 3.24 and 3.23.

To represent sequential behavior (Definition 3.16), a stack [BS57] is needed. This can be modeled using push-down automata.

Definition 3.25 (Push-down automata (with one state))
Here, we focus on push-down automata with one state only. Every language acceptable by a push-down automaton with one state can be accepted by a push-down automaton with several

states [HU79] and vice versa. A push-down automaton P is defined as $P \hat{=} (Q, \Sigma, \rightarrow, I)$, where

Q	*is a finite set of stack symbols,*
Σ	*is a finite set of (atomic) actions,*
$\rightarrow \subseteq Q^* \times \Sigma \times Q^*$	*is a finite set of transition rules,*
$I \in Q$	*is the stack symbol available at the stack initially.*

The following inference rules are valid (similar to Definition 3.21):

$$\frac{(t_1 \xrightarrow{a} t_2) \in \rightarrow}{t_1 \overset{a}{\Rightarrow} t_2}, \quad \frac{t_1 \overset{x}{\Rightarrow} t_2 \;\; t_2 \overset{a}{\Rightarrow} t_3}{t_1 \overset{x\,a}{\Longrightarrow} t_3}$$

where $t_1, t_2, t_3 \in Q^*$, $a \in \Sigma$ and $x \in \Sigma^*$.

A given input is evaluated stating at I. A token of the input queue can be removed if a corresponding transition rule is used (matching action and current state). If no stack symbol is held by the stack and the input is read completely, then the input is accepted.

Push-down automata can describe behavior containing sequential component behavior in general. Thus, they are capable to represent unbounded recursion (including recursive call-backs). In contrast, push-down automata cannot represent concurrent behavior in a general case [May98]. For example, push-down automata are used as representation of component behavior in [ZS06] (cf. Chapter 2). A well known approach to represent parallel behavior are Petri nets ([Pet73, Rei85]).

Definition 3.26 (Petri nets)
The classical Petri nets are bipartite graphs consisting of two types of nodes: "places" and "transitions". Places contain "token", while transitions are used to translate tokens.

A Petri net N is defined as $N \hat{=} (Q, \Sigma, \rightarrow, I)$, where

Q	*is a finite set of places or working symbols,*
Σ	*is a finite set of (labeled) transitions, no name $\varepsilon \in \Sigma$, for simplicity we demand that the following formula is valid: $Q \cap \Sigma = \emptyset$,*
$\rightarrow \subseteq (Q \times \Sigma) \cup (\Sigma \times Q)$	*is a finite multi set of arcs (flow relation),*
$I \subseteq Q^*$	*finite multiset, called "initial marking".*

The input can be considered as a multiset of working symbols, which is represented by the initial marking. The current multiset of working symbols (tokens) is named "marking" M. A transition $a \in \Sigma$ can "fire" if it is valid $\{\{q | (q, a) \in \rightarrow\}\} \subseteq M$. After a transition a has fired, the (resulting) marking M' is represented by $M' \hat{=} (M \setminus \{\{q | (q, a) \in \rightarrow\}\}) \cup \{\{q | (a, q) \in \rightarrow\}\}$.

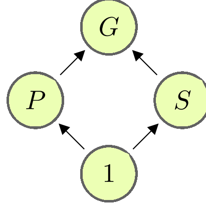
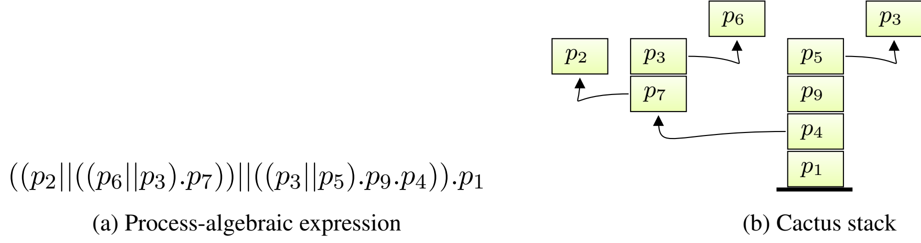


Figure 3.3.: Hierarchy of basic PRS operators.

Example 3.7: Process-algebraic expression and corresponding cactus stack.



In this work, we use a slightly different definition $N \hat{=} (Q, \Sigma, \rightarrow, I)$, where

$$\rightarrow \subseteq (Q^+ \times \Sigma \times Q^*) \quad \text{is a finite set of transition rules,}$$
$$I \in Q \quad \text{is an initial input symbol (working symbol).}$$

Transition rules eliminating a working symbol are represented by $t \xrightarrow{a} \varepsilon$, where $t \in Q \times Q^*$ and $a \in \Sigma$. A word is accepted, if the current marking contains no token: $M = \emptyset$. The language accepted by a Petri net P is defined as $L(P) \subseteq \Sigma^*$, thus the order of the fired transitions leading to an empty marking is represented. It is clear, that both definitions can accept the same languages.

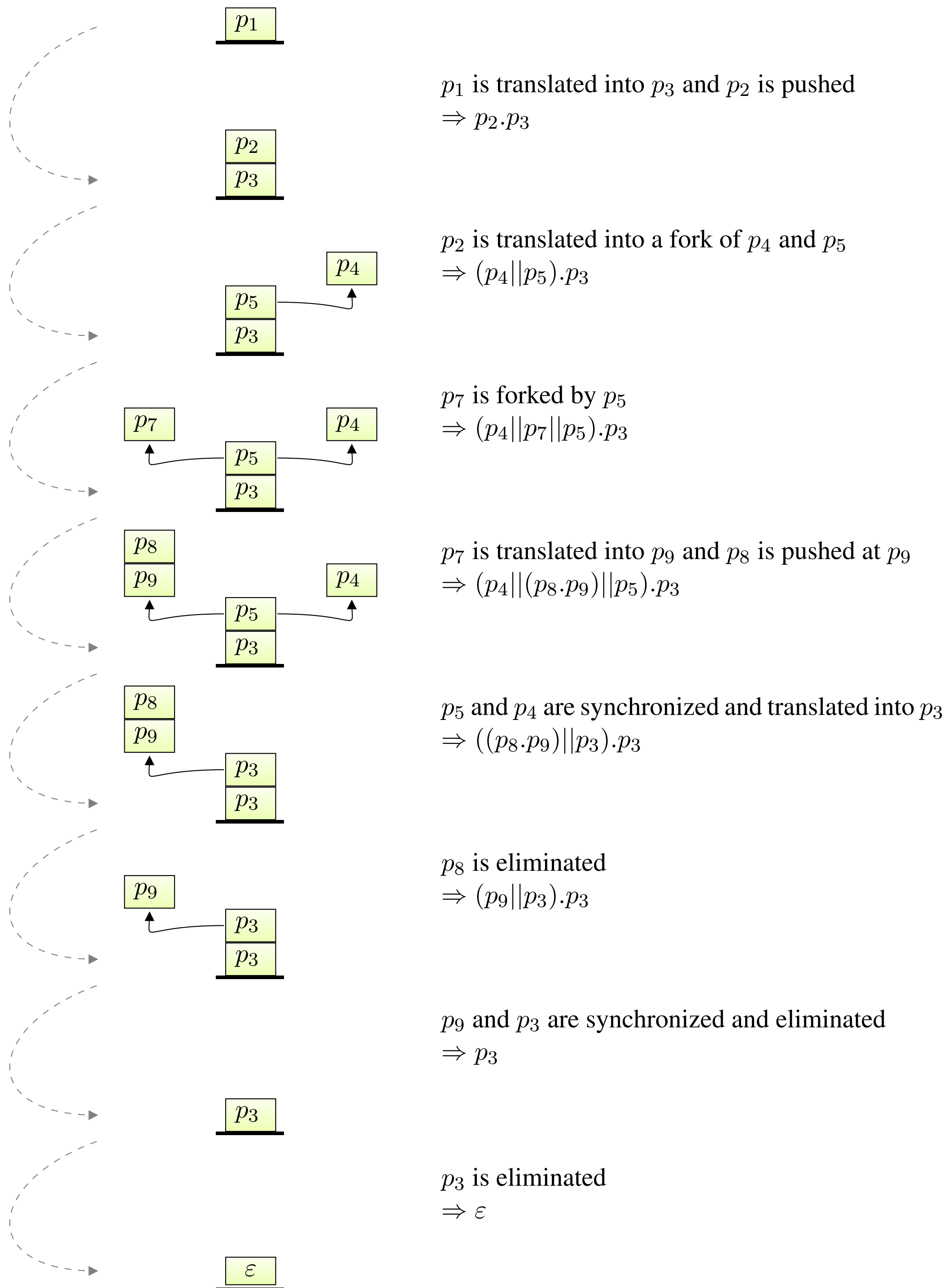
Petri nets are capable to represent unbounded concurrent behavior. However, recursion cannot be captured in general [May00]. For example, in [VdAvHvdT02] a subset of Petri nets is used as representation of component behavior (cf. Chapter 2).

3.2.2 Process Rewrite Systems (PRS)

As mentioned above, both representations (Petri nets and push-down automata) have advantages. We consider a representation which contains parallel semantics (like Petri nets) as well as sequential semantics (like push-down automata). Therefore, in this work a representation called *Process Rewrite Systems* (short PRS) is used, to represent the behavior (abstraction) of source code. Process Rewrite Systems are defined by Mayr [May98, May00].

A cactus stack naturally represents a process-algebraic expression (and vice versa), cf. Example 3.7. If a procedure frame p_1 is called on a stack p, this is represented by $p_1.p$. If a fork of a process p_2 on a stack p with the frame p_1 happens, this is represented by $(p_1||p_2).p$. Thus,

Example 3.8: Growing of cactus stack and corresponding process-algebraic expressions (corresponding to Example 3.6 on page 41).



we can model state transformations by transforming process-algebraic expressions into process-algebraic expressions. Example 3.6 is extended by process-algebraic expressions in Example 3.8.

Definition 3.27 (Process-algebraic expressions)
Let $PEX(Q)$ be the set of process-algebraic expressions over a finite set Q of atomic constants and the binary operators "." (associative) and "$||$" (commutative, associative). Formally, we define:

$$PEX(Q) = \{t : t \in Q\} \cup \{t_1 \otimes t_2 : t_1, t_2 \in PEX(Q) \wedge \otimes \in \{., ||\}\}$$

Process Rewrite Systems are a descriptive technique for such transformations of process-algebraic expressions into process-algebraic expressions.

Definition 3.28 (Process Rewrite Systems)
A Process Rewrite System *is defined as* $\Pi \hat{=} (Q, \Sigma, I, \rightarrow, F)$*, where*

$$\begin{aligned} Q &\text{ is a finite set of atomic processes,} \\ \Sigma &\text{ is a finite alphabet over actions, in other words calls} \\ &\text{ to interfaces of components,} \\ I \in Q &\text{ is the initial process,} \\ \rightarrow \subseteq PEX(Q) \times \Sigma \times PEX(Q) &\text{ is a set of process rewrite rules,} \\ F \subseteq PEX(Q) &\text{ is a finite set of final processes.} \end{aligned}$$

Thus, Process Rewrite Systems are capable to capture component behavior containing unbounded recursion (like push-down automata) and unbounded parallelism (like Petri nets). Example 3.9 on page 48 shows transformations of process-algebraic expressions.

Definition 3.29 (Empty process)
The process ε denotes the empty process. *It is the identity on "$||$" and the identity on ".".*

Remark
Thus, it is valid $\varepsilon.t = t = t.\varepsilon$ and $\varepsilon||t = t = t||\varepsilon$, where $t \in PEX(Q)$.

Definition 3.30 (Derivation relation of Process Rewrite Systems)
The process rewrite rules define a derivation relation $\Rightarrow \in PEX(Q) \times \Sigma^* \times PEX(Q)$ by the following inference rules:

$$\frac{(t_1 \xrightarrow{a} t_2) \in \rightarrow}{t_1 \xRightarrow{a} t_2}, \quad \frac{t_1 \xRightarrow{a} t_2}{t_1.t_3 \xRightarrow{a} t_2.t_3}, \quad \frac{t_1 \xRightarrow{a} t_2}{t_1||t_3 \xRightarrow{a} t_2||t_3}, \quad \frac{t_1 \xRightarrow{a} t_2}{t_3||t_1 \xRightarrow{a} t_3||t_2}, \quad \frac{t_1 \xRightarrow{x} t_2 \;\; t_2 \xRightarrow{a} t_3}{t_1 \xRightarrow{x\,a} t_3}, \quad \frac{}{t_1 \xRightarrow{\lambda} t_1}$$

where $t_1, t_2, t_3 \in PEX(Q)$, $a \in \Sigma$ and $x \in \Sigma^*$.

The second rule formalizes that only the frames on top of the stacks of a cactus stack can be considered for transformations. The third and fourth rule specify that any stack in a cactus stack can be considered (i. e., each of the processes currently being executed can be selected for state transformations). Thus, these two rules model interleaving semantics.

3.2.3 Hierarchy of Formal Representations

Based on these operators, Mayr defined a hierarchy of formal representations containing finite state machines, Petri nets and push-down automata. The hierarchy is called *PRS-hierarchy*. These PRS classes allow the classification of Process Rewrite Systems by the appearance of operands. He uses the following base classes:

- 1: terms are composed of atomic processes only
- P: terms are composed of atomic processes or parallel composition
- S: terms are composed of atomic processes or sequential composition
- G: terms can be formed with all operators

These classes model different behavior [May00]. Hence, it is not possible to model all behavior of a parallel system using only sequential composition and vice versa (cf. Figure 3.3 on page 44). With the four base classes, a strict hierarchy based on bisimulation was formed (cf. Figure 3.4 on page 49), which allows us to classify all possible and sensible Process Rewrite Systems.[5] The meaning of the abbreviations is described in Table A.2.

The hierarchy classifies the PRS classes by the appearance of the operators in the transition rules. The classes are named by the (L, R)-PRS schema, where L classifies the allowed operators on the left-hand side of the transition rules, while R classifies the allowed operators on the right-hand side of the transition rules, where $L, R \in \{1, P, S, G\}$.

As we see the $(1, S)$-PRS allows rules which contain an atomic process at the left-hand side and allows the sequential operator at the right-hand side. This class is equivalent to push-down automata with one state, which accept a language, if the stack is empty. The empty stack is represented in a $(1, S)$-PRS with the empty process ε. The (S, S)-PRS is the companion piece to PDA with several states.

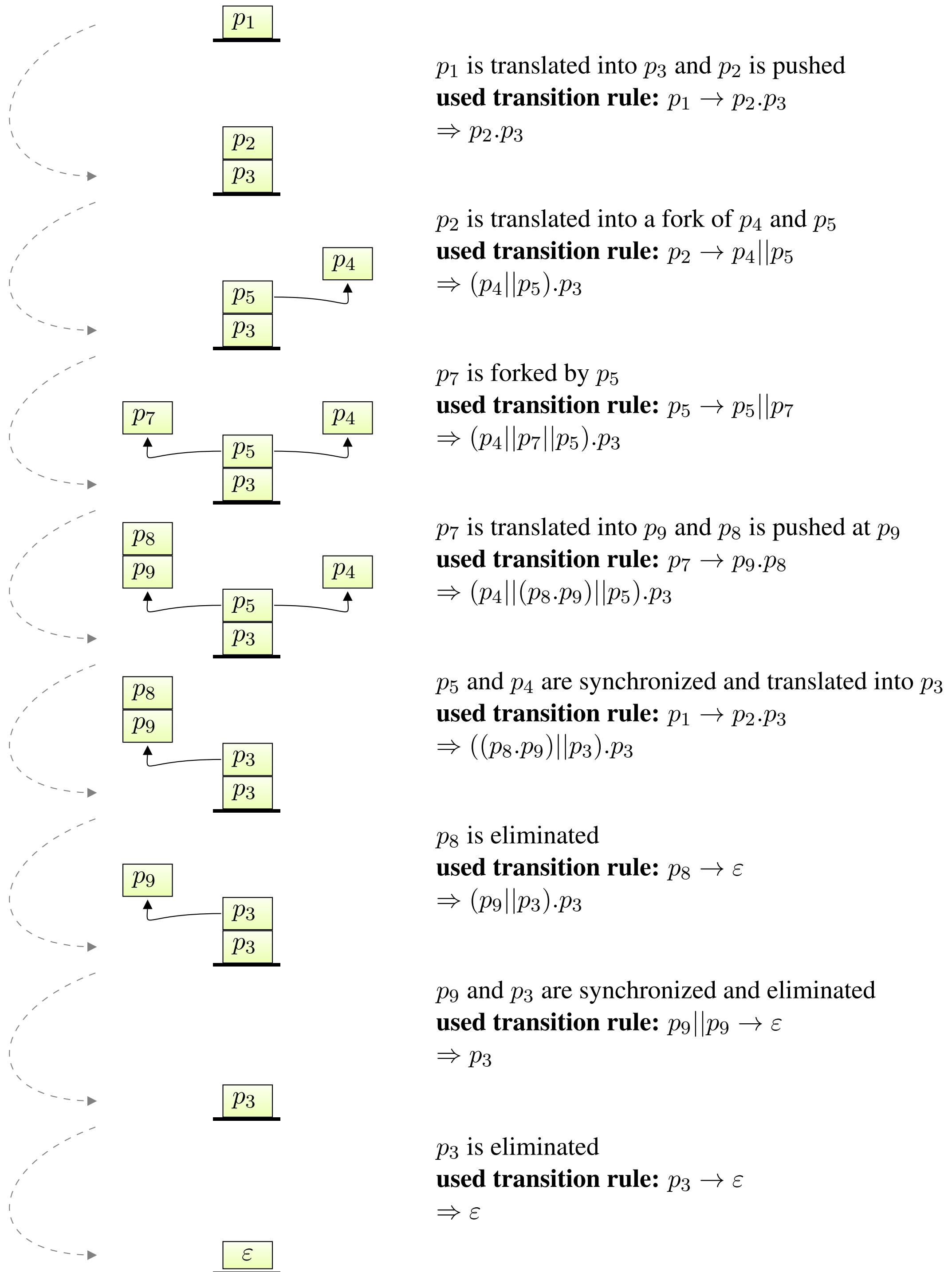
Process Rewrite Systems allowing the parallel operator are among others the class (P, P)-PRS. This class is equivalent to the Petri nets. The $(1, P)$-PRS are Petri nets, where every transition of the net has only one incoming arc, hence it does not contain synchronization.

Even the general Process Rewrite Systems are not Turing-powerful.

Mayr shows in [May00] that the rules of any Process Rewrite System can be transformed to a *normal form*, i. e., the left-hand side and the right-hand side have one of the forms t_1, $t_1.t_2$ or $t_1||t_2$, where t_1 and t_2 are atomic processes, formally:

[5][May00] points out that the left-hand side of a process rewrite rule must be at most as large as the right-hand side in the sense of Figure 3.4.

Example 3.9: Transformations of process-algebraic expressions using a given set of transition rules (corresponding to Example 3.8 on page 45).



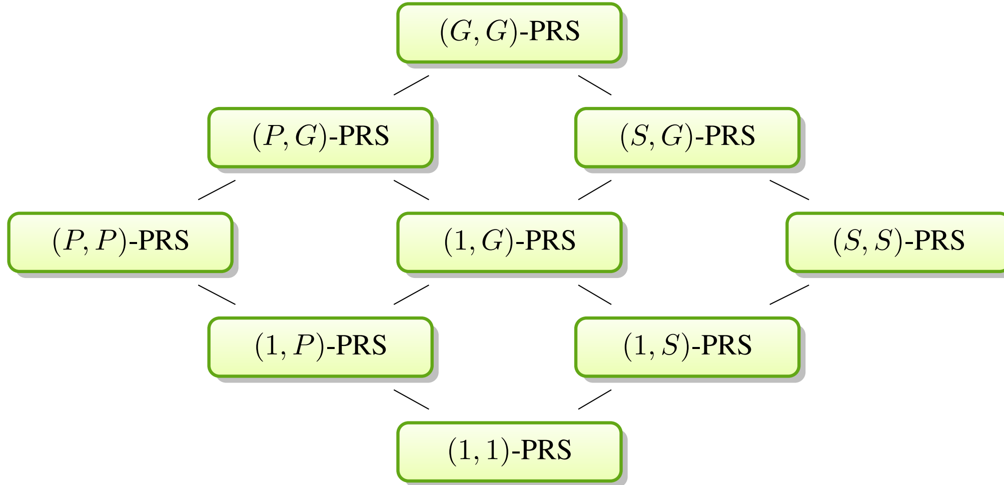


Figure 3.4.: Hierarchy of Process Rewrite Systems (cf. [May00]), classification by appearance of operands on the left-hand side and right-hand side, (LHS,RHS)-PRS.

Definition 3.31 (Normalform of Process Rewrite Systems [May00])
A Process Rewrite System $\Pi = (Q, \Sigma, I, \rightarrow, F)$ *is in normal form, if all transition rules* $\delta \in \rightarrow$ *are in normal form.*
A transition rule is in normal form, if it is in one of the following forms:

$p_1 \xrightarrow{a} p_2.p_3$	*called sequential rule,*
$p_1 \xrightarrow{a} p_2 \vert\vert p_3$	*called fork rule,*
$p_1.p_2 \xrightarrow{a} p_3$	*no name given,*
$p_1 \vert\vert p_2 \xrightarrow{a} p_3$	*called join rule,*
$p_1 \xrightarrow{a} p_2$	*called chain rule,*
$p_1 \xrightarrow{a} \varepsilon$	*called elimination rule.*

where $p_1, p_2, p_3 \in Q$ *and* $a \in \Sigma$.

Convention 3.3 (PRS always in normal form)
We assume in this work that every Process Rewrite System has been transformed to a Process Rewrite System in normal form.

3.2.4 Model Checking

Definition 3.32 (Model checking)
Given a model Π *(e. g., push-down automaton or Petri net) and a property a model checker decides whether the property is fulfilled by* Π *or not. Every part of* Π *which does not fulfill the given property is called* counterexample.

Not every model checker works precisely [CGJ+03].

logic	highest class, where the logic is decidable
linear-time μ-calculus	(P, P)-PRS "Petri nets"
alternation-free modal μ-calculus	(S, S)-PRS "push-down automata"
LTL	(P, P)-PRS "Petri nets", (S, S)-PRS "push-down automata"
CTL	(S, S)-PRS "push-down automata"
EF (branching-time temporal logic)	(S, G)-PRS "PAD"
Reachability	(G, G)-PRS "general PRS"

Table 3.1.: Model checking results considering Process Rewrite Systems.

Definition 3.33 (False negatives)
A false negative w is a counterexample of a model Π, where $w \notin L(\Pi)$.

False negatives appear often. They have to be checked by a human, in general. In contrast *false positives* are not acceptable:

Definition 3.34 (False positives)
A false positive is a violation of a property (i. e., counterexample), which is not discovered.

Hence, the application is accepted as fulfilling the requirements while it does not. For this reason false positives have to be excluded. A sound model checker will prevent false positives.

Model Checking of Process Rewrite Systems

Mayr has shown that the general PRS class can be model checked because reachability is decidable (not Turing-powerful) [May98]. Here, the reachability state STATE (cf. Definition 3.35) is a process-algebraic expression.

Definition 3.35 (Reachability)
Given a formal representation Π, it is decidable if a STATE *s of Π is computable. The definition of this* STATE *differs from representation to representation. E. g.,*

- *a state $q \in Q$ of a finite state machine (cf. Definition 3.21) is equal to the* STATE*,*
- *a marking is the* STATE *of a Petri net (cf. Definition 3.26),*
- *the stack content is the* STATE *of a push-down automaton with one state (cf. Definition 3.25).*

Model checking several logics is always decidable for a subset of PRS classes. A brief overview of Mayr's results is given in Table 3.1. An extended overview is shown in Figure A.2, A.3 and A.4.

3.3 Summary

In this work we will consider blackbox components. The only known attributes of them are name, required interfaces and provided interfaces.

An important issue is composition. Components are composed while matching their interfaces. This is a technical requirement. Hence, the composition is based on technical requirements.

Errors triggered just by component interactions cannot be excluded. They appear during the test phase or after the deployment (i. e., runtime, during usage by client). Delivering an application, which might contain serious problems, is not acceptable for some scopes of application. The problem might be triggered by a reused component as well as just by the composition of components. Using tests to find these errors is difficult, at least if components are allowed to choose peers dynamically. Moreover, tests can never be done exhaustively, in general.

4 Protocol Conformance and Abstractions

In this chapter we will describe and define the problem we will solve in this work. An important property of our approach is the distinction between the constraint of a component and the behavior of the component-based software. We start with a formal definition of component protocols describing the constraint which has to be fulfilled (Section 4.1). In contrast to other works (e. g., [PV02]) considering (behavioral) protocols (cf. Chapter 2), only the contract is described by the protocol. It contains only the callable operations. The behavior of the component is presented separately (cf. Section 4.3).

In Section 4.2 we define the protocol conformance problem, while Section 4.3 is concerned with, source code abstractions can be used to satisfy the requirements of the independent component developers and the properties of the problem.

4.1 Component Protocols (short: Protocol)

Simplicity is prerequisite for reliability.

(Edsger W. Dijkstra)

As discussed in Section 1.4 it is a difficult, expensive and error-prone task to ensure the correct functionality by testing only. We have in mind, that the developer should not be burdened of thinking about possible error-prone situations within the source code. Instead, he should formulate the purpose of the considered component. We assume that this is an easier task.

A *component protocol* (short: *protocol*) describes the valid use of all callable operations (cf. interfaces, Definition 3.2 on page 29) of a single component or instance[1], respectively. It can be used to dynamically verify (incoming remote) invocations, and also to statically verify, if the component is always used in the manner specified by the protocol. Protocols are used e. g., to avoid uncaught exceptions (e. g., avoid situations, where a division by zero is possible) during the execution of a component or obey business rules (e. g., ensure a specific workflow while interacting).

In accordance with other works (e. g., [ZS06, Reu02b, FLNT98]) we use finite state machines (short: FSM, cf. Definition 3.21) to represent the protocol P_C of a component C.

Definition 4.1 (Component protocol)
A protocol P_C of a component $C \hat{=} (\mathbb{P}, \mathbb{R})$ is a finite state machine (FSM).
The finite state machine $P_C \hat{=} (Q_{P_C}, \Sigma_{P_C}, I_{P_C}, \rightarrow_{P_C}, F_{P_C})$ is defined as usual (cf. Definition

[1] We use the term component for simplification.

3.21 on page 42), i. e.,

$$\begin{aligned} Q_{P_C} &\text{ is a finite set of states,} \\ \Sigma_{P_C} &\text{ is a finite set of atomic actions (interactions of the components), thus it is a subset of the provided interfaces of } C, \\ I_{P_C} \in Q_{P_C} &\text{ is the initial state,} \\ \rightarrow_{P_C} \subseteq Q_{P_C} \times \Sigma_{P_C} \times Q_{P_C} &\text{ is a finite set of transition rules,} \\ F_{P_C} \subseteq Q_{P_C} &\text{ is the set of final states.} \end{aligned}$$

For example, a SSO-component[2] with the following actions is offered:

a. register a new account and sign in instantly,

b. sign in using an existing account,

c. optionally change password,

d. logout.

The SSO-component (Example 4.1b) could have the following business rules to ensure a safe behavior. An unexpected situation should never be reached:

- users can log in if they already have an account or they might register for a new account (and thus log in immediately),
- after logged in, they can change their password (as often as desired),
- the connection is closed after the user is logged out.

This business rule describes some kind of a workflow within the component. It has to be obeyed by every caller of the component in any execution sequence. In Example 4.1 the protocol of the SSO-component mentioned above is shown in its different representations (cf. Examples 4.1c, 4.1d, 4.1e). As the protocol can only contain atomic actions provided by the implemented interface, the components interface is also shown (Example 4.1a).

Protocols forbid specific interaction sequences. Thus, they ensure that the component is only used in the manner the component developer has prepared. Protocols can be used to ensure a distinguished behavior. For example:

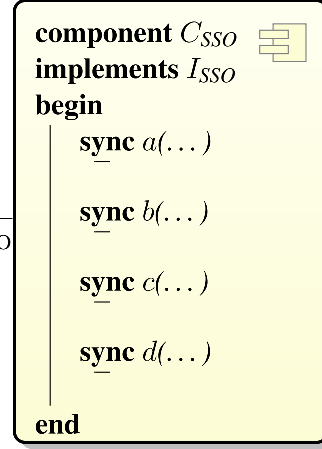
- ensure that a user of a component does not use the component in an unsafe manner (e. g., Example 4.1 describes such a restriction, in Example A.1 on page 199 a protocol derived from an industrial case study is shown),

[2] Single Sign On: A component which provides the functionality of a login/logout/session management, so different applications can use this mechanism to verify a user.

Example 4.1: SSO-component and corresponding protocol.

```
interface I_SSO begin
    /* register and sign in             */
    sync a(...)
    /* sign in                          */
    sync b(...)
    /* optionally change password       */
    sync c(...)
    /* user logout                      */
    sync d(...)
end
```

(a) SSO-Interface I_{SSO}.



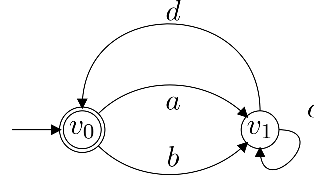
(b) SSO-component C_{SSO}.

$$\mathcal{P}_{\mathrm{SSO}} \hat{=} ((a|b)c^*d)^*$$

(c) Protocol of C_{SSO} as regular expression.

$$\begin{aligned}
\mathcal{P}_{\mathrm{SSO}} =& (Q, \Sigma, I, \rightarrow, F) \text{ with} \\
Q =& \{v_0, v_1\}, \\
\Sigma =& \{a, b, c, d\}, \\
I =& v_0, \\
\rightarrow =& \{v_0 \xrightarrow{a} v_1, v_0 \xrightarrow{b} v_1, v_1 \xrightarrow{c} v_1, v_1 \xrightarrow{d} v_0\}, \\
F =& \{v_0\}.
\end{aligned}$$

(d) Protocol of C_{SSO} as finite state machine.



(e) Graphical representation of protocol C_{SSO}.

- ensure that a specific interaction sequence can never appear because otherwise the application will crash, examples are shown [ZS06, BZ08b, BZ09e],
- ensure that a legal constraint or a business rule is obeyed (examples are shown in [BZ08a, BZ09b]).

The component developers or system designers should not be burdened with the internal view of a component's behavior. Therefore, we choose a component protocol representation, that contains only callable procedures of the component. Moreover, this allows the definition of a protocol without knowing the actual implementation. Besides that it is also possible to exchange the component's implementation while keeping the protocol the same. Thus, the requirements for an integration in a design or development process are fulfilled.

Remark

In some discussions the concept of protocols is compared to the concept of "design by contract" [Mey92b], which is lifted to the level of components. We would add to this statement that a protocol can be seen as a published workflow (sequence of operations) a corresponding component implements. However, this leads to the restriction that data flow is omitted. For this reasons,

this approach is not applicable to all kind of verification issues, e. g., the examples in [MK99] are mostly not verifiable while using this approach.

4.2 Protocol Conformance

n the last section we discussed the constraints to be obeyed if a component is used within a component-based software. This protocol is defined for each component, in general. Thus, the main problem is to decide, whether all given protocols are obeyed within the behavior of the component-based software. Hence, we consider the behavior of the full application. Here, it is assumed that the behavior of the application S is given by the Process Rewrite System Π_S (cf. Definition 4.7). It describes a language $L(\Pi_S)$ which contains all possible interaction sequences within S.

While verifying one protocol $P_C = (Q_{P_C}, \Sigma_{P_C}, I_{P_C}, \rightarrow_{P_C}, F_{P_C})$, we consider only the interactions $a \in \Sigma_{P_C}$. We define the behavior $\Pi_{S,C}$ which contains only interactions, that are part of the protocol P_C.

Definition 4.2 (Use of a component C in S: $\Pi_{S,C}$)
The use of a component C in a system S assembled from components is the set of all possible sequences of procedure invocations to C. Thus, this can be also modeled as a language $L(\Pi_{S,C})$. The behavior $\Pi_{S,C}$ describes the same behavior as Π_S, except that all interactions performed not to C are considered as no relevant action. Hence, $\Pi_{S,C}$ contains only symbols identifying operations of the provided interfaces of C. All other operations are masked. We denote $\Pi_{S,C} \hat{=} \varphi_C(\Pi_S)$. Thus, φ computes $\Pi_{S,C}$ of Π_S considering the operations provided by C.

The use of the component C_2 in Example 3.2 on page 31 is shown schematically in Example 4.2 using UML Sequence Diagrams [RJB04]. As it is shown in the second diagram, the interactions between other components are still present, but not named (represented as λ). Only interactions to the currently considered component C_2 are represented in an identifiable way. In Example 4.2b Π_{S,C_2} is visualized. It contains only operations of the provided interfaces of C_2.

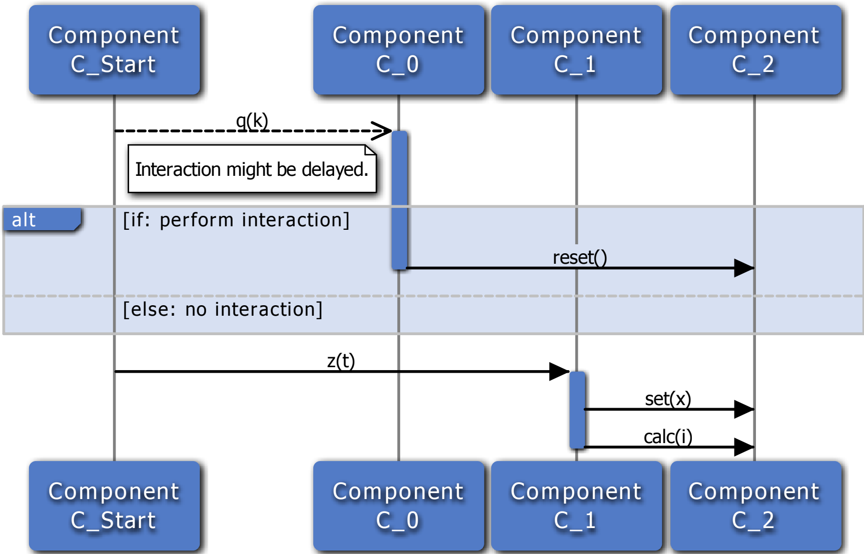
Now, it is possible to define the protocol conformance. To decide conservatively if an application S is conform, we have to check every given protocol $\mathcal{P}_{C_i}$. The protocol conformance problem is the question, whether the protocol P_C of one component C is obeyed in the component-based software S.

Definition 4.3 (Protocol conformance)
The protocol conformance describes the relation between the interaction sequences that are contained in $\Pi_{S,C}$ and the allowed sequences of interactions:

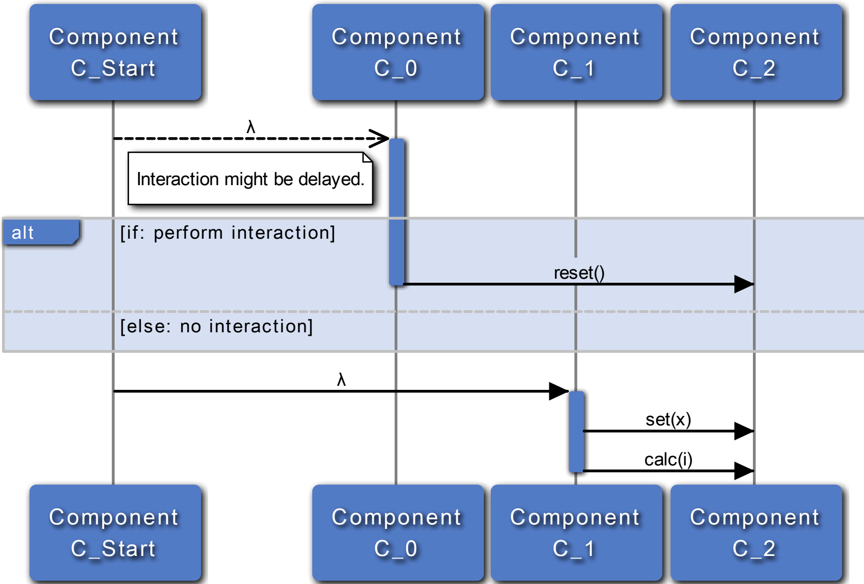
$$L(\Pi_{S,C}) \subseteq L(P_C)$$

As generalization we define the protocol conformance of a complete application.

Example 4.2: Sequence diagrams (drawn using [Han07]) showing Π_{S,C_2}, use of component C_2.



(a) Specification of application interactions (corresponding to Π_S).



(b) Considering interactions with C_2 only (corresponding to Π_{S,C_2}).

Definition 4.4 (Protocol conformance of a component-based software)
A complete application S is protocol conformant, *iff for each protocol $\mathcal{P}_{C_i}$ of S it is valid:*

$$L(\Pi_{S,C_i}) \subseteq L(\mathcal{P}_{C_i})$$

The statement $L(\Pi_{S,C_i}) \subseteq L(\mathcal{P}_{C_i})$ can be formulated equivalently by $L(\Pi_{S,C_i}) \cap L(\overline{\mathcal{P}_{C_i}}) = \emptyset$, where $\overline{\mathcal{P}_{C_i}}$ is the inverted protocol (inverted finite state machine, Definition 3.24 on page 42). Hence, we check if any sequence of interactions to C_i can be generated within S, that are forbidden by the protocol $\mathcal{P}_{C_i}$ (thus are contained in $\overline{\mathcal{P}_{C_i}}$). If $L(\Pi_{S,C_i}) \cap L(\overline{\mathcal{P}_{C_i}}) \neq \emptyset$ the application S contains a *protocol violation*.

As a result we are interested in the concrete sequences of interactions leading to a protocol violation. These sequences are called counterexamples. A counterexample w is a sequence of interactions that is not allowed by protocol P_C of a component, but nevertheless possible by the behavior $\Pi_{S,C}$ of the application S.

Definition 4.5 (Counterexample)
A counterexample in our sense is an interaction sequence $w \in L(\Pi_{S,C_i}) \cap L(\overline{\mathcal{P}_{C_i}})$.

Remark (Static vs. dynamic evaluation of the protocol)
Protocols can be used to verify the behavior of a component-based software dynamically *(i. e., at runtime). I. e., some mediator checks whether the next interaction is allowed and stops the interaction sequence if not.*

The static verification *of a protocol checks the behavior before the application is started (e. g., at composition time). In this case no actual interaction is performed and the behavior is evaluated without execution. In this work we consider static verification, because we are strongly interested in discovering errors before they appear during the usage by a customer.*

The protocol conformance problem describes the statement whether the rules (a component developer has established) are obeyed in the application or not. If not, we would like to deliver counterexamples which represent the execution trace leading to the forbidden interaction sequence.

In Example 4.3 an example of glassbox components is shown. There exists a problem within component C_2. If the procedure *calc* is called directly after calling procedure *reset*, a division by zero error will occur. The developer of component C_2 has foreseen this problem and defined a protocol for his component (Examples 4.3c, 4.3d, 4.3e). This protocol prescribes that after calling *reset* the procedure *set* has to be called, thereafter *calc* can be called (in Figure A.2 on page 202 a protocol of C_2 is shown, which is more specific). Looking at the available source code, we can discover such a situation. The variable y has to be equal to 42 (in component C_0).

The problem appears, if the execution of the asynchronous procedure call *q(k)* in C_{start} is delayed and interleaved with the interaction sequence *set(t)* and *calc(x)* initiated by C_1. The

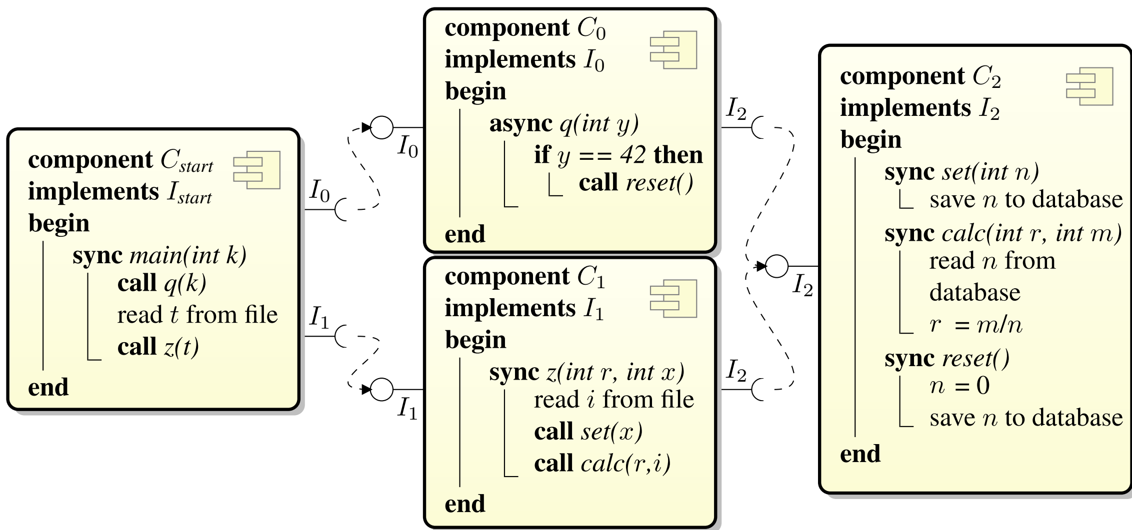
Example 4.3: Glassbox components, where the execution results in an error (cf. Example 4.4 on page 60).

interface I_0 **begin**
| **async** *q(int k)*
end

interface I_1 **begin**
| **sync** *z(***out** *int r, int x)*
end

interface I_2 **begin**
| **sync** *set(int n)*
| **sync** *calc(***out** *int r, int m)*
| **sync** *reset()*
end

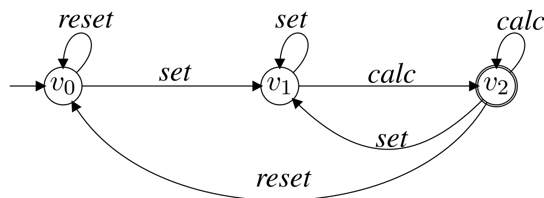
(a) Given interfaces.



(b) Given components, composed statically.

$\mathcal{P}_{C_2} \hat{=} (reset^{*}\ set^{+}\ calc^{+})^{+}$

(c) Protocol as regular expression.

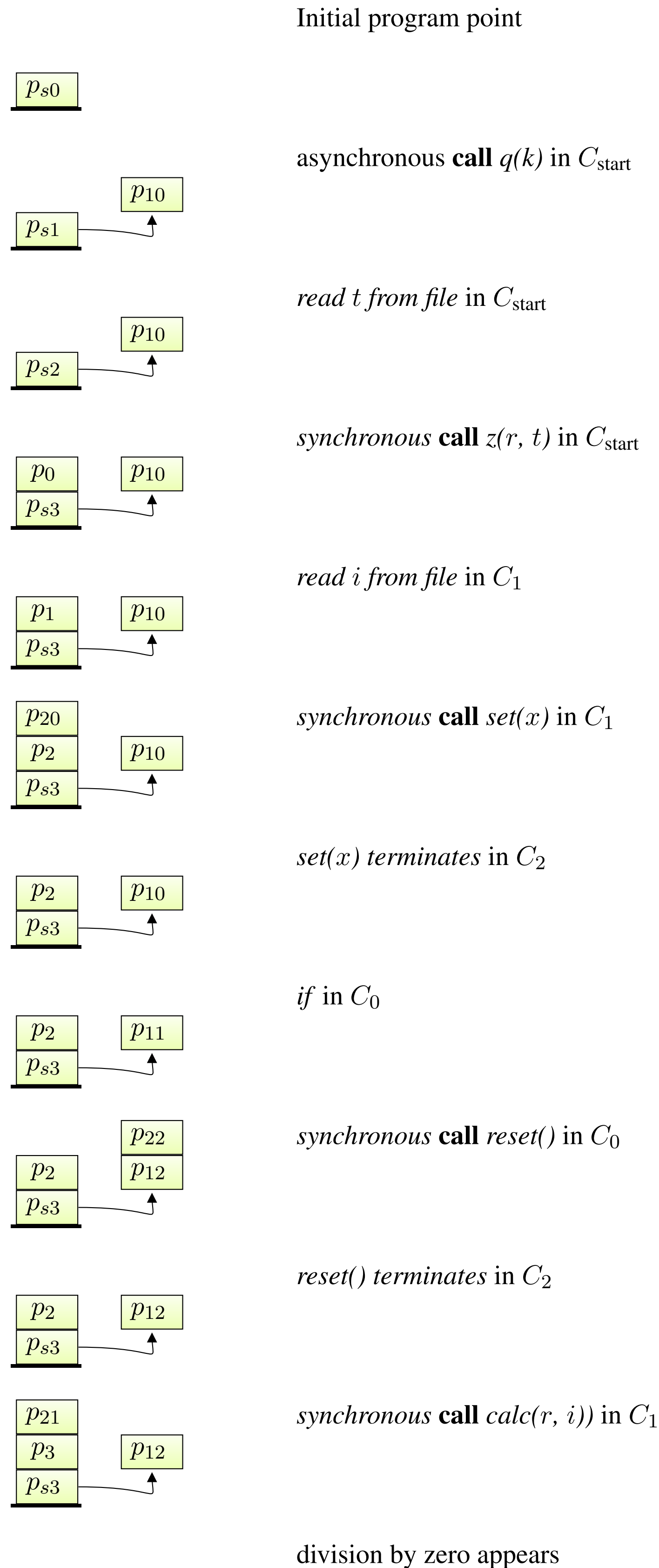


(d) Protocol $\mathcal{P}_{C_2}$ of component C_2 in a graphical representation.

$$
\begin{aligned}
\mathcal{P}_{C_2} =& (Q, \Sigma, I, \rightarrow, F), \text{ where} \\
Q =& \{v_0, v_1, v_2\}, \\
\Sigma =& \{reset, set, calc\}, \\
\rightarrow =& \{v_0 \overset{reset}{\rightarrow} v_0,\ v_0 \overset{set}{\rightarrow} v_1,\ v_1 \overset{set}{\rightarrow} v_1,\ v_1 \overset{calc}{\rightarrow} v_2,\ v_2 \overset{calc}{\rightarrow} v_2,\ v_2 \overset{set}{\rightarrow} v_1,\ v_2 \overset{reset}{\rightarrow} v_0\}, \\
I =& v_0, \\
F =& \{v_2\}
\end{aligned}
$$

(e) Protocol of component $\mathcal{P}_{C_2}$ as FSM.

Example 4.4: Cactus stack representing erroneous execution trace (counterexample) in Example 4.3b on page 59 (the labels of the frames are chosen randomly, at this point in time).



cactus stack representing this specific execution trace ($k = 42$) is shown in Example 4.4 on page 60 .

While looking at the source code (whitebox or glassbox components) we can discover the problem, but in a component-based software the components are blackboxes (premise of this work). Thus, it is hard to consider a problem, if only blackbox components are available, as we have no information about the behavior of the components. A verification method needs information about the behavior.

Remark

Several research [Bei95, Edw01, Mal95, KT04] has been done about blackbox testing. However, this cannot always lead to the discovery of all existing errors as the data ranges cannot be considered sufficiently, in the general case (state space explosion).

4.3 Abstractions of Source Code

> Program testing can be used to show the presence of bugs, but never to show their absence! Therefore, proof of program correctness should depend only upon the program text.
>
> *(Edsger W. Dijkstra [Dij69],[Dij70])*

In the previous section we have defined the protocol conformance problem. There, we used the behavior of the component-based software to determine whether every protocol is obeyed or not. However, we cannot assume that the source code of every component of a component-based software is accessible, because this contradicts the nature of components, which can be exist as blackbox components only. Moreover, it is possible that even the binary code is not accessible (e. g., Web Services). In an industrial environment the source code is protected, because of the encoded business secrets (intellectual property). This might result in a situation, where only blackbox components are present. In that case no statement about the source code behavior is provable (and the protocol conformance is undecidable). Hence, we have to choose a behavior representation, which complies the following requirements:

- It has to be a formal behavior representation, otherwise it cannot be used for model checking.
- It is not allowed to contain business secrets, hence data types and algorithms cannot be included in the representation, otherwise it could be possible to reconstruct intellectual properties.
- The representation has to contain every interaction with other components, or else not every protocol violation can be found.

- It has to represent the control-flow, such that it contains all possible interaction sequences.
- It has to be possible to represent synchronous as well as asynchronous interactions, because these are possible in several component systems or technologies (e. g., BPEL).

We conclude from these requirements, that we have to create a conservative representation of the given source code capturing sequential (cf. Definition 3.16 on page 38) as well as parallel behavior (cf. Definition 3.14 on page 38). This representation is called *abstraction*:

An abstraction $\mathcal{A}_C$ (abstract behavior) of a component C has to capture every observable behavior of C, thus at least every control flow path within C has to be considered. As representation of a component behavior the formal models shown in Section 3.2 can be used (e. g., finite state machines, push-down automata, Petri nets and many more).

Definition 4.6 (Component abstraction)
For an abstraction $\mathcal{A}_C$ of the component C the formula $L(C) \subseteq L(\mathcal{A}_C)$ is valid.

An abstraction $\mathcal{A}_S$ of an application S has to implement every executable path of S, thus every control-flow path has to be captured.

Definition 4.7 (System abstraction)
If $w \in L(S)$, then $w \in L(\mathcal{A}_S)$, in other words $L(S) \subseteq L(\mathcal{A}_S)$.

As mentioned in Section 2 it is possible to use different representations as abstraction of a source code. The most important ones are push-down automata and Petri nets. They are used in many works and were explored extensively (cf. Section 2). Push-down automata have significant advantages while dealing with recursion, whereas Petri nets are very suitable to represent concurrent behavior. While capturing the behavior of a component-based software, parallel as well as sequential behavior can appear. It is not possible to represent the behavior of a push-down automaton in general using a Petri net, and vice versa [May00].

A source code abstraction cannot capture the behavior exactly. In that case the resulting representation might be Turing-powerful, hence no model checking is possible. On the other hand an abstraction often means some kind of imprecise representation. While considering model checking (as we will do in this work), this could lead to *false negatives of the abstraction*. A false negative is a counterexample (Definition 4.5 on page 58) which is computable using the abstraction $\mathcal{A}_S$, but not computable in the actual implementation S.

Definition 4.8 (False negatives of an abstraction)
For a false negative w of an implementation S it is valid: $w \in L(\mathcal{A}_S) \wedge w \notin L(S)$.

False negatives often appear. If limited, they are acceptable. However, they should be reduced as far as possible, because the check (by a human) to determine whether a counterexample is a false negative might be very expensive.

Theorem 4.1 (False positives and conservative abstraction)
No false positives can appear, while assuming a conservative abstraction $\mathcal{A}_S$ of an application S

(Definition 4.7) and a sound model checker (which discovers all counterexamples).

Proof (Theorem 4.1)
A sound model checker checks $L(\mathcal{A}_S) \subseteq L(P_C)$, where P_C is the given constraint. Hence, it follows that $L(S) \subseteq L(P_C)$ (Definition 4.7).

It is clear that a conservative abstraction is possible, while computing the abstraction $\mathcal{A}_S$ of the application S, where $L(\mathcal{A}_S) = L(\mathbb{P}_i)$ (where $C_i \hat{=} (\mathbb{P}_i, \mathbb{R}_i)$ are the components of S). Thus, no false positives will appear. However, false negatives might appear. It has to be the aim to compute a more precise abstraction. i. e., it should contain less false negatives: $L(\mathcal{A}_S) \setminus L(S)$ should be minimized.

4.4 Use Process Rewrite Systems as Behavioral Representation

As discussed in Chapter 2 a representation of the component behavior should be capable to represent parallel and sequential behavior as well as asynchronous and synchronous interactions. Petri nets are suitable for parallel behavior and asynchronous interactions, while push-down automata are suitable for sequential behavior and synchronous interactions. To capture both behavior classes, we use Process Rewrite Systems (cf. Definition 3.28 on page 46). To our knowledge no other work uses Process Rewrite Systems for the purpose of representing component behavior.

Moreover, Process Rewrite Systems have the advantage that the theory contains the backward compatibility to other standard representations (like Petri nets and push-down automata). From our point of view, this is a main advantage in contrast to other (expressive) representations (e. g., Π-calculus [Mil92][3]). Therefore, if a specific representation is not needed, then another (standard) model can be used. All developed theory will be still applicable. However, it is possible to use research results of others to improve, for example, the moel checking time.

We will use Process Rewrite Systems, to represent the interactions between considered components and all other actions (with respect to the control flows) within the component-based software. To describe the latter explicitly, we define λ as a special action.

Definition 4.9 (No relevant action)
We introduce a special action $\lambda \notin \Sigma$. The action λ is equal to the empty word. *It denotes* no action *in the sense, that no interaction from one component to another is performed, which can influence the protocol conformance.*

We use the following conventions.

Convention 4.1 (λ-rules, action rules, explicit representation of λ)

- *Rules of the form $t \xrightarrow{\lambda} t'$, where $t, t' \in PEX(Q)$, are called λ-rules.*

[3] Furthermore, Π-calculus is Turing-powerful, thus model checking is not possible in general.

- *Rules of the form* $t \stackrel{a}{\to} t'$, *where* $a \in \Sigma$, $a \neq \lambda$, *and* $t, t' \in PEX(Q)$, *are called action rules.*
- *Because* λ *is an explicit representation of the empty word only, we omit noting explicitly that* λ *might be the action of a transition rule in the further thesis (if the meaning is clear by the context).*

Remark
Other authors (e. g., [Mil80, VdAvHvdT02]) use τ *to name internal actions (or silent actions) of a component. We use another representation of the empty word as not only internal actions are represented by* λ.

Using Process Rewrite Systems as abstraction representation makes it possible to formulate the language described by a behavior Π (cf. Definition 4.6 on page 62) in a formal way:

Definition 4.10 (Language accepted by an abstraction)
$L(\Pi) \hat{=} \{w : \exists f \in F | I \stackrel{w}{\Rightarrow} f\}$ *is the* language accepted *by* $\Pi = (Q, \Sigma, I, \to, F)$.

Likewise, it is possible to apply Definition 4.2 on page 56 on Process Rewrite Systems:

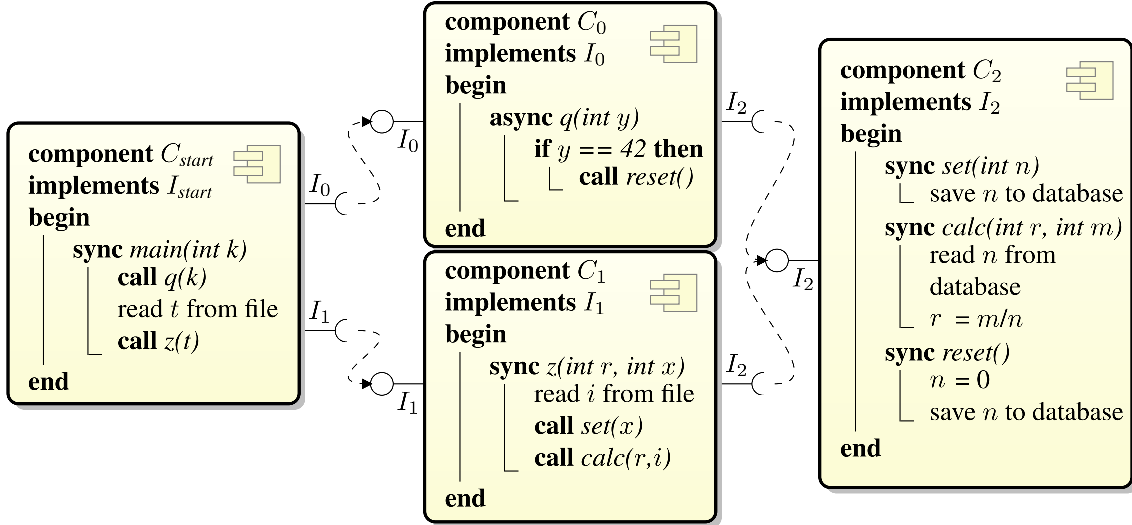
Definition 4.11 (Use of a component C in S in a Process Rewrite System Π_S)
The use of a component $\Pi_{S,C} = (Q_{S,C}, \Sigma_{S,C}, I_{S,C}, \to_{S,C}, F_{S,C})$ *based on a given system abstraction* $\Pi_S = (Q_S, \Sigma_S, I_S, \to_S, F_S)$ *and component* $C \hat{=} (\mathbb{P}, \mathbb{R})$ *is defined as follows:*

$$\begin{aligned}
Q_{S,C} &\hat{=} Q_S, \\
\Sigma_{S,C} &\hat{=} \Sigma_S \cap \mathbb{P}, \\
I_{S,C} &\hat{=} \mathbb{P}, \\
\to_{S,C} &\hat{=} \{t_1 \stackrel{b}{\to} t_2\}, \text{ where } b = \begin{cases} a: & (t_1 \stackrel{a}{\to} t_2) \in \to_S \wedge a \in \mathbb{P} \\ \lambda: & (t_1 \stackrel{a}{\to} t_2) \in \to_S \wedge a \notin \mathbb{P} \end{cases}, \\
F_{S,C} &\hat{=} F_S.
\end{aligned}$$

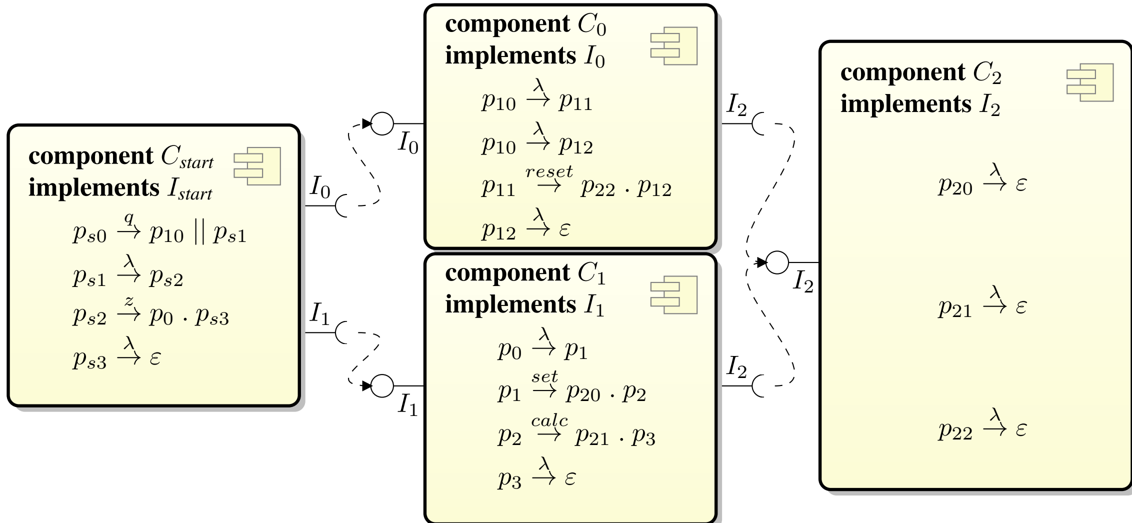
Convention 4.2 (Notation of use of component)
To mark the consideration of the interactions of one component $C \hat{=} (\mathbb{P}, \mathbb{R})$ *(in an application* S*), we use the notation* $\Pi_{S,C} \hat{=} \varphi_C(\Pi_S)$.

Example 4.5 shows the use of a component.

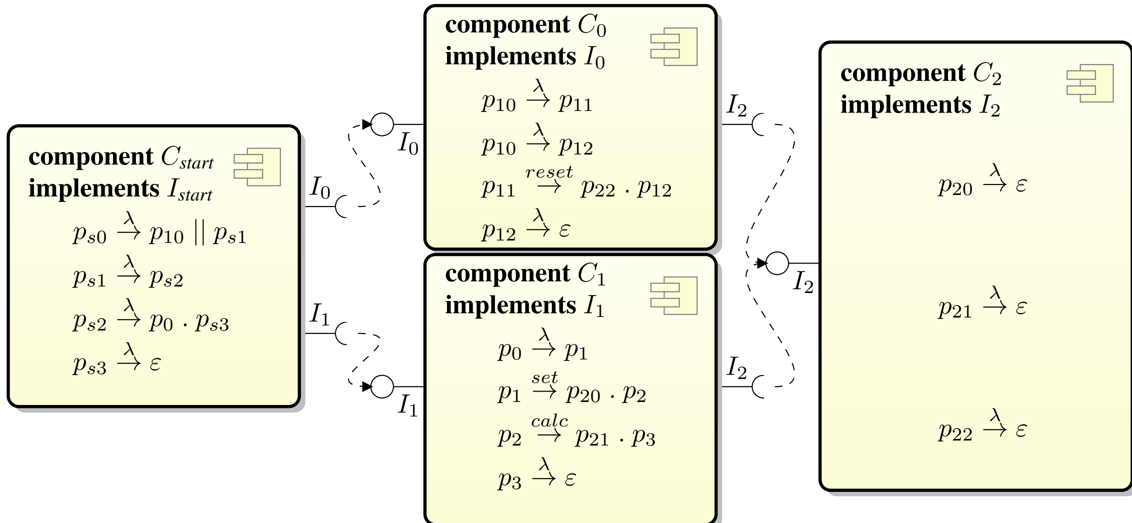
Example 4.5: System abstraction of S (cf. Example 4.3 on page 59) and use of component as Process Rewrite System.



(a) Components of Example 4.3 on page 59.



(b) Straight forward abstractions of application shown in Example 4.5a.



(c) Use of component C_2 (cf. Example 4.5b): The action of all rules performing no call to C_2 are replaced by λ (here only component C_{start} is affected).

4.4.1 PA-processes and Process Algebra Nets

By looking at the PRS hierarchy in Figure 3.4 on page 49 it is obvious that searching for a fusion of parallel concepts with sequential concepts requires to use the (P, G)-PRS "Process Algebra Nets"[4] (short: PAN) or the general PRS class (cf. Section 3.2.3 on page 47). As mentioned before we represent only the control flow in the abstract behavior. The data flow is omitted. In accordance with [Esp02, May00] the sequential operator on the left-hand side can be used to represent return values of procedure calls. As we capture no data flow, the return values are irrelevant, hence we can use Process Algebra Nets. Process Algebra Nets can handle unbounded recursion and unbounded parallelism including synchronization.

Moreover, the $(1, G)$-PRS "PA-processes" (short: PA) could be interesting, if no synchronization of concurrent control flows is allowed or captured in the abstraction.

4.4.2 Example Language

In this work we use a tiny programming language. The language definition is shown in Appendix A.2. It is defined with the usual semantics. An example was already introduced in Example 4.3 on page 59 and in Example 4.4 on page 60.

We request that a component (keyword in exemplary source code: "component") is used to encapsulate procedures. It implements at least one "interface". Signatures of procedures have to be marked with the modifier "sync" or "async". This defines whether a procedure has to be called synchronously or asynchronously.

We use a cactus stack (cf. Section 3.2.2) with interleaving semantics as execution model, i. e., every synchronous call to a procedure pushes a new frame onto the current stack, and every asynchronous call to a procedure results in the starting of a new process (branch of cactus stack). For simplification we allow no instances of a component in our example language.

4.4.3 Capturing Behavior with Process Rewrite Systems

In Section 3.2.2 we have shown that Process Rewrite Systems (process-algebraic expression) are a representation of cactus stacks. In Example 4.4 on page 60 we have gone through an erroneous execution (of the application in Example 4.3b on page 59) by using cactus stacks. In this section we will show that the behavior of the example programming language (cf. Section 4.4.2) can be represented using PRS transition rules.

We assume that the full source code of the application is available. The main ideas for the construction of an abstraction $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, \{\varepsilon\})$ are (where $Q_S = \{p_i : i \in \mathbb{N}\}$):

1. Create an atomic process for each program point p_i: Without loss of generality we assume, that every control flow path of a procedure ends (implicitly) with a *"return"-statement*. For "return"-statements no program point will be created.

[4] Caused by the fact that the grammar described by a (S, S)-PRS PDA can be accepted by a $(1, S)$-PRS called BPA, which is a push-down automaton with only one state.

2. Create transition rules, that map the control flow of the application in process rewrite rules (abstract behavior): We use the mapping function $next : Q_S \rightarrow 2^{Q_S}$, which results in the program points $p_j \in Q'$ ($Q' \in 2^{Q_S}$), where every p_j is a possible succeeding program point of $p_i \in Q_S$. The mapping result contains ε if there exists a control path, that ends in the next step ("return"-statement).

   - If at a program point p_i a synchronous procedure call a is performed, we create rewrite rules $p_i \overset{a}{\rightarrow}_S p_k \,.\, p_j$. We create the transition rule $p_i \overset{a}{\rightarrow}_S p_k \,||\, p_j$ if a is an asynchronous procedure call. The program point $p_k \in Q_S$ identifies the initial program point of the called procedure. If it is not known whether the procedure call is implemented synchronously or asynchronously, both transition rules are generated to ensure the conservative approximation.
   - If at a program point p_i another operation is performed, we create corresponding rewrite rules $t_i \overset{\lambda}{\rightarrow} t_k$, where $p_k \in t_k, p_i \in t_i$ and $p_k \in next(p_i)$. This transition rule has the semantics, that this operation is not interesting for the protocol verification (data calculations).

   We always update Q_S, when we create a new rewrite rule.

Theorem 4.2 (Conservative abstraction)
The abstraction leads to a conservative abstraction of any application S written in the example programming language with respect to the restrictions of the execution model (cactus stack).

To prove the theorem we will define the result of $next(p)$. We have to make a distinction, whether the program point is placed at the end of a block or not. We assume, that the last program point of the considered block is labeled with p_e. To ensure predictable behavior, we precise the values of the return set of $next(p_e)$ of the last statement of a "Block". We have to consider the following different situations:

- $Procedure$ ::= $type\ name$ "(" $ParameterDefs$ ")" "begin" $Block$ "end"

  The following formula is valid: $next(p_e) \hat{=} \{\varepsilon\}$.

- $Statement$ ::= "if" $Expr$ "then" $Block$
  $Statement$ ::= "if" $Expr$ "then" $Block$ "else" $Block$

  The following formula is valid: $next(p_e) \hat{=} \{p_k :$ where p_k identifies the next statement in the statement list after the "if"-statement$\}$.

- $Statement$ ::= "while" $Expr$ "do" $Block$

  The following formula is valid: $next(p_e) \hat{=} \{p_i :$ where p_i identifies the first statement within the "while"-block$\}$.

Proof (Theorem 4.2)
To prove the statement, we consider each possible statement. We always assume that the current

program point (statement) was labeled with p_0.

- *"if" $Expr$ "then" $Block$*

  The program counter can enter the block or not. Using the definition of $next(p_0)$, we know that a p_i exists identifying the initial program point of the block and a p_j exists identifying the program point, after the "if" statement[5]. As for each constant of the result set of $next(p_0)$ a PRS transition rule is generated and no interaction is performed, the following transition rules are generated:

$p_0 \xrightarrow{\lambda} p_i$	*enter the "if"-branch*
$p_0 \xrightarrow{\lambda} p_j$	*do not enter the "if"-branch*

  Thus, if an execution trace w contains interactions $\hat{w}$ initiated while executing the if-branch ($w \hat{=} w' \cdot \hat{w} \cdot w''$), it has to be valid $p_i \stackrel{\hat{w}}{\Rightarrow} \varepsilon$. Otherwise, $\hat{w}$ is empty ($\hat{w} = \lambda$). Hence, w can be constructed using the abstraction.

- *"if" $Expr$ "then" $Block$ "else" $Block$*

  The result set of $next(p_0)$ contains the initial program point of the first block p_i and the second block p_j. Hence, the following transition rules are generated:

$p_0 \xrightarrow{\lambda} p_i$	*enter the "if"-branch*
$p_0 \xrightarrow{\lambda} p_j$	*enter the "else"-branch*

  Thus, if an execution trace w contains interactions $\hat{w}$ initiated while executing the if-branch ($w \hat{=} w' \cdot \hat{w} \cdot w''$), it has to be valid $p_i \stackrel{\hat{w}}{\Rightarrow} \varepsilon$. If an execution trace w contains interactions w^{**}, initiated while executing the else-branch ($w \hat{=} w' \cdot w^{**} \cdot w'''$), it has to be valid $p_i \stackrel{w^{**}}{\Rightarrow} \varepsilon$. Hence, w can be constructed using the abstraction.

- *$Statement$ ::= "while" $Expr$ "do" $Block$*

  If the expression is evaluated to true, the block is entered. The constant p_i represents the initial program point of the block and p_j identifies the statement following the considered "while"-statement. The following transition represents the same behavior:

$p_0 \xrightarrow{\lambda} p_i$	*enter the "do"-block*
$p_0 \xrightarrow{\lambda} p_j$	*skip the "do"-block*

  Thus, if an execution trace w contains interactions $\hat{w}$ initiated while executing the while-block ($w \hat{=} w' \cdot \hat{w} \cdot w''$), it has to be valid $p_i \stackrel{\hat{w}}{\Rightarrow} \varepsilon$. If an execution trace is not executing the while-block, then $w^{**} = \lambda$ ($w \hat{=} w' \cdot w^{**} \cdot w'''$). Hence, w can be constructed using the abstraction.

[5] Note, that p_j can be ε, if the next statement is a "return"-statement.

- $Statement$ ::= "call" $name$ "(" $Parameters$ ")"

  If the procedure call is performed asynchronously, then a new process is forked. If a synchronous interaction is initiated, the current process waits until the result is returned. The following rewrite rules represent this behavior:

$$p_0 \overset{name}{\rightarrow} p_j \,||\, p_i,$$
$$p_0 \overset{name}{\rightarrow} p_j \cdot p_i.$$

  where $p_i \in next(p_0)$ and p_j is the initial program point of the called method.

  Thus, if $name$ is called while executing the application S, the execution path $w \hat{=} w' \cdot name \cdot w''$ is contained in the behavior. Because of the conservative abstraction, a derivation of Π_S exists so that $I \overset{w'}{\Rightarrow} t' \overset{name}{\Longrightarrow} t'' \overset{w''}{\Rightarrow} \varepsilon$.

As the abstract behavior of the programming language of each statement can be captured using PRS transition rules, the example programming language is captured conservatively. □

Remark (Expressions can be ignored)
In the used example programming language an expression ($Expr$) cannot initiate an interaction. Thus, the computation influences the data values only. Hence, no expressions can influence the protocol conformance while considering the abstraction. For this reason, expressions in "if"- and "while"-statements are not represented by an extra program point.

Remark (Automatic construction)
All necessary information can be derived automatically from the source code of the component and the interfaces (used by the component) using standard compiler construction technologies..

Remark
In [BZ08c, BZ08b] we have chosen a left-to-right evaluation order according to semantics of Java or C# and show the creation of the components in a general way. If the evaluation order of the considered component is implementation-dependent, one has to choose here the order used by a compiler to capture the source code conservatively. In Example 5.5 on page 80 an exemplary abstraction process is shown.

Convention 4.3 (Unique procedure names)
Without loss of generality we assume that the names of the remote procedures are unique, thus these names are used as interaction at the transition rules. This restriction is just made to simplify the notation in this work.

Remark
Process Rewrite Systems representing the abstract behavior (control flow) of an application will not always represent the behavior exactly (cf. Example 4.6 on page 71).

4.5 Summary

In this chapter we have defined the protocol conformance problem. As described, abstractions can be used to represent the behavior of applications. As mentioned, in this work we will focus on (P, G)-PRS called Process Algebra Nets, because they are capable to handle unbounded recursion and unbounded parallel behavior with synchronization. They extend Petri nets with sequential behavior. Hence, the main concepts of component systems can be represented using Process Algebra Nets and even recursive callbacks can be captured. Moreover, we will use PA-processes ($(1, G)$-PRS) if no synchronization has to be considered. We assume, that every Process Rewrite System is transformed to a Process Rewrite System in normal form.

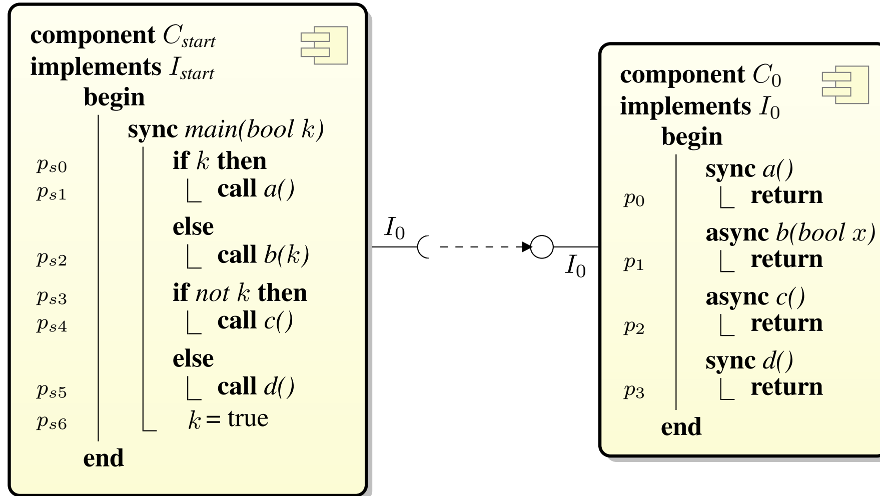
Based on the assumptions and definitions in this chapter, we will discuss our verification process in the next chapter. It captures the behavior of single components and processes them. Thereafter, a representation is created, where counterexamples can be computed.

Example 4.6: Behavior not represented exactly by a Process Rewrite System.

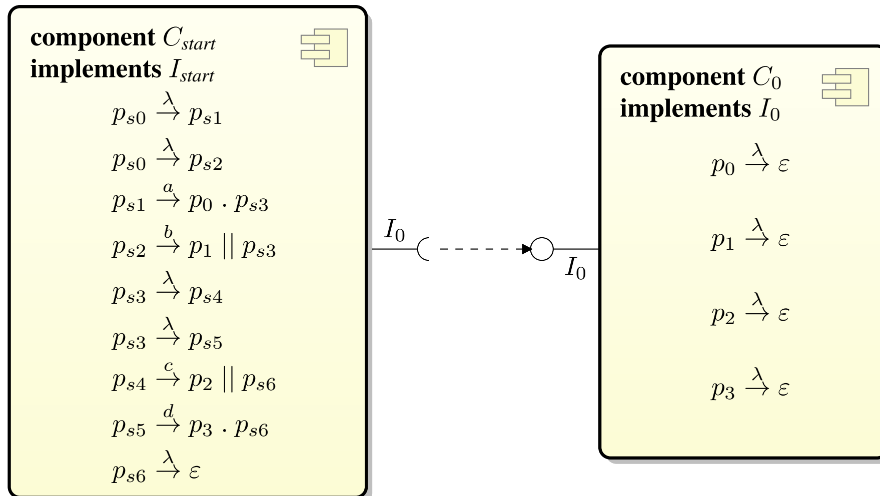
interface I_{start}
begin
  sync *main(bool k)*
end

interface I_0
begin
  sync *a()*
  async *b(bool x)*
  async *c()*
  sync *d()*
end

(a) Interface definition.



(b) Implementation of the components (with labeled program points).



(c) Behavioral abstractions of the components.

$a()\ c()$	this execution sequence is not possible within the actual implementation
$a()\ d()$	✓
$b()\ c()$	✓
$b()\ d()$	this execution sequence is not possible within the actual implementation
$c()\ b()$	✓
$d()\ b()$	this execution sequence is not possible within the actual implementation

(d) Possible execution sequences of the abstraction.

5 Verification Process

Here, we describe the heart of our approach: the verification process for protocol conformance checking.

The component abstractions of the application and the protocols that should be verified are needed as input. Moreover, we have to know, what kind of a component system is considered to imitate the composition of the component abstractions in the right manner. All these pieces of information are easily accessible. The steps of the verification process work automatically.

At the end the user of the verification process gets the answer whether the application is conform to all given protocols or not. In the latter case, the verification results in counterexamples visualizing the execution trace leading to the protocol violation.

In the following paragraphs, we will give a brief overview about the verification process. Later we will discuss the verification steps in detail.

The basic process is divided into five steps (cf. Figure 5.1). These steps are described in this section very briefly to give an overview of the whole process. In the Sections 5.1, 5.2, 5.3, 5.4 and 5.5 a detailed description of each step can be found.

The process was defined firstly in [BZ08a]. It works fully automatically. Only the verifying protocol has to be defined manually.

Step 1: Creating Component Abstractions (Brief Description)

In the first step a representation Π_C (Stripped Process Rewrite System) of only a single component $C = (\mathbb{P}, \mathbb{R})$ is created, containing $\mathbb{P}$ and $\mathbb{R}$. The abstraction is created conservatively, e. g., it captures at least the full behavior of the considered source code. An abstraction of every component can be generated independently. The Stripped Process Rewrite Systems $\Pi_{C_{\text{start}}}$, Π_{C_0}, Π_{C_1} and Π_{C_2} in Example 4.3 on page 59 can be seen in Example 5.1 on page 76 .

The abstraction process is an extension of the concept shown in Section 4.4. The sound description of this step can be found in Section 5.1.

Step 2: Creating System Abstractions (Brief Description)

In the second step we imitate the composition of components allowed in the considered component system. This transformation has to fulfill the properties of the component system or programming language, respectively.

As shown previously the system abstraction Π_S is a Process Rewrite System that represents all the possible behavior of the created component-based software S.

The transition rules shown in Example 4.5b on page 65 describe the application behavior in Example 4.3 on page 59 .

The sound description of this step can be found in Section 5.2.

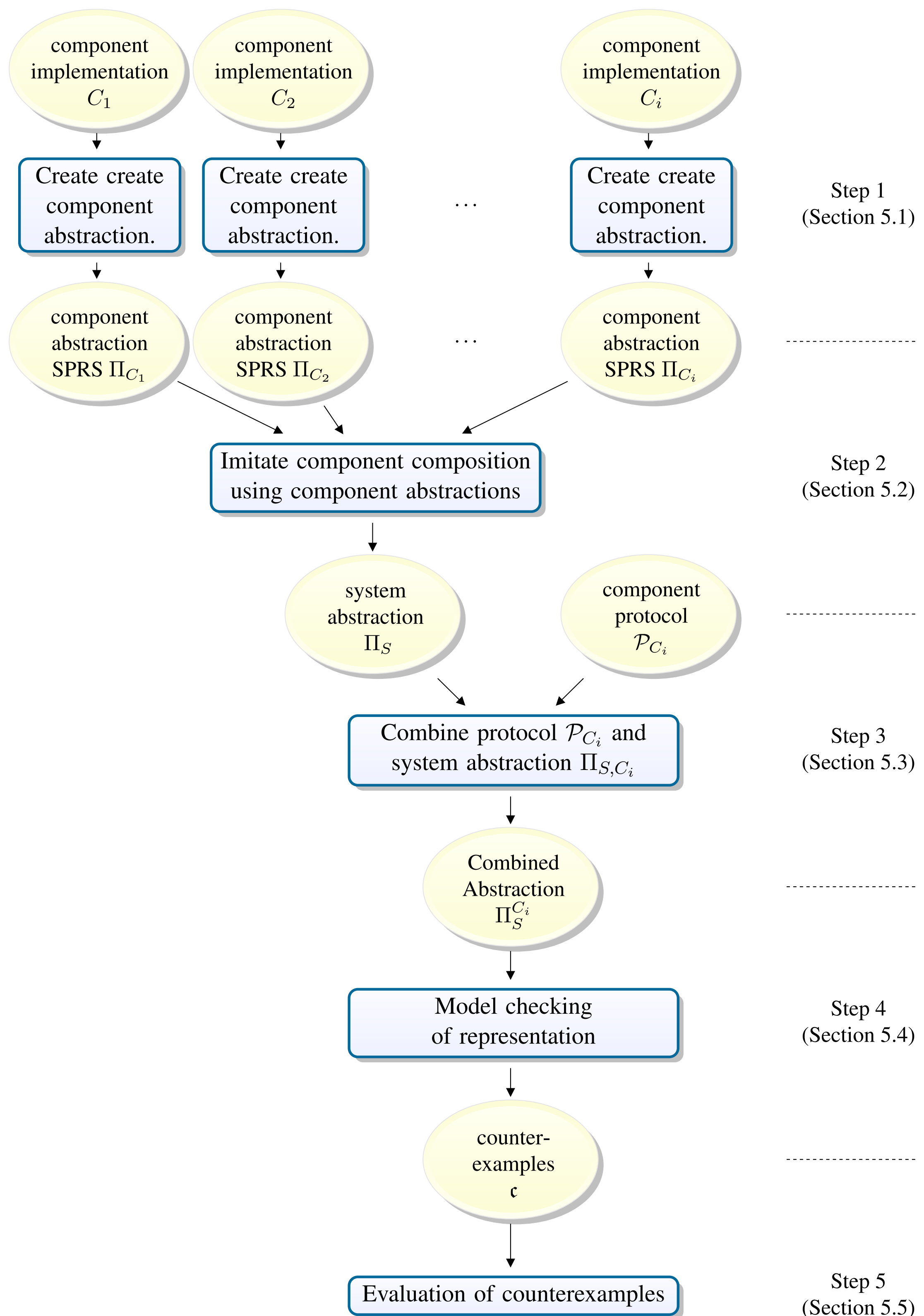


Figure 5.1.: Detailed preparation and verification process.

Step 3: Creating a Combined Abstraction (Brief Description)

In the third step a Process Rewrite System Π_S^C is constructed such that $L(\Pi_S^C) \supseteq \overline{L(P_C)} \cap L(\Pi_{S,C})$. The Process Rewrite System Π_S^C contains all sequences of interactions that are forbidden by the protocol P_C of one component C but nevertheless exists in the application S. The construction of the *Combined Abstraction* Π_S^C was presented in our papers [BZ08c, BZ08b, BZ08a]. We can find every error (protocol violation), because it is encoded as path from the initial program point to the final state of Π_S^C.

The sound description of this step can be found in Section 5.3.

Step 4: Model Checking the Combined Abstraction (Brief Description)

Every derivation from the initial state of Π_S^C to the final state identifies a *counterexample*. It contains all the interactions (or transition rules, respectively) resulting together in a forbidden use of the component C in the component-based software S.

Because we are interested in an applicable verification process we have to tackle the problem of long model checking time.

Example 4.3 on page 59 contains a counterexample (as described in Section 4.2). This counterexample can be constructed using our approach. It is not possible to give the user of the verification process only the information, that the application will crash when called with the parameter 42 (as shown in Example 4.5 on page 65). Nevertheless, we can describe directly, where the presented counterexample (based on the control flow) happens. Thus, the responsible component developer can inspect the problem easily.

The sound description of this step can be found in Section 5.4.

Step 5: Evaluating Counterexamples (Brief Description)

Computed counterexamples can be false negatives, which means that they are not reproducible in the model or in the source code. False negatives could in cases of doubt only be eliminated by the user of the verification process while reproducing the sequences of interactions with the components contained in a counterexample.

In this last step of the verification process we can use several techniques to reduce the number of counterexamples, which have to be reviewed. This is useful, because the reviewing process is very expensive.

The sound description of this step can be found in Section 5.5.

5.1 Step 1: Creating Single Component Abstractions

We define a uniform representation of a component behavior. Thus, it is possible to verify components from any programming language, without considering programming language specific pieces of information. These have to be considered in this step and obtained while translating it to the Process Rewrite System representation.

Example 5.1: $\Pi_{C_{\text{start}}}$, Π_{C_0}, Π_{C_1} and Π_{C_2} of Example 4.3 on page 59 as Stripped Process Rewrite Systems.

$$\begin{aligned}
\Pi_{C_{\text{start}}} &= (Q_{C_{\text{start}}}, \Sigma_{C_{\text{start}}}, \rightarrow_{C_{\text{start}}}, R_{C_{\text{start}}}, \mathbb{P}_{C_{\text{start}}}, \mathbb{M}_{C_{\text{start}}}) \text{ with} \\
Q_{C_{\text{start}}} &= \{p_{s0}, p_{s1}, p_{s2}, p_{s3}\} \\
\Sigma_{C_{\text{start}}} &= \{q, z\} \\
\rightarrow_{C_{\text{start}}} &= \{p_{s0} \overset{\lambda}{\rightrightarrows} q_{I_1,q} \mid\mid p_{s1}, p_{s1} \overset{\lambda}{\rightarrow} p_{s2}, p_{s2} \overset{\lambda}{\rightrightarrows} q_{I_0,z} \cdot p_{s3}, p_{s3} \overset{\lambda}{\rightarrow} \varepsilon, \} \\
R_{C_{\text{start}}} &= \{q_{I_1,q}, q_{I_0,z}\} \\
\mathbb{P}_{C_{\text{start}}} &= \{\} \\
\mathbb{M}_{C_{\text{start}}} &= \{\}
\end{aligned}$$

$$\begin{aligned}
\Pi_{C_0} &= (Q_{C_0}, \Sigma_{C_0}, \rightarrow_{C_0}, R_{C_0}, \mathbb{P}_{C_0}, \mathbb{M}_{C_0}) \text{ with} \\
Q_{C_0} &= \{p_0, p_1, p_2, p_3\} \\
\Sigma_{C_0} &= \{\, \textit{calc},\ \textit{set}\} \\
\rightarrow_{C_0} &= \{p_{10} \overset{\lambda}{\rightarrow} p_{11}, p_{10} \overset{\lambda}{\rightarrow} p_{12}, p_{11} \overset{\lambda}{\rightrightarrows} q_{I_2,reset} \cdot p_{12}, p_{12} \overset{\lambda}{\rightarrow} \varepsilon, \} \\
R_{C_0} &= \{q_{I_2,reset}\} \\
\mathbb{P}_{C_0} &= \{q_{I_0,z}\} \\
\mathbb{M}_{C_0} &= \{q_{I_0,z} \mapsto p_0\}
\end{aligned}$$

$$\begin{aligned}
\Pi_{C_1} &= (Q_{C_1}, \Sigma_{C_1}, \rightarrow_{C_1}, R_{C_1}, \mathbb{P}_{C_1}, \mathbb{M}_{C_1}) \text{ with} \\
Q_{C_1} &= \{p_{10}, p_{11}, p_{12}\} \\
\Sigma_{C_1} &= \{\, \textit{reset}\} \\
\rightarrow_{C_1} &= \{p_0 \overset{\lambda}{\rightarrow} p_1, p_1 \overset{\lambda}{\rightrightarrows} q_{I_2,set} \cdot p_2, p_2 \overset{\lambda}{\rightrightarrows} q_{I_2,calc} \cdot p_3, p_3 \overset{\lambda}{\rightarrow} \varepsilon\} \\
R_{C_1} &= \{q_{I_2,calc}, q_{I_2,set}\} \\
\mathbb{P}_{C_1} &= \{q_{I_1,q}\} \\
\mathbb{M}_{C_1} &= \{q_{I_1,q} \mapsto p_{10}\}
\end{aligned}$$

$$\begin{aligned}
\Pi_{C_2} &= (Q_{C_2}, \Sigma_{C_2}, \rightarrow_{C_2}, R_{C_2}, \mathbb{P}_{C_2}, \mathbb{M}_{C_2}) \text{ with} \\
Q_{C_2} &= \{p_{20}, p_{21}, p_{22}\} \\
\Sigma_{C_2} &= \{\} \\
\rightarrow_{C_2} &= \{p_{20} \overset{\lambda}{\rightarrow} \varepsilon,\ p_{21} \overset{\lambda}{\rightarrow} \varepsilon,\ p_{22} \overset{\lambda}{\rightarrow} \varepsilon\} \\
R_{C_2} &= \{\} \\
\mathbb{P}_{C_2} &= \{q_{I_2,set}, q_{I_2,calc}, q_{I_2,reset}\} \\
\mathbb{M}_{C_2} &= \{q_{I_2,set} \mapsto p_{20},\ q_{I_2,calc} \mapsto p_{21},\ q_{I_2,reset} \mapsto p_{22}\}
\end{aligned}$$

Example 5.2: Constructed system abstraction Π_S of a component-based software S assembled from component abstractions $\Pi_{C_{\text{start}}}$, Π_{C_0}, Π_{C_1} and Π_{C_2} in Example 5.1 on page 76.

$$
\begin{aligned}
\Pi_S &= (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S), \quad \text{where} \\
Q_S &= \{p_{s0}, p_{s1}, p_{s2}, p_0, p_1, p_2, p_3, p_{s3}, p_{10}, p_{11}, p_{12}, p_{20}, p_{21}, p_{22}\}, \\
\Sigma_S &= \{q, z, reset, set, calc\}, \\
I_S &= p_{s0}, \\
\rightarrow_S &= \{ \\
& \quad p_{s0} \overset{q}{\rightarrow} p_{10} \,||\, p_{s1}, \quad p_{s1} \overset{\lambda}{\rightarrow} p_{s2}, \quad p_{s2} \overset{z}{\rightarrow} p_0 \cdot p_{s3}, \\
& \quad p_{s3} \overset{\lambda}{\rightarrow} \varepsilon, \quad p_{10} \overset{\lambda}{\rightarrow} p_{11}, \quad p_{10} \overset{\lambda}{\rightarrow} p_{12}, \\
& \quad p_{11} \overset{reset}{\rightarrow} p_{22} \cdot p_{12}, \quad p_{12} \overset{\lambda}{\rightarrow} \varepsilon, \quad p_0 \overset{\lambda}{\rightarrow} p_1, \\
& \quad p_1 \overset{set}{\rightarrow} p_{20} \cdot p_2, \quad p_2 \overset{calc}{\rightarrow} p_{21} \cdot p_3, \quad p_3 \overset{\lambda}{\rightarrow} \varepsilon, \\
& \quad p_{20} \overset{\lambda}{\rightarrow} \varepsilon, \quad p_{21} \overset{\lambda}{\rightarrow} \varepsilon, \quad p_{22} \overset{\lambda}{\rightarrow} \varepsilon \\
& \}, \\
F_S &= \varepsilon
\end{aligned}
$$

In this chapter we will describe a general approach leading to the generation of a single component abstraction. In the first step a representation Π_C of only a single component $C = (\mathbb{P}, \mathbb{R})$ is created (Stripped Process Rewrite System), containing $\mathbb{P}$ and $\mathbb{R}$. In contrast to the regular Process Rewrite Systems a Stripped Process Rewrite Systems $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C, \mathbb{M}_C)$ contains also a mapping $\mathbb{M}_C : P_C \mapsto Q_C$ to program points Q_C identifying the provided interfaces P_C and a set R_C helping identifying the calls to required interfaces. Thereafter (Section 5.1.2), we will apply this approach to the programming language BPEL, which has a high relevance in industrial Web Services environments. The transition rules in $\rightarrow_C$ capture every relevant control flow inside the component C. This means that for every possible transition from a relevant program point to another one a transition rule in the Stripped Process Rewrite System is created. At the end of the section the different implementations performed for this work are discussed (Section 5.1.3). We will finish this chapter while considering optimizations of the abstraction leading to equivalent behavior but less transition rules.

5.1.1 General Approach

We will create a stripped abstraction Π_C as Process Rewrite System of the component C. This single component abstraction captures the behavior of the component without knowing any callee or context. Hence, this abstraction is as encapsulated as the component implementation.

Definition 5.1 (Stripped Process Rewrite Systems (SPRS))
A Stripped Process Rewrite Systems $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C, M_C)$ *of a component* $C =$

Example 5.3: Constructed system abstraction Π_{S,C_2} under consideration of only the interactions of component C_2 in Example 5.2.

$\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S)$ where
$Q_S = \{p_{s0}, p_{s1}, p_{s2}, p_0, p_1, p_2, p_3, p_{s3}, p_{10}, p_{11}, p_{12}, p_{20}, p_{21}, p_{22}\}$,
$\Sigma_S = \{reset, set, calc\}$,
$I_S = p_{s0}$,
$\rightarrow_S = \{$

$$
\begin{array}{lll}
p_{s0} \xrightarrow{\lambda} p_{10} \,||\, p_{s1}, & p_{s1} \xrightarrow{\lambda} p_{s2}, & p_{s2} \xrightarrow{\lambda} p_0 \cdot p_{s3}, \\
p_{s3} \xrightarrow{\lambda} \varepsilon, & p_{10} \xrightarrow{\lambda} p_{11}, & p_{10} \xrightarrow{\lambda} p_{12}, \\
p_{11} \xrightarrow{reset} p_{22} \cdot p_{12}, & p_{12} \xrightarrow{\lambda} \varepsilon, & p_0 \xrightarrow{\lambda} p_1, \\
p_1 \xrightarrow{set} p_{20} \cdot p_2, & p_2 \xrightarrow{calc} p_{21} \cdot p_3, & p_3 \xrightarrow{\lambda} \varepsilon, \\
p_{20} \xrightarrow{\lambda} \varepsilon, & p_{21} \xrightarrow{\lambda} \varepsilon, & p_{22} \xrightarrow{\lambda} \varepsilon
\end{array}
$$

$\}$,
$F_S = \varepsilon$

$(\mathbb{P}, \mathbb{R})$ *is defined as follows:*

Q_C *is a finite set of atomic processes,*

Σ_C *is a finite set of atomic actions,*

$\rightarrow_C \subseteq PEX(Q_C) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q_C)$ *is a set of process rewrite rules,*

$R_C \subseteq \mathbb{R} \times \mathsf{Sig}$ *is a finite set of required interfaces and the procedures called,*

$P_C \subseteq \mathbb{I} \times \mathsf{Sig}$ *is a finite set of the signatures* Sig *of all interfaces* $\mathbb{I}$*,*

$\mathbb{M}_C : P_C \mapsto Q_C$ *is a mapping from the provided interfaces to the atomic processes (program points)* $p \in Q_C$ *identifying the relevant initial program point of a method.*

I. e., R_C contains the callable operations or procedures of the current component (e. g., "I_2, a" is a required interface of C_1 in Figure 5.5b on page 80. In the same example, "I_1, m" is a provided interface of C_1.

We assume, that the full source code of the single component C is available. Nevertheless it is not needed after the automatic abstraction process (which is performed using standard compiler construction technologies). Then, just the constructed Stripped Process Rewrite System is used.

The main ideas for the construction of the Stripped Process Rewrite System are the same as discussed in Section 4.4.3. We just adapt the transition rules to remote procedures to allow the

Example 5.4: Erroneous execution trace (derivation) of Example 4.5 on page 65, based on the cactus stack translations in Example 4.4 on page 60, using the transition rules in Example 5.2 on page 77.

$\underline{p_{s0}}$	$\overset{q}{\Longrightarrow}$	$p_{10} \,\|\|\, \underline{p_{s1}}$	within component C_{start}
	$\overset{\lambda}{\Longrightarrow}$	$p_{10} \,\|\|\, \underline{p_{s2}}$	within component C_{start}
	$\overset{z}{\Longrightarrow}$	$p_{10} \,\|\|\, (\underline{p_0}.p_{s3})$	within component C_{start}
	$\overset{\lambda}{\Longrightarrow}$	$p_{10} \,\|\|\, (\underline{p_1}.p_{s3})$	within component C_1
	$\overset{set}{\Longrightarrow}$	$p_{10} \,\|\|\, (\underline{p_{20}}.p_2.p_{s3})$	within component C_1
	$\overset{\lambda}{\Longrightarrow}$	$\underline{p_{10}} \,\|\|\, (\varepsilon.p_2.p_{s3})$	within component C_2
	$\overset{\lambda}{\Longrightarrow}$	$\underline{p_{11}} \,\|\|\, (p_2.p_{s3})$	within component C_0
	$\overset{reset}{\Longrightarrow}$	$(\underline{p_{22}}.p_{12}) \,\|\|\, (p_2.p_{s3})$	within component C_0
	$\overset{\lambda}{\Longrightarrow}$	$(\varepsilon.p_{12}) \,\|\|\, (\underline{p_2}.p_{s3})$	within component C_2
	$\overset{calc}{\Longrightarrow}$	$p_{12} \,\|\|\, (\underline{p_{21}}.p_3.p_{s3})$	within component C_1
	$\overset{\lambda}{\Longrightarrow}$	$p_{12} \,\|\|\, (\varepsilon.\underline{p_3}.p_{s3})$	within component C_2
	$\overset{\lambda}{\Longrightarrow}$	$\underline{p_{12}}\|\|(\varepsilon.p_{s3})$	within component C_1
	$\overset{\lambda}{\Longrightarrow}$	$\varepsilon\|\|\underline{p_{s3}}$	within component C_0
	$\overset{\lambda}{\Longrightarrow}$	ε	within component C_{start}

mapping to the concrete bounded component or the possible bounded components.

1. If at a program point p_i a synchronous procedure call a of interface I is performed, we create rewrite rules $p_i \overset{a}{\to} q_{I,a}.\, p_k$
2. If procedure a of interface I is implemented asynchronously $p_i \overset{a}{\to} q_{I,a}||\, p_k$ is generated.[1]

The represented constant $q_{I,a}$ identifies the procedure a of a component implementing the interface I. Thus, it is possible to deduce the required interfaces and operations from the component abstraction.

Example 5.5 shows such a simple abstraction process.

Convention 5.1 (Unique name procedures)
Without loss of generality we assume, that the names of the remote procedures are unique, thus these names are used as interaction at the transition rules. This restriction is just made to simplify the notation in this work.

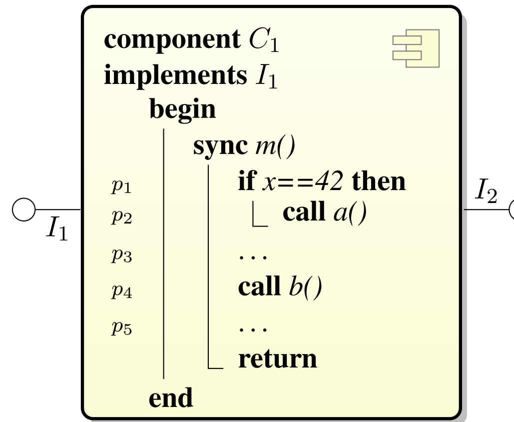
Remark
One premise of the source code abstraction is to make it possible to infer from the counterexam-

[1] If the context is clear, we will omit the interface in the transition rule, e. g., $p_i \overset{a}{\to} q_a||\, p_k$.

Example 5.5: Computing an abstraction of a component C.

```
interface I1          interface I2
begin                 begin
  | sync m()            | sync a()
end                     | async b()
                      end
```

(a) Interface definitions.



(b) Source code of a component (requiring interface I_2) with labeled program points.

$$
\begin{aligned}
\text{SPRS } \Pi_{C_1} \hat{=} (\ & \{p_1,\ p_2,\ p_3,\ p_4,\ p_5\}, \\
& \{a,\ b\}, \\
& \rightarrow_{C_1}, \\
& \{q_{I_1,m}\}, \\
& \{q_{I_2,a},\ q_{I_2,b}\}, \\
& \{q_{q_{I_1,m}} \mapsto p_1\} \\
)\ &
\end{aligned}
\qquad
\begin{aligned}
\text{where } \rightarrow_{C_1} \hat{=} \{\ & p_1 \overset{\lambda}{\rightarrow} p_2, \\
& p_1 \overset{\lambda}{\rightarrow} p_3, \\
& p_2 \overset{a}{\rightarrow} q_{I_1,a} \cdot p_3, \\
& p_3 \overset{\lambda}{\rightarrow} p_4, \\
& p_4 \overset{b}{\rightarrow} q_{I_1,b} \mid\mid p_5, \\
& p_5 \overset{\lambda}{\rightarrow} \varepsilon \quad \}
\end{aligned}
$$

(c) Stripped Process Rewrite Systems Π_{C_1}

ple to the source code. This property is needed to support the user of the development process in finding the problem. For this reason we assume, that every process constant p_i is unique in the whole application. This is a theoretical restriction only as we show in Section 6.1.1.

In [BZ08b] we use PA-processes as representation of the components behavior. The assumption is that neither a synchronization by interaction is allowed in the component system nor synchronizations of concurrent processes within the source code have to be captured. In [BZ08a] we show how the Web Services Business Process Execution Language (BPEL) can be represented (described in Section 5.1.2).

Remark
As in [ZS06], we can encode reference parameters in our component abstraction too, to regard even recursive callbacks. Also resolving the reference parameters to all possible dynamically chosen services is possible and equal to the mentioned earlier work. We do not describe these calculations here. For further details we refer to [ZS06].

Here, we have described the generation of an abstraction of a given source code or component, respectively. This abstraction is conservative with one restriction: The execution model has to be compatible with the cactus stack model (cf. Section 3.1.4).

5.1.2 Creating Stripped Process Algebra Nets from a Single BPEL Process

In this section we will describe the computation of single component abstractions of a specific programming language. We choose Web Services Business Process Execution Language, as this programming language is used explicitly for component-based applications. Another important property is, that Web Services Business Process Execution Language allows an explicit synchronization on a callback. Thus, we need the PRS class named Process Algebra Nets to represent the component behavior. The solution presented here, was published in [BZ08a].

5.1.2.1 Introduction of the Web Services Business Process Execution Language

The XML-based language Web Services Business Process Execution Language (short: BPEL)[2] [BPE03, BPE07] is used to specify processes, especially in a business context. Processes in BPEL use only Web Services to import functionality.

BPEL-processes can be executable (called executable business processes). They contain every information, that is needed to create a runnable process based only on BPEL-syntax. Abstract business processes are only partly specified processes, so they can hide implementation details. In this work we make no difference between both kinds.

In the following we will give a brief overview over the syntax and semantics of BPEL statements, called activities. From our point of view we do not have to distinguish between WS-BPEL 1.1 [BPE03] and WS-BPEL 2.0 [BPE07], but we focus on the WS-BPEL 1.1 standard.

[2]BPEL was initiated by IBM, BEA and Microsoft and is currently developed by OASIS. BPEL reach an high industrial acceptance level, thus is a de-facto-standard for business process implementation based on Web Services, today.

BPEL activity	short description
`invoke`	It is possible to call methods on *partners* using the `invoke` statement, this remote procedure call can be performed synchronously or asynchronously. A synchronous call blocks the currently executed trace of the caller until the callee returns the focus to the caller.
`reply`	Using this activity a callee returns the focus to the caller.
`receive`	If this statement exists at a program point in a BPEL process an invoked procedure call can be received.
`switch`	The control-flow can be influenced while using the `switch` statement (converted to `if` in WS-BPEL 2.0).
`while`	To repeat activities one can use this statement (moreover, WS-BPEL 2.0 allows `RepeatUntil`).
`flow`	To process a set of activities in parallel traces one has to use this statement.
`sequence`	A FIFO list of activities can be processed using this statement.
`empty`	Nothing will happen.

Table 5.1.: Considered BPEL activities.

BPEL processes (wrapped with the `process` statement) create references to other Web Services (*partners*) while using the command `partnerlinks`. We reduce our BPEL model to the statements mentioned in Figure 5.1.

We will not consider the entities `message`, `variable`, `copy` and `wait` because they have a data (flow) meaning, which is not (yet) considered in our approach. Moreover, we will exclude for example `event`, `faulthandler`, `pick`, etc. from our consideration.

5.1.2.2 Example

In Example 5.6 an example system assembled from Web Services is given. We use a trimmed graphical representation of BPEL and hide all irrelevant information. Moreover, we enumerate every activity.

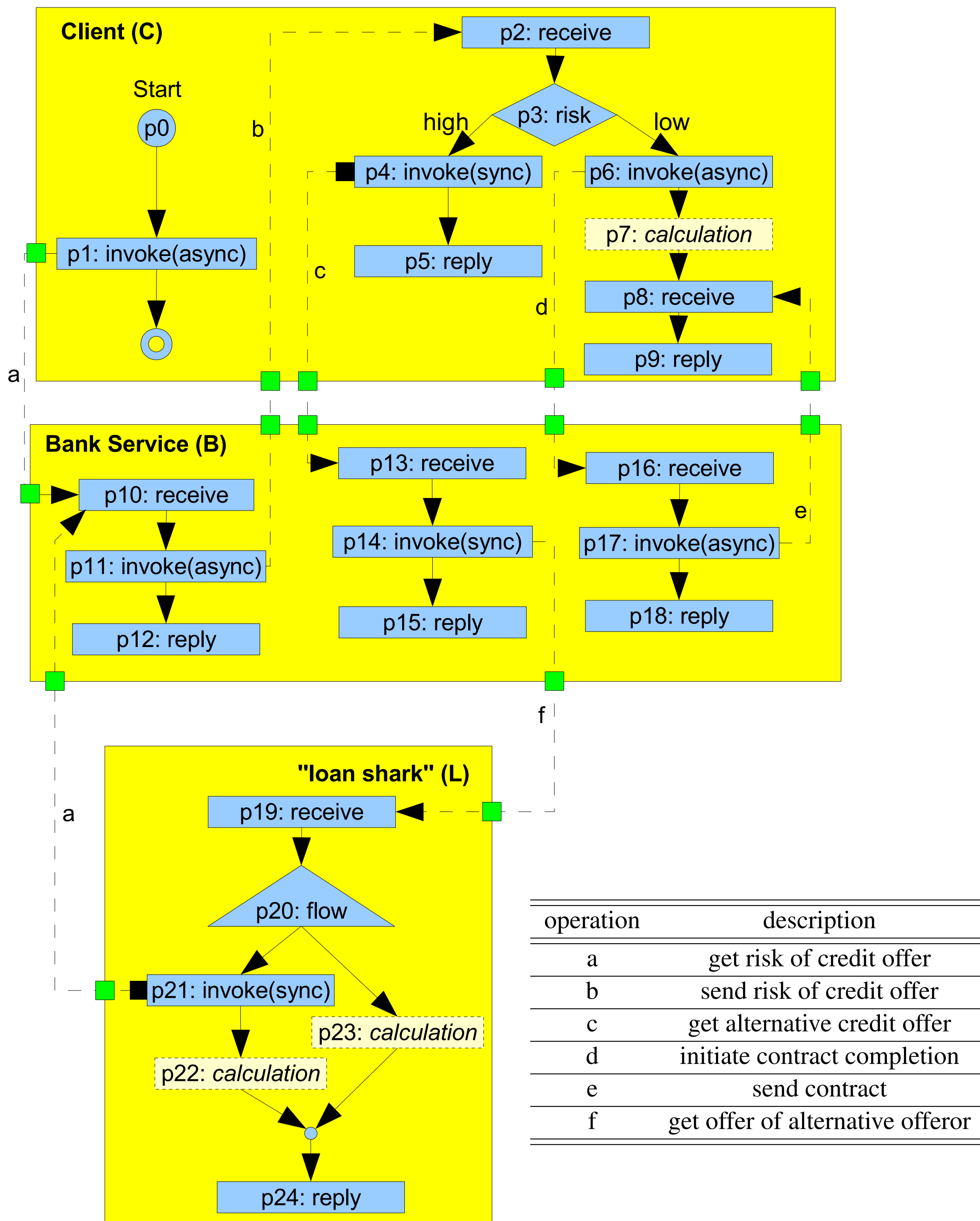
We see three Web Services. The service client C represents a terminal of a bank employee responsible for credit queries. The client starts the workflow (p_0) by asking the banks service B for a credit offer (invocation of a in activity p_1). The callback b sends back a personal risk, which is used to decide (`switch`, activity p_3) if a credit contract can be contracted (operations d and callback e). If the risk is too high the customer will be redirected to a *"loan shark"*, named Web Service L (operations c and f). This Web Service queries the bank B for the personal customer risk, while finishing some calculations (activity p_{23}) in parallel. Then L returns an adequate credit offer (activity p_{24})[3].

The protocols are given in Example 5.6.

It is likely that BPEL representations will not always be published, because they are used often only in an in-house context and include business secrets.

[3]We assume that correlation is used to ensure that the right Web Service instance will be called at every time.

Example 5.6: Bank system implemented using BPEL (with protocols).



operation	description
a	get risk of credit offer
b	send risk of credit offer
c	get alternative credit offer
d	initiate contract completion
e	send contract
f	get offer of alternative offeror

(a) Architecture of bank system.

$P_{\mathrm{C}} = (be)^*$ $\qquad$ $P_{\mathrm{B}} = (a(c|d))^+$ $\qquad$ $P_{\mathrm{L}} = f^*$

(b) Protocols of Web Services given as regular expressions.

5.1.2.3 Generating Abstractions of BPEL Web Services

Thus, we present an approach to generate an abstract representation of a single Web Service C. Here, we use *Stripped Process Algebra Nets*. This abstraction can be delivered together with the protocol of the Web Service without revealing business logic or business secrets.

Definition 5.2 (Stripped Process Algebra Nets)
A Stripped Process Algebra Net *is defined equivalently to a Stripped Process Rewrite System (Definition 5.1 on page 78) in Section 5.1.1. This abstraction does not contain transition rules of the form* $p_1.p_2 \overset{a}{\rightarrow} p_3$*. Thus, using Process Algebra Nets is sufficient.*

The set P_C contains all interfaces $q_{J,o}$ of C where o is an operation of a callable interface J[4]. $\mathbb{M}_C$ maps the set of interfaces P_C (provided by C) to the relevant atomic process of the abstraction of the callable Web Services.

In this abstraction no transition rule of the form $p_1.p_2 \overset{a}{\rightarrow} p_3$ is allowed.

Remark
As Process Algebra Nets can capture the same semantics as Petri nets it is possible to use other approaches (e. g., [VvdA05, Loh08, HSS05]) to represent BPEL behavior. E. g., in [Loh08] fault handler are considered, too.

Example 5.7: Process rewrite rules of the Stripped Process Algebra Nets Π_C, Π_B and Π_L of every single BPEL process C, B and L of Example 5.6 on page 83.

Process rewrite rules of the Stripped Process Algebra Nets (in normal form):

$$\begin{array}{llll} \rightarrow_C = \{ & p_0 \overset{a}{\rightarrow} q_{B,a}||p_1, & p_1 \overset{\lambda}{\rightarrow} \varepsilon, & p_2 \overset{\lambda}{\rightarrow} p_3, \\ & p_3 \overset{\lambda}{\rightarrow} p_4, & p_3 \overset{\lambda}{\rightarrow} p_6, & p_4 \overset{c}{\rightarrow} q_{B,c}.\varepsilon, \\ & p_6 \overset{d}{\rightarrow} q_{B,d}||p_7, & p_7||p_8 \overset{\lambda}{\rightarrow} \varepsilon & \} \end{array}$$

$$\begin{array}{llll} \rightarrow_B = \{ & p_{10} \overset{\lambda}{\rightarrow} p_{11}, & p_{11} \overset{b}{\rightarrow} q_{C,b}||\varepsilon, & p_{13} \overset{\lambda}{\rightarrow} p_{14}, \\ & p_{14} \overset{f}{\rightarrow} q_{L,f}.\varepsilon, & p_{16} \overset{\lambda}{\rightarrow} p_{17}, & p_{17} \overset{e}{\rightarrow} q_{C,e}||\varepsilon \quad \} \end{array}$$

$$\begin{array}{llll} \rightarrow_L = \{ & p_{19} \overset{\lambda}{\rightarrow} p_{20}, & p_{20} \overset{\lambda}{\rightarrow} p_{21'}||p_{23'}, & p_{21'} \overset{\lambda}{\rightarrow} p_{21}.e_0, \\ & p_{23'} \overset{\lambda}{\rightarrow} p_{23}.e_1, & p_{23} \overset{\lambda}{\rightarrow} \varepsilon, & p_{21} \overset{a}{\rightarrow} q_{B,a}.p_{22}, \\ & p_{22} \overset{\lambda}{\rightarrow} \varepsilon, & e_0||e_1 \overset{\lambda}{\rightarrow} \varepsilon & \} \end{array}$$

Mapping $\mathbb{M}$ of operations to atomic processes

$$\begin{array}{lll} \mathbb{M}_C : q_{C,b} \mapsto p_2, & \mathbb{M}_C : q_{C,e} \mapsto p_8, & \\ \mathbb{M}_B : q_{B,a} \mapsto p_{10}, & \mathbb{M}_B : q_{B,c} \mapsto p_{13}, & \mathbb{M}_B : q_{B,d} \mapsto p_{16}, \\ \mathbb{M}_L : q_{L,f} \mapsto p_{19} & & \end{array}$$

[4] J encapsulates all relevant information from the `partnerlink` statement or WSDL file, respectively.

Because we assume, that the functionality of a single Web Service C is available in BPEL syntax, we describe the translation of BPEL activities to process rewrite rules. The main ideas for the construction of Stripped Process Algebra Nets are (similar to Section 5.1.1):

1. Create an atomic process for each activity p_i of C. In Example 5.6 on page 83 we already marked the statements. Calculations (p_7, p_{22}, p_{23}) are represented in an aggregation. This is possible as they represent data flow only (which is not considered here).
2. Create transition rules, which map the control flow of C in process rewrite rules: We use the mapping function $next : p_i \rightarrow Q'$ (cf. Section 5.1.1).
   - If an activity p_i is an `invoke` statement, which performs a synchronous procedure a, we create a rewrite rule $p_i \stackrel{a}{\rightarrow}_C q_{J,a} . p_k$. If a is an asynchronous method call, we create the rewrite rule $p_i \stackrel{a}{\rightarrow}_C q_{J,a} \,||\, p_k$, where $q_{J,a}$ specifies the partner J of the operation a, and $p_k \in next(p_i)$.[5]
     In our example activity p_4 performs a synchronous interaction with the "Bank Service" captured by the transition rule $p_4 \stackrel{c}{\rightarrow} q_c.p_{13}$, whereas p_1 initiates an asynchronous interaction: $p_1 \stackrel{a}{\rightarrow} q_a||p_{10}$.
   - If the activity p_i is the first in a trace of the BPEL process (`receive`) implementing a provided operation q_o, we will extend the mapping $\mathbb{M}_C$ with $q_o \mapsto p_i$. q_o can be determined while analyzing the `partnerlink` statements (or WSDL description, respectively).
     Thus, for the "Bank Service" we create the following mapping: $\mathbb{M}_B = \{q_{B,a} \mapsto p_{10}, q_{B,c} \mapsto p_{13}, q_{B,d} \mapsto p_{16}\}$.
   - If a next statement p_j of p_i is a `receive` statement, the BPEL process expects an invoked operation a from a partner J (i. e., $q_{J,a}$). This results in a synchronization semantics, which is captured creating the rules $p_i||p_j \stackrel{\lambda}{\rightarrow} p_k$, with $p_j \in next(p_i), p_k \in next(p_j)$.
     The "Client" contains such a situation, represented by $p_7||\, p_8 \stackrel{\lambda}{\rightarrow} \varepsilon$.
   - If a `flow` statement p_i is used, we have to create a process rewrite rule $p_i \stackrel{\lambda}{\rightarrow} (p_0.e_0) \,||\, (p_1.e_1) \,||\, \ldots \,||\, (p_{j-1}.e_{j-1})$, where $p_0, p_1, \ldots, p_{j-1}$ are the j activities started by the `flow` in parallel. $e_0, e_1, \ldots, e_{j-1}$ are newly created, unique atomic processes, that mark the parallel executed activity as finished. To capture the defined `flow` behavior we create also the rules $e_0||e_1|| \ldots ||e_{j-1} \stackrel{\lambda}{\rightarrow} p_k$, where $p_k \in next(p_i)$. These ensure that the next activities cannot start until each activity $p_0, p_1, \ldots, p_{j-1}$ is finished.
     The transition rule $e_0||e_1 \stackrel{\lambda}{\rightarrow} \varepsilon$ represents the `flow` statement of Web Service L.
   - If a `sequence` statement p_i is used, we have to create process rewrite rules, $p_i \stackrel{\lambda}{\rightarrow} p_j.p_k$, with $p_k \in next(p_i)$, where p_j is the first activity performed inside p_i.
     Aiming simplification, sequences are omitted in the example.

[5]Note if we do not know how the `invoke` statement is implemented, we have to create both sets of rewrite rules to ensure that we create a conservative abstraction.

- If at a program point p_i another activity is performed, we create rewrite rules $p_i \xrightarrow{\lambda} p_k$, where $p_k \in next(p_i)$. This transition rule has the semantics that this activity is not interesting for the protocol verification.
  These rules capture for example the transition rules, that do not influence the control flow. E. g., the calculation p_{22} in Web Service L is bypassed by the transition rule $p_{22} \xrightarrow{\lambda} \varepsilon$.

Thus, the semantics is comparable to the general approach presented in Section 5.1.1. The representation of a `flow` activity requires a join rule.

Remark
All these pieces of information can be derived automatically from the BPEL code of the Web Service and the ports used by the Web Service.

Remark
In the abstraction rules shown above, the resolution of correlation settings in BPEL is not considered. We omit these details, because the concept is equal to encoding reference parameters in [ZS06]. Reference parameters are implicitly possible in BPEL using correlation. Also resolving the reference parameters to all possible dynamically chosen services is possible.

Using this construction we get a Stripped Process Algebra Net for every BPEL process C. This abstraction has to be delivered together with the protocol to the partner using C. The abstraction and the protocol could be part of the WSDL service description. In Example 5.7 on page 84 the abstractions of the Web Services B, C and L are shown together with the mapping M_B, M_C and M_L.

5.1.3 Implemented Translations to Process Rewrite Systems

5.1.3.1 Creating Stripped Process Algebra Nets from a Single Python Component

We consider Python [Fou09b], because it is one of the main programming languages used by our industrial partner (cf. Chapter 6).

Python is a dynamically-typed programming language [Fou09b]. It has a large standard library. The scopes of applicability are wide and therefore many products are developed using Python [Pyt09]. Python allows to implement parallel behavior in form of threads. A thread identifier can be handled as regular data (see the Listing A.3). Therefore, a conservative abstraction is not possible in any case using the cactus stack execution model. All interactions are performed synchronously.

In works supervised by us [Kra08, Kra09] abstractions of Python components are generated (only a part of the language specification was captured). The module “ast” [Fou09a] of the Python standard library is used for this purpose. It provides the access to the abstract syntax tree of a given application. The syntax tree contains only a rudimentary set of attributes (properties). Therefore, the main work was to decorate the abstract syntax tree to achieve an attribute syntax tree.

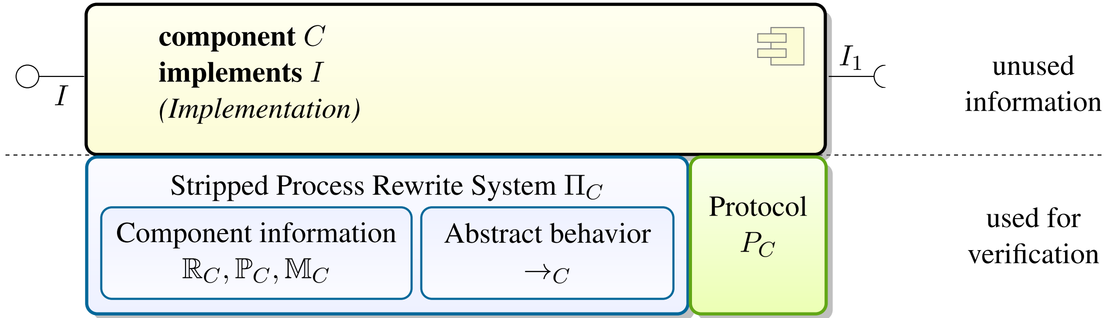


Figure 5.2.: Implementation of a component C, its abstraction Π_C and protocol P_C.

5.1.3.2 Creating Stripped Process Algebra Nets from a Single BPEL Webservices

Based on the approach discussed in 5.1.2 a tool is developed, currently. The tool is part of a diploma thesis [Hel10], which we are supervising.

Related work (e. g., [VvdA05, HSS05, Loh08] use Petri nets as representation. This has the advantage, that events and fault handlers can be implemented in a general case.

In the diploma thesis [Hel10] Process Algebra Nets are used to discover the problems discussed in Section 5.1.2. Unfortunately, the implemented abstraction approach used there seems reducing the capability to capture events and fault handlers (exception handlers).

5.1.3.3 Creating Stripped Process Algebra Nets From a Single PHP Component

PHP [Gro09] is an untyped programming language. It is considered in a diploma thesis, we are supervising currently. PHP is used while evaluating case study in an online shop context (cf. Chapter 6).

PHP allows no parallel behavior[6]. Therefore, push-down automata ($(1, S)$-PRS) are sufficient to represent the behavior of the source code. Nevertheless, it is interesting to consider PHP components, because they are often used steered by AJAX implementations. They allow parallel behavior, thus the complete application can be captured only by using Process Algebra Nets (or a higher PRS-class).

We extend a PHP to C compiler – named PHC [dVG07, BdVG09] – for our purpose.

5.1.4 Summary

In this section we have seen how source code abstractions are defined in this work. We use an adapted Process Rewrite System (named Stripped Process Rewrite Systems) to represent the behavior of the single component, in the sense of the component model. Hence, the required and provided interfaces are represented in the single component abstraction, too. To our knowledge this representation was not defined previously. In Figure 5.2 the correlation between a

[6] A few extensions exist, implementing a threading concept in PHP. None is adopted by the standard library.

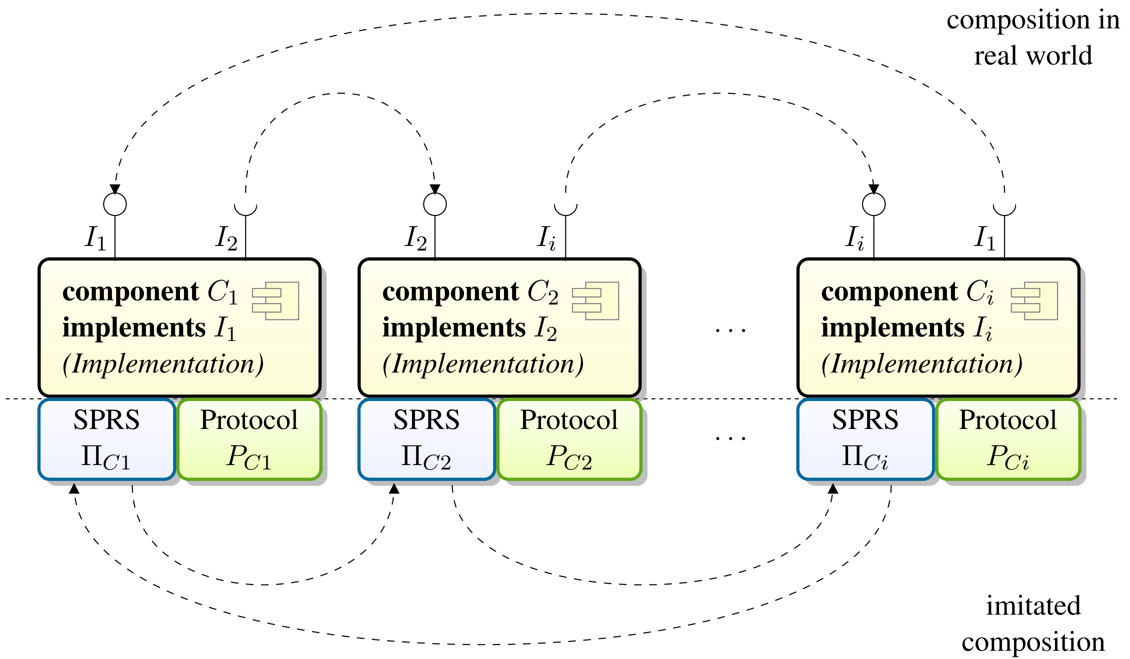


Figure 5.3.: Schematic representation of the construction of the system abstraction.

component's implementation, its protocol and abstraction is represented schematically. As it is visualized, the actual implementation is not used within the suggest approach.

The representation of the source code is more powerful than other abstractions like Petri nets (e. g., [VdAvHvdT02]) and push-down automata (e. g., [ZS06]). Thus, it enables new options of verification of component-based software, because it can capture unbounded parallelism and unbounded recursion.

Moreover, we have presented an approach to capture the behavior using standard compiler technologies. Developing tools for capturing various programming languages causes different hitches as seen in this chapter. We have shown, that this process is possible for programming languages (standard) object-oriented behavior (C++) as well as for dynamically typed programming languages (Pyhton, PHP) and component systems with irregular communication ways (join on receive in BPEL).

Because the abstraction is generated of exactly one single component (without knowing possible peer implementations) the source code is protected (e. g., to ensure the business secrets). Hence, it is possible using this approach in the standard case of component-based software engineering, where the implementations of the components are not available (e. g., because it is developed by different companies).

5.2 Step 2: Creating System Abstractions

In the previous chapter we have created an abstraction of each component separately. This component abstraction has to be assembled to an abstraction of the complete application, named *system abstraction*. The system abstraction represents the behavior of the application we want to

verify. Thus, we have to ensure, that all component abstractions are connected as the components in the considered component-based system.

We use the information about the required and provided interfaces to resolve the possible interactions within a component abstraction from a caller to a callee.

5.2.1 Process

Now, we will create a system abstraction of S using the Process Rewrite System representation.

Definition 5.3 (System abstraction as Process Rewrite System Π_S)
A system abstraction (as defined previously in Definition 4.7 on page 62) of a given application based on a binding $S = (\textsc{Comp}, \textsc{Bind})$*. Here, it is adapted as Process Rewrite System* $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, \{\varepsilon\})$*, where*

$$
\begin{aligned}
Q_S &= \bigcup_{C_i \in \textsc{Comp}} Q_{C_i} \uplus \{I_S\} \\
\Sigma_S &= \bigcup_{C_i \in \textsc{Comp}} \Sigma_{C_i} \\
I_S &\notin \bigcup_{C_i \in \textsc{Comp}} Q_{C_i}
\end{aligned}
$$

The construction of $\rightarrow_S$ *is discussed later.*

Hence, the system abstraction represents the behavior of the component-based software S using the standard Process Rewrite System representation. In contrast to the representation of the single component abstraction, it has in no provided and no required interfaces.

Because we search for a terminating execution of the application S, we define ε as final process. Thus, a derivation path $I_S \stackrel{w}{\Rightarrow} \varepsilon$ of Π_S describes an execution path within S, which terminates.

In Figure 5.3 the creation of a system abstraction is shown schematically. In Algorithm 5.1 on page 91 we describe how $\rightarrow_S$ is computed.

There, the interfaces of the contained components are resolved (line 3 to 19) resulting in the set of transition rules $\rightarrow_S$. This is implemented while adapting each transition rule containing a call to a required interface (lines 8,10,12)[7].

Moreover, a set of start rules is computed (line 21 to 29), ensuring the semantics of the considered application S. Depending on the component system three general cases are possible:

- line 21: If the application contains a client, then it is clear, which component has to be started. Thus, we generate a new transition rule $I_S \stackrel{\lambda}{\rightarrow} I_{C_i}$
- line 24: If it is possible, that each component starts firstly, we will generate a start rule for each of the components. This is the general case of the one in line 21.
- line 27: We add this rule to represent that several components can start concurrently and thus can be processed independently even in the abstraction.

[7]The consideration in Line 12 is needed as chain rules can be construction during optimizations.

Example 5.8: Rewrite rules of system abstraction $\Pi_{S,C}$ of Example 5.7 on page 84 according to the service protocol of the Web Service C. In rules of the form $p_i \stackrel{a}{\rightarrow} p_j.\varepsilon$ or $p_i \stackrel{a}{\rightarrow} p_j||\varepsilon$ the neutral element ε was eliminated.

$$
\begin{array}{lll}
I_S \stackrel{\lambda}{\rightarrow} p_0, & p_0 \stackrel{\lambda}{\rightarrow} p_{10}||p_1, & p_1 \stackrel{\lambda}{\rightarrow} \varepsilon, \\
p_2 \stackrel{\lambda}{\rightarrow} p_3, & p_3 \stackrel{\lambda}{\rightarrow} p_4, & p_3 \stackrel{\lambda}{\rightarrow} p_6, \\
p_4 \stackrel{\lambda}{\rightarrow} p_{13}.\varepsilon, & p_6 \stackrel{\lambda}{\rightarrow} p_{16}||p_7, & p_7||p_8 \stackrel{\lambda}{\rightarrow} \varepsilon, \\
p_{10} \stackrel{\lambda}{\rightarrow} p_{11}, & p_{11} \stackrel{b}{\rightarrow} p_2||\varepsilon, & p_{13} \stackrel{\lambda}{\rightarrow} p_{14}, \\
p_{14} \stackrel{\lambda}{\rightarrow} p_{19}, & p_{16} \stackrel{\lambda}{\rightarrow} p_{17}, & p_{17} \stackrel{e}{\rightarrow} p_8||\varepsilon, \\
p_{19} \stackrel{\lambda}{\rightarrow} p_{20}, & p_{20} \stackrel{\lambda}{\rightarrow} p_{21'}||p_{23'}, & p_{21'} \stackrel{\lambda}{\rightarrow} p_{21}.e_0, \\
p_{23'} \stackrel{\lambda}{\rightarrow} p_{23}.e_1, & p_{23} \stackrel{\lambda}{\rightarrow} \varepsilon, & p_{21} \stackrel{\lambda}{\rightarrow} p_{10}.p_{22}, \\
p_{22} \stackrel{\lambda}{\rightarrow} \varepsilon, & e_0||e_1 \stackrel{\lambda}{\rightarrow} \varepsilon &
\end{array}
$$

If a component system requires another directive describing the implementation of the start procedures, then this case has to be added in this part of the algorithm.

Example 5.9 on page 93 visualizes a component-based software which behavior is representable by a PA-process. In Example 5.10 on page 94 the corresponding system abstraction Π_{S,C_2} is shown. In Example 5.8 a Process Algebra Net used as representation of the behavior according to Example 5.7 on page 84.

Remark
The decision, which component of an application could or will start, depends strongly on the component system. Thus, we need the information too, which component system is used to build the application.

Property 5.1
If several implementations of the same (provided) interface exists, the algorithm generates a transition rule for each interface. Thus, it is represented in the system abstraction that a component can be chosen dynamically from a given set of components.

Remark
Transition rules of the form $\delta \hat{=} p_1 \stackrel{a}{\rightarrow} q$ are possible because of optimizations (e. g., Section 5.2.3.1).

After this construction we get a Process Rewrite System.

Theorem 5.1 (Correctness of construction of system abstraction)
The system abstraction $\Pi_S{}'$ captures the behavior of the complete component-based software S and contains all possible remote procedure calls.

Algorithm 5.1: Construction of $\rightarrow_S$ (needed in Definition 5.3 on page 89)

input : Set $\mathfrak{C}_S = \{C_i : C_i = (Q_{C_i}, \Sigma_{C_i}, \rightarrow_{C_i}, R_{C_i}, P_{C_i}, \mathbb{M}_{C_i})$ is contained in S }

output : $\rightarrow_S$

1 $\rightarrow_S = \emptyset$
2 $\mathbb{M} = \bigcup_{C_i \in \mathfrak{C}_S} \mathbb{M}_{C_i}$
3 **forall** *component abstractions* $C_i \in \mathfrak{C}_S$ **do**
4   /* q_a identifies a required interface call a */
5   /* q_a identifies the provided interface a */
6   **forall** $\delta \in \rightarrow_{C_i}$ **do**
7   **switch** δ **do**
8   **case** $\delta \hat{=} p_1 \stackrel{a}{\rightarrow} q_a.p_2$
9         $\rightarrow_S = \rightarrow_S \cup \{p_1 \stackrel{a}{\rightarrow} p_a.p_2 \ : \ p_x \in \mathbb{M}(q_a)\}$
10   **case** $\delta \hat{=} p_1 \stackrel{a}{\rightarrow} q_a || p_2$
11         $\rightarrow_S = \rightarrow_S \cup \{p_1 \stackrel{a}{\rightarrow} p_x || p_2 \ : \ p_x \in \mathbb{M}(q_a)\}$
12   **case** $\delta \hat{=} p_1 \stackrel{a}{\rightarrow} q_a$
13         $\rightarrow_S = \rightarrow_S \cup \{p_1 \stackrel{a}{\rightarrow} p_x \ : \ p_x \in \mathbb{M}(q_a)\}$
14   **otherwise**
15         $\rightarrow_S = \rightarrow_S \cup \{\delta\}$
16   **end**
17   **end**
18   **end**
19 **end**
20 /* Generating start rules */
21 **if** C_i *is the main component/the client of* S **then**
22   $\rightarrow_S = \rightarrow_S \cup \{I_S \stackrel{\lambda}{\rightarrow} I_{C_i}\}$
23 **end**
24 **if** *a set of components* $\mathfrak{C}_S$ *of* S *exists, containing the components* C_i *that can start* **then**
25   $\rightarrow_S = \rightarrow_S \cup \{I_S \stackrel{\lambda}{\rightarrow} I_{C_i} \mid C_i \in \mathfrak{C}_S\}$
26 **end**
27 **if** *a set of components* $\mathfrak{C}_S$ *of* S *exists, containing the components* C_i *which can be processed independently* **then**
28   $\rightarrow_S = \rightarrow_S \cup \{I_S \stackrel{\lambda}{\rightarrow}_S I_{C_0} || I_{C_1} || ... || I_{C_n}\}$
29 **end**

Proof (Theorem 5.1)

Proof by contraction: We assume, that not the complete behavior of S is captured.

We have to consider the following cases:

1. A component abstraction Π_C is not contained in Π_S, but C is contained in S.
   This problem cannot appear as in line 3 all components of S are included.

2. An execution trace of S is not included within Π_S.
   Suppose that the abstraction of the component-based software Π_S is computed considering whitebox components (cf. Section 4.3). We assume, that a derivation exits for every execution trace: $I \stackrel{w}{\Rightarrow} \varepsilon$. We will prove that every derivation is also computable while considering the system abstraction ${\Pi_S}'$ computed from component abstractions Π_{C_i} using

Algorithm 5.1. We prove it by considering the length of the interaction sequences $w \in \Sigma^*$ *of* Π_S.

$|w| = 0$: *If no interaction is performed in* Π_S*, then the execution terminates within the initial component. The same execution is possible using* ${\Pi_S}'$.

$|w| = 1$: *If one interaction is performed a transition rule* $t \overset{a}{\rightarrow} t' \in \rightarrow_{\Pi_S}$ $(t, t' \in PEX(Q_{\Pi_S}))$ *is used while performing the following derivation:*

$$I \overset{\lambda}{\Rightarrow}_{\Pi_S} t'' \overset{a}{\Rightarrow}_{\Pi_S} t''' \overset{\lambda}{\Rightarrow}_{\Pi_S} \varepsilon, t'', t''' \in PEX(Q_{\Pi_S})$$

To generate a derivation containing a *an interaction has to be performed. Such an interaction is captured by transition rules of the form* $p_1 \overset{a}{\rightarrow} q_a \otimes p_2 \in \rightarrow_{{\Pi_S}'}$ (p_2 *can be* ε). *While performing Algorithm 5.1 such a transition rule is captured in line 8, 10 or 12. Hence, the considered derivation is computable in* ${\Pi_S}'$*, too.*

hypothesis: *The same interactions sequences* w *are computable in* Π_S *and* ${\Pi_S}'$ *(which is assembled from single component abstractions), if* $|w| = n$.

$|w| = n + 1$: *In* Π_S *the following derivation is computable:*

$$I \overset{w}{\Rightarrow} t'' \overset{a}{\Rightarrow} t''' \overset{\lambda}{\Rightarrow} \varepsilon, \text{ where } a \in \Sigma$$

Using the induction basis, it is clear that the interaction a *has to be represented by a corresponding transition rule of* ${\Pi_S}'$. □

We have proven that every interaction sequence, which is computable using a system abstraction Π_S *(which is created while considering whitebox components), is also computable using a system abstraction* ${\Pi_S}'$ *(which is created while assembling component abstractions).*

So, we have to eliminate every action, which is not included in the protocol $\mathcal{P}_{C_i}$ of the component C_i that should be checked currently. For this purpose we use Definition 4.11 on page 64 resulting in the Process Rewrite System Π_{S,C_i}.

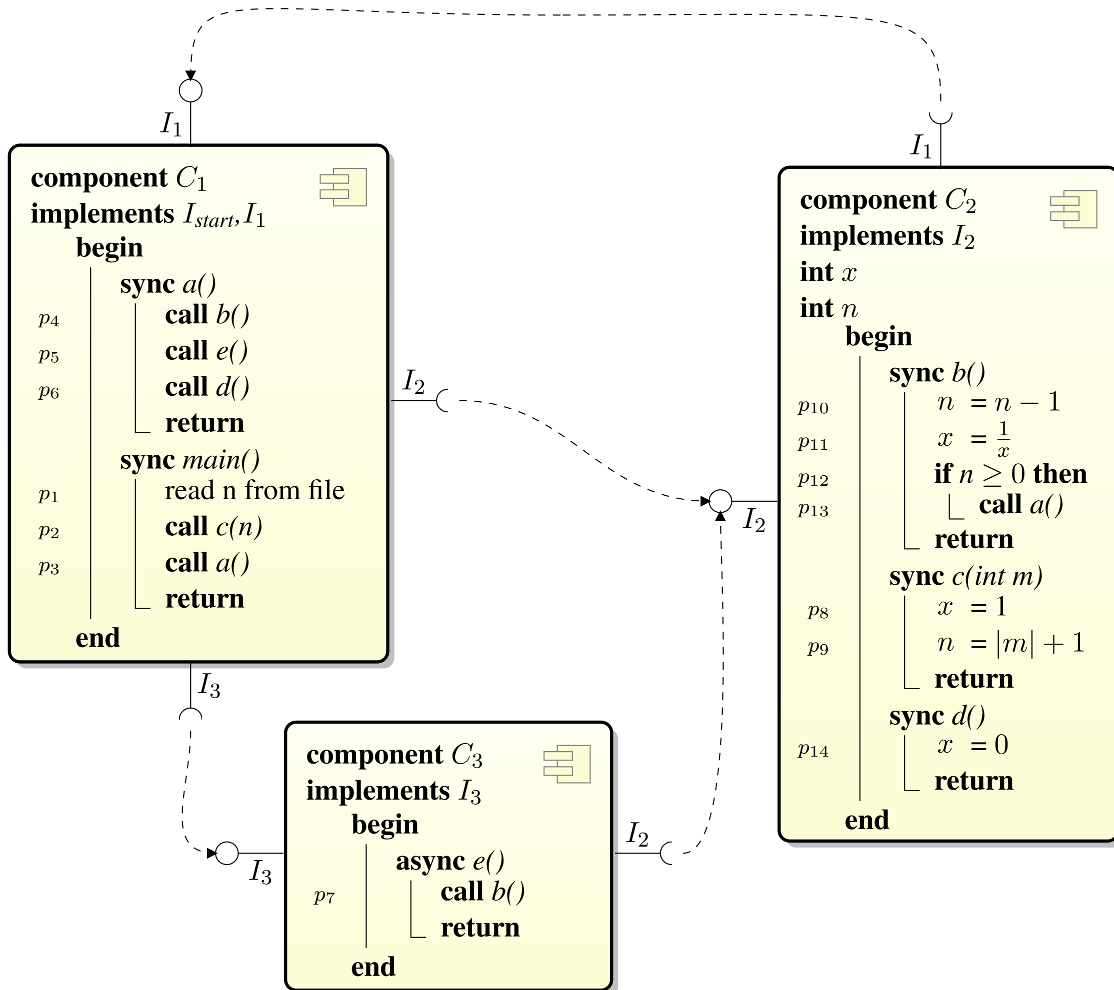
Property 5.2

If several protocols $\mathcal{P}_{C_i} \in \mathcal{P}_S$ *have to be checked, then as many* $\Pi_{S,C_i} = (Q, \Sigma_{S,C_i}, I, \rightarrow_{S,C_i}, \{\varepsilon\})$ *are created as protocols* $\mathcal{P}_{C_i} = (Q_{\mathcal{P}_{C_i}}, \Sigma_{\mathcal{P}_{C_i}}, I_{\mathcal{P}_{C_i}}, \rightarrow_{\mathcal{P}_{C_i}}, F_{\mathcal{P}_{C_i}})$ *exist. We have that:* $\Sigma_{S,C_i} = \Sigma_{\mathcal{P}_{C_i}}$.

Example 5.9: Component-based software with labeled program points.

```
interface I_start
begin
 | sync main()
end

interface I_1
begin
 | sync a()
end

interface I_2
begin
 | async e()
end

interface I_3
begin
 | sync b()
 | sync c()
 | sync d()
end
```

(a) Available interfaces of the component-based software.



(b) Architecture of the application.

$\mathcal{P}_{C_1} = a^*$ $\qquad$ $\mathcal{P}_{C_2} = e^*$ $\qquad$ $\mathcal{P}_{C_3} = c\, b^*\, d^*$

(c) Protocols of the composed components (as regular expressions).

Example 5.10: Rewrite rules of abstraction Π_{S,C_2} of the exemplary component-based software S according to the protocol of the component C_2 (cf. Example 5.9 on page 93).

$$
\begin{aligned}
\Pi_{S,C_2} &= (Q_{S,C_2}, \Sigma_{S,C_2}, I_{S,C_2}, \rightarrow_{S,C_2}, F_{S,C_2}) \text{ where} \\
Q_{S,C_2} &= \{I_S, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}\}, \\
\Sigma_{S,C_2} &= \{b, c, d\}, \\
I_{S,C_2} &= I_S, \\
\rightarrow_{S,C_2} &= \{ \\
& \begin{array}{llll}
I_S \overset{\lambda}{\rightarrow}_S p_1, & p_4 \overset{b}{\rightarrow}_S p_{10}.p_5, & p_8 \overset{\lambda}{\rightarrow}_S p_9, & p_{12} \overset{\lambda}{\rightarrow}_S p_{13}, \\
p_1 \overset{\lambda}{\rightarrow}_S p_2, & p_5 \overset{\lambda}{\rightarrow}_S p_7 || p_6, & p_9 \overset{\lambda}{\rightarrow}_S \varepsilon, & p_{12} \overset{\lambda}{\rightarrow}_S \varepsilon, \\
p_2 \overset{c}{\rightarrow}_S p_8.p_3, & p_6 \overset{d}{\rightarrow}_S p_{14}, & p_{10} \overset{\lambda}{\rightarrow}_S p_{11}, & p_{13} \overset{\lambda}{\rightarrow}_S p_4, \\
p_3 \overset{\lambda}{\rightarrow}_S p_4, & p_7 \overset{b}{\rightarrow}_S p_{10}, & p_{11} \overset{\lambda}{\rightarrow}_S p_{12}, & p_{14} \overset{\lambda}{\rightarrow}_S \varepsilon
\end{array} \\
& \}, \\
F_{S,C_2} &= \{\varepsilon\}
\end{aligned}
$$

5.2.2 Restrictions

Currently, this approach can only handle components which are known (component abstractions are accessible). These components can be bounded statically or dynamically. However, fully dynamically chosen (fully dynamically bound) components cannot captured in the abstraction, e. g., an imported unknown component, which is bounded while querying a dynamically chosen service repository. If the components are chosen by dynamic input (user input, predefined data), it is only possible to create a system containing all components, that are possible.

Remark

In [BZ09c] we have shown, how an extended verification process works dealing with component-based software with fully dynamically chosen components. Hence, the protocol conformance can be ensured in a fully dynamically context too. This approach is not described within this thesis.

5.2.3 Optimizations

In Section 5.1 we have shown that it is possible to generate Stripped Process Rewrite System abstractions of real source code. As we have shown, several approaches exist to perform this generation. An important issue is the tool support. However, just a poor tool support exists currently. Hence, we have to assume, that the computation of an abstraction might result in an inappropriate number of transition rules. This is caused by the fact, that we catch only the abstract behavior of the considered source code (e. g., generate a λ-rule for each statement, which influences only the data flow), while the abstraction process has to evaluate the complete source code to get a connected representation.

Example 5.11: Contracting λ-rules.

$$
\begin{aligned}
\text{given } \Pi =& (Q, \Sigma, I, \rightarrow, F) \text{ with}\\
Q =& \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}\}\\
\Sigma =& \{a, b, c, d\}\\
I =& p_1\\
\rightarrow =& \{p_1 \overset{\lambda}{\rightarrow} p_2,\ p_2 \overset{a}{\rightarrow} p_3.p_4,\ p_3 \overset{\lambda}{\rightarrow} p_7,\ p_4 \overset{b}{\rightarrow} p_5,\ p_5 \overset{\lambda}{\rightarrow} p_6,\\
& p_6 \overset{\lambda}{\rightarrow} \varepsilon,\ p_7 \overset{c}{\rightarrow} p_8.p_9,\ p_2 \overset{d}{\rightarrow} p_{10}.p_{11},\ p_{11} \overset{\lambda}{\rightarrow} \varepsilon\}\\
F =& \{\varepsilon\}
\end{aligned}
$$

$$
\begin{aligned}
\text{resulting } \Pi' =& (Q', \Sigma', I', \rightarrow', F') \text{ with}\\
Q' =& \{p_1, p_2, p_4, p_7, p_8, p_9, p_{10}\}\\
\Sigma' =& \{a, b, c, d\}\\
I' =& p_1\\
\rightarrow' =& \{p_1 \overset{\lambda}{\rightarrow} p_2,\ p_2 \overset{a}{\rightarrow} p_7.p_4,\ p_4 \overset{b}{\rightarrow} \varepsilon,\ p_7 \overset{c}{\rightarrow} p_8.p_9,\ p_2 \overset{d}{\rightarrow} p_{10}.\varepsilon,\ \}\\
F' =& \{\varepsilon\}
\end{aligned}
$$

The components of our verification architecture could be distributed (cf. Section 6.1). Thus, the abstractions have to be transferable in acceptable time. Moreover, it might be not model check an abstraction of an application with a high number of transition rules. Moreover, an abstraction containing a high number of transition rules might be not model checkable. The reason is the high effort that might occur, while performing the reachability analysis (cf. Section 5.4). Hence, an explosion of the number of transition rules should be avoided, if possible.

The statistics in Chapter 6 (case studies) show, that a significant number of transition rules are λ-rules. These rules are contained to steer the abstract behavior (control flow) or to connect parts of the source code. However, they do not represent interactions all interactons within the system. As the protocol is defined for the interactions with one component only,

Because of the high number of λ-rules, we derive the need for optimizations with the goal of reducing the transition rules. We focus on the λ-rules, as they do not influence the behavior representing relevant interactions.

5.2.3.1 Contracting λ-rules

By definition λ-rules have no relevant behavior except, that they describe rewriting sequences between action rules. Thus, we can reduce the number of λ-rules, since the semantics of the control-flow in the generated abstraction is obeyed. For this purpose we will define an algorithm, which calculates the Process Rewrite System Π' containing the contracted transition rules. An example is shown in Example 5.11.

We use the following auxiliary functions, which provide functionality to determine and replace process constants within process-algebraic terms of a Process Rewrite System $\Pi \widehat{=} (Q, \Sigma, I, \rightarrow, F)$.

- for calculating, which process constants a process-algebraic term contains:[8]

$$
\begin{aligned}
\mathsf{T}(\varepsilon) &= \{\{\}\} \\
\mathsf{T}(p) &= \{\{p\}\} \\
\mathsf{T}(p \otimes t) &= \mathsf{T}(p) \cup \mathsf{T}(t) \\
\text{where } & p \in Q \\
& t \in PEX(Q) \\
& \otimes \in \{||, .\}
\end{aligned}
$$

- for computing a term where a process constant p_1 is replaced by a constant p_2:

$$
\begin{aligned}
\mathsf{Rep}(p_1, p_2, \varepsilon) &= \varepsilon \\
\mathsf{Rep}(p_1, p_2, p_3) &= \begin{cases} p_2 : & p_3 = p_1 \\ p_3 : & p_3 \neq p_1 \end{cases} \\
\mathsf{Rep}(p_1, p_2, p_3 \otimes p_4) &= \mathsf{Rep}(p_1, p_2, p_3) \otimes \mathsf{Rep}(p_1, p_2, p_4) \\
\text{where } & p_1, p_2, p_3, p_4 \in Q \\
& \otimes \in \{||, .\}
\end{aligned}
$$

- for determining the number of transition rules using a constant p:

$$
\begin{aligned}
\#\mathsf{L}(p) &= |\{t_1 \xrightarrow{a} t_2 : \ t_1 \xrightarrow{a} t_2 \in \rightarrow \ \wedge p \in \mathsf{T}(t_1)\}| \\
\#\mathsf{R}(p) &= |\{t_1 \xrightarrow{a} t_2 : \ t_1 \xrightarrow{a} t_2 \in \rightarrow \ \wedge p \in \mathsf{T}(t_2)\}| \\
\#\mathsf{R}(\varepsilon) &= 1
\end{aligned}
$$

For bridging over transition rules, which have no influence on the derivations of the abstraction Π, we will compute a new set of transition rules where the old ones are replaced, without changing the semantics of the Process Rewrite System Π (right-hand side). The following example should show the idea:

$$
\begin{gathered}
p_1 \xrightarrow{\lambda} p_2, \ p_2 \xrightarrow{a} p_3, \ p_2 \xrightarrow{b} p_4 || p_5 \in \rightarrow_{\Pi} \\
\Downarrow \\
p_1 \xrightarrow{a} p_3, \ p_1 \xrightarrow{b} p_4 || p_5 \in \rightarrow_{\Pi'}
\end{gathered}
$$

We define the function TRBU to compute a new set of transition rules, while considering a λ-rule, which is also a chain rule, and the transition rules, using the process constant generated by

[8] Note, the considered Process Rewrite Systems is in normal form (cf. Definition 3.31).

Algorithm 5.2: Contraction of λ-rules.

input : Process Rewrite System $\Pi_C \hat{=} (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C, \mathbb{M}_C)$
output : Process Rewrite System $\Pi' \hat{=} (Q', \Sigma', \rightarrow', R', P', \mathbb{M}')$
1 **repeat**
2 $\quad \rightarrow\ = (\rightarrow \setminus \mathsf{TRBUR}(\Pi_C)) \cup \mathsf{TRBU}(\Pi_C)$
3 $\quad \rightarrow\ = (\rightarrow \setminus \mathsf{TRTDR}(\Pi_C)) \cup \mathsf{TRTD}(\Pi_C)$
4 **until** $\rightarrow$ *is not changed* ;
5 $\Pi' = \Pi_C$

the λ-rule:

$$\begin{aligned}\mathsf{TRBU}(\Pi) = \{ \quad \mathsf{Rep}(p_0, p_1, t) \xrightarrow{a} t' \quad | \quad & (p_0 \xrightarrow{\lambda} p_1) \in \rightarrow\ \wedge \\ & p_0, p_1 \in Q \wedge\ p_0 \neq p_1 \wedge\ p_0\ \neq I \wedge \\ & t \xrightarrow{a} t' \in \rightarrow\ \wedge \\ & p_1 \in \mathsf{T}(t) \wedge \\ & \#\mathsf{L}(p_0) = 1 \wedge \\ & \#\mathsf{R}(p_1) = 1 \quad \}\end{aligned}$$

As it is also possible that a contraction is only possible in the reverse way, e. g.,

$$p_1 \xrightarrow{a} p_2 || p_4,\ p_3 \xrightarrow{b} p_4,\ p_4 \xrightarrow{\lambda} p_5 \in \rightarrow_\Pi$$
$$\Downarrow$$
$$p_1 \xrightarrow{a} p_2 || p_5,\ p_3 \xrightarrow{b} p_5 \in \rightarrow_{\Pi'}$$

we also define a transformation, where the λ-rule follows:

$$\begin{aligned}\mathsf{TRTD}(\Pi) = \quad \{ \quad t \xrightarrow{a} \mathsf{Rep}(p_4, p_5, t') \quad | \quad & (p_4 \xrightarrow{\lambda} p_5) \in \rightarrow\ \wedge \\ & p_4, p_5 \in Q \wedge\ p_4 \neq p_5\ \wedge\ p_4\ \neq I \wedge \\ & t \xrightarrow{a} t' \in \rightarrow\ \wedge \\ & p_4 \in \mathsf{T}(t') \wedge \\ & \#\mathsf{L}(p_4) = 1 \wedge \\ & \#\mathsf{R}(p_5) = 1 \quad \}\end{aligned}$$

To calculate the set of the transition rules, that are contracted currently, we use the functions defined in Figure 5.4. Now, it is possible to define the contraction Algorithm 5.2. Transition rules are eliminated, if they are replaced by a transition rule calculated by $\mathsf{TRBU}(\Pi)$ or $\mathsf{TRTD}(\Pi)$. It is clear, that no new transition rules are added, TRBU and TRTD replace transition rules.

The algorithm works in the expected way, if the same words w (interaction sequences) can be constructed using Π and Π'.

$$
\begin{array}{rlll}
\mathsf{TRBUR}(\Pi) = & \{ & p_0 \xrightarrow{\lambda} p_1 \mid & (p_0 \xrightarrow{\lambda} p_1) \in \rightarrow \ \wedge \\
& & & p_0, p_1 \in Q \wedge p_0 \neq p_1 \ \wedge \ p_0 \ \neq I \wedge \\
& & & t \xrightarrow{a} t' \in \rightarrow \ \wedge \\
& & & p_1 \in \mathsf{T}(t) \wedge \\
& & & \#\mathsf{L}(p_0) = 1 \wedge \\
& & & \#\mathsf{R}(p_1) = 1 \ \ \} \\
\cup & \{ & t \xrightarrow{a} t' | & (p_0 \xrightarrow{\lambda} p_1) \in \rightarrow \ \wedge \\
& & & p_0, p_1 \in Q \wedge p_0 \neq p_1 \ \wedge \ p_0 \ \neq I \wedge \\
& & & t \xrightarrow{a} t' \in \rightarrow \ \wedge \\
& & & p_1 \in \mathsf{T}(t) \wedge \\
& & & \#\mathsf{L}(p_0) = 1 \wedge \\
& & & \#\mathsf{R}(p_1) = 1 \ \ \} \\
\mathsf{TRTDR}(\Pi) = & \{ & p_4 \xrightarrow{\lambda} p_5 \mid & (p_4 \xrightarrow{\lambda} p_5) \in \rightarrow \ \wedge \\
& & & p_4, p_5 \in Q \wedge \ p_4 \neq p_5 \ \wedge \ p_4 \ \neq I \wedge \\
& & & t \xrightarrow{a} t' \in \rightarrow \ \wedge \\
& & & p_4 \in \mathsf{T}(t') \wedge \\
& & & \#\mathsf{L}(p_4) = 1 \wedge \\
& & & \#\mathsf{R}(p_5) = 1 \ \ \} \\
\cup & \{ & t \xrightarrow{a} t' \mid & (p_4 \xrightarrow{\lambda} p_5) \in \rightarrow \ \wedge \\
& & & p_4, p_5 \in Q \wedge \ p_4 \neq p_5 \ \wedge \ p_4 \ \neq I \wedge \\
& & & t \xrightarrow{a} t' \in \rightarrow \ \wedge \\
& & & p_4 \in \mathsf{T}(t') \wedge \\
& & & \#\mathsf{L}(p_4) = 1 \wedge \\
& & & \#\mathsf{R}(p_5) = 1 \ \ \}
\end{array}
$$

Figure 5.4.: Calculation of optimized transition sets.

Theorem 5.2 (Optimization is correct)
If $t \overset{w}{\Rightarrow}_{\Pi} t'$, then $t \overset{w}{\Rightarrow}_{\Pi'} t'$.

Proof (Theorem 5.2)
We divide the proof into the consideration of the first and the second step. Induction on the number of replacements $n \in \mathbb{N}$:

$n = 0:$ *if no transition rule is replaced, then* $t \stackrel{w}{\Rightarrow}_{\Pi} t'$ *and* $t \stackrel{w}{\Rightarrow}_{\Pi'} t'$ *are equal.*

$n \geq 1:$ *suppose that* $t \stackrel{w}{\Rightarrow}_{\Pi} t'$ *can be decomposed as the following:*

$$t \stackrel{w'}{\Rightarrow}_{\Pi} \underbrace{t'' \stackrel{\lambda}{\Rightarrow}_{\Pi} t'''}_{\substack{\text{one}\\ \lambda\text{-rule}\\ \text{replaced}}} \underbrace{\stackrel{w''}{\Rightarrow}_{\Pi} t'}_{\substack{n-1\\ \lambda\text{-rules}\\ \text{replaced}}}$$

, when $t \stackrel{w'}{\Rightarrow}_{\Pi} t''$ *does not contain a replaced transition rules,*

where $w = w' \cdot w'',\ w, w', w'' \in \Sigma^*, p_1 \stackrel{\lambda}{\rightarrow} p_2 \in \mathsf{TRBUR}(\Pi), p_1 \in \mathsf{T}(t''), p_2 \in \mathsf{T}(t''')$

by induction hypothesis $t''' \stackrel{w''}{\Rightarrow}_{\Pi'} t'$

The second step (line 3) can be proven analogously. □

Corollary 5.1

The result of the optimized Process Rewrite System Π' *has a less or equal number of rules as the given Process Rewrite System* Π*. Formally:*

$$| \rightarrow_{\Pi'} | \leq | \rightarrow_{\Pi} |$$

5.2.4 Summary

In this section we have shown, how we can compose single component abstractions to a system abstraction. A system abstraction $\Pi_{S,C}$ describes the behavior of a complete application S, but nevertheless contains only interactions to the component C.

Using the approach in the current section we are also able to compose single component abstractions to system abstractions (also named application abstractions). The approach is composable like the component system, every component has its own abstraction. It is also able to capture the dynamically binding of components. It contains every possibility how components can be composed. The system abstraction can be composed according to the definitions of the system developer too. In this case a more precise system abstraction is needed.

Note, in this step several system abstractions (one for each component protocol) are generated. This enables optimizations. An optimization reduces the number of transition rules if a λ-rule exists, which can be contracted. Thus, it compensates lacks of the abstraction process (e. g., where for each statement a process constant is created and thereafter connected with λ-rules), too. The results of the case studies (cf. Chapter 6) show the impact of this specific approach. Thus, we can revert to smaller system abstractions in the next chapter.

Moreover, it separates the problem for each component. Hence, the properties of the application are not checked globally. They are checked for each component independently and individually. We assume, that this will help the component developer as he has to tackle only the problems appearing in his component (and does not have to care about the behavior of components he cannot control).

Example 5.12: Erroneous execution trace (derivation) in Example 4.5 on page 65 considering component C_2 only, based on the cactus stack translations in Example 4.4 on page 60, using the transition rules in Example 5.3 on page 78. In contrast to the derivation in Example 5.4 on page 79 only interactions with component C_2 are included here (however, the complete application abstraction is considered).

$$
\begin{array}{llrl}
\underline{p_{s0}} & \stackrel{\lambda}{\Longrightarrow} & p_{10} \,||\, \underline{p_{s1}} & \text{within component } C_{\text{start}} \\
 & \stackrel{\lambda}{\Longrightarrow} & p_{10} \,||\, \underline{p_{s2}} & \text{within component } C_{\text{start}} \\
 & \stackrel{\lambda}{\Longrightarrow} & p_{10} \,||\, (\underline{p_0}.p_{s3}) & \text{within component } C_{\text{start}} \\
 & \stackrel{\lambda}{\Longrightarrow} & p_{10} \,||\, (\underline{p_1}.p_{s3}) & \text{within component } C_1 \\
 & \stackrel{set}{\Longrightarrow} & p_{10} \,||\, (\underline{p_{20}}.p_2.p_{s3}) & \text{within component } C_1 \\
 & \stackrel{\lambda}{\Longrightarrow} & \underline{p_{10}} \,||\, (\varepsilon.p_2.p_{s3}) & \text{within component } C_2 \\
 & \stackrel{\lambda}{\Longrightarrow} & \underline{p_{11}} \,||\, (p_2.p_{s3}) & \text{within component } C_0 \\
 & \stackrel{reset}{\Longrightarrow} & (\underline{p_{22}}.p_{12}) \,||\, (p_2.p_{s3}) & \text{within component } C_0 \\
 & \stackrel{\lambda}{\Longrightarrow} & (\varepsilon.p_{12}) \,||\, (\underline{p_2}.p_{s3}) & \text{within component } C_2 \\
 & \stackrel{calc}{\Longrightarrow} & p_{12} \,||\, (\underline{p_{21}}.p_3.p_{s3}) & \text{within component } C_1 \\
 & \stackrel{\lambda}{\Longrightarrow} & p_{12} \,||\, (\varepsilon.\underline{p_3}.p_{s3}) & \text{within component } C_2 \\
 & \stackrel{\lambda}{\Longrightarrow} & \underline{p_{12}}||(\varepsilon.p_{s3}) & \text{within component } C_1 \\
 & \stackrel{\lambda}{\Longrightarrow} & \varepsilon||\underline{p_{s3}} & \text{within component } C_0 \\
 & \stackrel{\lambda}{\Longrightarrow} & \varepsilon & \text{within component } C_{\text{start}}
\end{array}
$$

5.3 Step 3: Creating Combined Abstractions

In the previous section we computed a representation of the behavior of the full application S for each protocol P_C: the system abstraction $\Pi_{S,C}$. In this section we will encode the protocol conformance problem, so it is solvable by using reachability analysis[9].

Earlier we motivated that a component developer can evaluate protocol violations only, if they involve the protocol P_C of "his" component C. Thus, it is the goal to inform the component developer about the problems with "his" protocol $\mathcal{P}_{C_i}$ only. Based on the counterexample represented informally in Example 5.4 on page 79 , we want to provide a counterexample containing the interactions with the considered component only (see Example 5.12).

For this reason we compute the *Combined Abstraction* $\Pi_S^{C_i}$ of a system abstraction Π_{S,C_i} and a protocol $\mathcal{P}_{C_i}$. This representation encodes the model checking problem, which we have to solve. It enables us to find sequences of interactions not allowed by the protocol $\mathcal{P}_{C_i}$.

We will discuss the properties of a $(1, P)$-PRS (PA-processes) and (P, G)-PRS (Process Al-

[9] Note, only reachability is defined for Process Algebra Nets (cf. Section 3.2.4).

gebra Nets) in relation to Combined Abstractions. In Section 5.3.2 we present the result of optimizations considering Combined Abstractions.

5.3.1 Process

5.3.1.1 Discussion of the Model Checking Problem

To verify the component-based software abstraction with respect to a component C_i, we have to check, whether $L(\Pi_{S,C_i}) \subseteq L(\mathcal{P}_{C_i})$, where $L(\mathcal{P}_{C_i})$ is the regular language described by the protocol $\mathcal{P}_{C_i}$ of component C_i, and $L(\Pi_{S,C_i})$ is the language over the actions in Π_{S,C_i}, specifying a superset of the use of C_i (cf. Definition 4.2 on page 56). In order to check $L(\Pi^i_S) \subseteq L(P_i)$ it is possible to check the equivalent problem $L(\Pi^i_S) \cap \overline{L(P_i)} = \emptyset$, where $\overline{L(\mathcal{P}_{C_i})}$ is the language described by the inverted FSM $\overline{\mathcal{P}_{C_i}}$ (cf. Definition 3.24 on page 42). Unfortunately, this question is undecidable.

Theorem 5.3 (Undecidability of protocol conformance checking problem)
It is undecidable if $L(\Pi) \subseteq L(P)$, where Π is a $(1, G)$-PRS and $L(P)$ is a regular language.

Proof
We can build a Büchi automaton which accepts all and only the infinite traces represented by a linear time logic (LTL) formula ϕ [WVS83, VW94, LP85]. A finite state machine P can be constructed efficiently [KMMP93, GPVW95, DGV99] so that $L(\phi) = L(P)$, where $L(\phi)$ is the set of action sequences specified by ϕ. Thus if the protocol checking problem would be decidable, we could also decide LTL-formula model checking for $(1, G)$-PRS. Contradiction, because LTL is undecidable in $(1, G)$-PRS PA-processes [BH96]. Therefore, it is also undecidable for (P, G)-PRS Process Algebra Nets. □

Therefore, we construct a $(1, G)$-PRS K which describes a language $L(K)$, where $L(\Pi_{S,C_i}) \cap \overline{L(\mathcal{P}_{C_i})} \subseteq L(K)$. We call K *Combined Abstraction*. Thus, $L(K) = \emptyset$ implies $L(\Pi_{S,C_i}) \subseteq L(\mathcal{P}_{C_i})$. However, there might be a sequence $w \in L(K)$ so that $w \notin L(\Pi_{S,C_i}) \cap \overline{L(\mathcal{P}_{C_i})}$.

Remark
We call these sequences spurious false negatives*. Details about false negatives follow in Section 5.4.1.*

Corollary 5.2 (Theorem 5.3)
The protocol conformance checking is undecidable for Process Algebra Nets too, because (P, G)-PRS are a superclass of $(1, G)$-PRS PA-processes.

Convention 5.2 (Notation of Combined Abstraction)
To mark, which Π_{S,C_i} and which protocol $\mathcal{P}_{C_i}$ are combined, we notate the Combined Abstraction with $\Pi^{C_i}_S$.

5.3.1.2 Construction of the Combined Abstraction

In the following, we present the construction of the Combined Abstraction Π_S^C. A combination of an inverted protocol $\overline{P_C}$ (FSM) and an abstraction $\Pi_{S,C}$ (PA) was to our knowledge defined for the first time in our scientific paper [BZ08b]. The extension to Process Algebra Nets was to our knowledge defined firstly in our paper [BZ08a]. We will discuss here the construction considering Process Algebra Nets only, as this construction is backward-compatible with the one of PA-processes.

Roughly spoken, the Combined Abstraction encodes in one model Π_S^C the parallel execution paths of the abstraction of the considered system abstraction $\Pi_{S,C}$ and the execution paths (which are forbidden by the considered protocol P_C of C) formulated as finite state machine $\overline{P_C}$.

Definition 5.4 (Combined Abstraction)
The Combined Abstraction is a Process Rewrite System $\Pi_S^C = (Q_S^C, \Sigma_S^C, I_S^C, \rightarrow_S^C, F_S^C)$ computed from an inverted protocol $\overline{P_C}$ and a system abstraction $\Pi_{S,C}$. It is defined as follows:

$Q_S^C = Q_{\overline{P_C}} \times Q_{S,C} \times Q_{\overline{P_C}}$	*is a finite set of processes,*
$\Sigma_S^C = \Sigma_{\overline{P_C}}$	*is a finite set of atomic actions,*
$I_S^C \in Q_S^C$	*is a start process,*
$\rightarrow_S^C \subseteq Q_S^C \times \Sigma_S^C \times Q_S^C$	*is a finite set of transition rules,*
$F_S^C \subseteq Q_S^C$	*is a finite set of final processes.*

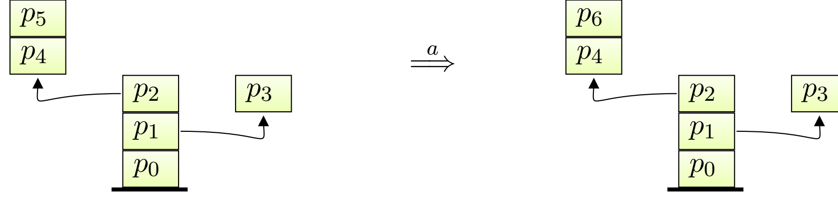
We restrict this definition to system Abstractions represented as Process Algebra Nets or a subclass of Process Algebra Nets.

In accordance with [HU79] a process $(v_i, p_j, v_k) \in Q_S^C$ encodes, that the FSM $\overline{P_C}$ is in the state v_i while the system abstraction $\Pi_{S,C}$ has the process constant p_j created. The aim of (v_i, p_j, v_k) is, that $v_i \overset{x}{\Rightarrow}_{\overline{P_C}} v_k$ while $p_j \overset{x}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ can be performed. Thus, a process constant of the Combined Abstraction Π_S^C encodes the current state of the considered FSM $\overline{P_C}$ and the system abstraction $\Pi_{S,C}$. We visualize this in Example 5.13, where the current state (cactus stack) of $\Pi_{S,C}$ and the inverted protocol $\overline{P_C}$ after the interaction sequence a is shown. In Figure 5.13a the transformation of a cactus stack is visualized (using the transition rule $p_5 \overset{a}{\rightarrow} p_6$). The corresponding process-algebraic expressions before and after performing interaction a are presented in Figure 5.13b. During this interaction the protocol has to switch accordingly the state using the transition rule $v_1 \overset{a}{\rightarrow} v_2$ (Figure 5.13c). To represent this corresponding a transition rule $((v_1, p_5, v_\zeta) \overset{a}{\rightarrow} (v_2, p_6, v_\zeta)) \in \rightarrow_S^C$ is created[10]. It encodes a simultaneous translation within the abstraction and the protocol performing a (cf. Figure 5.13d).

The transition rules of the Combined Abstraction Π_S^C have the same form and semantics as transition rules of regular Process Rewrite Systems.

[10]The construction of the last entry of the triple – here v_ζ is described in Figure 5.5 on page 104 . Here, it is present

Example 5.13: Visualization of the encoded state of the Combined Abstraction.



(a) Transformation of cactus stack.

$$(p_3||(((\underline{p_5}.p_4)||p_2).p_1)).p_0 \quad \stackrel{a}{\Longrightarrow} \quad (p_3||(((p_6.p_4)||p_2).p_1)).p_0$$

(b) Derivation of the abstraction $\Pi_{S,C}$.

$$v_1 \quad \stackrel{a}{\Longrightarrow} \quad v_2$$

(c) Derivation of the protocol $\overline{P_C}$.

$$(v_1, p_5, v_\zeta) \quad \stackrel{a}{\rightarrow} \quad (v_2, p_6, v_\zeta)$$

(d) Combined transition rule.

The construction of the transition rule $\rightarrow_S^C$ follows the directives shown in Figure 5.5 (described below) and is a generalization of the standard construction of the intersection of a finite state machine and a push-down automaton. Thus, the chain transition rules $\mathcal{R}_C^1$ and the set of sequential transition rules $\mathcal{R}_S^1$ are handled similar to create an intersection of a push-down automaton and a finite state machine in [HU79]. That means, the transition rule changes the state of the inverted protocol as well as the state of system abstraction simultaneously, if corresponding transition rules exist within $\overline{P_C}$ and $\Pi_{S,C}$.

The transition rules with a parallel operator $\mathcal{R}_P^1$ are constructed similar to $\mathcal{R}_C^1$. It has the same meaning: the state of the system abstraction is changed (in this case, while forking a new process), while the inverted protocol performs the same action a and changes the state, too. The same principle is used to represent the semantics of elimination rules of the system abstraction as $\mathcal{R}_{Elem}^1$ is constructed. Thereby, the middle entry of the triple (representing a state of the system abstraction) is transformed into the empty word.

We also have to deal with λ-rules, that can appear in the system abstraction $\Pi_{S,C}$. These rules perform an action, that is not relevant for the currently considered inverted protocol $\overline{P_C}$. The state of the protocol has not changed while performing a λ-rule within the system abstraction. Thus, the transition rules of the sets $\mathcal{R}_C^\lambda$, $\mathcal{R}_{Elem}^\lambda$, $\mathcal{R}_{Seq}^\lambda$, $\mathcal{R}_{Fork}^\lambda$ and $\mathcal{R}_{Sync}^\lambda$ transform the state of the Combined Abstraction, but change only the state p (middle entry of the triple) of the system

to show a sound definition.

$$
\begin{aligned}
\mathcal{R}^1_C &= \{(v,p,v'') \xrightarrow{a} (v',p',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p') && \}\\
\mathcal{R}^\lambda_C &= \{(v,p,v') \xrightarrow{\lambda} (v,p',v') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p') && \}\\
\mathcal{R}^1_{Elem} &= \{(v,p,v'') \xrightarrow{a} (v',\varepsilon,v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} \varepsilon) && \}\\
\mathcal{R}^\lambda_{Elem} &= \{(v,p,v') \xrightarrow{\lambda} (v,\varepsilon,v') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} \varepsilon) && \}\\
\mathcal{R}^1_{Seq} &= \{(v,p,v''') \xrightarrow{a} (v',p',v'').(v'',p'',v''') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'.p'') && \}\\
\mathcal{R}^\lambda_{Seq} &= \{(v,p,v'') \xrightarrow{\lambda} (v,p',v').(v',p'',v'') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'.p'') && \}\\
\mathcal{R}^1_{Fork} &= \{(v,p,v'') \xrightarrow{a} (v',p',v'')||(v',p'',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'||p'') && \}\\
\mathcal{R}^\lambda_{Fork} &= \{(v,p,v') \xrightarrow{\lambda} (v,p',v')||(v,p'',v') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'||p'') && \}\\
\mathcal{R}^1_{Sync} &= \{(v,p,v'')||(v,p',v'') \xrightarrow{a} (v',p'',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p||p' \xrightarrow{a}_{\Pi_{S,C}} p'') && \}\\
\mathcal{R}^\lambda_{Sync} &= \{(v,p,v')||(v,p',v') \xrightarrow{\lambda} (v,p'',v') : && (p||p' \xrightarrow{\lambda}_{\Pi_{S,C}} p'') && \}\\
\mathcal{R}^0 &= \{(v,p,v'') \xrightarrow{\lambda} (v',p,v'') : && (v \xrightarrow{a}_{P_C} v') && \}\\
\mathcal{R}^\varepsilon &= \{(v,\varepsilon,v) \xrightarrow{\lambda} \varepsilon && && \}
\end{aligned}
$$

$$
\begin{aligned}
\text{with } & v,v',v'',v''' \in Q_{P_C},\\
& p,p',p'' \in Q_{S,C},\\
& a \in \Sigma_{P_C} \wedge a \neq \lambda
\end{aligned}
$$

Figure 5.5.: Directives for construction of transition rules of a Combined Abstractions Π^C_S.

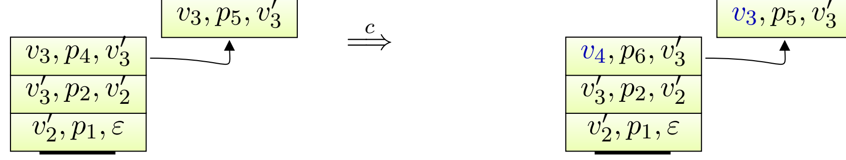
abstraction. The state of the inverted protocol v is unaffected.

The transition rules of the set $\mathcal{R}^\varepsilon$ are created to simplify the model checking. Using these rules semantics is represented explicitly, that the aimed protocol state v is reached (as first and third entry of the triple), while meanwhile the state corresponding process constant of the system abstraction is translated into the empty word ε (second entry). Thus this process stopped after performing a sequence of interactions, while the protocol performs a corresponding sequence of interactions. Hence, we can remove this process constant. Formally:

$$
p \overset{*}{\Rightarrow} \varepsilon \ \wedge \ v \overset{*}{\Rightarrow} v' \quad \Rightarrow \quad (v,p,v) \overset{*}{\Rightarrow} (v',\varepsilon,v') \xrightarrow{\lambda} \varepsilon
$$

During the transformation a problem can appear, while considering concurrent processes (e. g., $(v_3,p_5,v_3')||(v_3,p_4,v_3')$). While applying a transition rule to a process constant (v_3,p_4,v_3') the protocol state might change (e. g., to (v_4,p_6,v_3')). The second process (v_3,p_5,v_3') cannot recognize this protocol state, thus it remains in the protocol state v_3. Thus in the resulting process-algebraic term (e. g., $(v_3,p_5,v_3')||(v_4,p_6,v_3')$) two different protocol states are encoded, but the protocol can only be in one state. This situation is shown in Example 5.14, the different (and problematic) protocol states are marked. To deal with this problem we have to make it possible, that concurrent processes can synchronize their protocol state. This synchronization cannot be done explicitly, while the concurrent processes cannot communicate via a global communication

Example 5.14: Problematic situation while dealing with a Combined Abstraction Π_S^C.



(a) Transformation of a cactus stack.

$$
\begin{aligned}
\underline{(v_1, p_1, \varepsilon)} &\overset{a}{\Rightarrow} \underline{(v_2, p_2, v_2')}.(v_2', p_1, v_2) \\
&\overset{b}{\Rightarrow} \underline{(v_3, p_3, v_3')}.(v_3', p_2, v_2').(v_2', p_1, v_2) \\
&\overset{\lambda}{\Rightarrow} ((v_3, p_5, v_3')||\underline{(v_3, p_4, v_3')}).(v_3', p_2, v_2').(v_2', p_1, v_2) \\
&\overset{c}{\Rightarrow} ((\underset{\uparrow}{v_3}, p_5, v_3')||(\underset{\uparrow}{v_4}, p_6, v_3')).(v_3', p_2, v_2').(v_2', p_1, v_2)
\end{aligned}
$$

(b) Corresponding derivation of the Combined Abstraction.

channel (not allowed in PRS semantics). So, we will create transition rules allowing to imitate the adjustment of the protocol states at any time (this is an overapproximation of the behavior). These transition rules – named sleep rules – are contained.

In the example of Example 5.14 the following action rule was used in the fourth step:

$$(v_3, p_4, v_3') \overset{c}{\to} (v_4, p_6, v_3') \in \mathcal{R}_C^1$$

It is created using the transition rules $(v_3 \overset{a}{\to} v_4) \in \to_{P_C}$ and $(p_4 \overset{a}{\to} p_6) \in \to_{S,C}$. Therefore, the requirements are fulfilled to generate a transition rule $((v_3, p_5, v_3') \overset{\lambda}{\to} (v_4, p_5, v_3')) \in \mathcal{R}^0$. Applying this transition rule in the situation represented in the example leads to the imitation of a protocol change in the branch. Hence, the parallel processes can be justified in relation to their current protocol states.

Using these rules it is ensured, that each concurrent process of a process-algebraic expression of a Combined Abstraction can be transformed into the same protocol state if another process performs an interaction. The downside for this imprecise behavior is an overapproximation and thus false negatives (while considering model checking, cf. Section 5.4).

The set of transition rules $\to_S^C$ by unifying the sets of transition rules presented in Figure 5.5. Formally:

$$\to_S^C \hat{=} \mathcal{R}_C^1 \cup \mathcal{R}_C^\lambda \cup \mathcal{R}_{Elem}^1 \cup \mathcal{R}_{Elem}^\lambda \cup \mathcal{R}_{Seq}^1 \cup \mathcal{R}_{Seq}^\lambda \cup \mathcal{R}_{Fork}^1 \cup \mathcal{R}_{Fork}^\lambda \cup \mathcal{R}^0 \cup \mathcal{R}^\varepsilon$$

After constructing the transition rules of the Combined Abstraction, we obtain a Process Rewrite System $\Pi_S^C = (Q_S^C, \Sigma_S^C, I_S^C, \to_S^C, F_S^C)$ (we also call $\Pi_S^{C_i}$ *interleaving PRS*).

Lemma 5.1
By the construction of the Combined Abstraction a Process Rewrite System Π_S^C is computed which is in the same class of the PRS-hierarchy (Section 3.2.3) as the Process Rewrite System $\Pi_{S,C}$.

Proof (Lemma 5.1)
By the construction directives (Figure 5.5) the generated rules of Π_S^C are of the same kind as the considered transition rule of $\Pi_{S,C}$. Considering the transition rules of the system abstraction $\Pi_{S,C}$ resulting in a transition rule of Π_S^C we get the following mapping:

- *a chain rule of $\Pi_{S,C}$ results in a chain rule of Π_S^C implied by of $\mathcal{R}_C^1$ and $\mathcal{R}_C^\lambda$,*
- *a sequential rule of $\Pi_{S,C}$ results in a sequential rule of Π_S^C by reason of $\mathcal{R}_{Seq}^1$ and $\mathcal{R}_{Seq}^\lambda$,*
- *a fork rule of $\Pi_{S,C}$ results in a fork rule of Π_S^C by reason of $\mathcal{R}_{Fork}^1$ and $\mathcal{R}_{Fork}^\lambda$,*
- *a synchronization rule of $\Pi_{S,C}$ results in a synchronization of Π_S^C by reason of $\mathcal{R}_{Sync}^1$ and $\mathcal{R}_{Sync}^\lambda$,*

Moreover, $\mathcal{R}_{Elem}^1$, $\mathcal{R}_{Elem}^\lambda$, $\mathcal{R}^0$ and $\mathcal{R}^\varepsilon$ contain only chain transition rules, thus the PRS class of Π_S^C cannot be raised in comparison to the PRS class of $\Pi_{S,C}$. Hence, Π_S^C is contained in the same class of the PRS-hierarchy as $\Pi_{S,C}$. Because the transition rules determine the PRS class (cf. Definition 3.31). □

Remark
As the directives (cf. Figure 5.5 on page 104) for creating the Combined Abstraction compute a Process Rewrite System of the same PRS class as $\Pi_{S,C}$, the operation is closed.

Now, every possible interleaving sequence of the actions contained in the protocol is represented by at least one path in the Combined Abstraction Π_S^C. As we can easily see, using the explained construction, we create false negatives, too. These will be described in Section 5.4.1.

To show, that the presented construction is correct, we prove the following theorem.

Theorem 5.4 (Correctness of construction of the Combined Abstraction Π_S^C)
The construction of a Combined Abstraction Π_S^C results in a representation, so that $L(P_C) \cap L(\Pi_{S,C}) \subseteq L(\Pi_S^C)$.

Proof (Idea)
Counterexamples constructable only by sequential rules are gathered by using the rewrite rules of $\mathcal{R}_C^1$ and $\mathcal{R}_S^1$. Without loss of generality we only have to consider action rules. If a counterexample is computable while using parallel rules, we have to look at terms of the form $(p_1||p_2).p_p$. Using the construction directive computing $\mathcal{R}_{Fork}^1$ the parallel traces are calculated independently.
E. g., if a transition rule out of $\mathcal{R}_C^1$ is applied to p_1, then this trace reaches a new state v' in the inverted protocol automaton. However, in p_2 the protocol automaton has still the old state. With

Algorithm 5.3: Lazy construction of the Combined Abstraction

```
   input   : protocol P_C, system abstraction Π_{S,C} of component C
   output  : Combined Abstraction Π_S^C = (Q_S^C, Σ_S^C, I_S^C, →_S^C, F_S^C)
1  ‾P_C‾    = invert protocol P_C
2  →_start = generate start rules as described in Figure 5.5 on page 104
3  Q_unres = ∅
4  foreach transition rule δ in →_start do
5  |   add process constants of RHS of δ to Q_unres
6  end
7  foreach node (v, p, v) ∈ Q_unres do
8  |   Q_tmp  = ∅
9  |   generate set of transition rules R_new as described in Figure 5.5 on page 104 such
   |   that (v, p, v) is contained in the process constants of the RHS of each transition rule
10 |   foreach transition rule δ ∈ R_new do
11 |   |   foreach process constant (v, p, v)' contained in the RHS of δ do
12 |   |   |   add (v, p, v)' to Q_tmp
13 |   |   end
14 |   end
15 |   →_S^C  = →_S^C ∪ R_new
16 |   Q_unres = Q_tmp
17 end
```

a rule from $\mathcal{R}_0$, it is possible that p_2 reaches v', too, while using a rule out of $\mathcal{R}_0$. A extended approach is presented in Appendix A.4.

Lemma 5.2 (The definition of the construction is backward compatible)
The Definition 5.4 on page 102 (Figure 5.5 on page 104) of the Combined Abstraction is applicable to Process Algebra Nets. Moreover, it is applicable to the subclasses of Process Algebra Nets in the PRS-hierarchy, because of the consideration of each transition rule class.

Proof (Lemma 5.2)
The construction of the transition rules of a Combined Abstraction is partitioned into construction directives for each type of a transition rules (Definition 3.31 on page 49). Therefore, the irrelevant directives can be omitted, if the considered PRS representation does not allow the corresponding kind of transition rules.

Remark
The construction considering $(1, S)$-PRS is compatible with the definition of an intersection of push-down automata and finite state machines in [HU79].

5.3.2 Optimizations of the Combined Abstractions

5.3.2.1 Optimizations during Creation

We use Algorithm 5.3 to generate the transition rules of the Combined Abstraction. It implements a lazy construction as in every step the new process constants are considered only.

After the computation of the start transition rules, the algorithm computes in each step only transition rules, which contain a specific term at the RHS. The set $Q_{\text{unres}} \subseteq Q_S^C$ contains all process constants, which could be generated using the existing transition rules and never contained of the LHS of any existing transition rule. This ensures a lazy computation of the transition rules, because only rules are computed, which possibly can be used to translate a derivation from the initial process constant I_S^C into any term t: $I_S^C \overset{*}{\Rightarrow} t$.

Remark

It is possible, that Algorithm 5.3 results in no reduction of the set of transition rules in comparison to the formal algorithm derived from Figure 5.5 on page 104.

5.3.2.2 Remove Unresolvable Transition Rules

We subsume all transition rules from a Process Rewrite System $\Pi = (Q, \Sigma, I, \rightarrow, F)$ having on the left-hand side (LHS) a term which does not appear on the right-hand side (RHS) of any other transition rule.

Using this optimization we want to remove all transition rules, translating a term t at the left-hand side into a term t' at the right-hand side, so that t' can never be translated into ε. For this purpose we use a bottom-up definition.

Definition 5.5 (Unresolvable transition rules)
A transition $\delta \hat{=} t \overset{a}{\rightarrow} t'$ *is unresolvable if* $p \in LHS(\delta)$ *and* $p \overset{w}{\not\Rightarrow} \varepsilon$.

To compute the terms that are definitely unresolvable we define the function $\mathsf{usedConstants}(\{p\})$, which calculates the set of $p' \in Q$ ($p' \overset{*}{\Rightarrow} p$ and T is a set of process-algebraic expressions):

$$\begin{aligned}
\mathsf{usedConstants}(\{\}) &\hat{=} \{\} \\
\mathsf{usedConstants}(T) &\hat{=} \{p \in LHS(t \rightarrow t') \mid t \in T \wedge (t \rightarrow t') \in \rightarrow \wedge p \in Q\} \\
&\quad \cup \mathsf{usedConstants}(\{t \mid t' \in T \wedge (t \rightarrow t') \in \rightarrow \wedge t \notin T\})
\end{aligned}$$

Using $\mathsf{usedConstants}(\{t\})$ we define the function $\mathsf{unresolvableConstants}$ calculating the process constants, which cannot be part of a derivation $t' \overset{*}{\Rightarrow} t$.

$$\mathsf{unresolvableConstants}(T) \hat{=} Q \setminus \mathsf{usedConstants}(T)$$

Here, we calculate a Process Rewrite System $\Pi = (Q', \Sigma, I, \rightarrow', F)$ where the set of transition rules $\rightarrow' \hat{=} \{\delta \in \rightarrow \mid RHS(\delta) \cap \mathsf{unresolvableConstants}(\varepsilon) \neq \emptyset\}$ and $Q' \hat{=} \mathsf{usedConstants}(\varepsilon)$. This optimization removes the transition rules, which could never be a part of a derivation path: $I \overset{*}{\Rightarrow} \varepsilon$.

5.3.3 Summary

In this section we have shown how a representation – the Combined Abstraction Π_S^C – of the model checking problem can be constructed. The Combined Abstraction encodes the protocol

conformance problem in a Process Rewrite System representation, so that the reachability $I_S^C \overset{w}{\Rightarrow} \varepsilon$ has to be checked. The Combined Abstraction is constructed while combining the derivation paths contained in the system abstraction $\Pi_{S,C}$ and the inverted protocol $\overline{P_C}$ of component C, thus w describes an interaction sequence contained in the system abstraction $\Pi_{S,C}$ and in the considered inverted protocol $\overline{P_C}$. These calculations capture the behavior conservatively. Hence, we find every existing counterexample (no false positives can appear).

In the next section we will discuss how the model checking is performed.

5.4 Step 4: Performing Protocol Conformance Checking

In this section we discuss the model checking problem. We present the reachability algorithm defined by Richard Mayr [May97] and discuss the problems using this algorithm. Then, we will discuss our implementation and improvements of the model checking algorithm.

As the Combined Abstraction Π_S^C is an overapproximation (of the behavior possible in the system abstraction Π_S and forbidden by the protocol P_C), false negatives can appear. They are discussed in Section 5.4.1.

To raise the applicability of our context we develop two optimizations. The first – an a posteriori approach – improves the speed of the model checking while using the properties of the PRS-hierarchy (Section 5.4.3). This approach is similar to a CEGAR-loop[11] and was published as [BZ09d]. In Section 5.4.2 we present an a priori approach to reduce the number of false negatives while taking advantage of the properties of the computed Combined Abstraction. We published this adapted model checking algorithm in [BZ09b].

The theoretical foundations for the model checking algorithm are presented in [May97]. There, the model checking problem is reduced to the reachability of Petri nets. For this purpose the given Process Algebra Net is translated into a Petri net while considering the two process constants on the right-hand side of a sequential rule independently. Based on Mayr's Definition in [May97] we formulated the Algorithm 5.4.

The algorithm is divided into two parts. The function *PAN-Reachability* solves the problem for terms containing no sequential operator (Line 8) while calling a Petri nets reachability oracle as subprocess. For this purpose the sequential rules $p_1 \overset{a}{\to} p_2.p_3$ of Π are translated iteratively to chain transition rules $p_1 \overset{a}{\to} p_3$ if PAN-Reachability(p_2) is valid. The function *translateToParallelTerm* tries to translate a given term $t \in PEX(Q)$ into a term t' containing no sequential operator (Line 23). For details we refer to [May97].

The algorithm is EXPSPACE-hard, because the Petri nets reachability problem (called as subprocess) is EXPSPACE-hard as proven by Ernst W. Mayr [May81].

An implementation of a PAN model checker using Mayr's Algorithm was task of a diploma thesis [Prä09] we have supervised.

Using this algorithm the protocol conformance problem can be decided. Considering the practical applicability it is needed not to flood the user of the verification process with coun-

[11] *Counterexample guided abstraction refinement* loop [E. 00].

Example 5.15: Trace constructed by the reachability algorithm. It results in the counterexample $c\,b\,d\,b$ for the protocol of C_2 in Example 5.9 on page 93. For better understanding the processes of $\Pi_S^{C_2}$ rewritten in the considered step are underlined.

A finite state machine $\mathcal{P}_{C_2}{}'$, that describes a subset of the inverted protocol $\overline{\mathcal{P}_{C_2}}$ of component C_2.
$\mathcal{P}_{C_2}{}' = (\{I_A, x_2, x_3, x_4, x_F\}, \{b, c, d\}, I_A, \{I_A \xrightarrow{c} x_2, x_2 \xrightarrow{b} x_3, x_3 \xrightarrow{d} x_4, x_4 \xrightarrow{b} x_F\}, \{x_F\})$
We can see that $L(\mathcal{P}_{C_2}{}') \subseteq \overline{L(\mathcal{P}_{C_2})}$.
Trace of $\Pi_S^{C_2}$, constructing the protocol violation $c\,b\,d\,b$ in component C_2:

$$
\begin{array}{lll}
\underline{(I_A, I_S, x_F)} & \overset{\lambda}{\Rightarrow} \underline{(I_A, q_1, x_F)} & \overset{\lambda}{\Rightarrow} \underline{(I_A, p_2, x_F)} \\
\overset{c}{\Rightarrow} \underline{(x_2, p_8, x_2)}.(x_2, p_3, x_F) & \overset{\lambda}{\Rightarrow} \underline{(x_2, p_9, x_2)}.(x_2, p_3, x_F) & \overset{\lambda}{\Rightarrow} \underline{(x_2, \varepsilon, x_2)}.(x_2, p_3, x_F) \\
\overset{\lambda}{\Rightarrow} \underline{(x_2, p_3, x_F)} & \overset{\lambda}{\Rightarrow} \underline{(x_2, p_4, x_F)} & \overset{b}{\Rightarrow} \underline{(x_3, p_{10}, x_3)}.(x_3, p_5, x_F) \\
\overset{\lambda}{\Rightarrow} \underline{(x_3, p_{11}, x_3)}.(x_3, p_5, x_F) & \overset{\lambda}{\Rightarrow} \underline{(x_3, p_{12}, x_3)}.(x_3, p_5, x_F) & \overset{\lambda}{\Rightarrow} \underline{(x_3, \varepsilon, x_3)}.(x_3, p_5, x_F) \\
\overset{\lambda}{\Rightarrow} \underline{(x_3, p_5, x_F)} & \overset{\lambda}{\Rightarrow} (x_3, p_7, x_F)||\underline{(x_3, p_6, x_F)} & \overset{d}{\Rightarrow} (x_3, p_7, x_F)||\underline{(x_4, p_{14}, x_F)} \\
\overset{\lambda}{\Rightarrow} \underline{(x_3, p_7, x_F)}||(x_4, \varepsilon, x_F) & \overset{\lambda}{\Rightarrow} \underline{(x_4, p_7, x_F)}||(x_4, \varepsilon, x_F) & \overset{b}{\Rightarrow} \underline{(x_F, p_{10}, x_F)}||(x_4, \varepsilon, x_F) \\
\overset{\lambda}{\Rightarrow} \underline{(x_F, p_{11}, x_F)}||(x_4, \varepsilon, x_F) & \overset{\lambda}{\Rightarrow} (x_F, p_{12}, x_F)||\underline{(x_4, \varepsilon, x_F)} & \overset{\lambda}{\Rightarrow} (x_F, p_{12}, x_F)||\underline{(x_F, \varepsilon, x_F)} \\
\overset{\lambda}{\Rightarrow} \underline{(x_F, p_{12}, x_F)} & \overset{\lambda}{\Rightarrow} \underline{(x_F, \varepsilon, x_F)}\ \square &
\end{array}
$$

terexamples, which are provable no counterexample. These counterexamples are called false negatives. Reducing all false negatives is not possible as discussed in the next section.

5.4.1 False Negatives

The resulting counterexamples might not be real counterexamples. These counterexamples are called *false negatives*.

There are two causes of false negatives:

- *real false negatives*: Because the component abstractions could be created without any data-flow or control-flow analysis, it is possible that a trace will be contained in the component abstraction, which is not possible in the implemented component. E. g., a return value can route the control flow. Example 5.16 shows such a situation. There, the model checker will produce the interactions sequences ab, cd, ad and cb. The latter two are possible, as the data flow is not taken into account[12] (during the construction of the abstraction of C). Thus, if we have no access to the implementation of the other components of the component-based system, it is possible to create more false negatives. The first, fourth and sixth counterexample in Example 4.6d on page 71 are such false negatives.

- *spurious false negatives*: Because we only construct an approximated intersection Π_S^C of the language described by the component-based system $\Pi_{S,C}$ and the considered inverted

[12] This abstraction can be refined easily using standard compiler construction techniques.

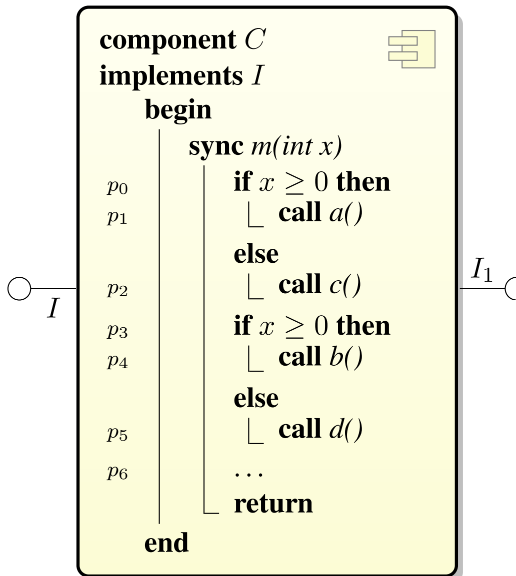
Example 5.16: Abstraction resulting in a real false negative.

interface I_1
begin
  sync void *a()*
  sync void *b()*
  sync void *c()*
  sync void *d()*
end

(a) Interface I_1.

$$\mathcal{P}_{C_1} \hat{=} (ab)|(cd)$$

(b) Protocol of component C_1.



(c) Component C requires an implementation of interface I_1.

$$p_0 \xrightarrow{\lambda} p_1$$
$$p_0 \xrightarrow{\lambda} p_2$$
$$p_1 \xrightarrow{a} q_{I_1,a} \cdot p_3$$
$$p_2 \xrightarrow{c} q_{I_1,c} \cdot p_3$$
$$p_3 \xrightarrow{\lambda} p_4$$
$$p_3 \xrightarrow{\lambda} p_5$$
$$p_4 \xrightarrow{b} q_{I_1,b} \cdot p_6$$
$$p_5 \xrightarrow{d} q_{I_1,d} \cdot p_6$$
$$p_6 \xrightarrow{\lambda} \varepsilon$$

(d) Transition rules of Π_C.

Algorithm 5.4: Mayr's Algorithm

```
    input   :  Process Algebra Net Π≙(Q, Σ, I, →, ε)
    output  :  Boolean reachable
 1  /* Compute PAN reachability                                   */
 2  Boolean PAN-Reachability(t ∈ PEX(Q))
 3  begin
 4      if FAIL ∈ t then
 5          return false
 6      end
 7      if t contains no sequential operator then
 8          return t ⇒_{Π_par} ε
 9      end
10      if t≙t1.t2 then
11          return (PAN-Reachability(t1) ∧ PAN-Reachability(t2))
12      end
13      if t≙t1||t2 then
14          return PAN-Reachability(translateToParallelTerm(t1||t2))
15      end
16  end
17  /* Translate a PAN term into a parallel term (Petri net) */
18  PEX(Q) ⊎ {FAIL}  translateToParallelTerm(t ∈ PEX(Q))
19  begin
20      if t≙t1||t2 then
21          return (translateToParallelTerm(t1)||translateToParallelTerm(t2))
22      else if t≙t1.t2 then
23          if PAN-Reachability(t1) then
24              return translateToParallelTerm(t2)
25          else
26              return FAIL
27          end
28      else
29          return t
30      end
31  end
32  /* Main                                                          */
33  reachable = PAN-Reachability(I)
```

protocol $\overline{P_C}$, it is possible to get false negatives. Some of the false negatives can be reduced, e. g., by taking the protocol into account.

As counterexamples could be false negatives they have to be evaluated.

If the component code is not available, it would not be possible to reduce the real false negatives. They have to be verified by the developer having access to the source code of the component.

As spurious counterexamples are generated while computing the Combined Abstraction, they may be erasable while evaluating the accessible pieces of information. We assume, that spurious counterexamples are very annoying for the user of the verification process as they are caused by

a representation, which cannot be evaluated by the user[13]. Thus, we see a need for reducing the number of spurious counterexamples. We will focus in the next section on this topic.

5.4.2 Reducing the Number of False Negatives

In the previous section we defined a CEGAR-loop improving the verification speed. For this approach we took the advantages of the PRS-hierarchy, which classifies the Process Rewrite System by the allowed operators. Since a counterexample is checked after it has been found, we call such approaches *a posteriori*. Because of the fact that all counterexamples of a conservative approximation have to be verified in the real software (by hand or tool supported), every spurious counterexample causes an expensive checking in detail. Thus, a large number of spurious counterexamples reduces the practical applicability of a verification.

In this section, we improve the search for counterexamples by cutting off branches in the search tree that definitely lead to spurious counterexamples. This is done within the model checking algorithm, thus it is an *a priori approach*.

5.4.2.1 Basic Idea

A counterexample $\mathfrak{c}$ is a sequence of interactions w used to create a derivation from the initial constant I to the final constant ε: $I \overset{w}{\Rightarrow} \varepsilon$. This sequence has to be contained in the language of the inverted protocol ${P_C}'$ (inverted protocol FSM), thus we can check $w \in L({P_C}')$. If $w \notin L({P_C}')$, we can eliminate this counterexample because it is a false negative. Using this approach we can eliminate the counterexamples $\mathfrak{c}_1$ and $\mathfrak{c}_2$ of Example 5.18 on page 115 (which is a reduced representation of the behavior of Example 5.17).

However, this check requires the explicit construction of a spurious counterexample. Here, our goal is to avoid this explicit construction. First, this leads to a more efficient search for counterexamples. Second, the method described above might give false hints to how counterexamples can be derived. The latter is demonstrated in Example 5.19 on page 116 : The counterexample $\mathfrak{c}_2$ can be constructed in the same way as in Example 5.18. However, the language inclusion check does not fail, because the word w of $\mathfrak{c}_2$ is contained in the language of the inverted protocol: $w = abb \in L(abb^*)$. A closer look shows, that $\mathfrak{c}_2$ contains two transitions from the protocol state v_1 to v_2 using the interaction b. As we know, this is not possible, because only one interaction is allowed at one point in time (interleaving semantics). This situation is confusing, since developers might conclude erroneously from $\mathfrak{c}_2$, that it is a false negative. Hence, this approach looses some pieces of information about the generated counterexamples.

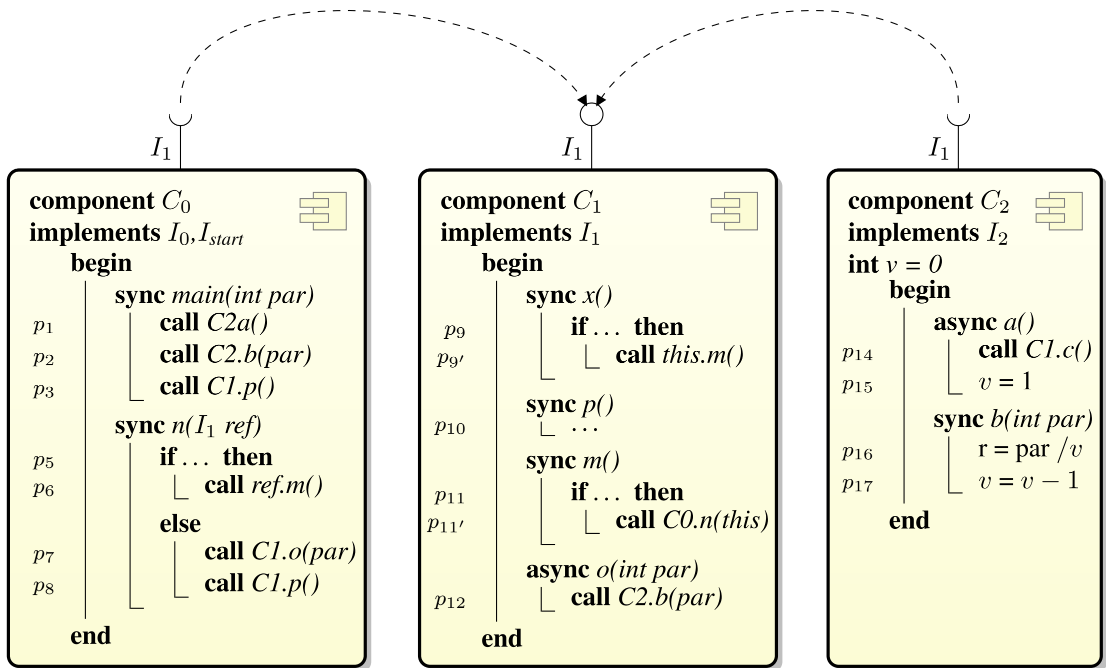
The idea is now: When a protocol state change happens in one of the terms where a rule is applicable (head terms), then we first change the protocol state in the other head terms before we continue the search. We call this heuristic the Round-robin reachability. The derivation of $\mathfrak{c}_0$ in Example 5.18 follows this strategy.

[13]Moreover, it is possible that the user is not poised to do an evalution of counterexamples.

Example 5.17: Abstraction containing a spurious false negative.

```
interface I0
begin
  sync n(I1 ref)
end

interface I1
begin
  sync x()
  sync p()
  sync m()
  async o(int par)
end

interface I2
begin
  async a()
  sync b(int par)
end
```

(a) Interfaces of three components.



(b) Composed three components.

$$P_{C_0} = n^* \qquad P_{C_1} = zpm^*(op|po) \qquad P_{C_2} = (ba^+)^*$$

(c) Protocols of three components.

Example 5.18: Spurious false negatives.

Given system abstraction (initial constant p_0):	$p_0 \xrightarrow{a} p_1 \|\| p_1$	$p_1 \xrightarrow{b} \varepsilon$	$p_1 \xrightarrow{\lambda} \varepsilon$
Given inverted protocol (initial state v_0, final state v_2):	$v_0 \xrightarrow{a} v_1$	$v_1 \xrightarrow{b} v_2$	

Generated Combined Abstraction (initial constant (v_0, p_0, v_2), only rules required for counterexamples are shown):

$$(v_0,p_0,v_2) \xrightarrow{a} (v_1,p_1,v_2)||(v_1,p_1,v_2) \quad (v_1,p_1,v_2) \xrightarrow{b} (v_2,\varepsilon,v_2) \quad (v_1,p_1,v_2) \xrightarrow{\lambda} (v_2,p_1,v_2)$$
$$(v_1,p_1,v_2) \xrightarrow{\lambda} (v_1,\varepsilon,v_2) \quad (v_1,\varepsilon,v_2) \xrightarrow{\lambda} (v_2,\varepsilon,v_2) \quad (v_2,p_1,v_2) \xrightarrow{\lambda} (v_2,\varepsilon,v_2)$$
$$(v_0,p_0,v_2) \xrightarrow{\lambda} (v_1,p_0,v_2) \quad (v_1,p_0,v_2) \xrightarrow{\lambda} (v_2,p_0,v_2)$$

Computed counterexamples:

$$\mathfrak{c}_0 \hat{=} \underline{(v_0,p_0,v_2)} \stackrel{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \stackrel{b}{\Rightarrow} \underline{(v_1,p_1,v_2)}||(v_2,\varepsilon,v_2) \stackrel{b}{\Rightarrow} \underline{(v_2,p_1,v_2)}||(v_2,\varepsilon,v_2)$$
$$\stackrel{\lambda}{\Rightarrow} (v_2,\varepsilon,v_2)||\underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \varepsilon$$
$$\mathfrak{c}_1 \hat{=} \underline{(v_0,p_0,v_2)} \stackrel{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \stackrel{\lambda}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,\varepsilon,v_2)} \stackrel{b}{\Rightarrow} \underline{(v_1,p_1,v_2)}||(v_2,\varepsilon,v_2)$$
$$\stackrel{\lambda}{\Rightarrow} \underline{(v_1,\varepsilon,v_2)}||(v_2,\varepsilon,v_2) \stackrel{b}{\Rightarrow} (v_2,\varepsilon,v_2)||\underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \varepsilon$$
$$\mathfrak{c}_2 \hat{=} \underline{(v_0,p_0,v_2)} \stackrel{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \stackrel{b}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \underline{(v_1,p_1,v_2)} \stackrel{b}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \stackrel{\lambda}{\Rightarrow} \varepsilon$$

5.4.2.2 The Round-Robin Reachability Algorithm

In this section we prove, that this heuristic does not exclude non-spurious counterexamples. Algorithm 5.5 shows a backtracking algorithm implementing this strategy.

It uses the following notations:

- As can be seen from the inference rules defining the derivations, the derivation relation in a term $(p_1.u_1|| \dots ||p_n.u_n).u_{n+1}$, where $n > 0, u_1, \dots, u_{n+1} \in PEX(Q), p_1, \dots, p_n \in Q$ is only applied to one of the terms p_i. Formally, we define the notion of the *set of heads* $H(t)$ for a process-algebraic term $t \in PEX(Q)$ inductively as follows: The multiset of (atomic) heads $H(t)$ and the multiset of synchronization possibilities $S(t)$ of a Process Rewrite System term t is defined as follows:

$$\begin{aligned} H(p) &\hat{=} \{\{p\}\} \\ H(t_1.t_2) &\hat{=} H(t_1) \\ H(t_1||t_2) &\hat{=} H(t_1)||H(t_2) \\ S(t) &\hat{=} \{\{p_i||p_j : p_i, p_j \in H(t), p_i \neq p_j \vee \xi_{A(t)}(p_i) \geq 2\}\} \end{aligned}$$

where $p, p_1, p_2 \in Q,\ t_1, t_2 \in PEX(Q)$ and $\xi_{A(t)}$ denotes the number of elements in a multiset A.

Intuitively the heads are the top stack frames in a cactus stack. E. g., the heads and the

Example 5.19: Adaption of Example 5.18 on page 115: language inclusion results in an impractical output (impractical, because it is difficult to capture the invalidity of the computed counterexample).

Given system abstraction rules (initial constant p_0):

$$p_0 \xrightarrow{a} p_1||p_1, \qquad p_1 \xrightarrow{b} \varepsilon, \qquad p_1 \xrightarrow{\lambda} \varepsilon$$

Given inverted protocol (initial state v_0, final state v_2):

$$v_0 \xrightarrow{a} v_1, \qquad v_1 \xrightarrow{b} v_2, \qquad v_2 \xrightarrow{b} v_2$$

A computable counterexample:

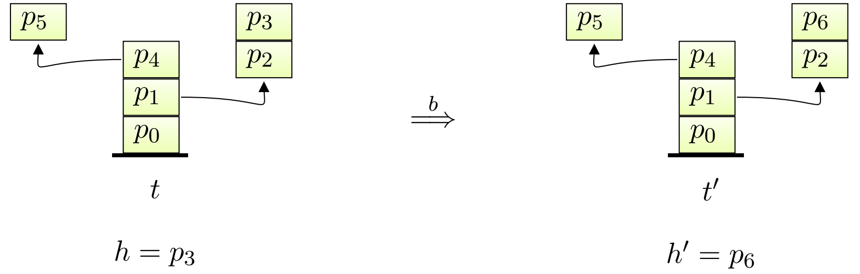
$\mathfrak{c}_2$ of Example 5.18 on page 115

Example 5.20: Situation described by Property 5.3.

considered transition rule of Process Rewrite System:

$$\delta \hat{=} p_3 \xrightarrow{b} p_6$$

translation of cactus stack:



$h = p_3$ $\qquad$ $h' = p_6$

$H(t) = \{\{p_3, p_4, p_5\}\}$ $\qquad$ $H(t') = \{\{p_4, p_5, p_6\}\}$

$S(t) = \{\{p_3||p_4, p_3||p_5\}\}$ $\qquad$ $S(t') = \{\{p_6||p_4, p_6||p_5\}\}$

synchronization possibilities of t in Example 3.7 on page 44 are:

$$H(t) = \{\{p_2, p_3, p_3, p_5, p_6\}\}$$
$$S(t) = \{\{p_2||p_3, p_2||p_3, p_2||p_5, p_2||p_6, p_3||p_3, p_3||p_5, p_3||p_6, p_3||p_5, p_3||p_6, p_5||p_6\}\}$$

A smaller example is part of Example 5.20. As we can see, the synchronization properties are an overapproximation – e. g., $p_5||p_6$ cannot be transformed using Process Rewrite System rules into the given cactus stack – but this definition satisfies our needs.

Property 5.3
If in a Process Rewrite System $\Pi = (Q, \Sigma, I, \rightarrow, F)$ a rule $\delta : t_1 \xrightarrow{b} t_2$ ($\delta \in \rightarrow$) is applied to process-algebraic term t (i. e., $t \xRightarrow{b} t'$, where $t, t' \in PEX(Q)$), then there is a head $h \in H(t)$ or a synchronization possibility $h \in S(t)$, so that t' is obtained from t by replacing h by h' by the same rule δ, where $h \xRightarrow{b} h'$ (cf. Example 5.20).

Property 5.3 states, that any action rule is either applied to a top frame of a cactus stack or

it synchronizes two threads (i. e., it merges two stacks into one).

- *Corresponding sleep rule*: A rule $\delta \in \mathcal{R}^0$ of the considered Combined Abstraction Π_S^C, which is created by using a rule $v \xrightarrow{a} v'$ of the protocol P_C is called the sleep rule corresponding to an action rule $t_1 \xrightarrow{a} t_2$.

- A *protocol state of a process-algebraic term* is the set of protocol states. The protocol state of a triple (v, p, v') is v. Formally, we define the function $\mathrm{ps}(t)$ for a process-algebraic term $t \in PEX(Q)$ as follows: $\mathrm{ps}(t) = \{v : (v, p, v') \in H(t)\}$.

In Algorithm 5.5 it is valid: if an action rule δ is applied to one head, all other heads are rewritten to the corresponding protocol state by using sleep rules corresponding to δ (cf. line 17-21).

At line 14 of the algorithm we choose one head $h \in H(t)$ or one synchronization possibility $h \in S(t)$ of the current term t to apply δ: $h \overset{a}{\Rightarrow} h'$. Following the definition of action rules this means that the protocol state of h could differ from the one of h'. At line 17 we ensure that all other heads are imitating rewriting of the protocol state in h' while applying a corresponding sleep rule.

Thus, in every derivation step, if one action rule δ has been applied, then all heads are translated into the same protocol state by applying the corresponding sleep rules of δ to extend the derivation d.

Hence, the problems of the language inclusion are eliminated:

- It is ensured, that only one protocol transition is performed in one derivation step of the Combined Abstraction.

- It is checked in every derivation step, whether such a derivation will create spurious false negatives.

The following theorem states, that each non-spurious counterexample can be constructed by the Round-robin reachability.

Theorem 5.5
Let $\Pi_{S,C}$ be a system abstraction (considering component C), P_C is the inverted protocol of component C, and $w \in \Sigma^*$, such that $t \overset{w}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ for a term $t \in PEX(Q_{\Pi_{S,C}})$ and $v \overset{w}{\Rightarrow}_{P_C} f$ for a protocol state $v \in Q_{P_C}$ and a final protocol state $f \in F_{P_C}$. Then there is a process-algebraic expression $s \in Q_{\Pi_S^C}$ over the Combined Abstraction, such that $s \overset{w}{\Rightarrow} \varepsilon$ and $|ps(s)| = 1$, i. e., the protocol states in the heads are equal.

The proof of Theorem 5.5 requires some technical definitions and lemmas. Let $s \in PEX(Q_{\Pi_S^C})$ be a process-algebraic expression over the Combined Abstraction Π_S^C. The process-algebraic expression $\mathrm{F}(s) \in PEX(Q_{\Pi_{S,C}})$ over the system abstraction $\Pi_{S,C}$ "forgets" the protocol states in

Algorithm 5.5: Round-robin reachability algorithm in pseudo code.

input : PRS term t
output : Derivation $t \stackrel{w}{\Rightarrow} \varepsilon$ if it exists, false otherwise

```
1  if t = ε then
2  |  return ε
3  end
4  let 𝓡 be
5  |  the set of rules applicable to t
6  end
7  foreach δ ∈ 𝓡 do
8  |  t' = t
9  |  switch δ do
10 |  |  case t1 -λ-> t2
11 |  |  |  t' = apply δ to t'
12 |  |  |  if newReach(t') = t' =w=> ε then
13 |  |  |  |  return t =λ=> t' =w=> ε
14 |  |  |  end
15 |  |  end
16 |  |  case t1 -a-> t2
17 |  |  |  choose h ∈ H(t') ∪ S(t')
18 |  |  |  t' = apply δ at h of t'
19 |  |  |  d = t =a=> t'
20 |  |  |  foreach h' ∈ H(t') ∧ ps(h') ≠ ps(t2) do
21 |  |  |  |  let δ' be
22 |  |  |  |  |  the corresponding sleep rule of δ;
23 |  |  |  |  end
24 |  |  |  |  t' = apply δ' at h' of t'
25 |  |  |  |  d = d =λ=> t'
26 |  |  |  end
27 |  |  end
28 |  end
29 |  if newReach(t' = t' =w=> ε then
30 |  |  return d =aw=> ε
31 |  end
32 end
33 return false
```

s. Formally:

$$\begin{aligned} \mathrm{F}((v,p,v')) &= p && \forall (v,p,v') \in Q_{\Pi_S^C} \\ \mathrm{F}(s.s') &= \mathrm{F}(s).\mathrm{F}(s') && \forall s,s' \in PEX(Q_{\Pi_S^C}) \\ \mathrm{F}(s||s') &= \mathrm{F}(s)||\mathrm{F}(s') && \forall s,s' \in PEX(Q_{\Pi_S^C}) \end{aligned}$$

E. g., $\mathrm{F}((v_1,p_2,v_5)||(v_1,p_3,v_2).(v_2,p_4,v_5)) = p_2||p_3.p_4$. Furthermore, let $\mathrm{F}^{-1}(t) = \{s|\mathrm{F}(s) = t\}$.

The following lemma states, that "λ-derivations" $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$ in the system abstraction $\Pi_{S,C}$ can be transformed into corresponding λ-derivations $s \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s'$ in the Combined Abstraction Π_S^C:

Lemma 5.3

Let $t,t' \in PEX(Q_{\Pi_{S,C}})$ such that $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$. Then, for all $s' \in F^{-1}(t')$, there is an $s \in F^{-1}(t)$ such that the following properties are satisfied.

1. *$s \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s'$ using only rules $\delta \in \mathcal{R}_C^\lambda \cup \mathcal{R}_{Seq}^\lambda \cup \mathcal{R}_{Fork}^\lambda \cup \mathcal{R}_{Sync}^\lambda$.*
2. *$(v,p,v') \in H(s)$ iff there is a $p' \in Q_{\Pi_{S,C}}$ such that either $(v,p',v') \in H(s')$ or $(v,p,v) \in H(s')$.*

Remark

2. implies that ps$(s) =$ ps(s'), i. e., no state change in the protocol happens.

Proof

Straightforward induction on the construction of $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$.

Lemma 5.4 states, that the Round-robin derivation can always be constructed, if there is exactly one protocol state change, i. e., an action rule is applied (cf. line 17-27 in Algorithm 5.5).

Lemma 5.4

Let $t_1,t_2 \in PEX(Q_{\Pi_{S,C}})$ two process-algebraic expressions over the system abstraction $\Pi_{S,C}$ such that $t_1 \overset{a}{\Rightarrow}_{\Pi_{S,C}} t_2$ for an $a \in \Sigma$ by If in the protocol P_C, there is a state transition $v \overset{a}{\rightarrow}_{P_C} v'$, then for any $s_2 \in F^{-1}(t_2)$ satisfying ps$(s_2) = \{v'\}$, there is a $s_1 \in F^{-1}(t_1)$ such that the following properties are satisfied:

1. *ps$(s_1) = \{v\}$*
2. *$s_1 \overset{a}{\Rightarrow}_{\Pi_S^C} s' \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s_2$ where $s_1 \overset{a}{\Rightarrow}_{\Pi_S^C} s'$ uses a single rule $\delta \in \mathcal{R}_C^1 \cup \mathcal{R}_{Seq}^1 \cup \mathcal{R}_{Fork}^1 \cup \mathcal{R}_{Sync}^1$ and $s' \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s_2$ only uses rules δ which are corresponding sleep rules of $v \overset{a}{\rightarrow}_{P_C} v'$.*

Proof (of Lemma 5.4)

<u>Case 1:</u> $\delta \hat{=} (p \overset{a}{\rightarrow}_{\Pi_{S,C}} p')$ or $\delta \hat{=} (p \overset{a}{\rightarrow}_{\Pi_{S,C}} p'||p'')$ or $\delta \hat{=} (p||p'' \overset{a}{\rightarrow}_{\Pi_{S,C}} p')$. Then, there must be a protocol state $v'' \in Q_{P_C}$ such that $(v',p',v'') \in H(s_2)$. Suppose that $H(s_2) \setminus \{(v',p',v'')\} =$

Example 5.21: Complete information of Example 5.18 on page 115.

$$\Pi_{S,C} = (\quad \{p_0, p_1, p_2\}, \\ \{a, b\}, \\ p_0, \\ \{p_0 \xrightarrow{a} p_1, p_1 \xrightarrow{b} \varepsilon, p_1 \xrightarrow{\lambda} \varepsilon\}, \\ \{\varepsilon\} \quad)$$

(a) System abstraction $\Pi_{S,C}$.

$$\overline{P_C} = (\quad \{v_0, v_1, v_2\}, \\ \{a, b\}, \\ \{v_0 \xrightarrow{a} v_1, v_1 \xrightarrow{b} v_2\}, \\ v_0, \\ \{v_2\} \quad)$$

(b) Inverted protocol $\overline{P_C}$.

$$\Pi_S^C = (\quad \{v_0, v_1, v_2\} \times \{p_0, p_1, p_2\} \times \{v_0, v_1, v_2\}, \\ \{a, b\}, \\ (v_0, p_0, v_2), \\ \mathcal{R}_C^1 \cup \mathcal{R}_C^\lambda \cup \mathcal{R}_{Fork}^1 \cup \mathcal{R}^0 \cup \mathcal{R}^\varepsilon, \\ \{\varepsilon\} \quad)$$

(c) Combined Abstraction Π_S^C.

$$\mathcal{R}_C^1 = \{(v_1, p_1, v_0) \xrightarrow{b} (v_2, \varepsilon, v_0),\ (v_1, p_1, v_1) \xrightarrow{b} (v_2, \varepsilon, v_1),\ (v_1, p_1, v_2) \xrightarrow{b} (v_2, \varepsilon, v_2)\}$$

$$\begin{aligned} \mathcal{R}_C^\lambda = \{&(v_0, p_1, v_0) \xrightarrow{\lambda} (v_0, \varepsilon, v_0),\ (v_0, p_1, v_1) \xrightarrow{\lambda} (v_0, \varepsilon, v_1),\ (v_0, p_1, v_2) \xrightarrow{\lambda} (v_0, \varepsilon, v_2), \\ &(v_1, p_1, v_0) \xrightarrow{\lambda} (v_1, \varepsilon, v_0),\ (v_1, p_1, v_1) \xrightarrow{\lambda} (v_1, \varepsilon, v_1),\ (v_1, p_1, v_2) \xrightarrow{\lambda} (v_1, \varepsilon, v_2), \\ &(v_2, p_1, v_0) \xrightarrow{\lambda} (v_2, \varepsilon, v_0),\ (v_2, p_1, v_1) \xrightarrow{\lambda} (v_2, \varepsilon, v_1),\ (v_2, p_1, v_2) \xrightarrow{\lambda} (v_2, \varepsilon, v_2)\} \end{aligned}$$

$$\begin{aligned} \mathcal{R}_{Fork}^1 = \{&(v_0, p_0, v_0) \xrightarrow{a} (v_1, p_1, v_0)||(v_1, p_1, v_0),\ (v_0, p_0, v_1) \xrightarrow{a} (v_1, p_1, v_1)||(v_1, p_1, v_1), \\ &(v_0, p_0, v_2) \xrightarrow{a} (v_1, p_1, v_2)||(v_1, p_1, v_2)\} \end{aligned}$$

$$\begin{aligned} \mathcal{R}^0 = \{&(v_0, p_0, v_0) \xrightarrow{\lambda} (v_1, p_0, v_0),\ (v_0, p_0, v_1) \xrightarrow{\lambda} (v_1, p_0, v_1),\ (v_0, p_0, v_2) \xrightarrow{\lambda} (v_1, p_1, v_2), \\ &(v_0, p_1, v_0) \xrightarrow{\lambda} (v_1, p_1, v_0),\ (v_0, p_1, v_1) \xrightarrow{\lambda} (v_1, p_1, v_1),\ (v_0, p_1, v_2) \xrightarrow{\lambda} (v_1, p_1, v_2), \\ &(v_1, p_0, v_0) \xrightarrow{\lambda} (v_2, p_0, v_0),\ (v_1, p_0, v_1) \xrightarrow{\lambda} (v_2, p_0, v_1),\ (v_1, p_2, v_2) \xrightarrow{\lambda} (v_2, p_1, v_2), \\ &(v_1, p_1, v_0) \xrightarrow{\lambda} (v_2, p_1, v_0),\ (v_1, p_1, v_1) \xrightarrow{\lambda} (v_2, p_1, v_1),\ (v_1, p_1, v_2) \xrightarrow{\lambda} (v_2, p_2, v_2), \\ &(v_0, \varepsilon, v_0) \xrightarrow{\lambda} (v_1, \varepsilon, v_0),\ (v_0, \varepsilon, v_1) \xrightarrow{\lambda} (v_1, \varepsilon, v_1),\ (v_0, \varepsilon, v_2) \xrightarrow{\lambda} (v_1, \varepsilon, v_2), \\ &(v_1, \varepsilon, v_0) \xrightarrow{\lambda} (v_2, \varepsilon, v_0),\ (v_1, \varepsilon, v_1) \xrightarrow{\lambda} (v_2, \varepsilon, v_1),\ (v_1, \varepsilon, v_2) \xrightarrow{\lambda} (v_2, \varepsilon, v_2)\} \end{aligned}$$

$$\mathcal{R}^\varepsilon = \{(v_0, \varepsilon, v_0) \xrightarrow{\lambda} \varepsilon,\ (v_1, \varepsilon, v_1) \xrightarrow{\lambda} \varepsilon,\ (v_2, \varepsilon, v_2) \xrightarrow{\lambda} \varepsilon\}$$

(d) Non-optimized sets of transition rules of Π_S^C.

$\{(v',p_1,v_1),\ldots,(v',p_n,v_n)\}$. Then a derivation $s'_n \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_{n-1} \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} \ldots \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_1 \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_0 = s_2$ where s'_{i-1} is the result of the application of the rule $(v,p_i,v_i) \stackrel{\lambda}{\rightarrow}_{\Pi_S^C} (v',p_i,v_i)$ at head (v,p_i,v_i) of term s'_i, $i = n,\ldots,1$. Thus, only corresponding sleep rules of $v \stackrel{a}{\rightarrow}_{\Pi_S^C} v'$ are applied, and $(v',p',v'') \in H(s'_n)$ is the only atomic process with a protocol state different from v. By induction on the construction of $t_1 \stackrel{a}{\Rightarrow}_{\Pi_{S,C}} t_2$, one can prove that $s \stackrel{a}{\Rightarrow}_{\Pi_S^C} s'_n$ using $(v,p,v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v',p,v'')$, $(v,p,v'')||(v,p',v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v',p'',v'')$ for a $v'' \in Q_{P_C}$, or $(v,p,v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v',p',v'')||(v',r'',v'')$, respectively.
Case 2: $\delta = p \stackrel{a}{\rightarrow}_{\Pi_{S,C}} p'||p''$. Then there is a protocol state $v'' \in Q_{P_C}$ such that $(v',p',v'') \in H(s_2)$, and $(v',p'',v'') \in H(s_2)$. The rest of the proof is analogous except that $H(s_2) \setminus \{\{(v',p',v''),(v',p'',v'')\}\}$ is used.

Proof (of Theorem 5.5)
Induction on w:
$w = \lambda$: By Lemma 5.3, there is a $s \in F^{-1}(t)$ such that $s \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ and, since $v \stackrel{\lambda}{\Rightarrow}_{P_C} f$ implies $v = f$, $H(s) = \{\{f\}\}$.
$w = ax$: for an $a \in \Sigma$ and $x \in \Sigma^*$.
Then, $t \stackrel{ax}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ has the form $t \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} t_1 \stackrel{a}{\Rightarrow}_{\Pi_{S,C}} t_2 \stackrel{x}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$, and $v \stackrel{w}{\Rightarrow}_{P_C} f$ has the form $v \stackrel{a}{\Rightarrow}_{P_C} v' \stackrel{x}{\Rightarrow}_{P_C} f$.
By induction hypothesis, there is an $s_2 \in PEX(Q_{\Pi_S^C})$ such that $s_2 \stackrel{a}{\Rightarrow}_{\Pi_S^C} \varepsilon$ and $|ps(s_2)| = 1$. By Lemma 5.4, there is an $s_1 \in PEX(Q_{\Pi_S^C})$ such that $s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s_2$, where $\Rightarrow_{\Pi_S^C}$ is constructed according to Round-robin reachability. By Lemma 5.3, there is an $s \in PEX(Q_{\Pi_S^C})$ such that $s \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_1$ only using rules of $\mathcal{R}^0$ and $|ps(s)| = 1$. Thus $s \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s_2 \stackrel{x}{\Rightarrow}_{\Pi_S^C} \varepsilon$.

Corollary 5.3
If $w \in L(\Pi_{S,C}) \cap L(P_C)$, then there is a Round-robin reachability derivation $t \stackrel{w}{\Rightarrow}_{\Pi_S^C} \varepsilon$ in the Combined Abstraction Π_S^C, where $t \in Q_{\Pi_S^C}$ is the initial state of Π_S^C.

Hence, the reachability problem can be solved by using the Round-robin reachability and will create fewer false negatives. This leads to a better applicability, because a component developer or quality management representative has to check a lower number of counterexamples. Moreover, because we cut branches during the verification, it will probably be finished faster.

Finally, while considering Example 5.18 on page 115, the counterexamples $\mathfrak{c}_1$ and $\mathfrak{c}_2$ are not created while using the Round-robin reachability. This will also reduce the number of false negatives in Example 5.22 on page 123, because Figure 5.18 on page 115 was a simplification of the behavior of the system abstraction shown in Example 5.3 on page 78.

5.4.2.3 Summary

In this section we have shown, how we can use the special properties of the Combined Abstraction representation (defined in Section 5.3) to reduce the number of false negatives. We call this approach Round-robin reachability, because it balances the derivation steps applied on each parallel term. This improvement reduces the costs of the quality check, because fewer

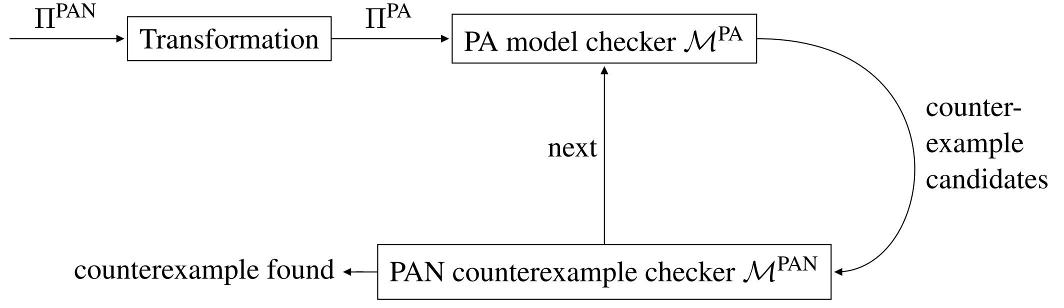


Figure 5.6.: Concept of a posteriori verification algorithm.

counterexamples have to be reviewed to find the real errors.

Although this improved verification process cannot reduce the complexity of the verification in a general case. We assume, that in an industrial setting the verification will be finished much faster. Checking this assumption is part of our future work because of a lack of large examples in the current case study.

5.4.3 Improving Runtime of Model Checking Using Process Rewrite System Properties

Using the standard reachability algorithm of Process Algebra Nets to compute statically so called counterexamples (forbidden interactions) may result in an unacceptable verification time. The reason for this is, that verification contains the reachability problem of Petri nets (which is EXPSPACE-hard, cf. Section 3.2.4). Thus, improving the model checking algorithm is essential to ensure a practical applicability.

In this section we focus on an a posteriori approach to improve the run-time of the model checker. We take advantage of the PRS-hierarchy (cf. Section 3.2.3) to reduce the effort needed to determine if a counterexample is contained in the Combined Abstraction. We use an approximated representation (PA-process) for this purpose. If this preparation step calculates a sequence of interactions (named *counterexample candidate*), then we have to check if it is also computable in the Combined Abstraction. Hence, this approach is similar to a CEGAR-loop (Figure 5.6).

According to the previous discussion, it is sufficient to provide an approach for checking whether $L(\Pi^{PAN}) \stackrel{?}{=} \emptyset$ for a Process Algebra Net. Figure 5.6 shows the algorithm. First a PA-process Π^{PA} is constructed, such that $L(\Pi^{PAN}) \subseteq L(\Pi^{PA})$. Thus, Π^{PA} contains all counterexamples of Π^{PAN}, but it might possibly has more counterexamples (spurious counterexamples $w \in L(\Pi^{PA}) \setminus L(\Pi^{PAN})$).

The counterexamples of Π^{PA} are obtained by using the model checker for PA-processes $\mathcal{M}^{PA}$. Then each counterexample of Π^{PA} is checked whether it is spurious until a counterexample $\mathfrak{c}'$ of Π^{PAN} is discovered or $L(\Pi^{PA}) = \emptyset$. The latter implies, that there is no counterexample for Π^{PAN}.

In order to check whether a counterexample $\mathfrak{c}$ is spurious, a Process Algebra Net such that $\Pi^{PA}_{\mathfrak{c}} = abstract(\Pi^{PAN}_{\mathfrak{c}})$ is constructed, where $\Pi^{PA}_{\mathfrak{c}}$ contains only those rules of Π^{PA} used to

Example 5.22: Example with four interfaces and components.

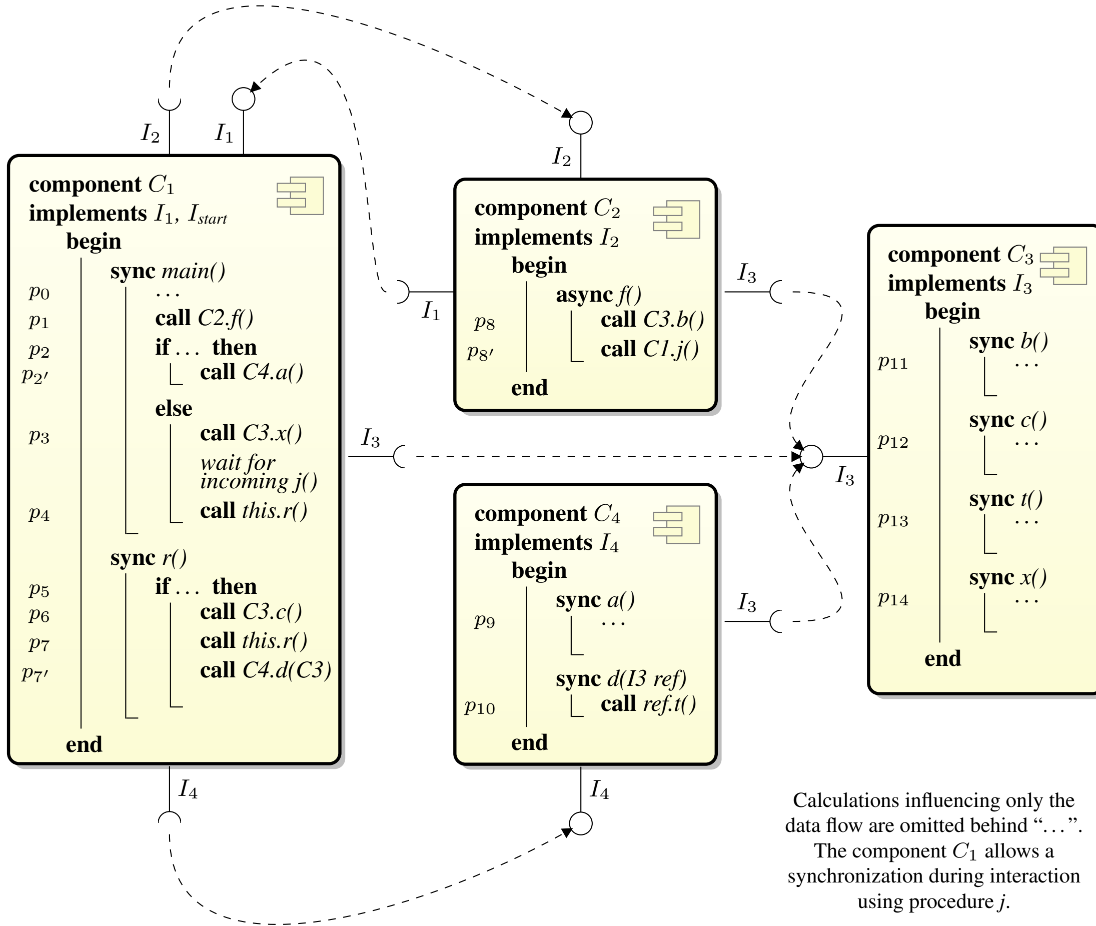
interface I_1
begin
  sync *r()*
  async *j()*
end

interface I_2
begin
  async *f()*
end

interface I_3
begin
  sync *b()*
  sync *c()*
  sync *t()*
  sync *x()*
end

interface I_4
begin
  sync *a()*
  sync *d(I3 ref)*
end

(a) Interfaces of considered components.



(b) Composed components.

$P_{C1} = (r|j)*$ $P_{C2} = f^*$ $P_{C4} = a\, d^*$

(c) Protocols P_{C1}, P_{C2} and P_{C4} as regular expressions.

forbidden sequences regular expression $\overline{P_{C3}} = \; b\, x^*\, c^+\, t^+$

inverted finite state machine $\overline{P_{C3}} = (\{v_0, v_1, v_2, v_3\}, \{b, c, t, x\},$

$\{v_0 \xrightarrow{b} v_1, v_1 \xrightarrow{x} v_1, v_1 \xrightarrow{c} v_2, v_2 \xrightarrow{c} v_2, v_2 \xrightarrow{t} v_3, v_3 \xrightarrow{t} v_3\},$

$v_0, \{v_3\})$

(d) Inverted protocol $\overline{P_{C3}}$.

Algorithm 5.6: Verification algorithm in pseudo code.

input : Π^{PAN}
output : Counterexamples

```
1  /* construct a PA Π^PA, so that L(Π^PAN) ⊆ L(Π^PA)                    */
2  Π^PA = abstract(Π^PAN)
3  /* compute counterexample candidate                                    */
4  𝔠 = M^PA(Π^PA)
5  while 𝔠 ≠ "L(Π^PA) = ∅" do
6  |  let Π_𝔠^PAN ⊆ Π^PAN be
7  |  |  a PAN, so that Π_𝔠^PA ≙ abstract(Π_𝔠^PAN) contains only the rules used in 𝔠
8  |  end
9  |  𝔠' = M^PAN(Π_𝔠^PAN)
10 |  if 𝔠' ≠ "L(Π_𝔠^PAN) = ∅" then
11 |  |  return 𝔠'
12 |  end
13 |  /* compute next counterexample                                       */
14 |  𝔠 = M^PA(Π^PA)
15 end
16 /* contains no counterexample                                          */
17 return "L(Π^PAN) = ∅"
```

compute $\mathfrak{c}$, and $\Pi_{\mathfrak{c}}^{\text{PAN}}$ only contains rules of Π^{PAN}. This is done by using a model checker for Process Algebra Nets $\mathcal{M}^{\text{PAN}}$.

Thus, the size of $\Pi_{\mathfrak{c}}^{\text{PAN}}$ is usually considerably smaller than the size of Π^{PAN} and $\Pi_{\mathfrak{c}}^{\text{PA}}$ contains a counterexample $\mathfrak{c}'$ iff $\mathfrak{c}$ is not spurious. In the following, we detail and prove these claims.

5.4.3.1 Construction of Π^{PA}

So, we will transform the Combined Abstraction in the Process Algebra Net form Π^{PAN} to a representation in the Process Algebra form Π^{PA}. Because we want to capture still a superset of the behavior of Π^{PAN}, we have to ensure, that every sequence of interactions is still createable. So, it has to be possible to find the counterexamples containing a synchronization rule.

Each transition rule $p_i||p_j \xrightarrow{a} p_k$ is replaced by the two rules:

$$p_i \xrightarrow{\lambda} \varepsilon \text{ and } p_j \xrightarrow{a} p_k \qquad (*)$$

It is also possible to use the rules $p_i \xrightarrow{a} p_k$ and $p_j \xrightarrow{\lambda} \varepsilon$, because it is only important to guarantee, that the same counterexample can be constructed (proof follows below).

Thus, all synchronization rules $p_i||p_j \xrightarrow{a} p_k$ are eliminated and a PA-process Π^{PA} is obtained. However, there might be more counterexamples constructable in Π^{PA} as in Π^{PAN} (spurious counterexamples).

Theorem 5.6
For each derivation $\mathfrak{c}_1 \hat{=}\, t \overset{w}{\Rightarrow} \varepsilon$ with application of synchronization rules there is a derivation $\mathfrak{c}_2 \hat{=}\, t \overset{w}{\Rightarrow} \varepsilon$ without application of synchronization rules.

Proof

Example 5.23: The constructed Π^{PA} from Π^{PAN} ($\Pi_{S,\mathrm{C3}}$) in Example 5.3 on page 78.

$$
\begin{array}{lll}
p_0 \xrightarrow{\lambda} p_{1*}||p_2, & p_{1*} \xrightarrow{f} p_8.p_1, & p_2 \xrightarrow{a} p_9.\varepsilon, \\
p_2 \xrightarrow{x} p_{14}.p_3, & p_1 \xrightarrow{\lambda} p_4, & p_3 \xrightarrow{\lambda} \varepsilon, \\
p_4 \xrightarrow{r} p_5.\varepsilon, & p_1 \xrightarrow{\lambda} \varepsilon, & p_5 \xrightarrow{\lambda} \varepsilon, \\
p_5 \xrightarrow{c} p_{12}.p_6, & p_6 \xrightarrow{r} p_5.p_7, & p_7 \xrightarrow{d} p_{10}.\varepsilon, \\
p_8 \xrightarrow{b} p_{11}.\varepsilon, & p_{11} \xrightarrow{\lambda} \varepsilon, & p_{12} \xrightarrow{\lambda} \varepsilon, \\
p_{13} \xrightarrow{\lambda} \varepsilon, & p_{14} \xrightarrow{\lambda} \varepsilon, & p_9 \xrightarrow{\lambda} \varepsilon, \\
p_{10} \xrightarrow{t} p_{13}.\varepsilon & &
\end{array}
$$

Example 5.24: A computable counterexample in Π^{PA} of Example 5.23 on page 125 that would not appear in Π^{PAN}. Here, the method F defined in Section 5.4.2.2 was used to simplify the transition rules of the counterexample (while "forgetting" the protocol states that only were available for verification reasons).

$$
\begin{array}{lllll}
\underline{p_0} & \overset{\lambda}{\Rightarrow} p_{1*}||p_2 & \overset{\lambda}{\Rightarrow} (p_8.p_1)||\underline{p_2} & \overset{\lambda}{\Rightarrow} (\underline{p_8}.p_1)||p_9 & \overset{b}{\Rightarrow} (\underline{p_{11}}.p_1)||p_9 \\
& \overset{\lambda}{\Rightarrow} p_1||\underline{p_9} & \overset{\lambda}{\Rightarrow} \underline{p_1} & \overset{\lambda}{\Rightarrow} \underline{p_4} & \overset{\lambda}{\Rightarrow} \underline{p_5} \\
& \overset{c}{\Rightarrow} \underline{p_{12}}.p_6 & \overset{\lambda}{\Rightarrow} \underline{p_6} & \overset{\lambda}{\Rightarrow} \underline{p_5}.p_7 & \overset{\lambda}{\Rightarrow} \underline{p_7} \\
& \overset{\lambda}{\Rightarrow} \underline{p_{10}} & \overset{t}{\Rightarrow} \underline{p_{13}} & \overset{\lambda}{\Rightarrow} \varepsilon &
\end{array}
$$

Induction on the number of synchronization rules.
$n = 0$: *Then* $\mathfrak{c}_1 = \mathfrak{c}_2$ *proofs the claim.*
$n > 0$: *If* $\mathfrak{c}_1$ *contains a synchronization rule, then it has the form*

$$\mathfrak{c}_1 \hat{=}\, t \overset{u}{\Rightarrow} ((p_i||p_j)||t_1).t_2 \overset{a}{\Rightarrow} (p_k||t_1).t_2 \overset{v}{\Rightarrow} \varepsilon$$

so that $t \overset{u}{\Rightarrow} ((p_i||p_j)||t_1).t_2$ *does not contain a synchronization rule,* $(p_k||t_1).t_2 \overset{v}{\Rightarrow} \varepsilon$ *contains* $(n-1)$ *synchronization rules, and* $w = uav$. *By induction hypothesis, there exists a derivation* $\mathfrak{c}_2' \hat{=}\, (p_k||t_1).t_2 \overset{v}{\Rightarrow} \varepsilon$ *without applying synchronization rules. Then*

$$
\begin{aligned}
\mathfrak{c}_2 \hat{=}\, t &\overset{u}{\Rightarrow} ((p_i||p_j)||t_1).t_2 \\
&\overset{\lambda}{\Rightarrow} ((\varepsilon||p_j)||t_1).t_2 = ((p_j||t_1).t_2) \\
&\overset{a}{\Rightarrow} \underbrace{(p_k||t_1).t_2 \overset{v}{\Rightarrow} \varepsilon}_{\mathfrak{c}_2'}
\end{aligned}
$$

is the desired derivation without synchronization rules.

Corollary 5.4
$L(\Pi^{PAN}) \subseteq L(\Pi^{PA})$

Remark
It is possible that $L(\Pi^{PA})$ contains more counterexamples than $L(\Pi^{PAN})$. Example 5.24 such a situation: The counterexample of Π^{PA} in Example 5.23 generates the sequence $b\,c\,t$. This counterexample can never result in a counterexample of Π^{PAN} in Example 5.3 on page 78, because it uses only one part of the replaced synchronization rule (underlined in both figures). Such spurious counterexamples should be excluded since they are not in the original model.

5.4.3.2 Discovering Spurious Counterexamples

We discover such spurious counterexamples by running the counterexample in Π^{PAN} in order to detect whether it is a real one or a spurious one.

We use a Process Algebra Net model checker $\mathcal{M}^{\text{PAN}}$ to check whether the counterexample candidate is a spurious counterexample or not. However, the size of the Process Algebra Net $\Pi_{\mathfrak{c}}^{\text{PAN}}$ is in most cases much smaller than the size of Π^{PAN}, since only the rules in the counterexample $\mathfrak{c}$ need to be considered.

The Process Algebra Net $\Pi_{\mathfrak{c}}^{\text{PAN}}$ has the transition rules $(\rightarrow_{\mathfrak{c}} \cup \rightarrow_{\mathfrak{c}}^{\text{sync}}) \cap \rightarrow_{\Pi}^{\text{PAN}}$ where $\rightarrow_{\mathfrak{c}}$ is the set of transition rules used in the derivation of the counterexample $\mathfrak{c}$ and $\rightarrow_{\mathfrak{c}}^{\text{sync}}$ contains synchronization rules $p_i||p_j \overset{a}{\rightarrow} p_k$ iff $p_i \overset{\lambda}{\rightarrow} \varepsilon$ and $p_i \overset{a}{\rightarrow} p_k$ are used in $\mathfrak{c}$ and these two rules stem from the transformation $(*)$. Then $L(\Pi_{\mathfrak{c}}^{\text{PAN}}) \subseteq L(\Pi^{\text{PAN}})$. Therefore, it is only necessary to use the model checker $\mathcal{M}^{\text{PAN}}$ on $\Pi_{\mathfrak{c}}^{\text{PAN}}$ for checking counterexamples. Since the size of $\Pi_{\mathfrak{c}}^{\text{PAN}}$ is typically much smaller than the size of Π^{PAN}, the model checking time is feasible even if $\mathcal{M}^{\text{PAN}}$ is used.

Theorem 5.7
If $\mathfrak{c}$ is a counterexample in Π^{PAN}, there is a counterexample $\mathfrak{c}'$ of Π^{PA} so that $\mathfrak{c}$ is a counterexample of $\Pi_{\mathfrak{c}'}^{PAN}$.

Proof
We proof the following stronger claim by induction on the number n of applied synchronization rules:
Claim: *If $\mathfrak{c} \hat{=} t \overset{w}{\Rightarrow} \varepsilon$ is a derivation in Π^{PAN}, then $\mathfrak{c}$ is also a derivation in $\Pi_{\mathfrak{c}'}^{PA}$ where $\mathfrak{c}'$ is the derivation constructed in Theorem 5.6 without synchronization rules. $\Pi_{\mathfrak{c}'}^{PAN}$ is the Process Rewrite System as defined above.*
$n = 0$: *Since no synchronization rules are applied, $\mathfrak{c}' = \mathfrak{c}$ and $\Pi_{\mathfrak{c}'}^{PAN}$ contains exactly the rules used in $\mathfrak{c}'$, $\mathfrak{c}$ is also a derivation in $\Pi_{\mathfrak{c}'}^{PAN}$.*
$n > 0$: *According to the proof of Theorem 5.6 $\mathfrak{c}$ can be decomposed into*

$$\underbrace{t \overset{u}{\Rightarrow} ((p_i||p_j)||t_1).t_2}_{\mathfrak{c}_1} \overset{a}{\Rightarrow} \underbrace{(p_k||t_1).t_2 \overset{v}{\Rightarrow} \varepsilon}_{\mathfrak{c}_2}$$

where $\mathfrak{c}_1$ does not apply a synchronization rule, and $\mathfrak{c}'$ has the form

$$\underbrace{t \stackrel{u}{\Rightarrow} ((p_i||p_j)||t_1)}_{\mathfrak{c}_1}.t_2$$
$$\stackrel{\lambda}{\Rightarrow} ((\varepsilon||p_i)||t_1).t_2 = (p_i||t_1).t_2$$
$$\stackrel{a}{\Rightarrow} \underbrace{(p_k||t_1).t_2 \stackrel{v}{\Rightarrow} \varepsilon}_{\mathfrak{c}_2'}$$

Since $\mathfrak{c}_2$ contains $n-1$ synchronization rules by induction hypothesis, $\mathfrak{c}_2$ is a derivation in $\Pi^{PAN}_{\mathfrak{c}_2'}$. Let $\Pi^{PA}_{\mathfrak{c}_1}$ the Process Rewrite System obtained from the rules in $\mathfrak{c}_1$. Analogous to the case $n = 0$, we can argue that $\mathfrak{c}_1$ is a derivation in $\Pi^{PAN}_{\mathfrak{c}_1}$. The rules $p_i \stackrel{\lambda}{\rightarrow} \varepsilon$ and $p_j \stackrel{a}{\rightarrow} p_k$ are transformed back to $p_i||p_j \stackrel{a}{\rightarrow} p_k$. Hence, $\mathfrak{c}$ is a derivation in $\Pi^{PAN}_{\mathfrak{c}'} = \{Q, \Sigma, I, \rightarrow_{\mathfrak{c}_1} \cup \{p_i||p_j \stackrel{a}{\rightarrow} p_k\} \cup \rightarrow_{\mathfrak{c}_2'}, \{\varepsilon\}\}$.

This theorem guarantees, that if the Process Algebra Net contains a counterexample $\mathfrak{c}$, then it will be found by the verification algorithm, since $\mathfrak{c}'$ becomes a counterexample candidate during the loop.

In the Section 6.3 we show the experiences we made with this approach.

5.4.4 Summary

In this section we have shown, how the reachability analysis of Process Algebra Nets are defined and can be improved.

Two problems appear while using Mayr's PAN reachability for our purpose. First, the number of false negatives increases, if we do not use all pieces of information available in the Combined Abstraction, thus we define an extended verification algorithm named Round-robin reachability [BZ09a]. Second, the performance of the reachability analysis results in too many timeouts considering a high number of transition rules. As this could be a limitation for industrial case studies we develop an a posteriori model checking approach. Thereby, we take advantage of the well defined properties of the PRS-hierarchy [BZ09d]. PA-processes allow a conservative approximation of the source code. A faster computation of the counterexamples is possible, while putting up more false negatives. They are reduced with a standard Process Algebra Nets model checker. Although the complexity class of reachability cannot be reduced using our approach, it is possible to speed up the Process Algebra Nets model checking while reducing the size of the input model. Experimental results are shown in Section 6.3.1.

In the next section, we will discuss the notation of counterexamples in this work. Moreover, we describe how the counterexamples are used for leading back to the relevant program points influencing the counterexample.

5.5 Step 5: Evaluating Counterexamples

In the previous section model checking of Process Algebra Nets was discussed. It results in counterexamples if any protocol violation exists. Every counterexample has to be evaluated by

a human or by an automatic process. This is needed as we cannot guarantee, that the counterexample is a real or a spurious counterexample (cf. Section 5.4.1). Here, we will discuss the notation and the evaluation of them, leading to a better understanding at the user side.

E. g., the problem can appear, that if repetitions (or recursions) exist within a counterexample, many slightly different counterexamples will be created. If the model checker returns a set or sequence of counterexamples (cf. Section 5.4.3, Figure 5.6 on page 122), we do not want to confuse the user of the verification process while listing many counterexamples that describe the same behavior.

The problem is shown in the example of Example 5.25. There, some of the existing counterexamples are represented showing the interaction sequences and the derivations. As we can see, repetitions, parallel behavior and recursion might lead to an infinite number of counterexamples.

We will define adaptions of our earlier definitions to reduce the effort needed to evaluate counterexamples by humans. This contains the options to show the user:

1. comprehensible counterexamples,

2. compact counterexamples,

3. less counterexamples.

We will show, how these goals are ensured.

5.5.1 Extending the Counterexamples

In this work, the counterexamples contain not only the sequences of interactions. We assume, that it will be easier for the user of our verification process, if the information is mapped directly to the behavior of the source code.

Remark
We assume, that it will be difficult (nevertheless not impossible) to describe an automatic process evaluating counterexamples, because then the data-flow of the actual implementation has to be taken into account. This is resulting in testing, which might be incapable to determine the absence of errors (i. e., determine, that the counterexample is a false negative). Thus, we can assume, that the case has to be respected, where a human evaluates the counterexample and has to decide if a real error exists or not.

In Definition 4.5 on page 58 we have described the formal properties a counterexample should have. We assume here, that these minimal properties (sequences of interactions) will not last for the evaluation by the user, because it does not explain the execution trace used to generate the counterexample. Moreover, the number of counterexamples could be inflated, if parallel behavior is present (Example 5.25). For this reason we define *extended counterexamples*.

Definition 5.6 (Extended counterexample)

Example 5.25: Inflating number of counterexamples.

$$
\begin{aligned}
\Pi_S^C =& (Q_S^C, \Sigma_S^C, I_S^C, \rightarrow_S^C, F_S^C), \text{ where} \\
Q_S^C =& \{p_1, p_2, p_3, p_4\} \ \wedge \ Q_S^C \subseteq (Q_C \times Q_{P_C} \times Q_C) \\
\Sigma_S^C =& \{a, b, c, d\} \\
I_S^C =& p_1 \\
\rightarrow_S^C =& \{p_1 \stackrel{a}{\rightarrow} p_2||p_3,\ p_2 \stackrel{b}{\rightarrow} p_1.p_4,\ p_2 \stackrel{g}{\rightarrow} \varepsilon,\ p_3 \stackrel{c}{\rightarrow} p_4,\ p_4 \stackrel{f}{\rightarrow} \varepsilon,\ p_4 \stackrel{d}{\rightarrow} p_3\} \\
F_S^C =& \{\varepsilon\}
\end{aligned}
$$

(a) Given Combined Abstraction Π_S^C.

$$agcf \quad \underline{p_1} \stackrel{a}{\Rightarrow} \underline{p_2}||p_3 \stackrel{g}{\Rightarrow} \underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C1.a}$$

$$acgf \quad \underline{p_1} \stackrel{a}{\Rightarrow} p_2||\underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_2}||p_4 \stackrel{g}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C1.b}$$

$$acfg \quad \underline{p_1} \stackrel{a}{\Rightarrow} p_2||\underline{p_3} \stackrel{c}{\Rightarrow} p_2||\underline{p_4} \stackrel{f}{\Rightarrow} \underline{p_2} \stackrel{g}{\Rightarrow} \varepsilon \tag{C1.c}$$

$$agcdcf \quad \underline{p_1} \stackrel{a}{\Rightarrow} \underline{p_2}||p_3 \stackrel{g}{\Rightarrow} \underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_4} \stackrel{d}{\Rightarrow} \underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C2.a}$$

$$acgdcf \quad \underline{p_1} \stackrel{a}{\Rightarrow} p_2||\underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_2}||p_4 \stackrel{g}{\Rightarrow} \underline{p_4} \stackrel{d}{\Rightarrow} \underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C2.b}$$

$$acdcfg \quad \underline{p_1} \stackrel{a}{\Rightarrow} p_2||\underline{p_3} \stackrel{c}{\Rightarrow} p_2||\underline{p_4} \stackrel{d}{\Rightarrow} p_2||\underline{p_3} \stackrel{c}{\Rightarrow} p_2||\underline{p_4} \stackrel{f}{\Rightarrow} \underline{p_2} \stackrel{g}{\Rightarrow} \varepsilon \tag{C2.c}$$

$$abagcccffdcf \quad \underline{p_1} \stackrel{a}{\Rightarrow} \underline{p_2}||p_3 \stackrel{b}{\Rightarrow} (p_2.p_4)||p_3 \stackrel{a}{\Rightarrow} ((\underline{p_2}||p_3).p_4)||p_3 \stackrel{g}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{c}{\Rightarrow} (p_4.p_4)||\underline{p_3} \stackrel{c}{\Rightarrow} (\underline{p_4}.p_4)||p_4 \stackrel{c}{\Rightarrow} p_4||\underline{p_4} \stackrel{f}{\Rightarrow} \underline{p_4} \stackrel{d}{\Rightarrow} \underline{p_3} \stackrel{c}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C3.a}$$

$$abagcdccfff \quad \underline{p_1} \stackrel{a}{\Rightarrow} \underline{p_2}||p_3 \stackrel{b}{\Rightarrow} (p_2.p_4)||p_3 \stackrel{a}{\Rightarrow} ((\underline{p_2}||p_3).p_4)||p_3 \stackrel{g}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{c}{\Rightarrow} (\underline{p_4}.p_4)||p_3 \stackrel{d}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{c}{\Rightarrow} (p_4.p_4)||\underline{p_3} \stackrel{c}{\Rightarrow} (\underline{p_4}.p_4)||p_4 \stackrel{f}{\Rightarrow} p_4||\underline{p_4} \stackrel{f}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C3.b}$$

$$abagcddcddcfff \quad \underline{p_1} \stackrel{a}{\Rightarrow} \underline{p_2}||p_3 \stackrel{b}{\Rightarrow} (p_2.p_4)||p_3 \stackrel{a}{\Rightarrow} ((\underline{p_2}||p_3).p_4)||p_3 \stackrel{g}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{c}{\Rightarrow} (\underline{p_4}.p_4)||p_3 \stackrel{d}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{d}{\Rightarrow} (\underline{p_3}.p_4)||p_3 \stackrel{c}{\Rightarrow} (p_4.p_4)||\underline{p_3} \stackrel{d}{\Rightarrow} (p_4.p_4)||\underline{p_4} \stackrel{d}{\Rightarrow} (p_4.p_4)||\underline{p_3} \stackrel{c}{\Rightarrow} (\underline{p_4}.p_4)||p_4 \stackrel{f}{\Rightarrow} p_4||\underline{p_4} \stackrel{f}{\Rightarrow} \underline{p_4} \stackrel{f}{\Rightarrow} \varepsilon \tag{C3.c}$$

(b) Some resulting counterexamples.

Example 5.26: Extended counterexample of Example 5.25a on page 129.

$$
\begin{aligned}
\mathfrak{c}_{\otimes 1} &= (p_0, w_{\otimes 1}, T_{\otimes 1}), \text{ where} \\
w_{\otimes 1} &= [\ a, g, c, f\ ] \\
T_{\otimes 1} &= \{\ p_1 \xrightarrow{a} p_2||p_3,\ p_2 \xrightarrow{g} \varepsilon,\ p_3 \xrightarrow{c} p_4,\ p_4 \xrightarrow{f} \varepsilon\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{c}_{\otimes 2} &= (p_0, w_{\otimes 2}, T_{\otimes 2}), \text{ where} \\
w_{\otimes 2} &= [\ a, g, c, d, c, f\ ] \\
T_{\otimes 2} &= \{\ p_1 \xrightarrow{a} p_2||p_3,\ p_2 \xrightarrow{g} \varepsilon,\ p_3 \xrightarrow{c} p_4,\ p_4 \xrightarrow{d} p_3 p_4 \xrightarrow{f} \varepsilon\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{c}_{\otimes 3} &= (p_0, w_{\otimes 3}, T_{\otimes 3}), \text{ where} \\
w_{\otimes 3} &= [\ a, b, a, g, c, c, c, f, f, d, c, f\ ] \\
T_{\otimes 3} &= \{\ p_1 \xrightarrow{a} p_2||p_3,\ p_2 \xrightarrow{b} p_1.p_4,\ p_2 \xrightarrow{g} \varepsilon,\ p_3 \xrightarrow{c} p_4,\ p_4 \xrightarrow{d} p_3,\ p_4 \xrightarrow{f} \varepsilon\ \}
\end{aligned}
$$

An extended counterexample defines a derivation $I_{\otimes} \overset{w_{\otimes}:T_{\otimes}}{\Longrightarrow} \varepsilon$, *where*

$I_{\otimes} \in Q$	*is an initial atomic process constant,*
$w_{\otimes} \hat{=} (\Sigma \times \mathbb{I})^n$	*is a sequence of interactions,*
$T_{\otimes} \subseteq \rightarrow$	*is the set of used transitions,*
$n \in \mathbb{N}$	*is the length of the counterexample .*

Thus, the extended counterexample describes that the interaction $a \in w_{\otimes}$ in the n-th step of the derivation $I_{\otimes} \overset{*}{\Rightarrow} \varepsilon$ is generated using the transition rule $\delta \in T_{\otimes}$. This definition is an explicit representation of the derivation path. It shows a compact representation in form of an interaction sequence without program points.

Using this definition and the fact, that each process constant can be mapped to a unique program point (Remark 5.1.1 on page 81) the user can browse through the program code (he can access) to determine the problem causing the counterexample.

Moreover, extended counterexamples are used to reduce the number of counterexamples, represented at the user interface. For this purpose every computed counterexample $\mathfrak{c}_{\otimes i} = (I_{\otimes i}, w_{\otimes i}, T_{\otimes i})$ is checked against the previously existing extended counterexamples $\mathfrak{c}_{\otimes j} = (I_{\otimes j}, w_{\otimes j}, T_{\otimes j})$. If a $T_{\otimes j}$ exists so that $T_{\otimes i} \subseteq T_{\otimes j}$ and $I_{\otimes i} = I_{\otimes j}$, then the interaction sequence of the currently computed counterexample $\mathfrak{c}_{\otimes i}$ is captured already by another counterexample $\mathfrak{c}_{\otimes j}$. Hence, we will not show $\mathfrak{c}_{\otimes i}$ to the user. Therefore, the number of showed counterexamples is reduced (Example 5.26). There, three extended counterexamples are shown. These counterexamples capture the (traditional) counterexamples shown in Figure 5.25b, precisely:

- the extended counterexample $\mathfrak{c}_{\otimes 1}$ captures the counterexamples C1.a, C1.b and C1.c of Figure 5.25b.

- the extended counterexample $\mathfrak{c}_{\text{Ex}2}$ captures the counterexamples C2.a, C2.b, C2.c (and many more) of Figure 5.25b.
- the extended counterexample $\mathfrak{c}_{\text{Ex}3}$ captures the counterexamples C3.a, C3.b, C3.c (and many more) of Figure 5.25b.

Remark
An extended counterexample describes not always a single forbidden trace. It is possible that a counterexample can be interpreted in different execution orders. We expect from the user to evaluate repetitions, too.

5.5.2 Evaluating Extended Counterexamples

If the counterexample is caused by the overapproximation, the user can mark the counterexample as invalid. In this case several options exist[14]:

- The counterexample can be used to refine the abstraction of the considered component. Thus, this specific counterexample will never appear again. As sideeffect the verification might speed up, because several derivations are excluded.
- The counterexample can be marked as irrelevant. In this case it is respected, that a user might not exclude a counterexample, only assumes that it cannot appear in the actual implementation. A combination with machine learning [GH88, BN07, Qui93] techniques seem to be promising.

Both options will reduce the counterexamples the user is confronted with.

Remark
The latter case is used in the frontend, we have developed for the case study (cf. Chapter 6). We choose this implementation, because we want to be sure, that every possible counterexample will appear at the user interface. We assume, that this is the safest representation as we have no access to the source code, which the user probably has.

5.5.3 Summary

The format we use as representation of the counterexamples allows the developer an aggregated view on the problems contained in the source code. We assume, that this representation allows an easy and isolated look on the component behavior. However, only long term case studies can show, which information will be useful for the developers too.

The counterexamples can be used in a CEGAR-loop. If a counterexample was marked by the developer as false negative, it is possible to sharpen the behavior of the abstractions or protocols using the defined approach (but this option was not implemented within this work). Thus, a verification run (based on this new information) might run faster and results in less counterexamples, probably.

[14]The list claims not to be exhausting.

6 Implemented Framework and Case Study

In this chapter the implemented framework and the industrial partners as well as the results of optimizations are presented. In particular, the target consideration of industrial source code leads to specific requirements that have to be fulfilled while implementing a framework. This framework is described in Section 6.1. In Section 6.2 the results of the optimizations are presented while considering three industrial case studies. In the same section an academic case is introduced. It is used to compensate lacks of the industrial case studies with respect to the properties of the verification approach. In Section 6.2 the influence of the earlier defined optimizations on abstractions (created from real source code) is considered. The performance of the optimized model checker is considered in Section 6.3.

Unfortunately, not all steps of the verification process could be evaluated within the industrial case studies. This was a matter of time. Therefore, component protocols are available rarely. Nevertheless, the infrastructure and the needed components are implemented completely.

6.1 Implemented Framework

While implementing the case studies several requirements have to be complied. The most important ones are:

- The industrial partner demands that no source code will be transferred over the internet (neither decrypted nor encrypted).
- The different steps should be deployable in different locations leading to a better usage rate (particularly with regard to the model checker).
- The parts of the verification process should be able for separate improvements.

To realize the case study and allow future use, we decided to implement a Web Service based architecture. This ensures an independent development of the components needed for the process. The architecture divides the tasks in four parts, which are similar to the steps of the verification process (cf. Figure 5.1 on page 74): It consists of components implementing the following interfaces:

- *Abstractions (A)*: generate the single component abstractions and compose them to a Π_S as requested by a user (discussed in Section 6.1.1),
- *PRS Operations (O)*: hide the different abstraction services, compute Combined Abstractions and optimize Combined Abstractions and system abstractions as requested by a user (discussed in Section 6.1.2),
- *Model Checker (B)*: solves the model checking problem, delivers the computed counterexamples to the user interfaces (discussed in Section 6.1.3),
- *User Interfaces (U)*: provides interfaces for end users, enabling a comfortable handling of the verification process (discussed in Section 6.1.4).

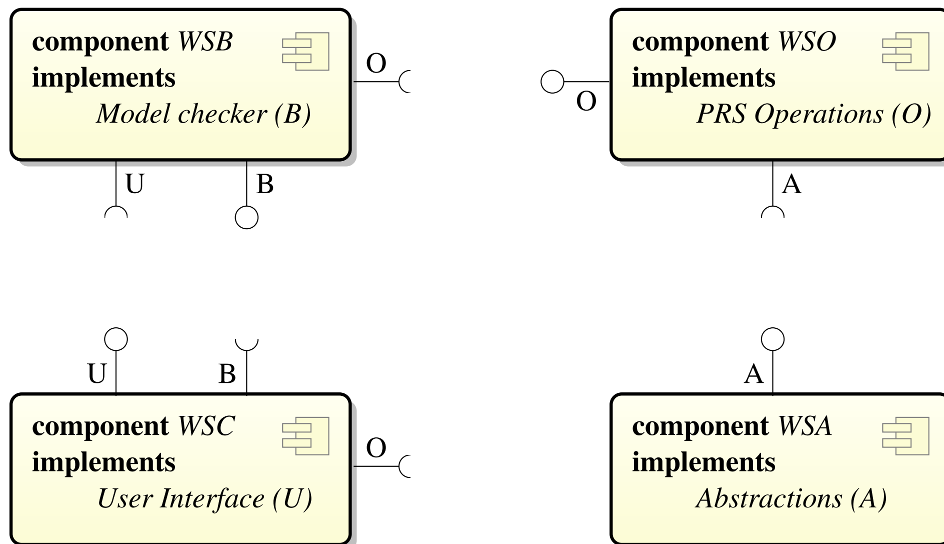


Figure 6.1.: Overview of implemented architecture.

An overview is given in Figure 6.1. The Web Services are designed to implement the main workflows shown in Figure 6.2 on page 135 . The main task of an end user is to get an overview of the implementation and thereafter create a model checking scenario which is checked at the end.

In the following the implemented Web Services are described in detail.

6.1.1 Abstractions (short: WSA)

Several works [Kra08, Kra09, Rud10, Hel10] were and are supervised during this thesis. The aim of these works was to implement generators of source code abstractions. A brief technical definition of this Web Service is given in Figure 6.3 on page 136 . The complete interface definition is shown in Listing A.1.

6.1.1.1 Translation of Python Statements into PRS Transition Rules

In [Kra08, Kra09] abstractions of Python [Fou09b] applications were computed partly. The abstraction process of Python source code is realized using the built-in Python "ast" module [Fou09a]. This module provides a syntax tree containing no information about the behavior of the source code, the computation of such information has to be implemented. The implementation of [Kra09] was improved vitally to make it usable for this thesis.

6.1.1.2 Translation of PHP Statements into PRS Transition Rules

In [Rud10] abstractions of applications written in PHP [Gro09] are considered, currently. Here, the "open source PHP compiler" (PHC) [dVG07, BdVG09] was used. PHC is a well structured and pluginable compiler frontend. It is implemented using the visitor design pattern [GHJV95]. Thus, the visitors, needed for creating statements of the target programming language are re-

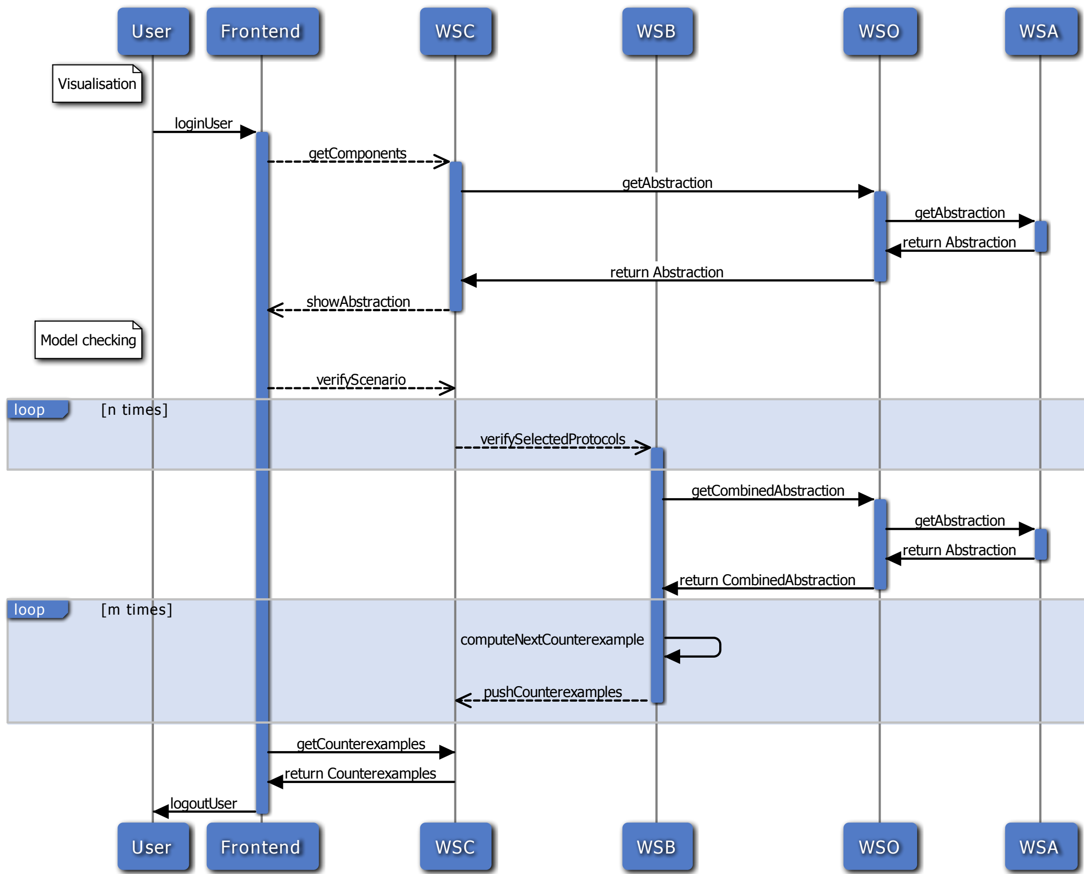


Figure 6.2.: Implemented workflow.

placed, so PRS transition rules, are computed. A preeminent property of this case study is that parallel operators only appear in the top level tier of the architecture. However, this level was currently not considered in [Rud10]. Hence, it was not available for evaluation.

6.1.1.3 Translation of BPEL Activities into PRS Transition Rules

In [Hel10] BPEL applications are translated into PRS transition rules, currently. For this purpose the XML files are parsed and evaluated. The implementation captures the following activities and handler:

- Acitivities
  - `Empty`
  - `Assign`
  - `Exit`
  - `Invoke`
  - `Receive`
  - `Reply`

interface ***Abstractions (A)***	
getListOfComponents	returns the list of available components
getAbstractions	returns the specified Process Rewrite Systems
getComponentImplementationInformation	returns a map of component's information
getProtocol	returns the specified protocol

Figure 6.3.: Definition of Web Service interface "Abstractions".

- `Rethrow`
- `Throw`
- `Wait`
- `Flow`
- `ForEach`
- `If`
- `Pick`
- `RepeatUntil`
- `Scope`
- `Sequence`
- `While`

• Handler

- `Fault Handler`
- `Event Handler`
- `TerminationHandler`

6.1.1.4 Representation of PRS Transition Rules

The transition rules need a deeper consideration to match the requirements of the case studies. PRS transition rules are defined in the implemented components as:

$$\rightarrow \hat{=} PEX(\texttt{NodeId}) \times \texttt{Integer} \times PEX(\texttt{NodeId})$$

Thus, the action is encoded as `Integer`. The struct `NodeId` is defined as:

$$\texttt{NodeId} \hat{=} \texttt{Integer} \times \texttt{Integer} \times \texttt{Integer} \times \texttt{Integer}$$

The current protocol state, the target protocol state, the current process in the system abstraction and a unique component identifier are used to represent a process constant (`NodeId`). All

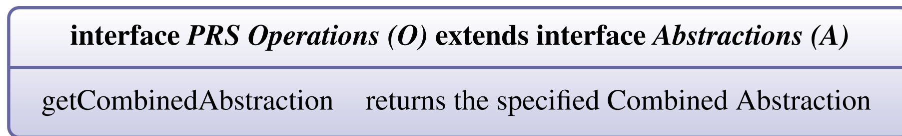


Figure 6.4.: Definition of Web Service interface “PRS Operations”.

services are implemented as Web Services. Hence, the unique component identifier can be derived from the uniform resource identifier of the current Web Service. As we use `Integer`-representations only, it is not possible to reconstruct the source code behavior using the PRS representation. This means in effect that we cannot be sure if a real counterexample is found while performing the case study.

Remark
The interface definitions (cf. Listings A.1, A.2, and A.3) contains much more information (e. g., `getComponentImplementation`, `getActionInfo`, `getNodeInfo`), that could be used to enrich the PRS representation at a user interface with more pieces of information (e. g., the version of the revision control system of the component). The provider of the source code decides which pieces of information are acceptable via these methods so that the behavior (that might contain business secrets) is still protected.

6.1.1.5 Summary

These implementations are accessible via a Web Service interface – called shortly *WSA*. They create abstractions of a given set of source code and translate them into Process Rewrite Systems. Thereby, optimizations specific for the considered programming language are performed. The source code of the given files is never made accessible for other services, only PRS representations are provided. Thus, one main requirement – the protection of the encoded business logic – is achieved.

6.1.2 PRS Operations (short: WSO)

The source code analyses implemented in the distinguished Web Services (described in the previous section) make the PRS representations of the considered source code available. They implement different optimizations to create smaller representations. It was the explicit aim to implement optimizations of the Process Rewrite Systems. This is done by the implementation of a Web Service called *PRS Operations* – abbreviated *WSO*.

This service is implemented as Web Service (a brief description is shown in Figure 6.4), too. The main purpose of this service is to create a Combined Abstraction by request and provide several optimizations for Process Rewrite Systems (a selection is discussed in Sections 5.2.3, 5.3.2). It requires a WSA implementation and implements the same interfaces as WSA (cf. Listing A.1). The idea behind this is a possible pipes and filters architecture. It enables to join onto optimizations implemented in future work or other groups.

interface ***Model checker (B)***	
startSearch	starts the search for counterexamples in the given Process Rewrite System
cancelTask	stops a verification task
getStatus	returns status of requested verification task

Figure 6.5.: Definition of Web Service interface "Model checker".

Remark
It is clear that it would not be possible to compose many implementations of optimizations (WSO) in a chain, as the performance might be reduced too much. Therefore, the current implementation implements a strategy design pattern enabling the extension by new optimizations within WSO, too.

Summarized, the service encapsulates the implementation of WSA. It provides the pieces of information to the user interfaces and the model checker.

6.1.3 Model Checker (short: WSB)

The Web Service *model checker* was defined and implemented in [Prä09], a diploma thesis we have supervised. After getting a task of the user interface, it solves the reachability problem for Process Algebra Nets and delivers counterexamples to the user interfaces asynchronously. The Web Service is called WSB shortly (definition is shown in Figure 6.5).

To our knowledge there exists no Process Algebra Nets model checker before this work. For this reason one has to define a new framework first. In [Prä09] the Petri nets tool TINA [BRV04] is drawn in comparison, which has in particular problems solving unbounded Petri nets (e. g., generic examples with less than 60 rules crashing after claiming 11 GB of RAM). Here, the goal was to define a process, which is not as vulnerable as other Petri nets reachability toolkits. For this reason the Petri nets reachability process (Petri nets oracle) was divided into three phases:

1. An overapproximated set of transition rules is calculated leading possibly to a derivation $I \overset{*}{\Rightarrow} \varepsilon$, where I is the initial marking of the Petri net (in the PRS context the initial marking always contains only one token). For the computation the state equation technique is used. The result is a symbolic solution representing all possible sets of transition rules.

2. The symbolic solution of the first step is translated into a concrete allocation using an integer linear programming solver. This is called *explicit candidate* ϑ.

3. In this last step the explicit candidate is checked while traversing through the Petri net. As the explicit candidate ϑ contains specific rules that have to be used for the computation of the counterexample, the derivation is aborted if the next derivation step is not possible using a rule of ϑ.

interface *User Interface (U)*	
appendCounterExample	pushs a counterexample to the database
calculationFinished	signals the completion of a specific verification task

Figure 6.6.: Definition of Web Service “user interface”.

At each step the counterexample or set of counterexamples can be marked as invalid. E. g., for many examples it could be computed in the first phase, that no counterexample is contained [Prä09]. This approach ensures a practical applicability. Model checking results are presented in [Prä09].

6.1.4 User Interfaces (short: WSC+P2)

In a project [FFB+09] supervised by us, a framework was implemented using Java and C#. It provides several alternatives for controlling and manipulating the phases of the verification process. The framework is called “P2” which is an abbreviation of the German word “Protokoll-Prüfung” meaning protocol checking. It was presented in [FB09].

The framework contains a server implementation (definition is shown in Figure 6.6) providing multi-user support and an interface for interactions with other Web Services. The frontend supports the following operations (cf. Figure 6.7):

- defining new protocols,
- showing a graphical representation of components and applications,
- defining applications and protocols (named scenarios) which should be verified,
- starting of verifications,
- evaluating delivered verifications (counterexamples) using the abstractions
- representing extended component information (e. g., author, last change, ...[1])

Moreover, an API was implemented providing Python support. It allows to start verifications and to evaluate counterexamples. This opens the option to an integration in the nightly checks of the source code at the industrial partners.

The evaluation of the counterexamples is used to improve the full set of counterexamples. If one user has marked a counterexample as invalid, all other users profit from this information as this counterexample is marked as *false negative* (cf. Section 5.4.1). The more users this counterexample exclude the less important it gets, until it disappears. This rating mechanism should help to improve the verification results.

[1] The set of information is flexible and depends on the configuration by the provider of the considered WSA implementation.

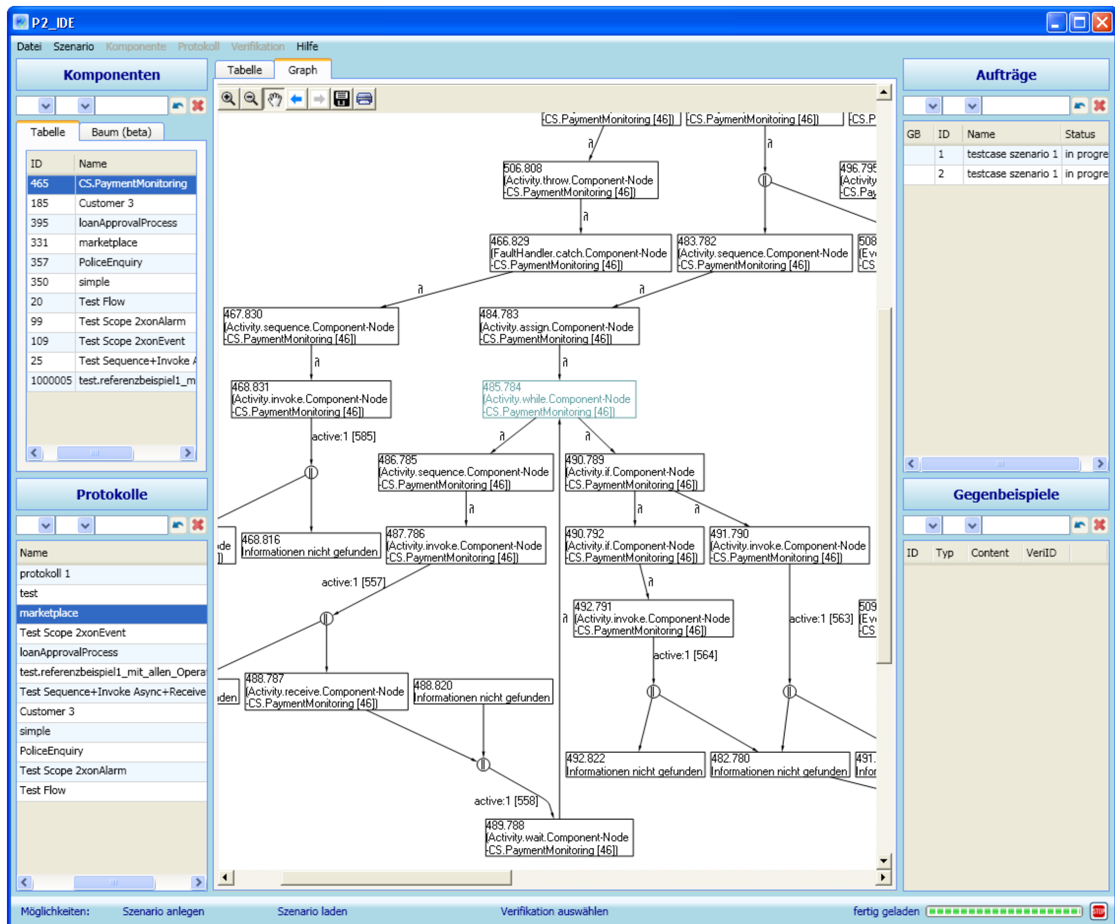


Figure 6.7.: Screenshot of implemented frontend “P2”.

This information gets weakened if the version (e. g., SVN revision number) or the abstraction of the source code file changes (if this information is provided by the Web Service “Abstraction”).

6.1.5 Summary

In this section we have presented the framework that was implemented. The correlation between the verification process and the implementations is sketched in Figure 6.8. The framework has well defined interfaces. It is implemented using Web Services. Thus, it is easy extendable and adaptable. Moreover, the main property for evaluating industrial application is ensured, as the source code is always hidden.

The user of the verification process can use a graphical user interface (GUI) or a Python interface providing the API (e. g., for frequently nightly performed verifications). The implementation of the Web Services for user interfaces omits the verification process and provides multi-user support. It is capable to evaluate the user interactions and conclude from them whether a delivered counterexample is a false negative or not. The GUI can represent the abstract behavior of components and composed components using a graphical representation (cf. Figure 6.7). PRS

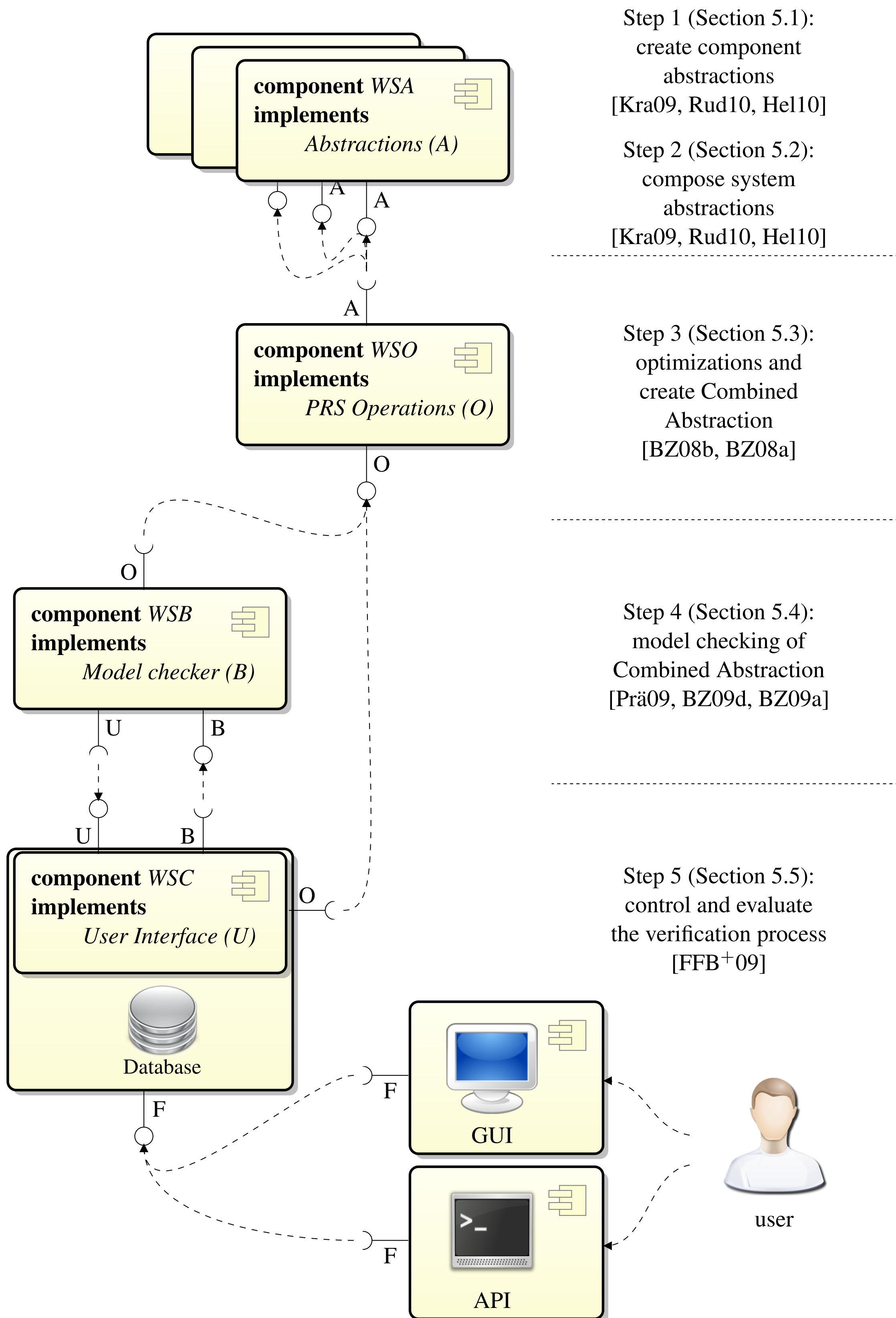


Figure 6.8.: Correlation of verification process (Figure 5.1 on page 74) and implemented components.

transition rules are represented only in this way.

To our experience a graphical representation of the considered extended counterexample is useful for the evaluation.

6.2 Considered Case Studies

In this section we will discuss abstract behavior generated from real source code. For this purpose four case studies were performed. Besides two industrial case studies (Section 6.2.1 and Section 6.2.2), an open source application (Section 6.2.3) and an academic case study (Section 6.2.4) are considered. At each case study the optimizations described in Section 5.2.3.1 and in Section 5.3.2.2 were processed on the given set of components.

6.2.1 OR Soft Workbench

The industrial partner – the OR Soft Jänicke GmbH (settled in Merseburg, Germany) – allowed us to access a part of the source code used to implement the "OR Soft Workbench". This is a comfortable frontend for SAP® ERP and SAP® SCM (SAP® R/3® and SAP® APO, respectively) written in Python [Fou09b] and C++ [ES90]. A part of product implemented in Python was made available. This implementation is interpreted as component-based software as several parts are composed to deployment units fitting the requirements of distinguish customers (from the areas Chemical, Pharmaceutical, Oil and Gas, Automotive, Consumer Products and Manufacturing).

One target in this cooperation was to check the components composed to different products continuously (nightly checks) in the sense of correct usage. This leads to more flexibility on combining components to new products. Moreover, a self-sufficient user of the verification process can be sure, that the interactions with other developer groups are in the manner they expect. Thus, we are interested in providing an integrated verification suite that allows a distributed verification by as many developers as possible. This goal was not reached finally as it is currently not possible to consider C++ source code files.

The case study contains:

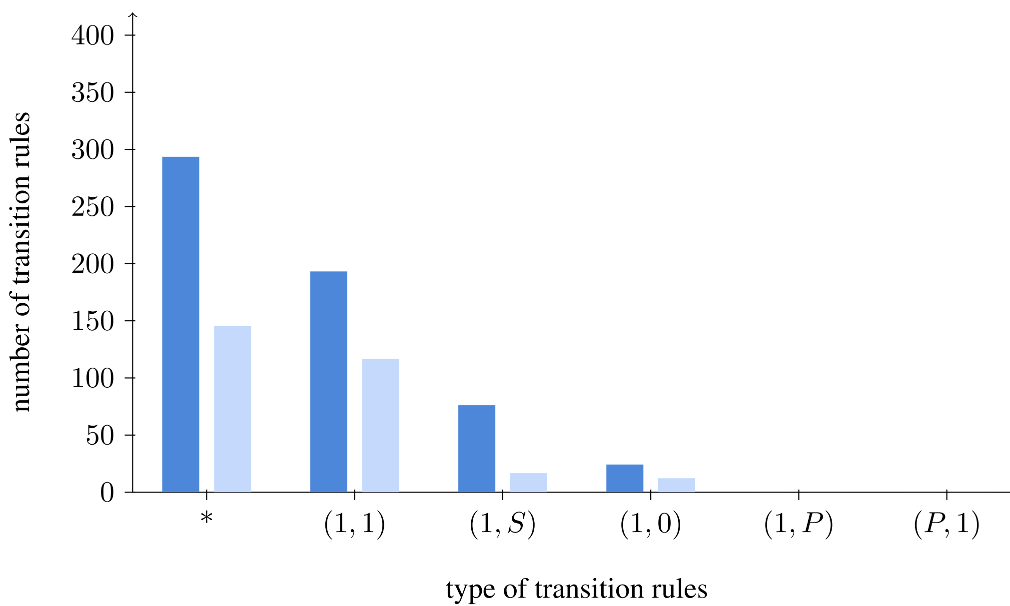
- 942 classes written in Python
- 219978 lines of code
- the largest considered scenario contains 3069 transition rules

As in Python it is possible to import each single method, we consider each class and each method as component in our sense. 300 components are randomly chosen from the existing set of components and considered in this part of the case study. From each of these components a scenario is created, while including all components that are used by the randomly picked components.

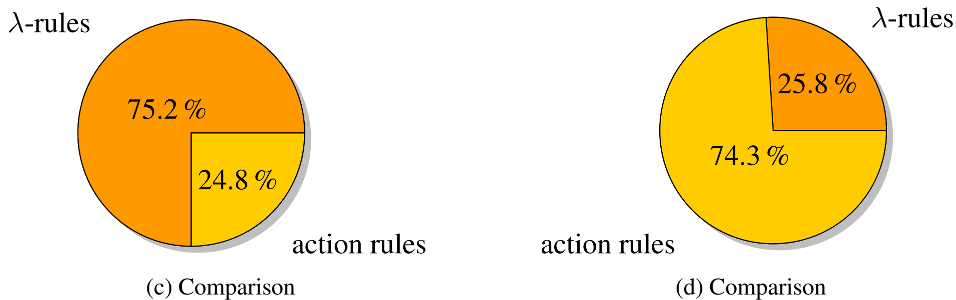
The abstraction service was implemented in [Kra08, Kra09] and strongly improved for this thesis. The results are represented in Figure 6.9. The same figure shows a comparison of the

type of transition rules	average number of transition rules	average number after optimizations	reduction rate
overall number (*)	293.27	145.26	-50.5 %
chain rules (1,1)	193.02	116.41	-39.7 %
sequential rules (1,S)	76.04	16.64	-78.1 %
elimination rules (1,0)	24.2	12.21	-49.5 %
fork rules (1,P)	0.01	0.01	0 %
synchronization rules (P,1)	0.01	0.01	0 %

(a) Comparison



(b) Comparison



(c) Comparison

(d) Comparison

Figure 6.9.: Case study: OR Soft Workbench.

transition rules, before and after the optimization. It shows a good reduction rate of the chain transition rules. The comparison of the λ-rules and action rules is shown in Figure 6.9c and Figure 6.9d.

The case study shows a significant lack of parallel behavior. An evaluation of the source code (of the considered branch of the product) shows that just at a few spots parallel behavior is contained. A discussion with the industrial partner points out that less parallel behavior is used in the sense of communication (in this part of the source code). Most parallel behavior is used to improve algorithms or is part of the frontend of the "OR Soft Workbench". Moreover, it is contained in parts of the product that is implemented using C++, in many cases. This part could not be evaluated yet, as no abstraction component was available that considers C++.

Only a few protocols are provided by the industrial partner. Nevertheless, one real counterexample was computed describing unexpected behavior contained in the source code. It matches the file handler pattern (open file, write values, close file).

6.2.2 EMenue.net

The second industrial partner is the ComServ Ingenieurgesellschaft mbH (settled in Halle (Saale), Germany). They would like to get more information about the structure and execution dependencies of the source code of their product "EMenue.net". It is a web-based shop for "Meals on Wheels" containing the administration of orders, drivers and billings. It is implemented in PHP [Gro09] using AJAX[2] technologies. The programming language PHP itself does not allow concurrent implementations.

The case study contains:

- 263 classes with 1523 methods
- 30821 lines of code
- the largest considered scenario contains 2302 transition rules

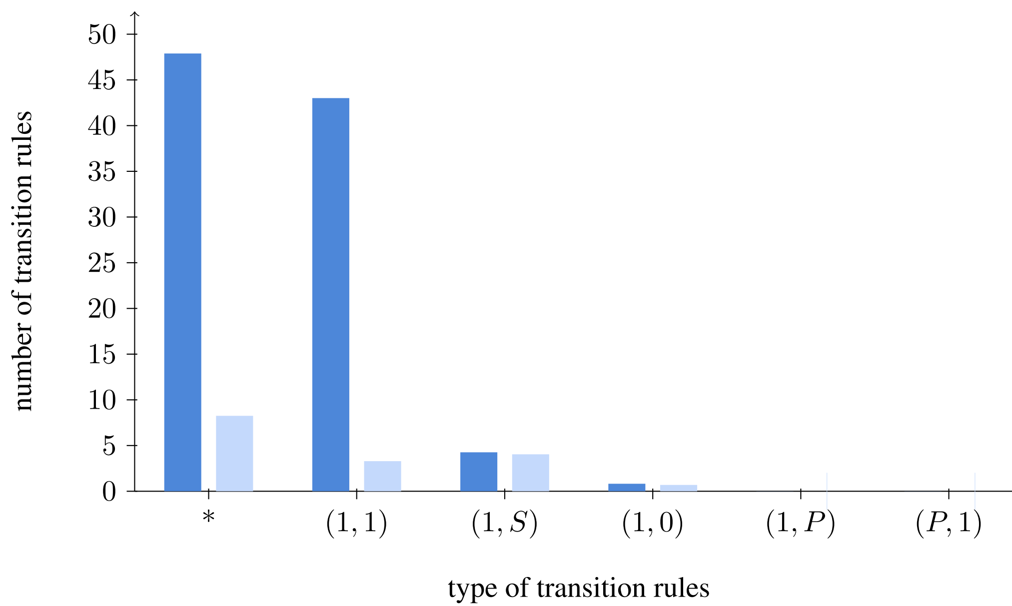
We interpret each method and class as component, where classes are components aggregated from components. 1068 components are considered in this part of the case study, all other components are already reduced to one transistion rule by Web Service "Abstractions". Here, each component is considered separately, because the current implementation of [Rud10] does not provide a system abstraction for all components (and their used components).

The results of the optimizations are represented in Figure 6.10. They show that a higher share of the transition rules are chain rules. This is caused by the used tool PHC (cf. Section 6.1.1.2). As this tool was developed for translating PHP to C++ it considers every data value. Using visitor patterns the code generation was exchanged. Thus, for each in former times generated C++ statement, now a PRS transition rules is computed. This leads to a significant higher number of chain transition rules (cf. Figure 6.10d on page 145), especially in contrast to the tool

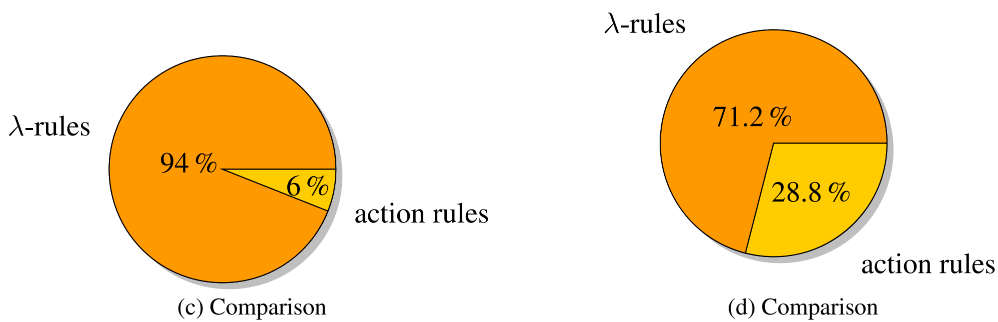
[2]AJAX $\widehat{=}$ Asynchronous JavaScript and XML.

type of transition rules	average number of transition rules	average number after optimizations	reduction rate
overall number (*)	47.88	8.25	-82.8 %
chain rules (1,1)	43	3.29	-92.3 %
sequential rules (1,S)	4.26	4.04	-5.4 %
elimination rules (1,0)	0.82	0.69	-15.9 %
fork rules (1,P)	0.01	0	–
synchronization rules (P,1)	0.01	0	–

(a) Comparison



(b) Comparison



(c) Comparison (d) Comparison

Figure 6.10.: Case study: EMenue.net.

presented in Section 6.1.1.1 that was implemented for the purpose of generation the abstract behavior of components only.

Because of these characteristics a reduction of the transition rules is necessary, in particular. The results of the optimizations are shown in Figure 6.10. A strong impact of the optimizations is visible leading to an overall reduction rate of 82.8%. Nevertheless, there are still improvements possible, as Figure 6.10d shows a share of 71.2% of λ-rules after the optimizations.

Early results of the model checking show a good applicability. Even now a real counterexample was found. Several parts of "EMenue.net" are implemented using the design patterns similar to Martin Fowler's Enterprise Application Architecture [Fow02]. These patterns were formulated as component protocols. More details will be accessible in [Rud10].

6.2.3 Fail2Ban

The application "Fail2Ban" [JB09] is an instrusion prevention framework. It is used to observe the login tries at a server and is able to prevent too many of them. Fail2Ban is an open source program written in Python. We chose this case study because parallel behavior is included in the source code. The same implementation was used for the abstractions process as used in Section 6.2.1. This case study considers the current version 0.84 (2009-09-07).

It contains:

- 45 python source code files
- 59 classes with 389 methods
- 5907 lines of code
- the largest considered scenario contains 4973 transition rules

As in the Section 6.2.1 classes and methods are interpreted as components.

The optimizations show a similar result as in Section 6.2.1 (Figure 6.11). Most of the λ-rules were eliminated (Figure 6.11d).

6.2.4 BPEL workflows

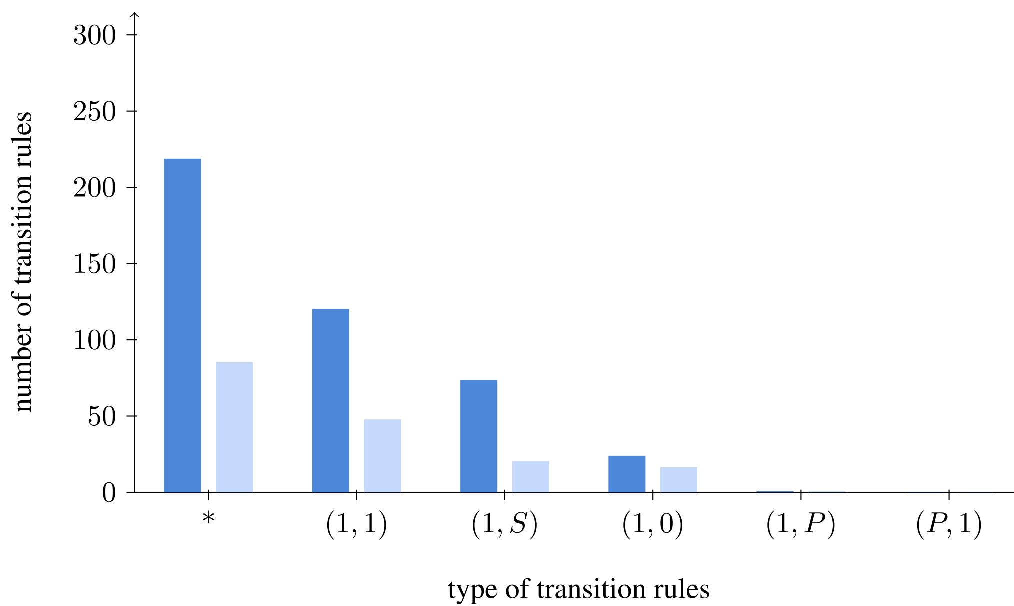
The previous case studies lack of parallel behavior. For this reason several workflows implemented using BPEL (cf. Section 5.1.2) are considered. The academic case study was created in [Hel10]. It is the smallest case study:

- 68 processes (Web Services, components)
- 4560 lines of code (lines containing XML statements)
- the largest considered scenario contains 69 transition rules

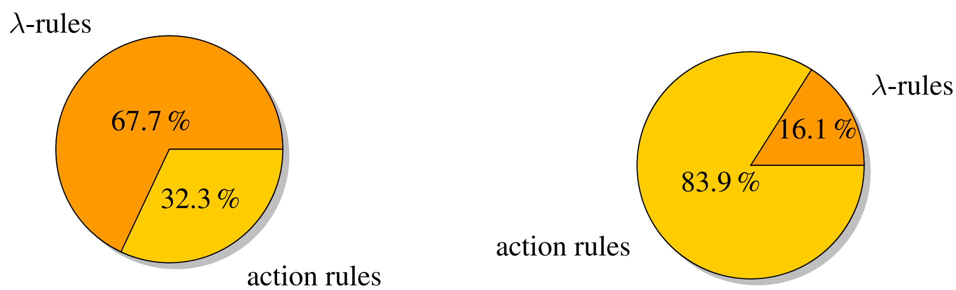
The source code of this case study was created for the purpose of evaluation in the context of this thesis only. The result of the optimizations is presented in Figure 6.12 on page 148 . It contains as less data definitions as possible. Hence, most of the λ-rules are used to steer the control flow. Therefore, their share after the optimization is still high as shown in Figure 6.12d.

type of transition rules	average number of transition rules	average number after optimizations	reduction rate
overall number (*)	219.22	85.48	-61 %
chain rules (1,1)	120.49	47.9	-60.2 %
sequential rules (1,S)	73.82	20.45	-72.3 %
elimination rules (1,0)	24.04	16.43	-31.7 %
fork rules (1,P)	0.67	0.6	-10.4 %
synchronization rules (P,1)	0.2	0.11	-45 %

(a) Comparison



(b) Comparison



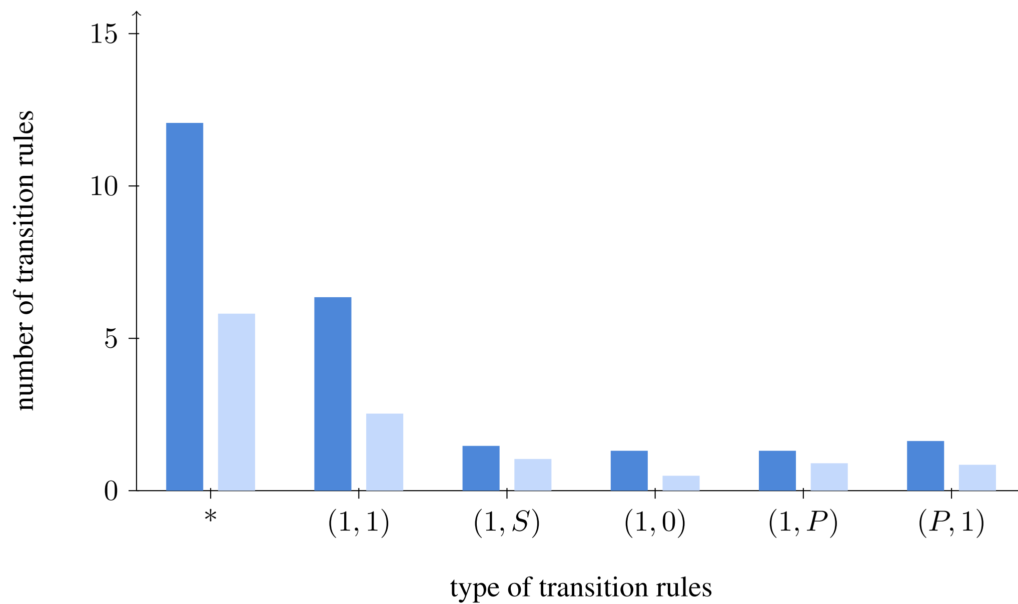
(c) Comparison

(d) Comparison

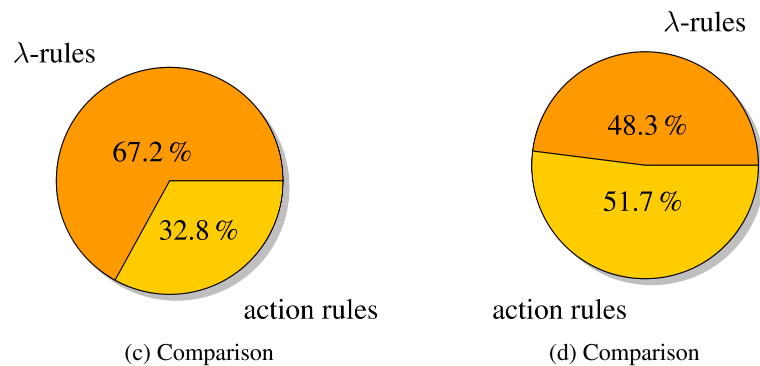
Figure 6.11.: Case study: Fail2Ban.

type of transition rules	average number of transition rules	average number after optimizations	reduction rate
overall number (*)	12.07	5.81	-51.9 %
chain rules (1,1)	6.35	2.53	-60.2 %
sequential rules (1,S)	1.47	1.04	-29.3 %
elimination rules (1,0)	1.31	0.49	-62.6 %
fork rules (1,P)	1.31	0.9	-31.3 %
synchronization rules (P,1)	1.63	0.85	-47.9 %

(a) Comparison



(b) Comparison



(c) Comparison (d) Comparison

Figure 6.12.: Case study: Web Services Business Process Execution Language.

6.2.5 Summary

The source code abstractions of the case study show a significant impact on the number of transition rules in contrast to the non-optimized abstraction. As the different implementations of the abstraction Web Services (WSA) are very distinguish, it is difficult to determine the reason for this observation.

We have shown at several examples (early results, not contained in this thesis), that it is possible to reduce the many Combined Abstractions to the existing counterexamples or eliminate all transition rules without using model checking techniques. This observation matches with [Ric08]. It should be evaluated in future work. Nevertheless, the implemented optimizations improve the practical applicability of the protocol conformance checking while reducing the transition rules significantly.

The reduced number of transition rules of the optimized Process Rewrite Systems is relatively low. After the composition to larger case studies the number of transition rules might increase to a size that cannot be handled by the current implementation of the model checker (WSB). Here a main advantage of our verification process has a good effect. As the number of action rules is reduced (translated into λ-rules) while creating the specific system abstraction $\Pi_{S,C}$ (considering component C), it exists more potential for reducing transition rules. This has also a positive impact on the model checking time in general. This can be reduced to the rule-of-thumb visualizing the impact on an algorithm which is EXPSPACE-hard:

$$2^n \geq k \cdot 2^{r \cdot n}$$

where n is the number of transition rules, k is the number of protocols that should be checked, and $r < 1$ is the average reduction rate (of scenarios in the current case study). If this rule-of-thumb also occurs, while evaluating large scenarios based on real source code, could not be evaluated within this work, because of the effort caused short-termed at the industrial partners. Only long term case studies can evaluate it sufficiently. Therefore, we consider in the next section a generic model checking case study.

6.3 Verifying Combined Abstractions

Because of the lack of benchmarks, we check our idea against a set of randomly generated Process Algebra Nets representing Combined Abstractions. This increases the likelihood of applicability in real systems. These examples were preselected so they do not contain trivial counterexamples[3]. We implement the model checkers for the reachability of PA-processes $\mathcal{M}^{\text{PA}}$ and a second one for the reachability of Process Algebra Nets $\mathcal{M}^{\text{PAN}}$ suggested in Section 5.4.3. The aim is to show, what improvement is gained while using the a posteriori approach for model checking.

[3] Elimination rules containg the initial process on the left-hand side are excluded.

number of transition rules per component	5000	7500	10000
average number of sequential interactions	609	914	1213
average number of fork transition rules	602	899	1211
average number of synchronization rules	628	922	1232
counterexample generated in < 1 min	49.0 %	50.0 %	51.4 %
counterexample generated in $1 - 30$ min	46.9 %	45.0 %	25.7 %
timeout (> 30 min)	4.1 %	5.0 %	22.9 %
no PA-processes counterexamples	14.3 %	7.5 %	8.6 %

Table 6.1.: Results of model checking.

6.3.1 The Experimental Setting

We use a standard PC with an AMD Athlon64 X2 4200+ and 2 GB RAM. The machine runs on Ubuntu Linux 8.04. We chose this system because it represents a performance which seems to be comparable to lower end workstations used by software developers in an industrial context.

The algorithms $\mathcal{M}^{\text{PA}}$ and $\mathcal{M}^{\text{PAN}}$ are implemented in Prolog using the SWI-Prolog framework [Wie03] in version 5.6.47. Although SWI-Prolog allows a multithreaded implementation, we implement a single threaded algorithm.

We create Process Algebra Nets with a much higher percentage of sequential, fork, synchronization and elimination rules than in an industrial case study. The models with 5000 rules have e. g., an average number of 609 sequential interactions and 602 fork rules (interpretable as start of a thread or asynchronous procedure call). This seems to be a high number and should ensure the expressiveness of the generic examples (cf. sections 6.2.1, 6.2.2, 6.2.3 and 6.2.4).

6.3.2 The Results

In Table 6.1 we show a classification of the verified models. It can be seen that a big share of models could be verified in less than a minute. This class contains many models, where counterexamples are computed by $\mathcal{M}^{\text{PA}}$, which use no replaced synchronization rules. Hence, the counterexamples generated from Π^{PA} need not to be checked using $\mathcal{M}^{\text{PAN}}$. Moreover, we can see that there is still a non-negligible number of models where $\mathcal{M}^{\text{PA}}$ could not generate counterexamples, which eliminates the $\mathcal{M}^{\text{PAN}}$-step.

For comparison we model check the generated models with two different justifications:

1. only $\mathcal{M}^{\text{PAN}}$, identifying the rate of timeouts without improvement
2. $\mathcal{M}^{\text{PA}} + \mathcal{M}^{\text{PAN}}$, identifying the rate of timeouts including the reduction of the input model

Process Algebra Net models which are not solvable by $\mathcal{M}^{\text{PA}} + \mathcal{M}^{\text{PAN}}$ in the fixed maximal run time of 30 minutes, are not solvable within the time limitation while using only $\mathcal{M}^{\text{PAN}}$ either. In Table 6.2 we show the improvement of the verification time. A comparison is difficult, because many Process Algebra Nets models cannot be solved by $\mathcal{M}^{\text{PAN}}$ within 30 minutes.

number of transition rules per component	5000	7500	10000
timeouts using $\mathcal{M}^{\text{PAN}}$	4.1 %	32.5 %	47.5 %
timeouts using $\mathcal{M}^{\text{PA}} + \mathcal{M}^{\text{PAN}}$	4.1 %	5.0 %	22.9 %

Table 6.2.: Timeouts while using $\mathcal{M}^{\text{PAN}}$ in comparison to $\mathcal{M}^{\text{PA}} + \mathcal{M}^{\text{PAN}}$.

So, as we see in the results the suggested improvement leads to a much better run time because of the reduction of the input model. The longest counterexample, we calculate in our generic models, contains 48 interactions until the protocol violation appears. We assume that such a long counterexample will happen only very rarely while verifying a real program. The reason is that the developer has had to define a very complex protocol for his component or the error has to be caused by a component arranged in a very high layer of the implemented architecture.

6.3.3 Summary

In this section the a posteri approach for model checking is applied on a generic case study. This is done to show the impact on this results while model checking Combined Abstractions with a large number of transition rules. Unfortunately, at this point in time no industrial case study is available containing as much sequential, fork and join transition rules (cf. Section 6.2) as needed to see an influence of the improvements presented in Section 5.4.3. This is a flaw of this case study. Nevertheless, the case study shows that the suggested improvements lead to a better model checking time in many cases.

6.4 Discussion

During the implementation of the architecture an important topic was to consider the practical applicability. For this reason we decided to respect several issues which we think are important for the usage in an industrial context.

- We obey the role and the knowledge of a component designer (top-level view of the architecture). To our experience a person performing this role has a top-level view of the architecture, only. Thus, it might be hard for him to decide, how the concrete implementation has to look like. Moreover, we think that the implementation details should be left to the component developers, because they might have more pieces of information about the technical and functional requirements than a high-level analyst. For this reason we implement a frontend allowing a comfortable and summarized view on the components and their abstract behavior (control flow only). The implementation details are left to the component developers, because they might have more pieces of information about the technical and functional requirements. Nevertheless, more concrete information can be represented within the frontend as we allow to extend the values of the operations `getComponentImplementationInformation`, `getNodeInfo`, `getActionInfo`, and `getProtocolInfo`.

- We support the component developer with our approach. In general, this role has low knowledge about behavior of complete application. In Service-oriented Architectures he should not have any information about the application context, the goal is here to establish a "market of components". A component has to be deployable in any context. Therefore, a component should provide a defined behavior. Using protocols this goal gets closer, because a formal definition of the usage is established. Moreover, the component developer has not to check and ensure every state of his component (which is expensive). Like by "design by contract" if the caller does not ensure the precondition (the protocol) the callee has not to provide any functionality. Thus, the developer is unburdened from many tasks. Furthermore, we use the knowledge of the component developer. As he knows the restrictions of "his" component at the best, he can define an adequate protocol. The implemented framework allows to file a component protocol at the component (`getProtocol` provides a local component's protocol).
- Third parties will not publish the component source code. Therefore, we define a representation of the behavior of a single component. Moreover, we provide a mechanism to generate the component abstraction locally and use it globally. For this purpose abstraction Web Services are implemented.
- In industrial environments one important requirement of a verification process is to work efficiently. On one hand this means, that the approach has an acceptable runtime, on the other hand an acceptable error ratio is needed. We have tackled these problems within this work with several developments. Furthermore, we have checked our developments in an industrial case study. However, the exclusion of counterexamples using the a priori approach (cf. Section 5.4.2) was not considered in a case study. Another important issue is the effort needed to get the process started. Again, the distinction between behavior and constraint is useful, as no specification has to be defined by a human, if the usage of a component is unproblematic. We assume, that a component protocol will be defined for the key components, which reduces the effort for the user in comparison with other approaches (cf. Chapter 2). The case study shows that even representations with many interactions are model checkable.
- Component abstraction and protocol can be published with the component interface description or service level agreement (SLA). This enables the integration of our approach into currently existing component systems or Service-oriented Architectures. The integration was shown as proof of concept while defining accordant Web Services in our case study.
- The infrastructure is oriented on the requirements of an industrial environment. The Web Services providing the functionality for computing component abstractions are separated and can be deployed distributively. This also takes the separated development in modern companies into account. The different development locations can be conflated under the roof of a WSO implementation.

In the case studies we have considered several programming languages and applications. We can see, that we have considered a comprehensive case study based on real programs. Nevertheless, there is the lack that it is no Service-oriented Architecture or a Component-based Architecture but an object-oriented architecture (exception the smallest case study in Section 6.2.4). However, we assume, that the results can be generalized, because it is a common consideration that classes are often smaller parts of aggregation than components.

As shown, optimizations lead to an applicable model checking time. Hence, our approach can be established in even a real software environment and is not only usable in academic milieus.

Moreover, we have recognized an educational effect at the developers defining protocols. As they have to think about the real behavior of the source code they think strongly about different execution sequences and possible errors. We assume that this can be lead back to simplicity of the constraints and their possible graphical representation, which are bound strongly at the interface of the components (in contrast to e. g., boolean formula). Thereby, the defined counterexamples format is very helpful as the user gets a specific hint, where a problem is suspected by the model checker.

We suppose that the integration of our model checking process into the development process, will lead to an improvement of the software quality. However, it seems to be difficult to quantify the influence on the software quality. Only a long-term case study could do this.

Nevertheless, we conclude from our observation that an integration of model checking techniques into the development process is possible.

7 Method for Using Protocol Conformance Checking in Iterative Component System Integration

In the previous chapters, we have presented and evaluated the main contribution of our work. It describes a new verification process for the verification of applications. In this scenario the abstractions of all needed components have to be accessible. The problem of this approach is the requirement that all components have to be implemented[1]. Thus, an error will be found at the end of the development process. This is no problem if we consider components which already exist (e. g., often the case in Service-oriented Architectures). In this chapter, we extend our verification approach, so it is possible to apply it on more scopes of applicability.

We consider the situations where components are implemented currently and composed to larger bricks. The complete application does not exist during this iterative development. We will support this approach by extending our verification approach for working with applications, which are implemented (or accessible) partly only. A statement is computed, which of the protocols of the aggregated component are always fulfilled. Moreover, it states what interaction sequences have to be excluded in the ongoing development, so that no protocol violation can occur. This approach was published in [BZ09e].

7.1 Motivation

In an industrial environment components are often developed concurrently by independent developers and bounded loosely. Attaching the verification at the end of the composition process might result in higher development costs, because problems are discovered late.

Often components are composed to larger parts, ensuring intended functionality, used in the further development, or sold for reuse as a part of unknown future applications. Thus, there is a big interest in components which can be composed to reliable systems.

From our point of view, providing (only) a verification process, that ensures the defined constraints, is not enough. This will ensure the properties of the application. However, using this verification process the problems are discovered later, which might cause higher costs for adaptions. Moreover, this approach is not suitable for verifying e. g., B2B scenarios or Service-oriented Architectures, where a complete application does not always exist or the components are bound dynamically.

Thus, the verification should be integrated into the development and composition process. In this section, we will show, how the available pieces of information about the behavior and the constraints of components can be used to support the development/composition of them. This will lead to a permanent verifiability of the current development state without having the full application implemented.

We allow to partly compose components and verify the created new aggregated component (also known as composite component). The verification of this aggregated component is also

[1] It is also possible to define the behavior by hand or generate it from another description language.

possible if no implementation of the required interfaces is available. For this purpose we generate verification drivers and verification dummies which are similar to the test driver and test dummy during test driven software validation. In contrast to tests, we consider every possible state of the software and we can prove the absence of the considered kind of errors (i. e., protocol violations). The verification results in indications what implementations might hurt the defined component protocols or that the component is reliable. This will lead to a bigger chance of discovering problems in an earlier phase, which will reduce the investments for changing the implementations or requirements and allow to apply the verification to more scopes of application.

In Example 7.1 a component-based software with three components is shown. Corresponding, in Example 7.2 on page 158 we see the abstraction Π_S of the system S composed by the abstractions of the components `C1`, `C2` and `C3`. The implemented components, their interfaces and protocols are shown in Example 3.2 on page 31. In this section the composition is made explicitly. It is possible using the *composition operator* $\oplus$, which resolves the required and matching provided interfaces of the integrated components. The result is the representation of an aggregated component.

Definition 7.1 (Composition operator $\oplus$)
The result of the composition operator $\oplus$ *is defined by the Stripped Process Rewrite System* $\Pi = \Pi_1 \oplus \Pi_2$*, where* $\Pi = (Q, \Sigma, \rightarrow, R, \mathbb{P}, \mathbb{M})$, $\Pi_1 = (Q_1, \Sigma_1, \rightarrow_1, R_1, \mathbb{P}_1, \mathbb{M}_1)$, $\Pi_2 = (Q_2, \Sigma_2, \rightarrow_2, R_2, \mathbb{P}_2, \mathbb{M}_2)$ *with* $Q = (Q_1 \cup Q_2) \setminus \{q_x : q_x \in \mathbb{P}_2\}$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\rightarrow = (\rightarrow_1 \cup \rightarrow_2) \setminus \{p \xrightarrow{x} p' \otimes p'' : p \xrightarrow{x} q_x \otimes p'' \in \rightarrow_1, \mathbb{M}(q_x) = p', \otimes \in \{., ||\}\}$, $R = (R_1 \setminus \{r_x : p_x \in \mathbb{P}_2\}) \cup R_2$, $\mathbb{P} = \mathbb{P}_1 \cup (\mathbb{P}_2 \setminus \{q_x : q_x \notin R_1\})$, $\mathbb{M} = \mathbb{M}_1 \cup (\mathbb{M}_2 \setminus \{\mathbb{M}(x) \mapsto p : x \in q_x \notin R_1\}$.
It is possible to bind the required to the own provided interfaces: $\Pi = \Pi_{C_1} \oplus \Pi_{C_1}$.

7.2 Iterative Verification

During the development process several scenarios exist concerning which components are already developed. The other components may not be available or in a state that cannot be used for verification. Figure 7.1 on page 159 shows the architecture of some typically aggregated components C_A. We assume that there is only one internal component C that provides the provided interfaces of the aggregated component C_A. We distinguish the following scenarios:

$a)$ C_A has provided interfaces, but has no required interfaces (Figure 7.1a on page 159).

$b)$ C_A has provided interfaces, required interfaces exist which are still unbounded, callbacks are not allowed (Figure 7.1b).

$c)$ C_A has provided interfaces, required interfaces exist which are still unbounded, and arbitrary callbacks (including callbacks) are allowed (Figure 7.1c).

Scenarios $a)$ and $b)$ are important for the hierarchical composition of components. However, a component-based software is not always built up hierarchically. It is also possible that the com-

Example 7.1: Application assembled from three component (this is a sub-scenario of the component-based software in Example 3.2 on page 31).

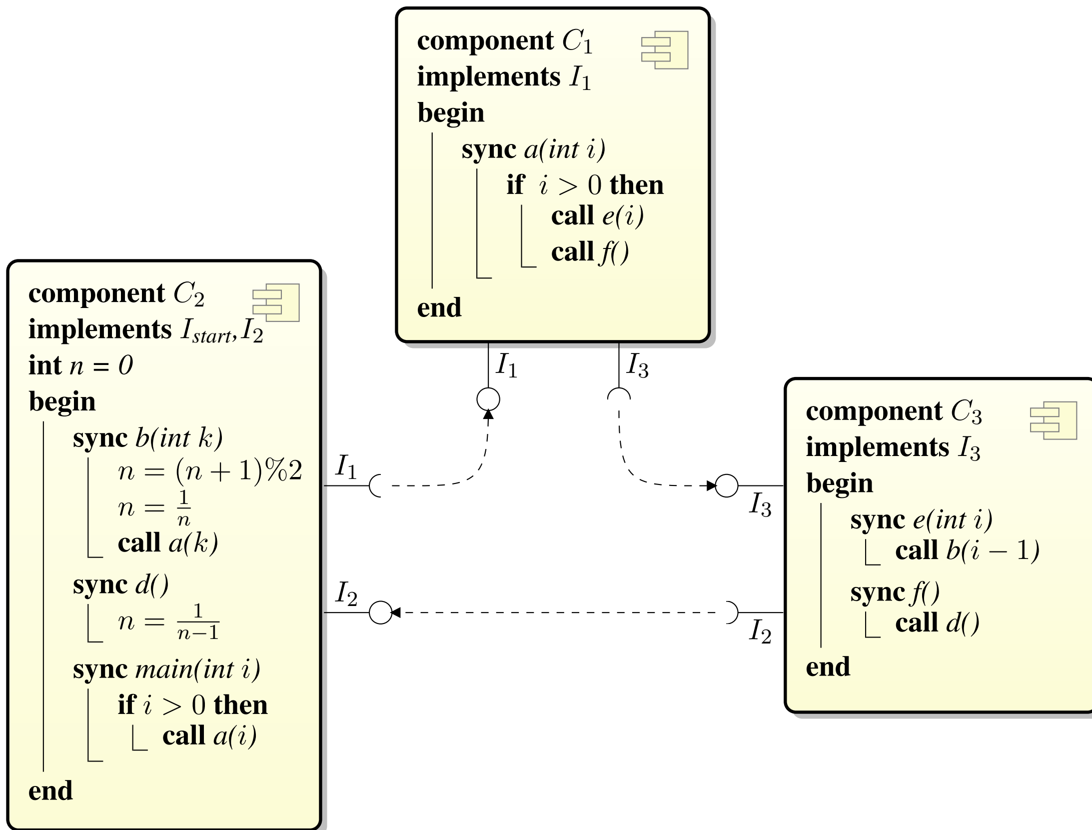
interface I_{start}
begin
  sync *main(int i)*
end

interface I_1
begin
  sync *a(int i)*
end

interface I_2
begin
  sync *b(int k)*
  sync *d()*
end

interface I_3
begin
  sync *e(int i)*
  sync *f()*
end

(a) Interfaces of components.



(b) Components and implementations.

$$\mathcal{P}_{\mathtt{C2}} = (bdd^*)^* \qquad \mathcal{P}_{\mathtt{C1}} = a^* \qquad \mathcal{P}_{\mathtt{C3}} = ef(ef)^*$$

(c) Component protocols (as regular expressions).

Example 7.2: Abstractions of the components in Example 7.1 on page 157 (we abbreviated the required interfaces and provided interfaces, e. g., q_e instead of $q_{I_3,e}$).

$$\begin{aligned}
Q_{\mathtt{C1}} &= \{p_{10}, p_{11}, p_{12}\},\\
\Sigma_{\mathtt{C1}} &= \{e, f\},\\
\rightarrow_{\mathtt{C1}} &= \{p_{10} \xrightarrow{\lambda} \varepsilon, p_{10} \xrightarrow{e} q_e.p_{11}, p_{11} \xrightarrow{f} q_f.p_{12}, p_{12} \xrightarrow{\lambda} \varepsilon\},\\
R_{\mathtt{C1}} &= \{q_e, q_f\},\\
\mathbb{P}_{\mathtt{C1}} &= \{q_a\},\\
\mathbb{M}_{\mathtt{C1}} &= \{q_a \mapsto p_{10}\}
\end{aligned}$$

(a) Abstraction $\Pi_{\mathtt{C1}} = (Q_{\mathtt{C1}}, \Sigma_{\mathtt{C1}}, \rightarrow_{\mathtt{C1}}, R_{\mathtt{C1}}, \mathbb{P}_{\mathtt{C1}}, \mathbb{M}_{\mathtt{C1}})$ of component `C1`.

$$\begin{aligned}
Q_{\mathtt{C2}} &= \{p_0, p_1, p_2, p_3, p_4\},\\
\Sigma_{\mathtt{C2}} &= \{a\},\\
\rightarrow_{\mathtt{C2}} &= \{p_0 \xrightarrow{a} q_a.p_1, p_1 \xrightarrow{\lambda} \varepsilon, p_2 \xrightarrow{\lambda} \varepsilon, p_3 \xrightarrow{a} q_a.p_4, p_4 \xrightarrow{\lambda} \varepsilon\},\\
R_{\mathtt{C2}} &= \{q_a\},\\
\mathbb{P}_{\mathtt{C2}} &= \{q_b, q_d\},\\
\mathbb{M}_{\mathtt{C2}} &= \{q_b \mapsto p_3, q_d \mapsto p_2\}
\end{aligned}$$

(b) Abstraction $\Pi_{\mathtt{C2}} = (Q_{\mathtt{C2}}, \Sigma_{\mathtt{C2}}, \rightarrow_{\mathtt{C2}}, R_{\mathtt{C2}}, \mathbb{P}_{\mathtt{C2}}, \mathbb{M}_{\mathtt{C2}})$ of component `C2`.

$$\begin{aligned}
Q_{\mathtt{C3}} &= \{p_{20}, p_{21}, p_{22}, p_{23}\},\\
\Sigma_{\mathtt{C3}} &= \{b, d\},\\
\rightarrow_{\mathtt{C3}} &= \{p_{20} \xrightarrow{b} q_b.p_{21}, p_{21} \xrightarrow{\lambda} \varepsilon, p_{22} \xrightarrow{d} q_d.p_{23}, p_{23} \xrightarrow{\lambda} \varepsilon\},\\
R_{\mathtt{C3}} &= \{q_b, q_d\},\\
\mathbb{P}_{\mathtt{C3}} &= \{q_e, q_f\},\\
\mathbb{M}_{\mathtt{C3}} &= \{q_e \mapsto p_{20}, q_f \mapsto p_{22}\}
\end{aligned}$$

(c) Abstraction $\Pi_{\mathtt{C3}} = (Q_{\mathtt{C3}}, \Sigma_{\mathtt{C3}}, \rightarrow_{\mathtt{C3}}, R_{\mathtt{C3}}, \mathbb{P}_{\mathtt{C3}}, \mathbb{M}_{\mathtt{C3}})$ of component `C3`.

ponents are composed in a flat manner, as it is often done in Service-oriented Architectures. This means that every component may have required and provided interfaces. In this case protocol violations can be caused by (recursive) callbacks, too. These kinds of protocol violations are not easy to discover (above all by humans) and a local protocol check is insufficient [ZS06]. Thus scenario c) matters especially if callbacks can happen.

In Figure 7.1d a more complicated example is shown. This scenario is also captured here. I. e., except the fact that the provided interface of an aggregated component is provided by a single internal component, there are no further restrictions on the captured architecture.

The example demonstrates that not every component required for composition is already implemented. For verification purpose, we need to know restrictions on the use of the aggregated component in a component-based software. We call these restrictions *context*. They play a similar role as the test driver and the test dummy for local tests.

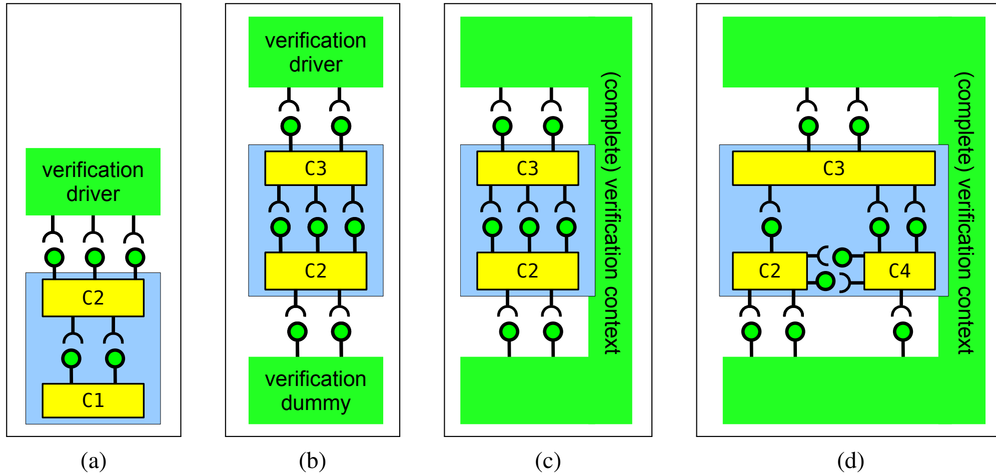


Figure 7.1.: Possible scenarios during development.

As we can see in Figure 7.1a (component C_1) there may be components without required interfaces. We call these components *base components* (cf. Definition 3.6). Usually bottom-up constructions of component-based software systems start with base components.

We assume that the integrated components C_i ($i \in \mathbb{N}$) are composed to a new component, which we call *aggregated component* C_A. The goal is to compose components C_i to a larger component C_A, which can be handled easier. Thus, required and provided interfaces of the components C_i are bounded closely. The aggregated component can also have provided and required interfaces.

Definition 7.2 (Architecture of the aggregated component)
Here, an aggregated component C_A consists of <u>*at least two*</u> *components, where the provided interfaces of components are bounded to the required interfaces of other components (cf. Figure 7.1). We restrict the model here with the assumption, that only the provided interfaces of* <u>*exactly one*</u> *component C are unbound and thus used as provided interfaces of the aggregated component C_A. Hence, the protocol of C_A is equal to the protocol $\mathcal{P}_A$ of C.*

Definition 7.3 (Top component)
A top component C is contained in an aggregated component C_A, where the set of provided interfaces of C_A is equal to the set of provided interfaces of C. Thus, C implements the well known design pattern "facade" [GHJV95].

A context represents every possible implementation of a component which can be bounded to interfaces of an aggregated component C_A.

Definition 7.4 (Context)
We distinguish three kinds of contexts:

- *A* verification driver *C_A^Ω of C_A has no provided interfaces, but for every* <u>*provided*</u> *interface q_i of the aggregated component C_A exists a matching required interface q_i in C_A^Ω.*

- *A* verification dummy $C_A^{\mho}$ *of* C_A *has no required interfaces, but for every* required *interface* q_i *of the aggregated component* C_A *a matching provided interface* q_i *in* C_A^{Ω} *exists. This context accepts all calls of the aggregated component* C_A *to its required interfaces.*

- *A (complete)* verification context $C_A^{\circlearrowleft}$ *of* C_A *has the same required interfaces* q_i *as the verification driver* C_A^{Ω} *and the same provided interfaces* q_i *as the verification dummy* $C_A^{\mho}$*.*

Figure 7.1 shows all different kinds of contexts.

An aggregated component C_A may have provided interfaces but no required interfaces, cf. Figure 7.1a. In this case the protocols of all components of the aggregated component should be obeyed if it has been proven that the aggregated component is reliable and C_A is bounded to other components.

Informally, to ensure that no protocol is violated, we create a context C_A^{Ω} of the aggregated component C_A called *verification driver* (Section 7.5.1). It initiates all possible and legal calls to C_A. The model checker calculates the statement if the component C_A is ready for the integration with other components (Section 7.4).

For the same reasons, we define the *verification dummy* $C_A^{\mho}$. It has at its provided interfaces the required interfaces R_{C_A} of C_A, and each initial state of a call x is terminated immediately (because no callbacks are initiated).

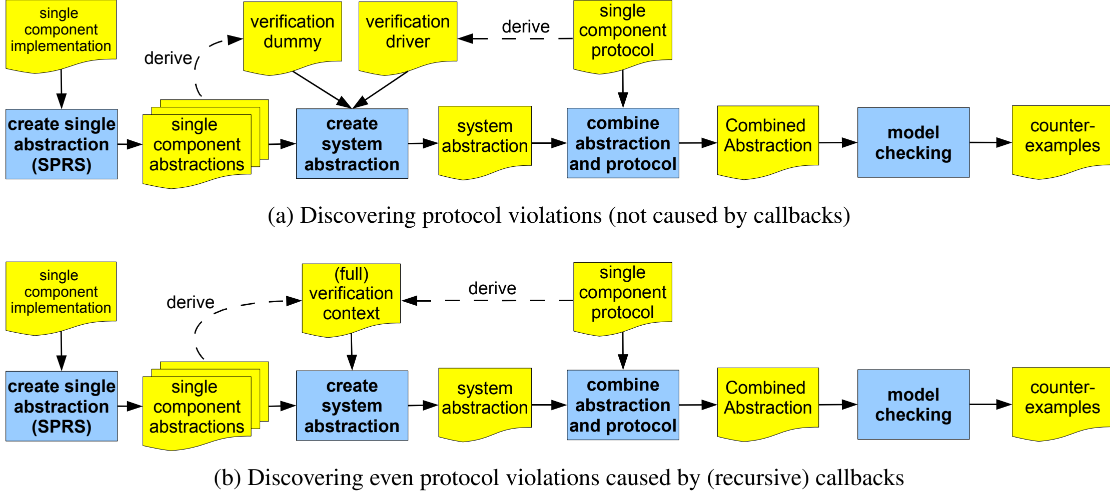
If no callbacks are allowed, it is sufficient to have a verification driver as well as a verification dummy. However, if callbacks are present, a (direct or indirect) call to a required interface of the aggregated component C_A may initiate a call to a provided interface of C_A. Thus, a (complete) verification context $C_A^{\circlearrowleft}$ (Section 7.5.2) is required, cf. Figure 7.1c and 7.1d.

The contexts are used as a substitute for the unknown components. Hence, the model checking approach described in Chapter 5 can be applied.

7.3 Verification Process for Iterative Development

As shown in Chapter 5 we need the abstractions of every single component which is needed for the application. Only these abstractions will ensure that the behavior of the defined component protocols are obeyed. If the behavior of a component is not present, it is currently not possible to decide whether the protocol of the considered component is obeyed.

The new verification process is shown in Figure 7.2. We divide the verification into two parts. The first considers (hierarchical) programs without callbacks, while the second considers also callbacks. The users are enabled to consider only the problems, they are interested in. Moreover, we assume that this will improve the model checking speed [Prä09], as the system abstraction contains less rules than in a non-iterative model checking approach.



(a) Discovering protocol violations (not caused by callbacks)

(b) Discovering even protocol violations caused by (recursive) callbacks

Figure 7.2.: Verification processes for iterative development.

7.4 Evaluating the Result of the Model Checker

From the model checker we will get the result if a component using the provided interfaces of C and obeying the protocol $\mathcal{P}_A$ can trigger errors (i. e., protocol violations) in the other components C_i of the aggregated component. If a model checking results in a counterexample, we can describe directly, in which constellation the counterexample appears. I. e., how the protocol $\mathcal{P}_i$ is violated.

If a counterexample violating protocol $\mathcal{P}_i$ was found (which is no false negative), then we can react with the following options:

1. It can be checked locally, whether the considered component protocol $\mathcal{P}_i$ should be weakened. This is only possible if no fault in the program will result by this adaption.

2. The other components C_j (where $i \neq j$) participating at the counterexample in order to decide whether an adaption of their source code should be recommended.

3. The error can be consciously neglected while assuming that such an error will not occur in a real implementation, where C_i is using the provided interfaces of C_A.

The first and second option will result in a situation where no protocol violation can appear, if any user of the aggregated component C_A obeys the protocol $\mathcal{P}_A$. The advantage of this extension is that unviolated protocols $\mathcal{P}_j$ of C_A need to be considered (again), because all calls to the component C are guarded by the protocol $\mathcal{P}_A$. In Theorem 7.1 we will prove that this is sufficient for avoiding protocol violations of internal components.

As described, this is only possible if no counterexample is constructed (or the first or second option is implemented). In the third case all protocols $\mathcal{P}_i$ of the components C_i, which are violated while using the verification context, have to be considered.

The abstraction of the aggregated component can now be reused in the further development or composition process.

Remark

Summarized, it is possible that a component protocol $\mathcal{P}_A$ can provable never be violated with respect to the current composition step. In this case $\mathcal{P}_A$ can be removed from the current scenario. Moreover, all transition rules of the form $t_1 \stackrel{a}{\to} t_2$ can be translated into $t_1 \stackrel{\lambda}{\to} t_2$, where $a \in \Sigma_{\mathcal{P}_i}$. This enables more static reductions of the number of transition rules, resulting in a faster model checking (cf. Chapter 6).

We formalize and prove these assumptions and the verification process in the following sections.

7.5 Verification Contexts (No Callbacks Allowed)

We use the component term to describe the behavior of a context. If the context is (really) implemented as component C, we would generate a new abstraction Π_C to use this in our verification process (cf. Chapter 5). Thus, C is translated to a Stripped Process Rewrite System Π_C. To represent the behavior of a context C' we also use a Stripped Process Rewrite System $\Pi_{C'}$. For technical reasons (which become clear later) we assume that each call to C_A has at least one terminating computation (this only excludes the possibility that a call never terminates).

7.5.1 Components with no Unbounded Required Interfaces

In this section we consider the scenario $a)$ (cf. Section 7.2) of Figure 7.1a (considering base components).

The verification driver C_A^Ω of a component C_A has no provided interfaces. The set of the required interfaces is equal to the set of provided interfaces of the component C_A (cf. Definition 7.4).

Definition 7.5 (Verification driver C_A^Ω)
A verification driver for C_A is a Stripped Process Rewrite System $C_A^\Omega \hat{=} (Q_{C_A^\Omega}, \Sigma_{C_A^\Omega}, \to_{C_A^\Omega}, R_{C_A^\Omega}, \mathbb{P}_{C_A^\Omega}, \mathbb{M}_{C_A^\Omega})$, where:

$$
\begin{aligned}
Q_{C_A^\Omega} =& \{p_{v_i} : v_i \in Q_{\mathcal{P}_A}\},\\
\Sigma_{C_A^\Omega} =& \Sigma_{\mathcal{P}_A},\\
R_{C_A^\Omega} =& \{q_i : q_i \in \mathbb{P}_{C_A}\},\\
\mathbb{P}_{C_A^\Omega} =& \{\},\\
\mathbb{M}_{C_A^\Omega} =& \{\},\\
\to_{C_A^\Omega} =& \{p_{v_i} \stackrel{x}{\to} q_x \otimes p_{v_j} : v_i \stackrel{x}{\to} v_j \in \to_{\mathcal{P}_A} \wedge p_{q_x} \in Q_{\Pi_{C_A}} \wedge q_x \in \mathbb{P}_{\Pi_{C_A}}\}
\end{aligned}
$$

$$
\cup \{p_{v_k} \stackrel{\lambda}{\to} \varepsilon : p_{v_k} \in F_{\mathcal{P}_A}\},
$$

$$
\textit{with } \otimes = \begin{cases} . & \textit{if } x \textit{ is implemented by } C_A \textit{ as synchronous interface,}\\ || & \textit{if } x \textit{ is implemented by } C_A \textit{ as asynchronous interface.}\end{cases}
$$

Example 7.3: Construction of the verification driver.

$$\mathcal{P}_{\texttt{C3}} = (\{v_1, v_2, v_3\}, \quad \{e, f\}, \quad \{v_1 \xrightarrow{e} v_2, v_2 \xrightarrow{f} v_3, v_3 \xrightarrow{\lambda} v_3\}, \quad v_0, \quad \{v_3\} \quad)$$

(a) Given protocol $\mathcal{P}_{\texttt{C3}}$

$$\Pi^{\Omega}_{\texttt{C3}} = (\{p_{v_1}, p_{v_2}, p_{v_3}\}, \quad \{e, f\}, \quad \{p_{v_1} \xrightarrow{e} q_e.p_{v_2}, p_{v_2} \xrightarrow{f} q_f.p_{v_3}, p_{v_3} \xrightarrow{\lambda} p_{v_1}, p_{v_3} \xrightarrow{\lambda} \varepsilon\}, \\ \{q_e, q_f\}, \{\}, \{\})$$

(b) Generated C^{Ω}_{A} of component C3

By this definition, for each transition rule $(v_i \xrightarrow{x} v_j) \in \rightarrow_{\mathcal{P}_A}$ of the protocol $\mathcal{P}_A$ of the aggregated component C_A a new transition rule $p_{v_i} \xrightarrow{x} q_x \otimes p_{v_j}$ of C^{Ω}_{A} is created. An example for this construction is shown in Example 7.3.

Lemma 7.1
Let be $\Pi^A_S = \varphi_{\mathcal{P}_A}(\Pi_S)$, where $\Pi_S = C^{\Omega}_{A} \oplus \Pi_{C_A}$. Then $L(\Pi^A_S) = L(\mathcal{P}_A)$, where $\mathcal{P}_A$ is the protocol of C_A.

Proof (Sketch)
By the architectural assumption in this section, C_A cannot (neither directly nor indirectly) call to its provided interfaces, because C_A has no required interfaces. Thus, for any provided interface $q_x \in \mathbb{P}_{C_A}$ of C_A, it holds $p_{q_x} \overset{\lambda}{\Rightarrow}_{\Pi^A_S} \varepsilon$ where p_{q_x} is the initial state of interface x (otherwise a call to x would be always non-terminating). We prove now the following claim by induction sufficient for $L(\Pi^A_S) = L(\mathcal{P}_A)$:

For each $w \in \Sigma^*_{\mathcal{P}_A}$ it holds $v_0 \overset{w}{\Rightarrow}_{\mathcal{P}_A} v_k$ iff $p_{v_0} \overset{w}{\Rightarrow}_{\Pi^A_S} p_{v_k}$.
if: We only consider the case where the interface is implemented synchronously. The asynchronous case can be proven analogously.
case $w = \lambda$: Then $v_0 \overset{\lambda}{\Rightarrow}_{\mathcal{P}_A} v_1$ and $p_{v_0} \overset{\lambda}{\Rightarrow} p_{v_1}$ proves the claim.
case $w = w'a$ for a $w' \in \Sigma^*_{\mathcal{P}_A}, a \in \Sigma_{\mathcal{P}_A}$: Then $v_0 \overset{w'}{\Rightarrow}_{\mathcal{P}_A} v_{k-1} \overset{a}{\Rightarrow}_{\mathcal{P}_A} v_k$. Thus, $v_{k-1} \xrightarrow{a} v_k \in \rightarrow_{\mathcal{P}_A}$ and therefore $p_{v_{k-1}} \overset{a}{\Rightarrow}_{\mathcal{P}_A} q_a.p_{v_k} \in \rightarrow_{C^{\Omega}_{A}}$ (by Definition 7.5).

$$\text{Thus, we have } p_{v_{k-1}} \overset{a}{\Rightarrow}_{\Pi^A_S} p_{q_a}.p_{v_k} \overset{\lambda}{\Rightarrow}_{\Pi^A_S} \varepsilon.p_k \quad \text{(by the above observation)}$$
$$= p_k \quad \text{(since } \varepsilon \text{ is the identify of .)} \quad \square$$

Now we show that a derivation in $\Pi^{C_i}_S \hat{=} \varphi_{\mathcal{P}_{C_i}}(\Pi_{S\text{Im}})$ (where $\Pi_{S\text{Im}} \hat{=} \Pi_{C_{\text{Im}}} \oplus \Pi_{C_A}$) corresponds to a derivation of $C^{\Omega}_{A} \oplus \Pi_{C_A}$. In particular, our goal is to show that $L(\varphi_{C_i}(\Pi_{C_{\text{Im}}} \oplus \Pi_{C_A})) \subseteq L(\varphi_{C_i}(C^{\Omega}_{A} \oplus \Pi_{C_A}))$ for all internal components C_i of C_A, if the protocol of C_A has been checked. Such a result implies that replacing C^{Ω}_{A} by C_{Im} does not lead to new protocol violations except possibly for C_A. In order to show a correspondence between derivation $\Pi_{S\text{Im}}$ and Π^A_S, we need a correspondence between process-algebraic expressions $t \in PEX(Q_{\Pi^A_S})$ and $t' \in PEX(Q_{\Pi^A_S})$.

Note, that $Q_{C_A} \subseteq Q_{\Pi_{S\text{Im}}} \cap Q_{\Pi_S^A}$ because these states stem from Π_{C_A}. The main idea for defining the correspondence basically requires that sequences initiated by $\Pi_{C_{\text{Im}}}$ are sequences initiated by $\Pi_{C_A^\Omega}$. Intuitively, a t corresponds to t' iff forgetting the states not in Q_{C_A} leads to the same process-algebraic expression, and the same sequence of interactions to C_A can be initiated from t and t'.

Lemma 7.2

If $L(\varphi_{\mathcal{P}_A}(\Pi_{S\mathit{Im}})) \subseteq L(\mathcal{P}_A)$, and there is a $w \in \Sigma_{C_A}^*$ and $t \in PEX(Q_{\Pi_{S\mathit{Im}}})$ such that $I \overset{w}{\Rightarrow}_{\Pi_{\Pi_{S\mathit{Im}}}} t$, then there is a $t' \in PEX(Q_{\Pi_S^A})$ with t corresponds to t' and $p_{v_0} \overset{w}{\Rightarrow} t'$.

Proof (Sketch)

We use induction to prove the claim. If a word w with $|w| = 1$ is created, a transition rule in C_{Im} has to be used which has the form $p_1 \overset{x}{\rightarrow} p_{q_x} \otimes p_2$, $x \in L(\mathcal{P}_A)$, $\otimes \in \{., ||\}$. Because C_{Im} obeys the protocol $\mathcal{P}_A$, there exists a transition rule $p_{v_0} \overset{x}{\rightarrow} p_{q_x} \otimes p_{v_1}$ in Π_S^A (more precise in C_A^Ω). Thus, the term t' is constructed using this rule. t corresponds to t'.

Assuming that the claim is true for words of length n. A word $w' = w \cdot a$ with $|w'| = n+1$ can only be constructed using a rule $\delta \hat{=} p_3 \overset{a}{\rightarrow} p_{q_a} \otimes p_4$ in C_{Im}. A similar rule δ' has to be part of C_A^Ω, such that the same word $w \cdot a$ can be constructed. While constructing w in C_{Im} a term t has to be computed, encoding a situation where δ is applicable. Because t' encodes the same situation, t corresponds to t'. $\square$

Corollary 7.1

$L(\varphi_{C_A}(\Pi_{S\mathit{Im}})) \subseteq L(\Pi_S^A)$.

Hence, a verification driver C_A^Ω can generate all sequences that are permitted by the protocol $\mathcal{P}_A$ of C_A.

Now, it is possible to model check the scenario $a)$ described in Section 7.2.

We generate a system abstraction Π_S while unifying the behavior of the aggregated component C_A and the verification dummy $C_A^\mho$ as described in Chapter 5 (an example is shown in Example 7.3b). The resulting Π_S is used to generate a Combined Abstraction for each protocol $\mathcal{P}_i$ of each component C_i, where C_i is included in the aggregated component C_A.

Now, we have to prove that no protocol violation is missed, while considering the verification driver C_A^Ω.

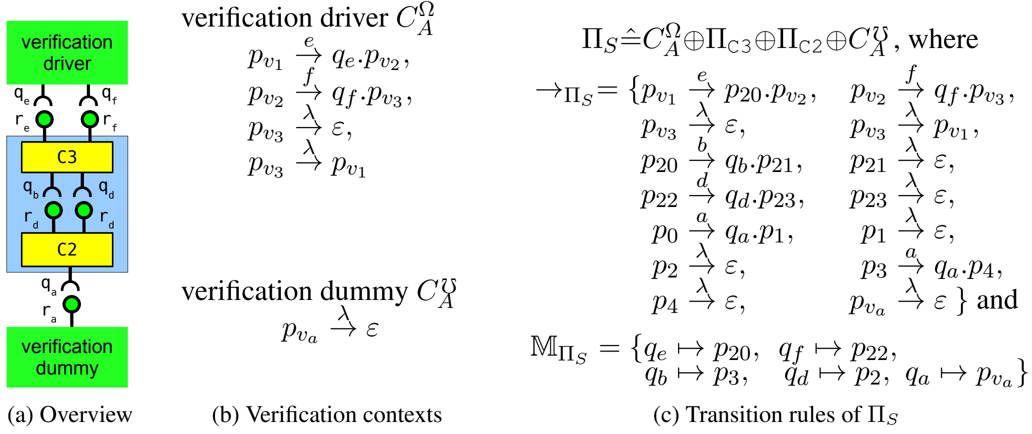
Theorem 7.1

If $L(\varphi_{\mathcal{P}_{C_i}}(\Pi_S^A)) \subseteq L(\mathcal{P}_{C_i})$ for each protocol $\mathcal{P}_{C_i}$ of a component C_i in C_A (i. e., we have protocol conformance proven using the verification driver), and $L(\varphi_{\mathcal{P}_A}(\Pi_{S\mathit{Im}})) \subseteq L(\mathcal{P}_A)$ (i. e., the component C_{Im} uses C_A according to the protocol $\mathcal{P}_A$) then for each component C_i of C_A, it holds $L(\varphi_{\mathcal{P}_{C_i}}(\Pi_{S\mathit{Im}})) \subseteq L(\mathcal{P}_{C_i})$ (i. e., the protocol is satisfied if C_{Im} is used instead of the verification driver C_A^Ω).

Remark

It is clear that if $L(\Pi_1) \subseteq L(\Pi_2)$, then $L(\varphi_{C_i}(\Pi_1)) \subseteq L(\varphi_{C_i}(\Pi_2))$.

Example 7.4: Verification context, we use C2 and C3 (and $\mathcal{P}_{\texttt{C2}}$) from Example 3.2 on page 31 and 7.2 on page 158.



(a) Overview (b) Verification contexts (c) Transition rules of Π_S

Proof (Theorem 7.1)

$$
\begin{aligned}
L(\varphi_{\mathcal{P}_{C_i}}(\Pi_{S\,Im})) &\subseteq L(\varphi_{\mathcal{P}_{C_i}}(\Pi_S^A)) && \textit{by Corollary 7.1 and the remark on page 164} \\
&\subseteq L(\mathcal{P}_A) && \textit{by assumptions of Theorem 7.1} \quad \square
\end{aligned}
$$

Hence, all possible protocol violations by any implementation (without callbacks) can be found.

7.5.2 Components with Unbounded Provided and Unbounded Required Interfaces

In this section we consider the scenario b) (cf. Section 7.2) of Figure 7.1b on page 159 (allowing no callbacks).

For model checking a complete Process Rewrite System is needed. Thus, a verification driver C_A^{Ω} and the abstraction of the aggregated component Π_{C_A} are not sufficient, because there are still unbounded required interfaces of C_A. We therefore add a Stripped Process Rewrite System $C_A^{\mho}$, called verification dummy. It is bounded to the required interfaces of $C_A^{\mho}$.

The verification dummy $C_A^{\mho}$ has to accept all calls from the aggregated component C_A to its unbounded required interfaces $q_i \in R_{C_A}$, but imitates no callbacks to a $q_j \in \mathcal{P}_{C_A}$. Thus, $C_A^{\mho}$ imitates all possible base components, which could be used to complete the application integrating C_A. Here we define the verification dummy formally as Stripped Process Rewrite System $\Pi_{C_A^{\mho}}$.

Definition 7.6 (Verification dummy $C_A^{\mho}$)
A verification dummy is a Stripped Process Rewrite System $\Pi_{C_A^{\mho}} \hat{=} (Q_{C_A^{\mho}}, \Sigma_{C_A^{\mho}}, \rightarrow_{C_A^{\mho}}, R_{C_A^{\mho}}, \mathbb{P}_{C_A^{\mho}}, \mathbb{M}_{C_A^{\mho}})$*, where:*

$$
\begin{aligned}
Q_{C_A^{\mho}} &= \{p_{q_i} : q_i \in \mathbb{P}_{C_A}\}, \\
\Sigma_{C_A^{\mho}} &= \{\}, \\
\rightarrow_{C_A^{\mho}} &= \{p_{q_i} \xrightarrow{\lambda} \varepsilon : q_i \in \mathbb{P}_{C_A^{\mho}}\}, \\
R_{C_A^{\mho}} &= \{\}, \\
\mathbb{P}_{C_A^{\mho}} &= \{q_i : q_i \in R_{C_A}\}, \\
\mathbb{M}_{C_A^{\mho}} &= \{q_i \mapsto p_{q_i} : q_i \in R_{C_A}\}.
\end{aligned}
$$

Thus, the verification dummy $C_A^{\mho}$ of C_A can be bounded at C_A and capture all behavior which is possible by a component C_2 implementing the context. Hence, $C_A^{\Omega} \oplus C_A \oplus C_A^{\mho}$ is a Process Rewrite System.

Theorem 7.2
An implementation C_{Im} *of the verification dummy* $C_A^{\mho}$ *cannot initiate a protocol violation.*

Proof (Theorem 7.2)
Callbacks are forbidden. Thus, no call to an interface of C_{Im} *can result in a call to* C_A*. No protocol violation can appear.* □

Corollary 7.2
Let be $\Pi_{SIm} \hat{=} C_1 \oplus \Pi_{C_A} \oplus C_2$*, where* C_1 *is an implementation of* C_A^{Ω} *and* C_2 *is an implementation of* $C_A^{\mho}$*. Violations of* $\mathcal{P}_{C_i}$ *of* C_i *can only be initiated by* C_1 *(captured by Theorem 7.1).*

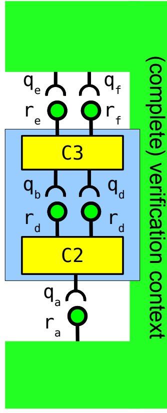
An example is shown in Example 7.4. As we can see no protocol violation happens while verifying the protocols $\mathcal{P}_{\mathrm{C2}}$, because all computable interaction sequences w are contained in $L(\mathcal{P}_{\mathrm{C2}})$.

7.6 Verification Context (Allowing Callbacks)

For humans it is usually difficult to discover protocol violations if callbacks are present, in particular if these are recursive. Thus, it is an important task to find or exclude protocol violations in this case. For this reason we generate a general context, which contains all possible callbacks, too. As before we use a Stripped Process Rewrite System $\Pi_{C_A^{\circlearrowleft}}$ representation to describe the behavior of the verification context.

Definition 7.7 (Verification context $\Pi_{C_A^{\circlearrowleft}}$)
A verification context is a Stripped Process Rewrite System $\Pi_{C_A^{\circlearrowleft}} \hat{=} (Q_{C_A^{\circlearrowleft}}, \Sigma_{C_A^{\circlearrowleft}}, \rightarrow_{C_A^{\circlearrowleft}}, R_{C_A^{\circlearrowleft}}, \mathbb{P}_{C_A^{\circlearrowleft}},$

Example 7.5: $\Pi_S \hat{=} C_A^{\circlearrowleft} \oplus \Pi_{\mathtt{C2}} \oplus \Pi_{\mathtt{C3}} \oplus C_A^{\circlearrowleft}$ (cf. Example 7.2 on page 158) with callbacks.



(a) Overview

$\rightarrow$ of C_A^{Ω}

$p_{v_1} \xrightarrow{e} q_e{\cdot}p_{v_2}$,

$p_{v_2} \xrightarrow{f} q_f{\cdot}p_{v_3}$,

$p_{v_3} \xrightarrow{\lambda} \varepsilon$,

$p_{v_3} \xrightarrow{\lambda} p_{v_1}$

$\rightarrow_{\circlearrowleft}$ (callbacks)

$p_{v_a} \xrightarrow{\lambda} p_{v_1}$

$p_{v_a} \xrightarrow{\lambda} p_{v_2}$

$p_{v_a} \xrightarrow{\lambda} p_{v_3}$

$\rightarrow$ of $C_A^{\mho}$

$p_{v_a} \xrightarrow{\lambda} \varepsilon$

(b) $\rightarrow$ of $C_A^{\circlearrowleft}$

$$
\begin{aligned}
&\underline{p_0} \overset{a}{\Rightarrow} \underline{p_{v_a}}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{v_1}}{\cdot}p_1 \overset{e}{\Rightarrow} \underline{p_{20}}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{b}{\Rightarrow} \underline{p_3}{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \overset{a}{\Rightarrow} \underline{p_{v_a}}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{\lambda}{\Rightarrow} \underline{p_{v_1}}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \overset{e}{\Rightarrow} \underline{p_{20}}{\cdot}p_{v_2}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{b}{\Rightarrow} \underline{p_3}{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{a}{\Rightarrow} \underline{p_{v_a}}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{\lambda}{\Rightarrow} \underline{p_4}{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{\lambda}{\Rightarrow} \underline{p_{21}}{\cdot}p_{v_2}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{v_2}}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{f}{\Rightarrow} \underline{p_{22}}{\cdot}p_{v_3}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{d}{\Rightarrow} \underline{p_2}{\cdot}p_{23}{\cdot}p_{v_3}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{\lambda}{\Rightarrow} \underline{p_{23}}{\cdot}p_{v_3}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{v_3}}{\cdot}p_4{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \\
&\overset{\lambda}{\Rightarrow} \underline{p_4}{\cdot}p_{21}{\cdot}p_{v_2}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{21}}{\cdot}p_{v_2}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{v_2}}{\cdot}p_1 \\
&\overset{f}{\Rightarrow} \underline{p_{22}}{\cdot}p_{v_3}{\cdot}p_1 \overset{d}{\Rightarrow} \underline{p_2}{\cdot}p_{v_3}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_{v_3}}{\cdot}p_1 \overset{\lambda}{\Rightarrow} \underline{p_1} \overset{\lambda}{\Rightarrow} \varepsilon
\end{aligned}
$$

(c) Computed counterexample

$\mathbb{M}_{C_A^{\circlearrowleft}}$), *where*

$$
\begin{aligned}
Q_{C_A^{\circlearrowleft}} &= Q_{C_A^{\Omega}} \cup Q_{C_A^{\mho}}, \\
\Sigma_{C_A^{\circlearrowleft}} &= \Sigma_{C_A^{\Omega}} \cup \Sigma_{C_A^{\mho}} = \Sigma_{C_A^{\Omega}}, \\
\rightarrow_{C_A^{\circlearrowleft}} &= \rightarrow_{C_A^{\Omega}} \cup \rightarrow_{C_A^{\mho}} \cup \rightarrow_{\circlearrowleft}, \\
R_{C_A^{\circlearrowleft}} &= R_{C_A^{\Omega}} \cup R_{C_A^{\mho}} = R_{C_A^{\Omega}}, \\
\mathbb{P}_{C_A^{\circlearrowleft}} &= \mathbb{P}_{C_A^{\Omega}} \cup \mathbb{P}_{C_A^{\mho}} = \mathbb{P}_{C_A^{\mho}}, \\
\mathbb{M}_{C_A^{\circlearrowleft}} &= \mathbb{M}_{C_A^{\Omega}} \cup \mathbb{M}_{C_A^{\mho}} = \mathbb{M}_{C_A^{\mho}}
\end{aligned}
$$

To capture the semantics of callbacks we introduce the special set of transition rules $\rightarrow_{\circlearrowleft} = \{p_{q_i} \xrightarrow{\lambda} p_{v_i} : p_{q_i} \in Q_{C_A^{\mho}} \wedge p_{v_i} \in Q_{C_A^{\Omega}}\}$ (*callback rules*). The transition rules $\rightarrow_{C_A^{\circlearrowleft}}$ allow all possible interactions with C_A. Using rules from $\rightarrow_{C_A^{\Omega}}$, it is possible to generate any possible interaction $a \in \mathcal{P}_A$ to a provided interface $q_i \in \mathcal{P}_{C_A}$ of Π_{C_A} (cf. Section 7.5.1). Using rules from $\rightarrow_{C_A^{\mho}}$ any external interaction of C_A is captured (cf. Section 7.5.2). I. e., all interactions with the context are captured.

The semantics of callbacks are that any call to a required interface $q_i \in R_{C_A}$ of C_A can result in a sequence of calls to any provided interface $q_j \in \mathbb{P}_{C_A}$ of C_A. To represent this behavior we generate the transition rules $\rightarrow_{\circlearrowleft}$. Informally this ensures that after each call to a required interface q_i (mapped by $\mathbb{M}_{C_A^{\Omega}}$ to p_{q_i}) any of the provided interfaces $q_x \in \mathbb{P}_{C_A^{\Omega}}$ of C_A can be called, too, using $p_{v_i} \xrightarrow{x} q_x \otimes p_k$ (contained in $\rightarrow_{C_A^{\Omega}}$).

An example is shown in Example 7.5. There, a protocol violation is initiated if a call to a could result in some way to a call of b of component `C2`. This counterexample (Example 7.5c) precisely describes the problem observed in Example 7.1 on page 157. Thus, the problem discussed in the motivation will be discovered before component `C1` has been developed, which is a great benefit and good support for the developers.

The verification context captures the behavior of every possible implementation.

Theorem 7.3

Let C_{Im} be an implementation of the verification context $C_A^{\circlearrowleft}$ of the aggregated component C_A and $\mathcal{P}_A$ the protocol of C_A. We consider $\Pi_{S\,Im} \hat{=} \Pi_{C_{Im}} \oplus \Pi_{C_A} \oplus \Pi_{C_{Im}}$ and $\Pi_S^{C_{Im}} = \varphi_{C_{Im}}(\Pi_{S\,Im})$ (thus it contains only actions of C_{Im}). If $L(\Pi_S^{C_{Im}}) \subseteq L(\mathcal{P}_A)$ and there is no protocol violation in C_A (i. e., $L(\Pi_{S\,Im}^i) \subseteq L(\mathcal{P}_{C_i})$) for each internal component C_i of C_A, then for each internal component C_i of C_A, there is no protocol violation if C_A is replaced by C_{Im} (i. e., $L(\Pi_S^{C_{Im}}) \subseteq L(\mathcal{P}_{C_i})$).

Remark

If no callback exists within the implementation C_{Im} of the verification context $C_A^{\circlearrowleft}$ we can derive from Theorem 7.1 that all protocol violations are already described with the transition rules of C_A^{Ω}.

In order to prove Theorem 7.3 the following lemma is required.

Lemma 7.3

Let be $\Pi_{S\,Im}^i \hat{=} \varphi_{C_i}(\Pi_S^{C_{Im}})$ and $\Pi_S^{C_i} \hat{=} \varphi_{C_i}(\Pi_S)$, where $\Pi_S^{C_{Im}} \hat{=} \Pi_{C_{Im}} \oplus \Pi_{C_A} \oplus \Pi_{C_{Im}}$ and $\Pi_S \hat{=} \Pi_{C_A^{\circlearrowleft}} \oplus \Pi_{C_A} \oplus \Pi_{C_A^{\circlearrowleft}}$. If $p_{q_x} \overset{w}{\Rightarrow}_{\Pi_{S\,Im}^i} t_E$ then $p_{q_x} \overset{w}{\Rightarrow}_{\Pi_S^{C_i}} t_E$, where $w \in \Sigma_{\mathcal{P}_{C_i}}^*, t_E \in PEX(Q_{\Pi_S^{C_i}}), q_x \in \mathbb{P}_{C_A}$ and a component C_i is considered which is not the top component.[2]

Proof (Lemma 7.3)

Induction over the length n of the sequence of interactions w.

<u>$n = 1$:</u> If $p_{q_x} \overset{w}{\Rightarrow}_{\Pi_S^{C_{Im}}} t_E$, with $w = a \in \Sigma_{\mathcal{P}_{C_i}}^*$ then a transition rule $p_1 \overset{a}{\to} p_2 \otimes p_3$ has to be used. This rule is part of C_A, thus w can be generated in $\Pi_S^{C_i}$, too.

<u>$n = 2$:</u> If $p_{q_x} \overset{a}{\Rightarrow}_{\Pi_S^{C_{Im}}} t \overset{b}{\Rightarrow}_{\Pi_S^{C_{Im}}} t_E$, with $a, b \in \Sigma_{\mathcal{P}_{C_i}}$. It can be refined to $p_{q_x} \overset{a}{\Rightarrow}_{\Pi_S^{C_{Im}}} t \overset{\lambda}{\Rightarrow}_{\Pi_S^{C_{Im}}} t' \overset{b}{\Rightarrow}_{\Pi_S^{C_{Im}}} t_E$. A transition rule $p_1 \overset{a}{\to} p_2 \otimes p_3$ and a transition rule $p_4 \overset{b}{\to} p_5 \otimes' p_6$ have to be used to generate the counterexample, both rules are contained in $\to_{C_A}$, thus they are also available in Π_S ($\otimes, \otimes' \in \{., ||\}$). $t \overset{\lambda}{\Rightarrow} t'$ can be computed within $\Pi_S^{C_{Im}}$ only in three cases:

- *To compute $t \overset{\lambda}{\Rightarrow} t'$ only rules of $\to_{C_A}$ are applied. Thus, the derivation is also computable in Π_S (especially $t = t'$).*
- *To compute $t \overset{\lambda}{\Rightarrow} t'$ rules of the verification context $C_A^{\circlearrowleft}$ are used, but no callbacks. By the remark on page 164 the derivation is also computable in Π_S.*
- *To compute $t \overset{\lambda}{\Rightarrow} t'$ a callback can be used. Thus, the derivation can be refined to $t \overset{\lambda}{\Rightarrow} t_{q_j} \overset{\lambda}{\Rightarrow} t_{q_k} \overset{\lambda}{\Rightarrow} t'$, where $t_{q_j}, t_{q_k} \in PEX(Q_{\Pi_S^{C_{Im}}})$ and in t_{q_j} a transition rule is applicable to a required interface q_j of C_A and t_{q_k} a transition rule is applicable to a provided interface q_k of C_A. Because of the rule $p_{q_j} \overset{\lambda}{\to} p_{q_k} \in \to_{C_A^{\circlearrowleft}}$ this derivation is also possible in Π_S.*

<u>$n = m$:</u> The lemma is valid for any interaction sequence: $p_{q_x} \overset{w}{\Rightarrow}_{\Pi_S^{C_{Im}}} t$.

[2] By assumption the formula $t_E \overset{w'}{\Rightarrow} \varepsilon$ is valid, where $w \in \Sigma_{C_i}$.

$\underline{n = m+1}$: If $p_{q_x} \overset{w}{\Rightarrow}_{\Pi_S^{C_{Im}}} t \overset{a}{\Rightarrow}_{\Pi_S^{C_{Im}}} \varepsilon$, then a rule $p_1 \overset{a}{\to} p_2 \otimes p_3$ has to be applied. This rule is part of $\to_{C_A}$, hence this derivation is also computable in $\Pi_S^{C_i}$, because the first part is covered by the induction hypothesis. □

Corollary 7.3
$L(\varphi_{C_A}(\Pi_{S\,Im})) \subseteq L(\Pi_S^A)$

Proof (Theorem 7.3)
Analogous to Theorem 7.1.

Summary

In this section we have shown, how to support the development process of reliable components and applications, respectively. We use component protocols as constraints for the usage of components and define a formal approach based on model checking. This allows system architects or component developers to check the implementations, even if the components are implemented in different programming languages. They can check whether the implementation of every possible application matches the constraints of the protocols of the components that are aggregated. In contrast to testing procedures we can prove absence of errors, because we find every possible error. This is an important new option.

Moreover, the development might speed up, because if the user of a component has to interact conformably with the protocol, the developer of this component does not have to implement checks and catch for all errors which can be triggered from any unexpected interaction.

The approach allows to check the protocols of components aggregated to a larger component. This common step during the development is now suitably supported by our verification process. Thus, a permanent verification of each composition or development step is possible. The result is a reliable aggregated component that can be composed easier. As a side effect the costs of the verification can be reduced, because component protocols need not to be checked again, if they are integrated into an aggregated component.

Moreover, the constructed counterexamples of an aggregated component can be used to evaluate the design of the application in an early phase as well as they can point the developers of the currently unfinished components to possible implementation errors. Both will lead to a more predictable implementation of the application, where errors could be discovered earlier.

We combined here the assumption about the application software design with a verification approach. Hence, our verification approach can be applied to consider B2B-scenarios, client server and tier architectures, Service-oriented Architectures and many more.

7.7 Discussion

The traditional verification approach considers given applications, which have to be (almost) implemented completely.

In this chapter we have shown, how our approach can be extended to become applicable for iterative component development or composition, respectively. We have shown that a verification in an early development step is possible. These topics have a big importance in an industrial context, because they can delay the time to market and raise the development costs. As we know from several considerations [Boe81, Bal96] eliminating an error caused in an early development step is very expensive. Moreover, iterative development is a common approach for industrial software development.

Summarized, our approach can provide pieces of information about the current state of the architecture and the compositionality of the currently existing components. Furthermore, we can point to possible problems which might appear while developing a currently missing component. And last but not least, the model checking problem can be simplified, if component protocols can be omitted.

8 Conclusions and Future Work

In this work we consider the verification of component composition, which is one of the most important issues of component-based software engineering [PTDL07]. Currently, component composition is based on functional properties of the components (signature-based). The users of our approach can define an easy understandable constraint, all clients of a component have to obey. For the definition of a *component protocol* neither mathematical background nor implementation details are needed. The goal is to ensure statically that a component-based software works in the defined manner.

The *main contribution of our work* is, that we define the verification of component behavior based on a representation, which is capable to represent unbounded recursion (like push-down automata) and unbounded parallelism (like Petri nets). These *Process Rewrite Systems* enable to discover problems triggered by recursive callbacks and asynchronous behavior, too. Even synchronization through external interactions (e. g., used in BPEL) can be represented.

We translate source code into abstractions automatically using standard compiler techniques. Thus, the correct behavior of the real implementation (source code) of an application can be proved at compile time or deploy time. Although we break up the blackbox principle a little bit, while publishing the abstract behavior of components, we are obeying the principles of component-based software and Service-oriented Architectures. Accordingly the scientific questions from Section 1.2 can be answered positively.

An important property of our approach is that we distinguish between the component contract and the component behavior. This is a distinguishing feature to many other approaches. Our approach can be interpreted as another option for design by contract. We define an automatic method (Figure 8.1). It captures the process from the abstraction of a single component to the computation of the protocol violations (counterexamples), which is capable to deal with Process Rewrite Systems.

1. The source code of each component is translated into an abstract representation (capable to represent unbounded parallelism and unbounded recursion). This component abstraction represents the relevant behavior. Hence, a component abstraction imitates the component behavior. The encoded business secrets are kept safe. The source code is not needed any more.
2. In the second phase the components are composed like the real application. We also respect the possible dynamic binding of components.
3. To verify the behavior of the application, the abstraction has to be evaluated. We have defined a new representation, named *Combined Abstraction*. It encodes the model checking problem, while considering a single protocol and the complete application behavior.
4. The model checking of the Combined Abstraction is based on the model checking of Process Rewrite Systems. We compute counterexamples for each protocol violation. They describe precisely which execution trace (program points) leads to an error.
5. The constructed and comprehensive counterexamples can be evaluted easily by a human.

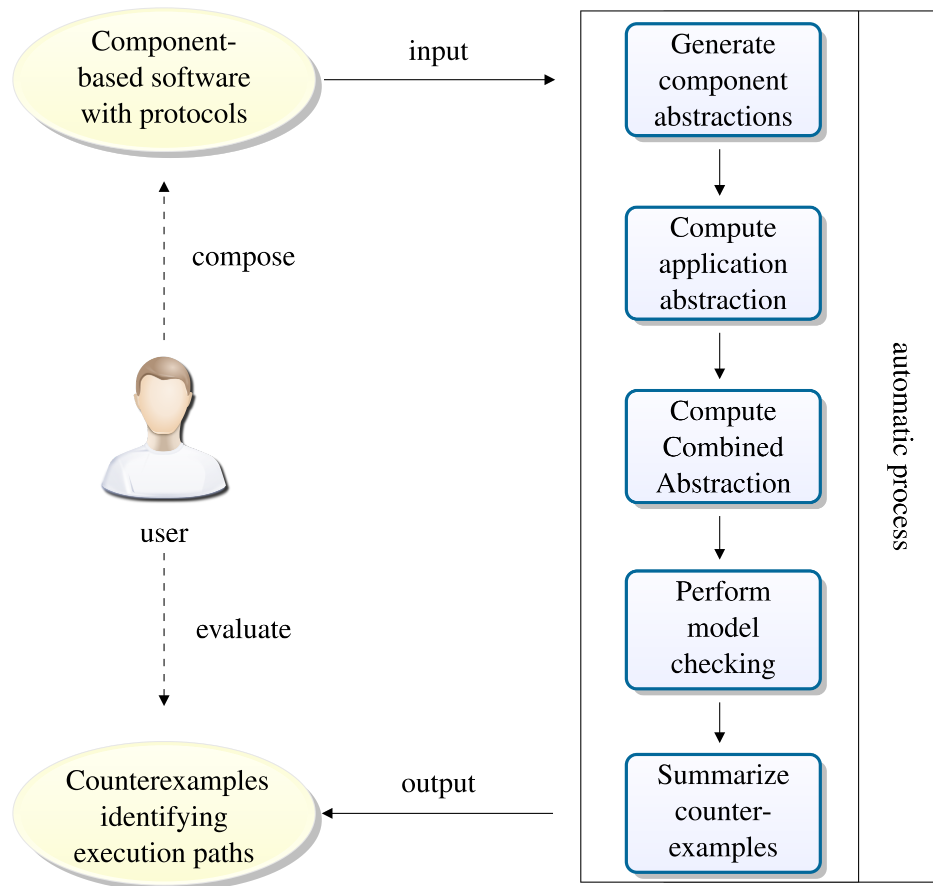


Figure 8.1.: Overview: Verification process.

8.1 Component Properties

Our approach is capable to deal with the properties of components (corresponding to Section 1.3), component systems and Service-oriented Architectures:

- The used formal representation of the component behavior (Process Rewrite Systems) is capable to deal with many programming and interaction paradigms. It is possible to represent unbounded recursion and unbounded parallelism. Thus, synchronous and asynchronous interactions as well as synchronization by interaction can be captured.
- Our approach is independent from the (imperative or object-oriented) programming language a component is implemented with. The platforms and the location on which a component is deployed have no influence. Moreover, we can deal with different component systems.
- Dealing with the context-insensitive development of components is the main task of our verification process. Moreover, our approach is capable to deal with components binding other components dynamically depending on the environment (component repository) which is common in Service-oriented Architectures.

- Because errors can be triggered by the complete behavior of a component-based software we always consider the abstraction of the complete application to decide, whether a component protocol is obeyed in any case. The protocol is defined by using provided interfaces only, thus a component developer needs no global view on the Component-based Architecture or Service-oriented Architecture. Hence, we overcome the problems of peer to peer verification.
- Although our approach considers the complete behavior of the component-based software, it is focused on a single component. A component developer has only to deal with "his" component, its protocol and protocol violations.
- Our approach can deal with stateful and stateless components. Since earlier results [ZS06] are applicable, it is possible to deal with references, too.
- Our approach keeps the implemented business secrets safe. Only the control flow is considered and the component abstraction is generated locally. The blackbox semantics are obeyed.

Our approach is capable to deal with software created while assembling components provided through component systems or Service-oriented Architectures. Under the premise of using a compatible execution model (cactus stack), it is possible to verify if a component is used in the expected way by the complete implemented component-based software.

8.2 Comparison with other Approaches

Our approach is different from other works in several properties. The main distinction is the more powerful behavior representation. The formal model used for representing the abstracted component behavior is the corner stone of the properties following. The used Process Rewrite Systems allow the presentation of unbounded recursion and unbounded parallelism (properties I, II in Table 8.1). Therefore, it is possible to represent synchronous and asynchronous interactions as well as synchronization by interaction and recursive callbacks (III, IV). This was not possible while using any other known approach.

The reason, for many improvements, is the separation of constraint and behavior of components (V). This idea is borrowed by the work of Zimmermann and Schaarschmidt [ZS06]. Our work can be viewed as ancestor of their approach (approach number 9 in Table 8.1).

The abstractions we consider here are based on the source code (VI) and the verification takes into account the complete application behavior (VII). This is also not common sense, but we consider these properties as very important. Otherwise the actual component behavior can differ from the (human-defined) specification or an error can be missed. Both might be bad for model checking approaches, therefore other approaches have to reduce the scopes of applicability stronger than we did. Moreover, it is possible to consider only a couple of components and verify their interactions (VIII). However, during this possible iterative component integration we warn for errors, that might appear, while adding the "missing" parts. From property V and VI

approach is using	I. capture recursive behavior	II. capture parallel behavior	III. capture synchronous calls	IV. capture asynchronous calls	V. contract and behavior separated	VI. use source code as behavior	VII. verify complete system behavior	VIII. verify peer to peer behavior	IX. use provided interface, only	X. consider substitutability	XI. allow non-determinismus	XII. more powerful constraints
1. finite state machines [YS94, YS97]	/	N	Y	N	N	N	N	Y	N	Y	N	N
2. CSP^{-} [AG97]	Y	O	N	O	Y	N	N	Y	N	N	Y	Y
3. counter automata [Reu02b, Reu02a]	O	N	/	/	Y	N	N	Y	N	Y	Y	Y
4. FSM^{+} [PV02]	/	Y	O	Y	N	O	N	Y	N	Y	Y	Y
5. workflow nets [VdAvHvdT02]	N	Y	O	Y	N	N	Y	N	N	Y	Y	Y
6. DFSM [SKPR04]	/	Y	Y	N	N	N	/	Y	N	Y	O	Y
7. non-regular protocols [Sud05]	/	N	N	Y	N	N	N	Y	N	Y	N	Y
8. STS^{+} [PNPR05b]	/	N	Y	Y	N	O	N	Y	N	N	/	Y
9. context free grammars [ZS03, ZS06]	Y	N	Y	N	Y	Y	Y	N	Y	N	Y	N
10. eLTS [AAA05, AAA06, AAA07]	/	N	Y	Y	N	N	N	Y	N	/	Y	Y
11. **PRS (this work)**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**O**	**Y**	**N**

The sign "**Y**" is used if the approaches fulfill the property. If not, "**N**" is used. If the property is only fulfilled partly, "**O**" is used. Some attributes have no meaning for the considered approach, in this case "/" is used.

Table 8.1.: Comparison with related work (extended version of Table 2.1 on page 19).

follows the advantage that not for every component a protocol has to be defined (by a human) like in the other works. This reduces the effort for the application of our approach.

Process Rewrite Systems provide another important advantage: As this model unifies Petri nets and push-down automata it is possible to take advantage of model checking approaches developed for subclasses of Process Rewrite Systems. E. g., if a component-based software makes no use of a parallel behavior (or if it is removable), we can take advantage of the well known push-down automaton techniques and logics.

We assume that it is easier for the component developer to define a protocol based on the provided interfaces only (IX). Thus, he has not to predefine or redefine the component behavior

as it is done in the approach using "behavioral protocols". Anyway, these approaches have the advantages that substitutability can be considered in an easier and more concrete way (X). In this work substitutability of components can only be considered under the question if a component can be used within an existing component-based software replacing another component. Other works have a more powerful definition, where it can be decided locally if a substitute has the same behavior as a preexisting component.

Our approach is capable to deal with non-deterministic constraints and behavior (XI). Thus we have not to deal with state-space explosion while transforming them into a deterministic one, which might obfuscating the model checking. However, it is not possible to define component protocols which are more powerful than finite state machines (XII). Hence, the formulation for some components is not possible (e. g., of a stack implementation).

In this work we have presented a unique approach. It eliminates several lacks of other approaches. Many decisions are taken into account the planned use in an industrial context.

8.3 Implementation and Practical Applicability

The approach developed in this work was implemented in a component-based software. The architecture of the framework was defined under the aim to keep the implementation extendable and the components exchangeable. Moreover, we have taken the requirements of our industrial partners into account.

In detail the functionality was divided into the following scopes:

- generators for abstractions of source code of components,
- optimizations of component abstractions and system abstractions as well as operations like the computation of Combined Abstractions,
- user interfaces, user tracking and evalutions considering the counterexample evaluations,
- model checking of Process Algebra Nets.

For each of these areas exist at least one implementation.

The framework was applied and evaluated in a (industrial) case study. The results show that our approach is applicable on problems of practical relevance in respect to the formulation of problems as well as an adequate solution.

8.4 Future Work

In this section, we will describe what further (scientific) questions our approach created, where potential for extensions is available and what other applications could be possible.

The approach we have presented in this work is capable to deal with many other scopes of applicability. Considering workflows is a valuable task. Nowadays workflows – also known

as business process models (BPM) – are often used in a preparation step to specify new applications for Service-oriented Architectures. Using the pieces of information defined already in these models allow a check whether the workflow can be implemented using the pre-existing components. In [BZ09b] we have already defined an extension of our approach leading to the answer of this question. It allows the consideration of each BPM which can be translated into Petri nets. Therefore, it is applicable to the extensively used business process modeling notation (BPMN [BPM]) and event-driven process chains (EPC [Kel92]). Another extension is the consideration of Service-oriented Architectures where the available components are not known at deployment time, e. g., if the component repositories are chosen dynamically. To verify applications, that are assembled in this dynamic way, we have defined an extension of the component model and a new verification process in [BZ09c].

Other scopes of applicability can be derived from the protocols existing after applying our approach. Test techniques have the problem of state space explosion while considering data values. As the component protocol reduces the interaction sequences the state space could be reduced. In the same context component protocols could be validated using test or model checking techniques. This will lead to more reliable components, as our approach requires a component working in the defined way. However, we leave it to the component developer to actually check if the component behaves as expected. An extension and simplification of our verification process could be the generation of component protocols based on the actual implementation. For this purpose the evaluation and model checking techniques (e. g., [CGP99]) can be used to discover unsafe situations within a component, which should be forbidden by the protocol.

An interesting adaption of our approach might be the consideration of adapted Process Rewrite Systems (e. g., [prs04]). This might enable the consideration of further problems, because more model checking problems are applicable.

As mentioned before (see Chapter 2) other approaches lack at the correspondence between source code and component specification (behavioral protocol). Nevertheless, the tool support in this research is better (e. g., [BHP06]). We are sure that both approaches can be combined. The idea is to separate the behavioral protocol in a provided and required protocol using the function φ we have defined. Using the process shown in Chapter 7 it should be possible to derive the verification context from the behavioral protocol.

Interesting seems to be the consideration of legacy components. The integration of such components is a common task. Therefore, an approach for handling these components should be developed.

Other approaches consider performance predication [BGMO06], component substitutability [SCCS05, PV02] and deadlock freeness [IT03, AINT07, GS03a], too. We assume that the source code abstraction contained in our approach can be extended to allow statements about the performance of components. Component exchangeability will need a significant extension of our approach. While applying or adapting the Liskov substitution principle [LW93, LW94, LW01] (exchangeability) for component representations using Process Rewrite Systems, the languages defined over the required interface usage have to be considered. To our knowledge no work ex-

ists considering these languages. In contrast statements about deadlocks are already considered in [May98]. Although the expressiveness of other works (e. g., Petri nets [CX97]) cannot be reached, several problems might be solvable. Thereafter, it could be possible to evaluate if a component-based software can ever reach a state where no termination is possible. Developing just criteria for these problems is interesting, too.

The abstractions should be considered in respect to the question, whether an abstraction could be inaccurate (because the execution model is not matching fully). By considering data values, more accurate abstractions seem to be possible, while computing component abstractions. Considering *general Process Rewrite Systems* might be helpful (cf. [Esp02]).

Moreover, the abstractions should be considered with the aim of reducing the number of transition rules. Early prototypes of further optimizations show that it is possible to reduce the model checking problem dramatically (sometimes only the actual counterexample is still contained in the optimized Combined Abstraction). Hence, it seems to be interesting to consider optimizations in more detail (other work show that this is useful on source code level, too [DR07]), to reduce the model checking effort.

In general the model checking Process Rewrite Systems was not considered sufficiently in current research. There seems to be much potential for optimizations and improvements. Adapting existing model checking optimizations [Ric09, Ric08] might be possible and suitable. We assume that on-the-fly model checking (where the Combined Abstraction is computed lazily during the traversing) and multi-threaded algorithms have a great potential to improve the model checking performance (especially while considering PA-processes).

Besides the academic questions, the practical application should not be lost out of sight. The approach is capable to be integrated into integrated development environments (IDE) and component models. The integration into well established industrial middleware (like IBM WebSphere Message Broker® [ibm05]) or service level agreements (SLA) should lead to faster implementations and more reliability because of the provability of the protocols. Moreover, this would lead to larger case studies, a wider use and more research. The influence on software quality of the non-functional requirements (encoded in the component protocols) should be evaluated in industrial case studies leading to an even better understanding of the needs in industrial contexts, e. g., to our experience leads the preoccupation with component protocols by developers to an educational effect of the actual behavior of their implementations.

Index of Definitions

Aggregated component, 37
Architecture of the aggregated component, 159
Asynchronous interactions, 39

Base component, 33
Blackbox component, 30

Cactus stack, 39
Callbacks, 39
Combined Abstraction, 102
Component, 29
Component abstaction, 62
Component protocol, 53
Component-based software, 37
Composition, 34
Composition operator $\oplus$, 156
Context, 159
Counterexample, 58

Derivation relation of PRS, 46

Empty process, 46
Execution trace, 38
Extended counterexample, 128

False negatives, 50
False negatives of an abstraction, 62
False positives, 50
Finite state machine (FSM), 40

Initial component, 33
Interface, 29
Interleaving semantics, 38
Inverted finite state machine, 42

Language accepted by an abstraction, 64

Model checking, 49

No relevant action, 63
Normalform of PRS, 49

Observable behavior of a component-based software or aggregated component, 38
Observable component behavior, 38

Parallel behavior, 38
Petri nets, 43
Process Rewrite Systems (PRS), 46
Process-algebraic expressions (PEX), 46
Protocol, 54
Protocol conformance, 56
    of a component-based software, 58
Push-down automata (with one state), 42

Reachability, 50
Regular expression, 42
Regular language, 42

Sequential behavior, 38
Signature, 29
Stripped PAN, 84
Stripped PRS (SPRS), 77
Synchronous interactions, 39
System abstraction, 62
    as Process Rewrite System Π_S, 89

Top component, 159

Unbounded interfaces, 37
Unresolvable transition rules, 108
Use of a component C in S: $\Pi_{S,C}$, 56
Use of a component C in S in a Process Rewrite System Π_S, 64

Verification context $\Pi_{C_A^{\circlearrowleft}}$, 166
Verification driver C_A^{Ω}, 162
Verification dummy $C_A^{\mho}$, 166

Bibliography

[AAA05] P. Andre, G. Ardourel, and C. Attiogbe. Behavioural Verification of Service Composition. *Engineering Service Compositions (WESC'05)*, pages 77–84, 2005.

[AAA06] C. Attiogbe, P. Andre, and G. Ardourel. Checking component composability. *Lecture Notes in Computer Science*, 4089:18, 2006.

[AAA07] P. Andre, G. Ardourel, and C. Attiogbe. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In *Software Composition: 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*. Springer, 2007.

[AAH98] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.

[Aal00] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 161–183, London, UK, 2000. Springer-Verlag.

[ABB97] J.M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. *Handbook of Formal Languages. Word, Language, Grammar*, 1:111–174, 1997.

[ABD$^+$04] Alain Abran, Pierre Bourque, Robert Dupuis, James W. Moore, and Leonard L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2004 version edition, 2004.

[Ada06] J. Adamek. Addressing unbounded parallelism in verification of software components. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006. SNPD 2006. Seventh ACIS International Conference on*, pages 49–56, 2006.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[AINT07] Marco Autili, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 784–787, Washington, DC, USA, 2007. IEEE Computer Society.

[All09] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.2*. IOS Press, Inc., September 2009.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Bal96] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, 1996.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005.

[BCS03] E. Bruneton, T. Coupaye, and J. Stefani. The fractal component model. Technical report, ObjectWeb Consortium, 2003. Specification V2.

[BdVG09] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916–1923, New York, NY, USA, 2009. ACM.

[Bei95] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc. New York, NY, USA, 1995.

[BGMO06] Steffen Becker, Lars Grunske, Raffaela Mir, and Sven Overhage. Performance prediction of component-based systems: A survey from an engineering perspective. In *Architecting Systems with Trustworthy Components, volume 3938 of Lecture Notes in Computer Science*, pages 169–192. Springer, 2006.

[BH96] Ahmed Bouajjani and Peter Habermehl. Constrained properties, semilinear systems, and Petri nets. In *CONCUR '96: Proc. of the 7th Int. Conf. on Concurrency Theory*, pages 481–497, London, UK, 1996. Springer.

[BHJ09] Sebastian S. Bauer, Rolf Hennicker, and Stephan Janisch. Behaviour protocols for interacting stateful components with ports. In *Proccedings of International Workshop on Formal Aspects of Component Software (FACS'09)*. Centrum Wiskunde & Informatica, Amsterdam, November 2009.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder–a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BL74] W.S. Brainerd and L.H. Landweber. *Theory of computation*. John Wiley & Sons, Inc. New York, NY, USA, 1974.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., May 2006.

[BN07] C.M. Bishop and N.M. Nasrabadi. Pattern Recognition and Machine Learning. *Journal of Electronic Imaging*, 16(4):9901, 2007.

[Boe81] B.W. Boehm. *Software engineering economics*. Prentice-Hall, 1981.

[BPE03] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language for Web Sevices, Version 1.1*, May 2003.

[BPE07] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language, Version 2.0*, OASIS standard edition, April 2007.

[BPM] BPMI.org. Business Process Modeling Notation (BPMN) Version 1.0.

[BR04] S. Becker and R.H. Reussner. The impact of software component adaptors on quality of service properties. *Issues on Coordination and Adaptation Techniques*, page 25, 2004.

[BRV04] B. Berthomieu, P.O. Ribet, and F. Vernadat. The tool TINA-construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[BS57] Friedrich Ludwig Bauer and Klaus Samelson. Auslegeschrift DE1094019. Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens. (Anmeldetag: 30. März 1957. Bekanntmachung der Anmeldung und Ausgabe der Auslegeschrift: 1. Dezember 1960. Erteilt 12. August 1971. DE-PS 1094019.), 1957.

[BS03] E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[BW96] B. Beizer and J. Wiley. Black box testing: Techniques for functional testing of software and systems. *IEEE Software*, 13(5), 1996.

[BZ08a] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel BPEL systems. *IEEE Sixth European Conference on Web Services (ECOWS '08)*, 0:81–91, 2008.

[BZ08b] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel component-based systems. In Michel R. V. Chaudron, Clemens A. Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, volume 5282 of *Lecture Notes in Computer Science*, pages 163–179. Springer, October 2008.

[BZ08c] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel component-based systems. Technical Report 2008/01, University Halle-Wittenberg, Institute of Computer Science, June 2008.

[BZ09a] Andreas Both and Wolf Zimmermann. Model checking of component protocol conformance – optimizations by reducing false negatives. In *Proccedings of International Workshop on Formal Aspects of Component Software (FACS'09)*. Centrum Wiskunde & Informatica, Amsterdam, November 2009.

[BZ09b] Andreas Both and Wolf Zimmermann. On more predictable implementations of reliable workflows in service-oriented architectures. *IEEE Seventh European Conference on Web Services (ECOWS '09)*, November 2009.

[BZ09c] Andreas Both and Wolf Zimmermann. Sicherstellung der Funktionalität in Komponentensystemen und Service-orientierten Architekturen. In Erik Maehle Stefan Fischer and Rüdiger Reischuk, editors, *GI Jahrestagung*, volume 154 of *Lecture Notes in Informatics*, pages 425;3336–3349. GI, 2009. (in German).

[BZ09d] Andreas Both and Wolf Zimmermann. A step towards a more practical protocol conformance checking algorithm. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)*, pages 458–465. IEEE Computer Society, August 2009.

[BZ09e] Andreas Both and Wolf Zimmermann. Supporting the development process of reliable software during the composition process using interaction protocols. Technical Report 2009/4, University Halle-Wittenberg, Institute of Computer Science, September 2009.

[CCK$^+$06] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.

[CGJ$^+$03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):794, 2003.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT, 1999.

[Cho56] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[Cho57] N. Chomsky. Syntactic structure. *The Hague: Mouton*, 1957.

[CMS02] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS based on Symbolic Transition Systems. *Computer Journal*, 45(1):55–61, 2002.

[COR01] CORBA/IIOP Specification. Technical report, Object Management Group, Inc, 2001. Revision 2.4.2, OMG 01-02-01.

[CX97] F. Chu and X.L. Xie. Deadlock analysis of Petri nets using siphons and mathematical programming. *IEEE Transactions on Robotics and Automation*, 13(6):793–804, 1997.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, 1972.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.

[DGV99] M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. *Computer Aided Verification*, 1999.

[Dij69] Edsger W. Dijkstra. Structured programming. circulated privately, August 1969.

[Dij70] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.

[DOT] .NET Framework. http://msdn.microsoft.com/netframework/.

[DR95] V. Diekert and G. Rozenberg. *The book of traces*. World Scientific Pub Co Inc, 1995.

[DR07] W. Zimmermann D. Richter. Slicing zur Modellreduktion von symbolischen Kellersystemen. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.

[dVG07] Edsko de Vries and John Gilbert. Design and implementation of a PHP compiler front-end. Technical report, Department of Computer Science, School of Computer Science and Statistics, Trinity College Dublin, Ireland, 2007.

[E. 00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-guided abstraction refinement*, pages 154–169. Number 1855 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Edw01] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11:97 – 111, 2001.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[ES90] M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1990.

[Esp02] J. Esparza. Grammars as processes. *Lecture Notes In Computer Science*, pages 277–297, 2002.

[FB09] Rene Franke and Andreas Both. Tool demo: Component-based infrastructure for protocol conformance checking of component-based software. International Workshop on Formal Aspects of Component Software (FACS'09), Eindhoven, The Netherlands, November 2009.

[FFB+09] Rene Franke, Francesco Freder, Christian Blaar, Patrick Hiesinger, Monique Argus, Martin Hörig, Johannes Jahn, and Andre Zepezauer. P2 – Erstellen einer Grafische Benutzeroberfläche für Protokolldefinition und Auswertung. student project work, 2009. Institute of Computer Science, University of Halle, Germany, supervisor: Andreas Both.

[FLNT98] J. Freudig, W. Löwe, R. Neumann, and M. Trapp. Subtyping of context-free classes. In *Proc. 3rd White Object Oriented Nights*, 1998.

[Fou09a] Python Software Foundation. Python compiler package, October 2009. http://docs.python.org/library/compiler.html.

[Fou09b] Python Software Foundation. Python v2.6.4 documentation, October 2009. http://docs.python.org/library/compiler.html.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[Fri99] Nat Friedman. The bonobo component and document model. In *ALS'99: Proceedings of the 3rd annual conference on Atlanta Linux Showcase*, pages 40–40, Berkeley, CA, USA, 1999. USENIX Association.

[GH88] D.E. Goldberg and J.H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99, 1988.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley, 1995.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification XV*, page 18. Chapman & Hall, Ltd., 1995.

[Gri93] M.L. Griss. Software reuse: from library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[Gro09] The PHP Group. PHP Manual, November 2009.

[GS03a] G. Gossler and J. Sifakis. Component-Based Construction of Deadlock-Free Systems. In *FST TCS 2003: foundations of software technology and theoretical computer science: 23rd conference, Mumbai, India, December 15-17, 2003: proceedings*, page 420. Springer-Verlag New York Inc, 2003.

[GS03b] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM.

[Han07] Steve Hanov. websequencediagrams, 2007. `http://www.websequencediagrams.com/` (last visited: 2009-10-31).

[HC01] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[HD68] E. A. Hauck and B. A. Dent. Burroughs' b6500/b7500 stack mechanism. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 245–251, New York, NY, USA, 1968. ACM.

[Hel10] Hannes Hellmig. Abstraktion von BPEL-Prozessen für Protokollprüfungen (in progress). Master's thesis, Institute of Computer Science, University of Halle, Germany, 2010. In German, supervisor: Andreas Both.

[HMM86] R. Harper, D. MacQueen, and R. Milner. *Standard ML*. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes*. electronic edition, 2004.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[HP00] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proc. of the Third Int. Conf. on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[ibm05] *Websphere message broker basics*. IBM Corp., Riverton, NJ, USA, 2005.

[IL01] A. Ingolfsdottir and H. Lin. A Symbolic Approach to Value-passing Processes, chapter Handbook of Process Algebra, 2001.

[IT03] Paola Inverardi and Massimo Tivoli. Deadlock-free software architectures for com/dcom applications. *J. Syst. Softw.*, 65(3):173–183, 2003.

[JB09] Cyril Jaquier and Arturo Busleiman. Fail2ban, November 2009. http://www.fail2ban.org/.

[JC00] John Daniels John Cheesman. *UML Components - A Simple Process for Specifying Component-based Software*. Addison-Wesley Longman, Amsterdam, 2000. ISBN-10: 0201708515 ISBN-13: 978-0201708516.

[Jen91] Kurt Jensen. Coloured Petri Nets: a high level language for system design and analysis. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 342–416, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[JHA+99] S.P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Haskell 98. *Language and Libraries, The Revised Report*, 1999.

[JJ01] C. John and D. John. *UML components: a simple process for specifying component-based software*. Addison-Wesley, Boston, 2001.

[Jon03] S.P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[Jos08] Nicolai Josuttis. Der Business-Case von SOA: Ein Erfahrungsbericht. *OBJEKTspektrum*, 01(01), 1 2008.

[KDE] KDE.org. KParts – component framework for the KDE desktop environment.

[Kel92] A.-W. Scheer G. Keller. Semantische Prozemodellierung auf der Grundlage Ereignisgesteuerter Prozeketten. Technical Report 89, Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, Germany, 1992.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full prepositional temporal logic. In *Computer aided verification: 5th international conference, CAV'93, Elounda, Greece, June 28-July 1, 1993: proceedings*, volume 10, page 97. Springer, 1993.

[KMZB02] Lars M. Kristensen, Brice Mitchell, Lin Zhang, and Jonathan Billington. Modelling and initial analysis of operational planning processes using coloured petri nets. In *CRPIT '02: Proceedings of the conference on Application and theory of petri nets*, pages 105–114, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[Kra08] Johannes E. Krause. Erstellen von Abstraktionen für Python- und C/C++-Programme. student project work, April 2008. Institute of Computer Science, University of Halle, Germany, In German, supervisor: Andreas Both.

[Kra09] Johannes E. Krause. Implementierung einer gemeinsamen Zwischenschicht für C/C++/Python zum Zwecke gemeinsamer Programmanalysen. Master's thesis, Institute of Computer Science, University of Halle, Germany, March 2009. In German, supervisor: Andreas Both.

[KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2004.

[LL02] J. Lee and L. F. Lai. A high-level petri nets-based approach to verifying task structures. *IEEE Trans. on Knowl. and Data Eng.*, 14(2):316–335, 2002.

[Loh08] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. *Lecture Notes in Computer Science*, 4937:77–91, 2008.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM.

[LTKR07] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. Technical Report 1598, Computer Sciences Department, University of Wisconsin, 2007.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2008.

[LW93] Barbara Liskov and Jeannette M. Wing. Family values: A behavioral notion of subtyping. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[LW94] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM transactions on programming languages and systems*, 16(6):1811–1841, 1994.

[LW01] B.H. Liskov and J.M. Wing. Behavioural subtyping using invariants and constraints. In *Formal methods for distributed processing*, page 280. Cambridge University Press, 2001.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[Mal95] Y. Malaiya. Antirandom testing: Getting the most out of black-box testing. pages 86–95, October 1995.

[May81] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM.

[May97] Richard Mayr. Combining Petri nets and PA-processes. In *TACS'97: Proc. of the Third Int. Symposium on Theoretical Aspects of Computer Software*, pages 547–561, London, UK, 1997. Springer.

[May98] Richard Mayr. *Decidability and complexity of model checking problems for infinite-state systems*. PhD thesis, Technical University of Munich, 1998.

[May00] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1-2):264–286, 2000.

[May01] Richard Mayr. Decidability of model checking with the temporal logic EF. *Theoretical Computer Science*, 256(1-2):31–62, 2001.

[MBTS04] G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The art of software testing*. Wiley, second edition, 2004.

[Mee01] Michael Meeks. Bonobo and free software gnome components. *Component-based software engineering: putting the pieces together*, pages 607–619, 2001.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction, 1st editon*. Prentice-Hall, 1988.

[Mey92a] B. Meyer. Eiffel: the language. *Prentice-Hall Object-Oriented Series*, page 594, 1992.

[Mey92b] Bertrand Meyer. Applying ”design by contract”. *Computer*, 25(10):40–51, 1992.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil92] D. Miller. The pi-calculus as a theory in linear logic: Preliminary results. In *Extensions of logic programming: third international workshop, ELP'92, Bologna, Italy, February 26-28, 1992: proceedings*, page 242. Springer, 1992.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[MLM$^+$06] C.M. MacKenzie, K. Laskey, F. McCabe, P.F. Brown, and R. Metz. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.

[MT00] L.I. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):343–349, 2000.

[MTMH97] R. Milner, M. Tofte, D. Macqueen, and R. Harper. *The definition of standard ML: revised*. The MIT Press, 1997.

[Mye79] Glenford J. Myers. *The art of software testing*. John Wiley & Sons, Inc. New York, NY, USA, 1979.

[Nie93] O. Nierstrasz. Regular types for active objects. In *OOPSALA '93*, volume 28:10 of *ACM SIGPLAN Notices*, 1993.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NM95] O. Nierstrasz and T.D. Meijler. Requirements for a composition language. *Lecture Notes in Computer Science*, pages 147–147, 1995.

[NR69] P. Naur and B. Randell. *The 1968/69 NATO Software Engineering Reports*. NATO, 1969.

[NTD+95] O. Nierstrasz, D. Tsichritzis, L. Dami, D. Konstantas, and X. Pintado. *Object-oriented software composition*. Prentice Hall, 1995.

[Pet73] C.A. Petri. Concepts of net theory. In *Proceedings of MFCS*, volume 73, pages 137–146, 1973.

[Pet77] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[PJHA+99] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Report on the programming language Haskell 98, a non-strict, purely functional language, 1999.

[PNPR05a] S. Pavel, J. Noye, P. Poizat, and J.C. Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In *Software composition: 4th international workshop, SC 2005, Edinburgh, UK, April 9, 2005: revised selected papers*, page 115. Springer Verlag, 2005.

[PNPR05b] S. Pavel, J. Noye, P. Poizat, and J.C. Royer. A Java implementation of a component model with explicit symbolic protocols. *Lecture Notes in Computer Science*, 3628:115–124, 2005.

[PP09] Tomáš Poch and František Plášil. Extracting behavior specification of components in legacy applications. In *CBSE '09: Proceedings of the 12th International Symposium on Component-Based Software Engineering*, pages 87–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[Prä09] Stephan Prätsch. Implementierung eines Modellprfungsalgorithmus für Prozess-Algebra-Netze. Master's thesis, Institute of Computer Science, University of Halle, Germany, September 2009. In German, supervisor: Andreas Both.

[prs04] *Extended process rewrite systems: Expressiveness and reachability*. Springer-Verlag New York Inc, 2004.

[PST07] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 254–257. Springer, 2007.

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer, Innovative Technology for Computer Professionals*, 40(11):38–45, November 2007.

[Pun99] F. Puntigam. Non-regular Process Types. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 1334–1343. Springer-Verlag London, UK, 1999.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[Pyt09] List of python software, November 2009. `http://en.wikipedia.org/wiki/List_of_Python_software`.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Rei85] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc. New York, NY, USA, 1985.

[Reu02a] Ralf H. Reussner. Counter-constrained finite state machines: Modelling component protocols with rescource dependencies. Technical Report 6, Fakultät für Informatik, Institut fr Algorithmen und Kognitive Systeme (IAKS), Universität Karlsruhe, 2002.

[Reu02b] Ralf H. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In Bill Grosky, Frantisek Plasil, and Ales Krenek, editors, *Proc. of the 29th Annual Conf. in Current Trends in Theory and Practice of Informatics (SOFSEM 2002), Milovy, Czech Republic*, volume 2540 of *Lecture Notes in Computer Science*, pages 20–40. Springer-Verlag, Berlin, Germany, November 2002.

[Ric08] Dirk Richter. Modellreduktionstechniken für symbolische Kellersysteme. Proc. of the 25. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008.

[Ric09] Dirk Richter. Rekursionspräzise Intervallanalysen. Im Rahmen des 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS), 2009.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[Rud10] Andreas Rudolf. Erzeugen von Process Rewrite Systems - Abstraktionen aus PHP-Code (in progress). Master's thesis, Institute of Computer Science, University of Halle, Germany, 2010. In German, supervisor: Andreas Both.

[SCCS05] Natasha Sharygina, Sagar Chaki, Edmund M. Clarke, and Nishant Sinha. Dynamic component substitutability analysis. In John Fitzgerald, Ian J. Hayes,

and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2005.

[Sch63] M. P. Schutzenberger. On Context-Free Languages and Push-Down Automata. *Information and control*, 6:246–264, 1963.

[Sch96] Roy W. Schulte. "Service Oriented" Architectures, Part 2. Research Note SPA-401-069, the Gartner Group, April 1996.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley Longman, Amsterdam, 2nd ed. (15. November 2002) edition, 2002.

[SK00] Patrick J. Schroeder and Bogdan Korel. Black-box test reduction using input-output analysis. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 173–177, New York, NY, USA, 2000. ACM.

[SKPR04] Heinz W. Schmidt, Bernd J. Krämer, Iman Poernomo, and Ralf Reussner. Predictable component architectures using dependent finite state machines. In *Proc. of the NATO Workshop Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2004.

[SN96] Roy W. Schulte and Yefim V. Natis. "Service Oriented" Architectures, Part 1. Research Note SPA-401-068, the Gartner Group, April 1996.

[Sud05] M. Sudholt. A model of components with non-regular protocols. *Lecture Notes in Computer Science*, 3628:99–113, 2005.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[TMP08] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008.

[Ull98] J.D. Ullman. *Elements of ML programming (ML97 ed.)*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Form. Methods Syst. Des.*, 1(4):297–322, 1992.

[VBvdA01] H. M. W. Verbeek, T. Basten, and W. M . P. van der Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44, 2001.

[vdA97] Wil M. P. van der Aalst. Verification of workflow nets. *Lecture Notes in Computer Science*, 1248:407–426, 1997.

[vdA98] Wil M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[VdAvHvdT02] W. M. P. Van der Aalst, K. M. van Hee, and R. A. van der Toorn. Component-based software architectures: a framework based on inheritance of behavior. *Science of Computer Programming*, 42(2-3):129–172, 2002.

[Ver01] R. Veryard. *Component-based business: plug and play*. Springer Verlag, 2001.

[VvdA05] H. M. W. Verbeek and W. M. P. van der Aalst. Analyzing BPEL processes using Petri nets. In D. Marinescu, editor, *2nd Int. Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PNCWB 2005)*, pages 59–78. Florida International University, Miami, Florida, 2005.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Web07a] Web Services Description Working Group. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Technical report, World Wide Web Consortium (W3C), April 2007.

[Web07b] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, World Wide Web Consortium (W3C), June 2007.

[Whi00] J. A. Whittaker. What is software testing? And why is it so hard? *IEEE software*, 17(1):70–79, 2000.

[Wie03] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *SFCS '83: Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 185–194, Washington, DC, USA, 1983. IEEE Computer Society.

[YG] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. volume 25, pages 67–96. Formal Methods in System Design.

[YS94] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *SIGPLAN Not.*, 29(10):176–190, 1994.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[ZS03] W. Zimmermann and M. Schaarschmidt. Model checking of client-component conformance. In *2nd Nordic Conf. on Web-Services*, number 008 in Mathematical Modelling in Physics, Engineering and Cognitive Sciences, pages 63–74, 2003.

[ZS06] Wolf Zimmermann and Michael Schaarschmidt. Automatic checking of component protocols in component-based systems. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2006.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

A Appendix

A.1 Notations Used in this Thesis

Notation	Description
C_i	a component named C_i
Π	abstract behavior
Π_{C_i}	abstract behavior of the component C_i
$\mathcal{P}_{C_i}$	protocol of a component C_i (represented as finite state machine or regular expression)
Π_S	system abstraction of a complete component-based software
$\Pi_{S,C}$	system abstraction, considering only the interactions of a component C
Π_S^C	Combined Abstraction considering the calls to the component C
$PEX(Q)$	set of process-algebraic expressions
s, t	process-algebraic terms, $s, t \in PEX(Q)$
v	states of an (inverted) protocol, $v \in \mathcal{P}_A$
a, b, λ	interactions in the considered system, $a, b \in \Sigma$
λ	empty sequence of interactions
w, x, y, λ	sequences of interactions, $w, x, y \in \Sigma^*$
p, ε	atomic processes, $p \in Q$, empty process, $\varepsilon \in Q$

Table A.1.: Overview of notations.

A.2 Syntax of Used Programming Language

The syntax of the example language allow procedures only. Therefore, result parameters are possible using the keyword “out”. All other parameters are value parameters. To simplify the notation of source code, we allow also natural language within the source code. We demand that one statement per line is written.

Prog ::= Interface* Component*

Interface ::= “interface” name “begin” Signature* “end”

Signature ::= mod type name “(” types “)”

types ::= `ListOf` type

Component ::= “component” name “implements” Interfaces

Interfaces ::= Init “begin” Procedure* “end”

Init ::= type identifier “=” value

Procedure ::= “proc” name “(” ParameterDefs “)” “begin” Block “end”

ParameterDefs ::= `ListOf` (inout type name)

Parameter ::= `ListOf` (type name)

Block ::= Statement*

Statement ::= “if” Expr “then” Block

Statement ::= “if” Expr “then” Block “else” Block

Statement ::= “call” name “(” Parameters “)”

Statement ::= “while” Expr “do” Block

Statement ::= Expr

Statement ::= name “=” Expr

Expr ::= identifier

Expr ::= Expr op identifier

Expr ::= “not” identifier

name ::= identifier

type ::= identifier

mod ::= “sync”

mod ::= “async”

inout ::= “out”

inout ::= ε

op ::= “&&”

op ::= “||||”

op ::= “+”

op ::= “−”

op ::= “∗”

op ::= “/”

op ::= “==”

op ::= “<”

op ::= “>”

The semantics of our programming language is defined as usual:

keyword	semantics
“sync”	The marked interaction is performed synchronously.
“async”	The marked interaction is performed asynchronously.
“out”	The value of the marked parameter is set within the procedure.
“call”	This keyword labels a call to a procedure explicitly.
“begin”	Labels the beginning of a scope.
“end”	Labels the end of a scope.
“+”	Arithmetic addiction.
“&&”	Logical and.
“not”	Logical negation.

A.3 More Figures, Tables and Listings

Example A.1: Service for registration and confirmation of service items based on time and material. The example is discussed intensely in [BZ09b].

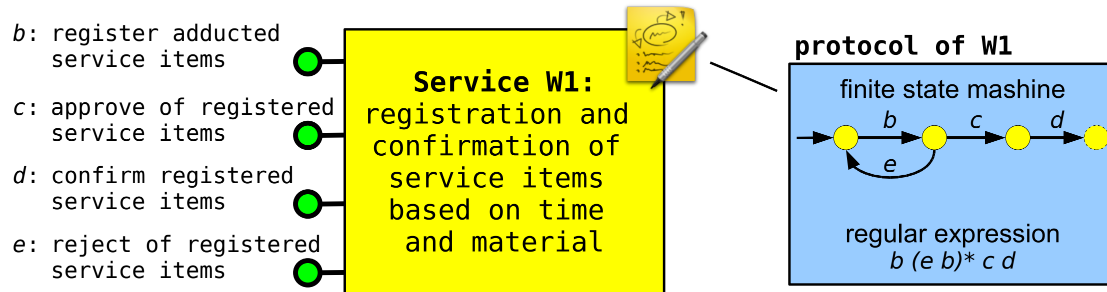


Figure A.1.: A Saguaro cactus (Carnegiea gigantea), `wikimedia.org`.

abbr.	PRS classification	description
FS	$(1,1)$-PRS	Finite-State Systems (finite state machines)
BPA	$(1,S)$-PRS	Basic Parallel Algebra, push-down automata with one state
BPP	$(1,P)$-PRS	Basic Parallel Processes
PA	$(1,G)$-PRS	PA-Processes
PDA	(S,S)-PRS	push-down automata
PN	(P,P)-PRS	Petri nets
PRS	(G,G)-PRS	(general) Process Rewrite Systems

Table A.2.: Meaning of the acronyms of the PRS-hierarchy.

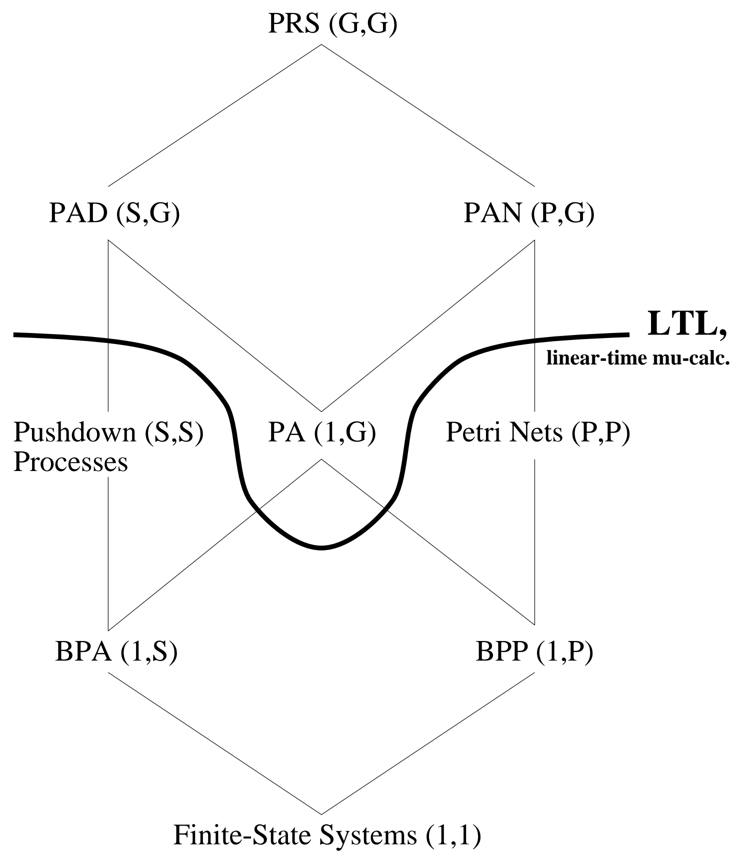


Figure A.2.: Limits of the decidability of linear-time logics (taken from [May98, May01]).

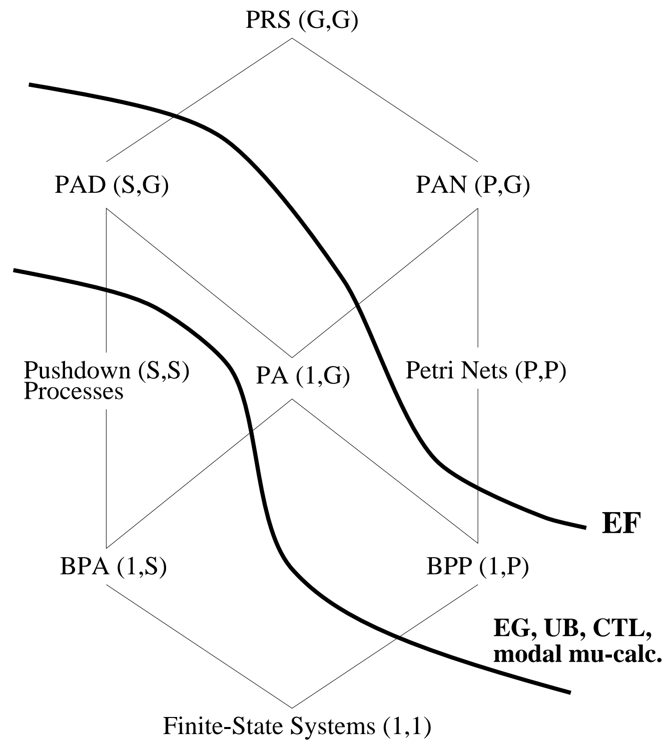


Figure A.3.: Limits of the decidability of branching-time logics (taken from [May98, May01]).

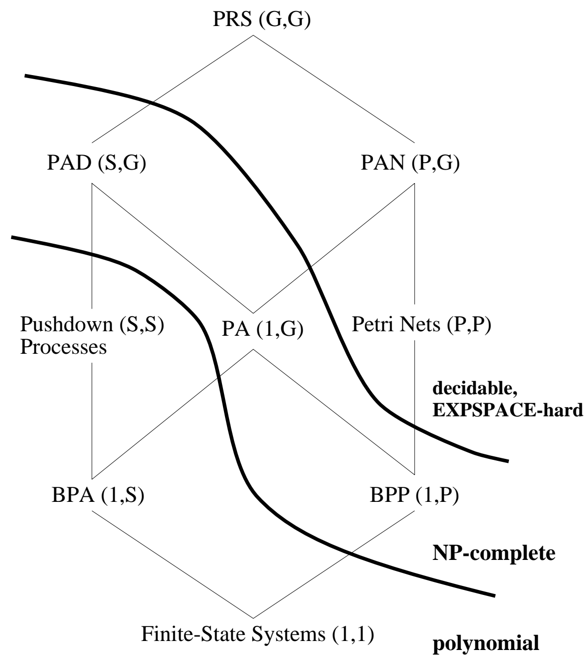


Figure A.4.: The complexity of reachability of Process Rewrite Systems (taken from [May98]).

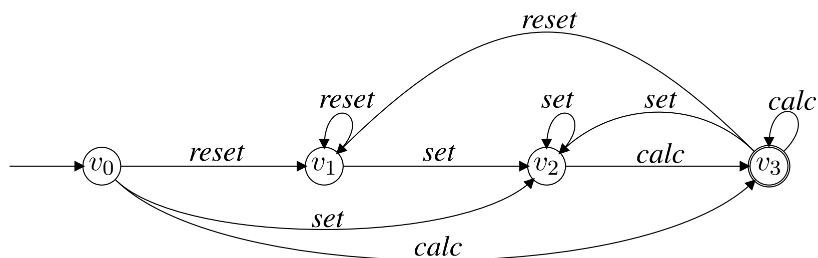
```python
#!/bin/python

import time
from threading import Thread

class testit(Thread):
    def __init__ (self, name):
        Thread.__init__(self)
        self.name = name

    def run(self):
        print self.name+" starts."
        time.sleep(5);
        print self.name+" finishs."

def startThread():
    current = testit("t1")
    current.start()
    return current


def main():
    mythread = startThread();
    print "main is waiting."
    mythread.join()
    print "main has finished."


main()
```

Example A.3: Program returning thread identifier as return value.

Example A.2: A more specific protocol of component C_2 of Example 4.3 on page 59.

$$\mathcal{P}_{C_2} \hat{=} (((reset^+ \; set^+)^* | set^*) \; calc^+)^*$$

(a) Protocol $\mathcal{P}_{C_2}$ of component C_2 as regular expression.



(b) Protocol of component $\mathcal{P}_{C_2}$ in a graphical representation.

Listing A.1: WSDL Definition of WSA and WSO

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WSO"
 targetNamespace="http://localhost:80/WSO.wsdl"
 xmlns:tns="http://localhost:80/WSO.wsdl"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:WS-A="http://tempuri.org/WS-O.xsd"
 xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:MIME="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:DIME="http://schemas.xmlsoap.org/ws/2002/04/dime/wsdl/"
 xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>

 <schema targetNamespace="http://tempuri.org/WS-A.xsd"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:WS-A="http://tempuri.org/WS-A.xsd"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">
  <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
  <simpleType name="Operation">
   <restriction base="xsd:string">
    <enumeration value="no-op"/><!-- enum const = 0 -->
    <enumeration value="sequ-op"/><!-- enum const = 1 -->
    <enumeration value="sync-op"/><!-- enum const = 2 -->
    <enumeration value="fork-op"/><!-- enum const = 3 -->
    <enumeration value="eps-op"/><!-- enum const = 4 -->
   </restriction>
  </simpleType>
  <simpleType name="TransitionTyp">
   <restriction base="xsd:string">
    <enumeration value="no-typ"/><!-- enum const = 0 -->
    <enumeration value="active"/><!-- enum const = 1 -->
    <enumeration value="inactive"/><!-- enum const = 2 -->
    <enumeration value="lambda"/><!-- enum const = 3 -->
   </restriction>
  </simpleType>
  <complexType name="IntVector">
   <sequence>
     <element name="item" type="xsd:int" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
  </complexType>
  <complexType name="NodeId">
   <sequence>
     <element name="componentId" type="xsd:int" minOccurs="1" maxOccurs="1"/>
     <element name="componentNodeId" type="xsd:int" minOccurs="1" maxOccurs="1"/>
     <element name="pre" type="xsd:int" minOccurs="1" maxOccurs="1"/>
     <element name="post" type="xsd:int" minOccurs="1" maxOccurs="1"/>
   </sequence>
  </complexType>
  <complexType name="ArrayOfNodeIds">
   <sequence>
     <element name="nodeIds" type="WS-A:NodeId" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
  </complexType>
  <complexType name="NodeInfo">
   <sequence>
     <element name="nodeId" type="WS-A:NodeId" minOccurs="1" maxOccurs="1"/>
     <element name="name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <element name="fileName" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <element name="lineInCode" type="xsd:int" minOccurs="1" maxOccurs="1"/>
     <element name="posInLine" type="xsd:int" minOccurs="1" maxOccurs="1"/>
   </sequence>
  </complexType>
  <complexType name="ActionId">
   <sequence>
     <element name="actionId" type="xsd:int" minOccurs="1" maxOccurs="1"/>
   </sequence>
  </complexType>
  <complexType name="ArrayOfActionIds">
   <sequence>
     <element name="ActionIds" type="WS-A:ActionId" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
  </complexType>
  <complexType name="ActionInfo">
   <sequence>
     <element name="actionId" type="WS-A:ActionId" minOccurs="1" maxOccurs="1"/>
     <element name="action" type="xsd:string" minOccurs="1" maxOccurs="1"/>
```

```
86    </sequence>
87   </complexType>
88   <complexType name="ProtocolId">
89    <sequence>
90      <element name="protocolId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
91    </sequence>
92   </complexType>
93   <complexType name="ComponentId">
94    <sequence>
95      <element name="componentId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
96    </sequence>
97   </complexType>
98   <complexType name="AbstractionTransition">
99    <sequence>
100     <element name="op1" type="WS-A:NodeId" minOccurs="1"
          maxOccurs="1"/>
101     <element name="op2" type="WS-A:NodeId" minOccurs="1"
          maxOccurs="1"/>
102     <element name="op3" type="WS-A:NodeId" minOccurs="1"
          maxOccurs="1"/>
103     <element name="operation" type="WS-A:Operation" minOccurs="1"
           maxOccurs="1"/>
104     <element name="typ" type="WS-A:TransitionTyp" minOccurs="1"
          maxOccurs="1"/>
105     <element name="action" type="WS-A:ActionId" minOccurs="1"
          maxOccurs="1"/>
106   </sequence>
107  </complexType>
108  <complexType name="AbstractionAutomaton">
109   <sequence>
110     <element name="compId" type="WS-A:ComponentId" minOccurs="1"
          maxOccurs="1"/>
111     <element name="startId" type="WS-A:NodeId" minOccurs="1"
          maxOccurs="1"/>
112     <element name="Transitions" type="WS-A:AbstractionTransition"
           minOccurs="0" maxOccurs="unbounded"/>
113   </sequence>
114  </complexType>
115  <complexType name="Component">
116   <sequence>
117     <element name="compId" type="WS-A:ComponentId" minOccurs="1"
          maxOccurs="1"/>
118     <element name="protocolId" type="WS-A:ProtocolId" minOccurs="
          1" maxOccurs="1"/>
119     <element name="name" type="xsd:string" minOccurs="1"
          maxOccurs="1"/>
120   </sequence>
121  </complexType>
122  <complexType name="ComponentResponse">
123   <sequence>
124     <element name="compId" type="WS-A:ComponentId" minOccurs="1"
          maxOccurs="1"/>
125     <element name="protocolId" type="WS-A:ProtocolId" minOccurs="
          1" maxOccurs="1"/>
126     <element name="name" type="xsd:string" minOccurs="1"
          maxOccurs="1"/>
127     <element name="rev" type="xsd:string" minOccurs="1" maxOccurs
          ="1"/>
128     <element name="more" type="xsd:string" minOccurs="1"
          maxOccurs="1"/>
129   </sequence>
130  </complexType>
131  <!-- operation request element -->
132  <element name="getAbstraction">
133   <complexType>
134    <sequence>
135     <element name="IdOfComponent" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
136     <element name="Analyses" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
137    </sequence>
138   </complexType>
139  </element>
140  <!-- operation response element -->
141  <element name="AbstractionAutomatonResponse">
142   <complexType>
143    <sequence>
144     <element name="compId" type="WS-A:ComponentId" minOccurs="1"
          maxOccurs="1"/>
145     <element name="startId" type="WS-A:NodeId" minOccurs="1"
          maxOccurs="1"/>
146     <element name="Transitions" type="WS-A:AbstractionTransition"
           minOccurs="0" maxOccurs="unbounded"/>
147    </sequence>
148   </complexType>
149  </element>
150  <!-- operation request element -->
151  <element name="getAbstractions">
152   <complexType>
153    <sequence>
154     <element name="IdsOfComponents" type="WS-A:IntVector"
          minOccurs="1" maxOccurs="1"/>
155     <element name="Analyses" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
156    </sequence>
157   </complexType>
158  </element>
159  <!-- operation response element -->
160  <element name="ArrayOfAbstractions">
161   <complexType>
```

```
162      <sequence>
163       <element name="AbsArray" type="WS-A:AbstractionAutomaton"
              minOccurs="0" maxOccurs="unbounded"/>
164      </sequence>
165     </complexType>
166    </element>
167    <!-- operation request element -->
168    <element name="getActionInfo">
169     <complexType>
170      <sequence>
171       <element name="ActionIds" type="WS-A:ArrayOfActionIds"
              minOccurs="1" maxOccurs="1"/>
172      </sequence>
173     </complexType>
174    </element>
175    <!-- operation response element -->
176    <element name="ArrayOfActionInfos">
177     <complexType>
178      <sequence>
179       <element name="Infos" type="WS-A:ActionInfo" minOccurs="0"
              maxOccurs="unbounded"/>
180      </sequence>
181     </complexType>
182    </element>
183    <!-- operation request element -->
184    <element name="getCombinedAbstraction">
185     <complexType>
186      <sequence>
187       <element name="CompSystem" type="WS-A:IntVector" minOccurs="1
              " maxOccurs="1"/>
188       <element name="Analyses" type="xsd:int" minOccurs="1"
              maxOccurs="1"/>
189       <element name="IdOfComponentsProtocol" type="xsd:int"
              minOccurs="1" maxOccurs="1"/>
190      </sequence>
191     </complexType>
192    </element>
193    <!-- operation request element -->
194    <element name="getCombinedAbstractionWithProtocol">
195     <complexType>
196      <sequence>
197       <element name="CompSystem" type="WS-A:IntVector" minOccurs="1
              " maxOccurs="1"/>
198       <element name="Analyses" type="xsd:int" minOccurs="1"
              maxOccurs="1"/>
199       <element name="protocol" type="WS-A:AbstractionAutomaton"
              minOccurs="1" maxOccurs="1"/>
200      </sequence>
201     </complexType>
202    </element>
203    <!-- operation request element -->
```

```
204    <element name="getComponentImplementationInformation">
205     <complexType>
206      <sequence>
207       <element name="IdOfComponent" type="xsd:int" minOccurs="1"
              maxOccurs="1"/>
208      </sequence>
209     </complexType>
210    </element>
211    <!-- operation response element -->
212    <element name="ComponentInfo">
213     <complexType>
214      <sequence>
215       <element name="compId" type="WS-A:ComponentId" minOccurs="1"
              maxOccurs="1"/>
216       <element name="name" type="xsd:string" minOccurs="1"
              maxOccurs="1"/>
217       <element name="information" type="xsd:string" minOccurs="1"
              maxOccurs="1"/>
218       <element name="parentComponentId" type="xsd:int" minOccurs="1
              " maxOccurs="1"/>
219       <element name="childrenComponentIds" type="WS-A:IntVector"
              minOccurs="1" maxOccurs="1"/>
220      </sequence>
221     </complexType>
222    </element>
223    <!-- operation request element -->
224    <element name="getListOfComponents">
225     <complexType>
226      <sequence>
227      </sequence>
228     </complexType>
229    </element>
230    <!-- operation response element -->
231    <element name="ArrayOfComponents">
232     <complexType>
233      <sequence>
234       <element name="Components" type="WS-A:ComponentResponse"
              minOccurs="0" maxOccurs="unbounded"/>
235      </sequence>
236     </complexType>
237    </element>
238    <!-- operation request element -->
239    <element name="getNodeInfo">
240     <complexType>
241      <sequence>
242       <element name="NodeIds" type="WS-A:ArrayOfNodeIds" minOccurs=
              "1" maxOccurs="1"/>
243      </sequence>
244     </complexType>
245    </element>
246    <!-- operation response element -->
```

```
247   <element name="ArrayOfNodeInfos">
248    <complexType>
249     <sequence>
250      <element name="Infos" type="WS-A:NodeInfo" minOccurs="0"
           maxOccurs="unbounded"/>
251     </sequence>
252    </complexType>
253   </element>
254   <!-- operation request element -->
255   <element name="getProtocol">
256    <complexType>
257     <sequence>
258      <element name="IdOfProtocol" type="xsd:int" minOccurs="1"
           maxOccurs="1"/>
259     </sequence>
260    </complexType>
261   </element>
262   <!-- operation request element -->
263   <element name="getProtocolInfo">
264    <complexType>
265     <sequence>
266      <element name="IdOfProtocol" type="xsd:int" minOccurs="1"
           maxOccurs="1"/>
267     </sequence>
268    </complexType>
269   </element>
270   <!-- operation response element -->
271   <element name="ProtocolInfo">
272    <complexType>
273     <sequence>
274      <element name="protocolId" type="WS-A:ProtocolId" minOccurs="
           1" maxOccurs="1"/>
275      <element name="regex" type="xsd:string" minOccurs="1"
           maxOccurs="1"/>
276      <element name="location" type="WS-A:ComponentId" minOccurs="1
           " maxOccurs="1"/>
277     </sequence>
278    </complexType>
279   </element>
280  </schema>
281
282 </types>
283
284 <message name="getAbstraction">
285  <part name="parameters" element="WS-A:getAbstraction"/>
286 </message>
287
288 <message name="AbstractionAutomatonResponse">
289  <part name="parameters" element="WS-
       A:AbstractionAutomatonResponse"/>
290 </message>
291
292 <message name="getAbstractions">
293  <part name="parameters" element="WS-A:getAbstractions"/>
294 </message>
295
296 <message name="ArrayOfAbstractions">
297  <part name="parameters" element="WS-A:ArrayOfAbstractions"/>
298 </message>
299
300 <message name="getActionInfo">
301  <part name="parameters" element="WS-A:getActionInfo"/>
302 </message>
303
304 <message name="ArrayOfActionInfos">
305  <part name="parameters" element="WS-A:ArrayOfActionInfos"/>
306 </message>
307
308 <message name="getCombinedAbstraction">
309  <part name="parameters" element="WS-A:getCombinedAbstraction"/>
310 </message>
311
312 <message name="getCombinedAbstractionWithProtocol">
313  <part name="parameters" element="WS-
       A:getCombinedAbstractionWithProtocol"/>
314 </message>
315
316 <message name="getComponentImplementationInformation">
317  <part name="parameters" element="WS-
       A:getComponentImplementationInformation"/>
318 </message>
319
320 <message name="ComponentInfo">
321  <part name="parameters" element="WS-A:ComponentInfo"/>
322 </message>
323
324 <message name="getListOfComponents">
325  <part name="parameters" element="WS-A:getListOfComponents"/>
326 </message>
327
328 <message name="ArrayOfComponents">
329  <part name="parameters" element="WS-A:ArrayOfComponents"/>
330 </message>
331
332 <message name="getNodeInfo">
333  <part name="parameters" element="WS-A:getNodeInfo"/>
334 </message>
335
336 <message name="ArrayOfNodeInfos">
337  <part name="parameters" element="WS-A:ArrayOfNodeInfos"/>
338 </message>
339
```

```
340 <message name="getProtocol">
341  <part name="parameters" element="WS-A:getProtocol"/>
342 </message>
343
344 <message name="getProtocolInfo">
345  <part name="parameters" element="WS-A:getProtocolInfo"/>
346 </message>
347
348 <message name="ProtocolInfo">
349  <part name="parameters" element="WS-A:ProtocolInfo"/>
350 </message>
351
352 <portType name="WSAPortType">
353  <operation name="getAbstraction">
354   <documentation>Liefert eine Abstraktion von einer Komponente</
        documentation>
355   <input message="tns:getAbstraction"/>
356   <output message="tns:AbstractionAutomatonResponse"/>
357  </operation>
358  <operation name="getAbstractions">
359   <documentation>Liefert aufgeloeste Abstraktionen der
        uebergebenen Komponenten</documentation>
360   <input message="tns:getAbstractions"/>
361   <output message="tns:ArrayOfAbstractions"/>
362  </operation>
363  <operation name="getActionInfo">
364   <documentation>loest Action-Ids auf</documentation>
365   <input message="tns:getActionInfo"/>
366   <output message="tns:ArrayOfActionInfos"/>
367  </operation>
368  <operation name="getCombinedAbstraction">
369   <documentation>Liefert Combined Abstraction eines Systems von
        Komponenten in Bezug auf eine spezifiziertes Protokoll</
        documentation>
370   <input message="tns:getCombinedAbstraction"/>
371   <output message="tns:AbstractionAutomatonResponse"/>
372  </operation>
373  <operation name="getCombinedAbstractionWithProtocol">
374   <documentation>Service definition of function
        WS_A__getCombinedAbstractionWithProtocol</documentation>
375   <input message="tns:getCombinedAbstractionWithProtocol"/>
376   <output message="tns:AbstractionAutomatonResponse"/>
377  </operation>
378  <operation name="getComponentImplementationInformation">
379   <documentation>Liefert Informationen ueber eine Komponente, z. B
        . welche Programmiersprache verwendet wurde, welche
        Interfaces diese Komponente besitzt, ... (dynamisch) </
        documentation>
380   <input message="tns:getComponentImplementationInformation"/>
381   <output message="tns:ComponentInfo"/>
382  </operation>
383  <operation name="getListOfComponents">
384   <documentation>Service definition of function
        WS_A__getListOfComponents</documentation>
385   <input message="tns:getListOfComponents"/>
386   <output message="tns:ArrayOfComponents"/>
387  </operation>
388  <operation name="getNodeInfo">
389   <documentation>loest Node-Ids auf</documentation>
390   <input message="tns:getNodeInfo"/>
391   <output message="tns:ArrayOfNodeInfos"/>
392  </operation>
393  <operation name="getProtocol">
394   <documentation>Service definition of function WS_A__getProtocol<
        /documentation>
395   <input message="tns:getProtocol"/>
396   <output message="tns:AbstractionAutomatonResponse"/>
397  </operation>
398  <operation name="getProtocolInfo">
399   <documentation>loest Node-Ids auf</documentation>
400   <input message="tns:getProtocolInfo"/>
401   <output message="tns:ProtocolInfo"/>
402  </operation>
403 </portType>
404
405 <binding name="WSO" type="tns:WSOPortType">
406  <SOAP:binding style="document" transport="http://schemas.xmlsoap.
       org/soap/http"/>
407  <operation name="getAbstraction">
408   <SOAP:operation soapAction=""/>
409   <input>
410      <SOAP:body parts="parameters" use="literal"/>
411   </input>
412   <output>
413      <SOAP:body parts="parameters" use="literal"/>
414   </output>
415  </operation>
416  <operation name="getAbstractions">
417   <SOAP:operation soapAction=""/>
418   <input>
419      <SOAP:body parts="parameters" use="literal"/>
420   </input>
421   <output>
422      <SOAP:body parts="parameters" use="literal"/>
423   </output>
424  </operation>
425  <operation name="getActionInfo">
426   <SOAP:operation soapAction=""/>
427   <input>
428      <SOAP:body parts="parameters" use="literal"/>
429   </input>
430   <output>
```

```
431        <SOAP:body parts="parameters" use="literal"/>
432    </output>
433   </operation>
434   <operation name="getCombinedAbstraction">
435    <SOAP:operation soapAction=""/>
436    <input>
437        <SOAP:body parts="parameters" use="literal"/>
438    </input>
439    <output>
440        <SOAP:body parts="parameters" use="literal"/>
441    </output>
442   </operation>
443   <operation name="getCombinedAbstractionWithProtocol">
444    <SOAP:operation soapAction=""/>
445    <input>
446        <SOAP:body parts="parameters" use="literal"/>
447    </input>
448    <output>
449        <SOAP:body parts="parameters" use="literal"/>
450    </output>
451   </operation>
452   <operation name="getComponentImplementationInformation">
453    <SOAP:operation soapAction=""/>
454    <input>
455        <SOAP:body parts="parameters" use="literal"/>
456    </input>
457    <output>
458        <SOAP:body parts="parameters" use="literal"/>
459    </output>
460   </operation>
461   <operation name="getListOfComponents">
462    <SOAP:operation soapAction=""/>
463    <input>
464        <SOAP:body parts="parameters" use="literal"/>
465    </input>
466    <output>
467        <SOAP:body parts="parameters" use="literal"/>
468    </output>
469   </operation>
470   <operation name="getNodeInfo">
471    <SOAP:operation soapAction=""/>
472    <input>
473        <SOAP:body parts="parameters" use="literal"/>
474    </input>
475    <output>
476        <SOAP:body parts="parameters" use="literal"/>
477    </output>
478   </operation>
479   <operation name="getProtocol">
480    <SOAP:operation soapAction=""/>
481    <input>
482        <SOAP:body parts="parameters" use="literal"/>
483    </input>
484    <output>
485        <SOAP:body parts="parameters" use="literal"/>
486    </output>
487   </operation>
488   <operation name="getProtocolInfo">
489    <SOAP:operation soapAction=""/>
490    <input>
491        <SOAP:body parts="parameters" use="literal"/>
492    </input>
493    <output>
494        <SOAP:body parts="parameters" use="literal"/>
495    </output>
496   </operation>
497  </binding>
498
499  <service name="WSO">
500   <documentation>WSA and WSO service definition</documentation>
501   <port name="WSO" binding="tns:WSO">
502  <SOAP:address location="http://127.0.0.1:8080/wso"/>
503    </port>
504  </service>
505
506  </definitions>
```

Listing A.2: WSDL Definition of WSB

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <definitions name="WSB"
3   targetNamespace="http://localhost:80/WSB.wsdl"
4   xmlns:tns="http://localhost:80/WSB.wsdl"
5   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
6   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
9   xmlns:WS-B="http://tempuri.org/WS-B.xsd"
10  xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
11  xmlns:MIME="http://schemas.xmlsoap.org/wsdl/mime/"
12  xmlns:DIME="http://schemas.xmlsoap.org/ws/2002/04/dime/wsdl/"
13  xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
14  xmlns="http://schemas.xmlsoap.org/wsdl/">
15
16 <types>
17
18  <schema targetNamespace="http://tempuri.org/WS-B.xsd"
19   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
20   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
21   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
22   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
23   xmlns:WS-B="http://tempuri.org/WS-B.xsd"
```

```
24   xmlns="http://www.w3.org/2001/XMLSchema"
25   elementFormDefault="unqualified"
26   attributeFormDefault="unqualified">
27  <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
28  <simpleType name="Operation">
29   <restriction base="xsd:string">
30    <enumeration value="no-op"/><!-- enum const = 0 -->
31    <enumeration value="sequ-op"/><!-- enum const = 1 -->
32    <enumeration value="sync-op"/><!-- enum const = 2 -->
33    <enumeration value="fork-op"/><!-- enum const = 3 -->
34    <enumeration value="eps-op"/><!-- enum const = 4 -->
35   </restriction>
36  </simpleType>
37  <simpleType name="TransitionTyp">
38   <restriction base="xsd:string">
39    <enumeration value="no-typ"/><!-- enum const = 0 -->
40    <enumeration value="active"/><!-- enum const = 1 -->
41    <enumeration value="inactive"/><!-- enum const = 2 -->
42    <enumeration value="lambda"/><!-- enum const = 3 -->
43   </restriction>
44  </simpleType>
45  <simpleType name="Status">
46   <restriction base="xsd:string">
47    <enumeration value="not-found"/><!-- enum const = 0 -->
48    <enumeration value="queued"/><!-- enum const = 1 -->
49    <enumeration value="working"/><!-- enum const = 2 -->
50    <enumeration value="finished"/><!-- enum const = 3 -->
51    <enumeration value="canceled"/><!-- enum const = 4 -->
52    <enumeration value="error"/><!-- enum const = 5 -->
53   </restriction>
54  </simpleType>
55  <complexType name="IntVector">
56   <sequence>
57     <element name="item" type="xsd:int" minOccurs="0" maxOccurs="
          unbounded"/>
58   </sequence>
59  </complexType>
60  <complexType name="NodeId">
61   <sequence>
62     <element name="componentId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
63     <element name="componentNodeId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
64     <element name="pre" type="xsd:int" minOccurs="1" maxOccurs="1
          "/>
65     <element name="post" type="xsd:int" minOccurs="1" maxOccurs="
          1"/>
66   </sequence>
67  </complexType>
68  <complexType name="ActionId">
69   <sequence>
70     <element name="actionId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
71   </sequence>
72  </complexType>
73  <complexType name="ProtocolId">
74   <sequence>
75     <element name="protocolId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
76   </sequence>
77  </complexType>
78  <complexType name="ComponentId">
79   <sequence>
80     <element name="componentId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
81   </sequence>
82  </complexType>
83  <complexType name="AbstractionTransition">
84   <sequence>
85     <element name="op1" type="WS-B:NodeId" minOccurs="1"
          maxOccurs="1"/>
86     <element name="op2" type="WS-B:NodeId" minOccurs="1"
          maxOccurs="1"/>
87     <element name="op3" type="WS-B:NodeId" minOccurs="1"
          maxOccurs="1"/>
88     <element name="operation" type="WS-B:Operation" minOccurs="1"
           maxOccurs="1"/>
89     <element name="typ" type="WS-B:TransitionTyp" minOccurs="1"
          maxOccurs="1"/>
90     <element name="action" type="WS-B:ActionId" minOccurs="1"
          maxOccurs="1"/>
91   </sequence>
92  </complexType>
93  <complexType name="AbstractionAutomaton">
94   <sequence>
95     <element name="compId" type="WS-B:ComponentId" minOccurs="1"
          maxOccurs="1"/>
96     <element name="startId" type="WS-B:NodeId" minOccurs="1"
          maxOccurs="1"/>
97     <element name="Transitions" type="WS-B:AbstractionTransition"
           minOccurs="0" maxOccurs="unbounded"/>
98   </sequence>
99  </complexType>
100 <complexType name="HtAcces">
101  <sequence>
102    <element name="usr" type="xsd:string" minOccurs="1" maxOccurs
          ="1"/>
103    <element name="pw" type="xsd:string" minOccurs="1" maxOccurs=
          "1"/>
104  </sequence>
105 </complexType>
106 <!-- operation request element -->
```

```
107 <element name="cancelTask">
108  <complexType>
109   <sequence>
110    <element name="IdOfTask" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
111   </sequence>
112  </complexType>
113 </element>
114 <!-- operation response element -->
115 <element name="cancelTaskResponse">
116  <complexType>
117   <sequence>
118    <element name="status" type="xsd:int" minOccurs="1" maxOccurs
         ="1"/>
119   </sequence>
120  </complexType>
121 </element>
122 <!-- operation request element -->
123 <element name="startSearch">
124  <complexType>
125   <sequence>
126    <element name="IdsOfComponents" type="WS-B:IntVector"
         minOccurs="1" maxOccurs="1"/>
127    <element name="Analyses" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
128    <element name="IdOfProtocol" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
129    <element name="HtAccess" type="WS-B:HtAcces" minOccurs="1"
         maxOccurs="1"/>
130    <element name="IdOfURLs" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
131   </sequence>
132  </complexType>
133 </element>
134 <!-- operation response element -->
135 <element name="startSearchResponse">
136  <complexType>
137   <sequence>
138    <element name="IdOfTask" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
139   </sequence>
140  </complexType>
141 </element>
142 <!-- operation request element -->
143 <element name="getStatus">
144  <complexType>
145   <sequence>
146    <element name="IdOfTask" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
147   </sequence>
148  </complexType>
149 </element>
150 <!-- operation response element -->
151 <element name="getStatusResponse">
152  <complexType>
153   <sequence>
154    <element name="Status" type="WS-B:Status" minOccurs="1"
         maxOccurs="1"/>
155   </sequence>
156  </complexType>
157 </element>
158 <!-- operation request element -->
159 <element name="startSearchAbs">
160  <complexType>
161   <sequence>
162    <element name="CombAbs" type="WS-B:AbstractionAutomaton"
         minOccurs="1" maxOccurs="1"/>
163    <element name="Protocol" type="WS-B:AbstractionAutomaton"
         minOccurs="1" maxOccurs="1"/>
164    <element name="HtAccess" type="WS-B:HtAcces" minOccurs="1"
         maxOccurs="1"/>
165    <element name="IdOfURLs" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
166   </sequence>
167  </complexType>
168 </element>
169 <!-- operation response element -->
170 <element name="startSearchAbsResponse">
171  <complexType>
172   <sequence>
173    <element name="IdOfTask" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
174   </sequence>
175  </complexType>
176 </element>
177 <!-- operation request element -->
178 <element name="registerURL">
179  <complexType>
180   <sequence>
181    <element name="URLA" type="xsd:string" minOccurs="1"
         maxOccurs="1"/>
182    <element name="URLC" type="xsd:string" minOccurs="1"
         maxOccurs="1"/>
183   </sequence>
184  </complexType>
185 </element>
186 <!-- operation response element -->
187 <element name="registerURLResponse">
188  <complexType>
189   <sequence>
190    <element name="IdOfURLs" type="xsd:int" minOccurs="1"
         maxOccurs="1"/>
```

```
191     </sequence>
192    </complexType>
193   </element>
194  </schema>
195
196 </types>
197
198 <message name="cancelTaskRequest">
199  <part name="parameters" element="WS-B:cancelTask"/>
200 </message>
201
202 <message name="cancelTaskResponse">
203  <part name="parameters" element="WS-B:cancelTaskResponse"/>
204 </message>
205
206 <message name="startSearchRequest">
207  <part name="parameters" element="WS-B:startSearch"/>
208 </message>
209
210 <message name="startSearchResponse">
211  <part name="parameters" element="WS-B:startSearchResponse"/>
212 </message>
213
214 <message name="getStatusRequest">
215  <part name="parameters" element="WS-B:getStatus"/>
216 </message>
217
218 <message name="getStatusResponse">
219  <part name="parameters" element="WS-B:getStatusResponse"/>
220 </message>
221
222 <message name="startSearchAbsRequest">
223  <part name="parameters" element="WS-B:startSearchAbs"/>
224 </message>
225
226 <message name="startSearchAbsResponse">
227  <part name="parameters" element="WS-B:startSearchAbsResponse"/>
228 </message>
229
230 <message name="registerURLRequest">
231  <part name="parameters" element="WS-B:registerURL"/>
232 </message>
233
234 <message name="registerURLResponse">
235  <part name="parameters" element="WS-B:registerURLResponse"/>
236 </message>
237
238 <portType name="WSBPortType">
239  <operation name="cancelTask">
240   <documentation>Service definition of function WS_B__cancelTask</
        documentation>
241   <input message="tns:cancelTaskRequest"/>
242   <output message="tns:cancelTaskResponse"/>
243  </operation>
244  <operation name="startSearch">
245   <documentation>erhaelt Ids eines Komponentensystems und eines
        Protokolls: Die Komponenten sollen gegen das Protokoll
        geprueft werden.</documentation>
246   <input message="tns:startSearchRequest"/>
247   <output message="tns:startSearchResponse"/>
248  </operation>
249  <operation name="getStatus">
250   <documentation>Service definition of function WS_B__getStatus</
        documentation>
251   <input message="tns:getStatusRequest"/>
252   <output message="tns:getStatusResponse"/>
253  </operation>
254  <operation name="startSearchAbs">
255   <documentation>erhaelt eine CombinedAbstraction und ein
        Protokoll, gegen das die Abstraktion geprueft werden soll</
        documentation>
256   <input message="tns:startSearchAbsRequest"/>
257   <output message="tns:startSearchAbsResponse"/>
258  </operation>
259  <operation name="registerURL">
260   <documentation>Service definition of function WS_B__registerURL<
        /documentation>
261   <input message="tns:registerURLRequest"/>
262   <output message="tns:registerURLResponse"/>
263  </operation>
264 </portType>
265
266 <binding name="WSB" type="tns:WSBPortType">
267  <SOAP:binding style="document" transport="http://schemas.xmlsoap.
       org/soap/http"/>
268  <operation name="cancelTask">
269   <SOAP:operation soapAction=""/>
270   <input>
271      <SOAP:body parts="parameters" use="literal"/>
272   </input>
273   <output>
274      <SOAP:body parts="parameters" use="literal"/>
275   </output>
276  </operation>
277  <operation name="startSearch">
278   <SOAP:operation soapAction=""/>
279   <input>
280      <SOAP:body parts="parameters" use="literal"/>
281   </input>
282   <output>
283      <SOAP:body parts="parameters" use="literal"/>
284   </output>
```

```
285  </operation>
286  <operation name="getStatus">
287   <SOAP:operation soapAction=""/>
288   <input>
289      <SOAP:body parts="parameters" use="literal"/>
290   </input>
291   <output>
292      <SOAP:body parts="parameters" use="literal"/>
293   </output>
294  </operation>
295  <operation name="startSearchAbs">
296   <SOAP:operation soapAction=""/>
297   <input>
298      <SOAP:body parts="parameters" use="literal"/>
299   </input>
300   <output>
301      <SOAP:body parts="parameters" use="literal"/>
302   </output>
303  </operation>
304  <operation name="registerURL">
305   <SOAP:operation soapAction=""/>
306   <input>
307      <SOAP:body parts="parameters" use="literal"/>
308   </input>
309   <output>
310      <SOAP:body parts="parameters" use="literal"/>
311   </output>
312  </operation>
313 </binding>
314
315 <service name="WSB">
316  <documentation>gSOAP 2.7.10 generated service definition</
         documentation>
317  <port name="WSB" binding="tns:WSB">
318   <SOAP:address location="http://localhost:80"/>
319  </port>
320 </service>
321
322 </definitions>
```

Listing A.3: WSDL Definition of WSC

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions name="WSC"
3  targetNamespace="http://localhost:80/WSC.wsdl"
4  xmlns:tns="http://localhost:80/WSC.wsdl"
5  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
6  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
7  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
9  xmlns:WS-C="http://tempuri.org/WS-C.xsd"
10  xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
11  xmlns:MIME="http://schemas.xmlsoap.org/wsdl/mime/"
12  xmlns:DIME="http://schemas.xmlsoap.org/ws/2002/04/dime/wsdl/"
13  xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
14  xmlns="http://schemas.xmlsoap.org/wsdl/">
15
16 <types>
17
18  <schema targetNamespace="http://tempuri.org/WS-C.xsd"
19   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
20   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
21   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
22   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
23   xmlns:WS-C="http://tempuri.org/WS-C.xsd"
24   xmlns="http://www.w3.org/2001/XMLSchema"
25   elementFormDefault="unqualified"
26   attributeFormDefault="unqualified">
27   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
28   <simpleType name="Operation">
29    <restriction base="xsd:string">
30     <enumeration value="no-op"/><!-- enum const = 0 -->
31     <enumeration value="sequ-op"/><!-- enum const = 1 -->
32     <enumeration value="sync-op"/><!-- enum const = 2 -->
33     <enumeration value="fork-op"/><!-- enum const = 3 -->
34     <enumeration value="eps-op"/><!-- enum const = 4 -->
35    </restriction>
36   </simpleType>
37   <simpleType name="TransitionTyp">
38    <restriction base="xsd:string">
39     <enumeration value="no-typ"/><!-- enum const = 0 -->
40     <enumeration value="active"/><!-- enum const = 1 -->
41     <enumeration value="inactive"/><!-- enum const = 2 -->
42     <enumeration value="lambda"/><!-- enum const = 3 -->
43    </restriction>
44   </simpleType>
45   <complexType name="HtAcces">
46    <sequence>
47      <element name="usr" type="xsd:string" minOccurs="1" maxOccurs
          ="1"/>
48      <element name="pw" type="xsd:string" minOccurs="1" maxOccurs=
          "1"/>
49    </sequence>
50   </complexType>
51   <complexType name="NodeId">
52    <sequence>
53      <element name="componentId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
54      <element name="componentNodeId" type="xsd:int" minOccurs="1"
          maxOccurs="1"/>
55      <element name="pre" type="xsd:int" minOccurs="1" maxOccurs="1
          "/>
```

```
56      <element name="post" type="xsd:int" minOccurs="1" maxOccurs="1"/>
57    </sequence>
58   </complexType>
59   <complexType name="ActionId">
60    <sequence>
61      <element name="actionId" type="xsd:int" minOccurs="1" maxOccurs="1"/>
62    </sequence>
63   </complexType>
64   <complexType name="ComponentId">
65    <sequence>
66      <element name="componentId" type="xsd:int" minOccurs="1" maxOccurs="1"/>
67    </sequence>
68   </complexType>
69   <complexType name="AbstractionTransition">
70    <sequence>
71      <element name="op1" type="WS-C:NodeId" minOccurs="1" maxOccurs="1"/>
72      <element name="op2" type="WS-C:NodeId" minOccurs="1" maxOccurs="1"/>
73      <element name="op3" type="WS-C:NodeId" minOccurs="1" maxOccurs="1"/>
74      <element name="operation" type="WS-C:Operation" minOccurs="1" maxOccurs="1"/>
75      <element name="typ" type="WS-C:TransitionTyp" minOccurs="1" maxOccurs="1"/>
76      <element name="action" type="WS-C:ActionId" minOccurs="1" maxOccurs="1"/>
77    </sequence>
78   </complexType>
79   <complexType name="AbstractionAutomaton">
80    <sequence>
81      <element name="compId" type="WS-C:ComponentId" minOccurs="1" maxOccurs="1"/>
82      <element name="startId" type="WS-C:NodeId" minOccurs="1" maxOccurs="1"/>
83      <element name="Transitions" type="WS-C:AbstractionTransition" minOccurs="0" maxOccurs="unbounded"/>
84    </sequence>
85   </complexType>
86   <!-- operation request element -->
87   <element name="appendCounterExample">
88    <complexType>
89     <sequence>
90      <element name="IdOfTask" type="xsd:int" minOccurs="1" maxOccurs="1"/>
91      <element name="Abstraction" type="WS-C:AbstractionAutomaton" minOccurs="1" maxOccurs="1"/>
92      <element name="CounterExample" type="xsd:string" minOccurs="1" maxOccurs="1"/>
93     </sequence>
94    </complexType>
95   </element>
96   <!-- operation response element -->
97   <element name="appendCounterExampleResponse">
98    <complexType>
99     <sequence>
100     <element name="status" type="xsd:int" minOccurs="1" maxOccurs="1"/>
101    </sequence>
102   </complexType>
103  </element>
104  <!-- operation request element -->
105  <element name="calculationFinished">
106   <complexType>
107    <sequence>
108     <element name="IdOfTask" type="xsd:int" minOccurs="1" maxOccurs="1"/>
109    </sequence>
110   </complexType>
111  </element>
112  <!-- operation response element -->
113  <element name="calculationFinishedResponse">
114   <complexType>
115    <sequence>
116     <element name="status" type="xsd:int" minOccurs="1" maxOccurs="1"/>
117    </sequence>
118   </complexType>
119  </element>
120  <!-- operation request element -->
121  <element name="startGuiLike">
122   <complexType>
123    <sequence>
124     <element name="HtAccess" type="WS-C:HtAcces" minOccurs="1" maxOccurs="1"/>
125    </sequence>
126   </complexType>
127  </element>
128 </schema>
129
130 </types>
131
132 <message name="appendCounterExampleRequest">
133  <part name="parameters" element="WS-C:appendCounterExample"/>
134 </message>
135
136 <message name="appendCounterExampleResponse">
137  <part name="parameters" element="WS-C:appendCounterExampleResponse"/>
```

```
138 </message>
139
140 <message name="calculationFinishedRequest">
141  <part name="parameters" element="WS-C:calculationFinished"/>
142 </message>
143
144 <message name="calculationFinishedResponse">
145  <part name="parameters" element="WS-C:calculationFinishedResponse
        "/>
146 </message>
147
148 <message name="startGuiLike">
149  <part name="parameters" element="WS-C:startGuiLike"/>
150 </message>
151
152 <portType name="WSCPortType">
153  <operation name="appendCounterExample">
154   <documentation>ein Gegenbeispiel zu einer spezifischen
        Berechnung wird uebergeben (async)</documentation>
155   <input message="tns:appendCounterExampleRequest"/>
156   <output message="tns:appendCounterExampleResponse"/>
157  </operation>
158  <operation name="calculationFinished">
159   <documentation>Markiert das Ende einer spezifischen Berechnung</
        documentation>
160   <input message="tns:calculationFinishedRequest"/>
161   <output message="tns:calculationFinishedResponse"/>
162  </operation>
163  <operation name="startGuiLike">
164   <documentation>Es soll ein Ansprechen des WS C simuliert werden,
         als sei es die Projektgruppe</documentation>
165   <input message="tns:startGuiLike"/>
166  </operation>
167 </portType>
168
169 <binding name="WSC" type="tns:WSCPortType">
170  <SOAP:binding style="document" transport="http://schemas.xmlsoap.
        org/soap/http"/>
171  <operation name="appendCounterExample">
172   <SOAP:operation soapAction=""/>
173   <input>
174      <SOAP:body parts="parameters" use="literal"/>
175   </input>
176   <output>
177      <SOAP:body parts="parameters" use="literal"/>
178   </output>
179  </operation>
180  <operation name="calculationFinished">
181   <SOAP:operation soapAction=""/>
182   <input>
183      <SOAP:body parts="parameters" use="literal"/>
184   </input>
185   <output>
186      <SOAP:body parts="parameters" use="literal"/>
187   </output>
188  </operation>
189  <operation name="startGuiLike">
190   <SOAP:operation soapAction=""/>
191   <input>
192      <SOAP:body parts="parameters" use="literal"/>
193   </input>
194  </operation>
195 </binding>
196
197 <service name="WSC">
198  <documentation>gSOAP 2.7.10 generated service definition</
        documentation>
199  <port name="WSC" binding="tns:WSC">
200   <SOAP:address location="http://localhost:80"/>
201  </port>
202 </service>
203
204 </definitions>
```

A.4 Extended Consideration of Combined Abstraction

Theorem (Correctness of construction of the Combined Abstraction Π_S^C)
The construction of a Combined Abstraction Π_S^C results in a representation, so that $L(\mathcal{P}_A) \cap L(\Pi_{S,C}) \subseteq L(\Pi_S^C)$.

Proof
Let be given the system abstraction $\Pi_{S,C} = (Q_{\Pi_{S,C}}, \Sigma_{\Pi_{S,C}}, I_{\Pi_{S,C}}, \rightarrow_{\Pi_{S,C}}, F_{\Pi_{S,C}})$ and the inverted protocol $\overline{\mathcal{P}_C} = (Q_{\overline{\mathcal{P}_C}}, \Sigma_{\overline{\mathcal{P}_C}}, I_{\overline{\mathcal{P}_C}}, \rightarrow_{\overline{\mathcal{P}_C}}, F_{\overline{\mathcal{P}_C}})$.

We perform an induction over the length n of the counterexample w. It is assumed that $I_{\overline{\mathcal{P}_A}} \overset{w}{\Rightarrow}_{\overline{\mathcal{P}_A}} f$ is constructed using the transition rules of $\Pi_{S,C}$, where $f \in F_{\overline{\mathcal{P}_A}}$, while $I_{\Pi_{S,C}} \overset{w}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ is constructed using the transition rules of the system abstraction $\Pi_{S,C}$. Thus, it should be valid: $w \in L(\Pi_S^C)$. Without loss of generality, we assume that precisely one counterexample w is contained in the system abstraction $\Pi_{S,C}$.

Induction over the length n.

$|w| = 1$*: There has to exist a transition rule $I_{\overline{\mathcal{P}_A}} \overset{a}{\rightarrow} f$, where $f \in F_{\overline{\mathcal{P}_A}}$ and $w = a$ describing the forbidden interaction sequence. We have to consider the different types of transition rules (Section 3.28).*

We assume here, that in the derivation step, where the relevant action rule is performed, just one $q \in Q_S^C$ exists. A discussion of the generalization follows after this step. We consider in distinguished cases which kind of transition rule is used in the system abstraction $\Pi_{S,C}$ to construct the counterexample.

chain rule: *In the system abstraction $\Pi_{S,C}$ has to exist a transition rule $p \overset{a}{\rightarrow} p' \in \rightarrow_{\Pi_{S,C}}$. Thus, the transition rule $(I, p, f) \overset{a}{\rightarrow} (f, p, f) \in \rightarrow_{\Pi_S^C}$ is constructed using the rules $\mathcal{R}_C^1$. If $I \overset{a}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ is constructable, then a derivation $I_{\Pi_{S,C}} \Rightarrow_{\Pi_{S,C}} \varepsilon$ and the corresponding λ-rules have to exist. Hence, the following derivation exists in the Combined Abstraction Π_S^C:*

$$(I_{\overline{\mathcal{P}_A}}, I_{\Pi_{S,C}}, f) \underset{Ⓐ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} (I_{\overline{\mathcal{P}_A}}, p, f) \underset{Ⓑ}{\overset{a}{\Rightarrow}_{\Pi_S^C}} (f, p', f) \underset{Ⓒ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} (f, \varepsilon, f) \underset{Ⓓ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} \varepsilon$$

For performing Ⓐ transition rules of $\mathcal{R}_C^\lambda$, $\mathcal{R}_{Seq}^\lambda$, $\mathcal{R}_{Fork}^\lambda$ and $\mathcal{R}_{Sync}^\lambda$ are used. The protocol states are leaved unaffected. Thus, the needed transition rules are constructed while considering all transition rules used in the derivation $I_{\Pi_{S,C}} \overset{\lambda}{\Rightarrow} p$. Ⓑ is performed while using the constructed transition rule as described above. Following the same arguments as for Ⓐ the transition rules for the derivation Ⓒ are constructed, while considering $p \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$. While using a transition rule of $\mathcal{R}^\varepsilon$ the derivation Ⓓ is performed. Hence, if $w \in L(\overline{\mathcal{P}_C})$ and $w \in L(\Pi_{S,C})$ while using a chain rule in $\Pi_{S,C}$, then $w \in L(\Pi_S^C)$.

elimination rule: *Let $p \overset{a}{\rightarrow}_{\Pi_{S,C}} \varepsilon$ be given. This case is analogous to the previous case, while the step Ⓒ is not needed and a transition rule of $\mathcal{R}^\varepsilon$ is used in step Ⓑ.*

sequential rule: *Let* $p \xrightarrow{a}_{\Pi_{S,C}} p'.p''$ *be given. Thus, the transition rules* $(I_{\overline{\mathcal{P}_A}}, p, f) \xrightarrow{a}_{\Pi_S^C} (f, p', x).(x, p'', f)$ *are constructed, where* $x \in \{I_{\overline{\mathcal{P}_A}}, f\}$ *(the only two states of the protocol in this case). Hence, a the following derivation exists in the Combined Abstraction* Π_S^C*:*

$$(I_{\overline{\mathcal{P}_A}}, I_{\Pi_{S,C}}, f) \underset{Ⓐ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} (I_{\overline{\mathcal{P}_A}}, p, f) \underset{Ⓑ}{\overset{a}{\Rightarrow}_{\Pi_S^C}} (f, p', f).(f, p'', f)$$

$$\underset{Ⓒ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} (f, \varepsilon, f).(f, p'', f) \underset{Ⓓ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} \varepsilon.(f, p'', f) \underset{Ⓔ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} (f, \varepsilon, f) \underset{Ⓕ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} \varepsilon$$

The transition rule used for Ⓑ is constructed as described above. Considering $\Pi_{S,C}$ *it has to exist a derivation* $p' \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ *and* $p'' \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$*. Thus, a construction of transition rule in* Π_S^C *is possible while using* $\mathcal{R}_C^\lambda$ *and* $\mathcal{R}_{Elem}^\lambda$*. They allow the derivations Ⓒ and Ⓔ. Ⓓ and Ⓕ are possible while using a transition rule of* $\mathcal{R}^\varepsilon$*.*

fork rule: *Let* $p \xrightarrow{a}_{\Pi_{S,C}} p'||p''$ *be given. Thus, the transition rule* $(I_{\overline{\mathcal{P}_A}}, p, f) \xrightarrow{a}_{\Pi_S^C} (f, p', f)||(f, p'', f)$ *is constructed. The derivation of* Π_S^C *is analogous to the step discussed previously.*

synchronization rule: *Let* $p||p' \xrightarrow{a}_{\Pi_{S,C}} p''$ *be given. Thus, the transition rule* $(I_{\overline{\mathcal{P}_A}}, p, f)||(I_{\overline{\mathcal{P}_A}}, p', f) \xrightarrow{a}_{\Pi_S^C} (f, p'', f)$ *is constructed. Again, as no other action rule is performed in* $\Pi_{S,C}$*,* $(f, p'', f) \overset{\lambda}{\Rightarrow}{}^C_S \varepsilon$ *is possible. Therefore, the following derivation is possible:*

$$(I_{\overline{\mathcal{P}_A}}, I_{\Pi_{S,C}}, f) \overset{\lambda}{\Rightarrow}_{\Pi_S^C} (I_{\overline{\mathcal{P}_A}}, p, f)||(I_{\overline{\mathcal{P}_A}}, p', f)$$

$$\overset{a}{\Rightarrow}_{\Pi_S^C} (f, p'', f) \overset{\lambda}{\Rightarrow}_{\Pi_S^C} (f, \varepsilon, f) \overset{\lambda}{\Rightarrow}_{\Pi_S^C} \varepsilon$$

Thus, it is possible to construct the counterexample w *using the permitted transition rules of a Process Algebra Nets* Π_S^C*. However, the terms (process-algebraic expressions) in the considered derivation steps of* Π_S^C *might contain more process constants* $q \in Q_{\Pi_S^C}$*. Nevertheless, the same constructions are possible, while demanding that the considered process constants used in the steps above are contained in the corresponding terms. For example, the derivation used while considering the chain rules (first case, above) looks like the following in the general form:*

$$(I_{\overline{\mathcal{P}_A}}, I_{\Pi_{S,C}}, f) \underset{Ⓐ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} t \underset{Ⓑ}{\overset{a}{\Rightarrow}_{\Pi_S^C}} t' \underset{Ⓒ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} \varepsilon \quad \text{(GeneralCase)}$$

where $t, t', t'' \in PEX(Q_{\Pi_S^C})$ *as well as* t *contains* $(I_{\overline{\mathcal{P}_A}}, p, f)$*,* t' *contains* (f, p', f) *that has replaced* $(I_{\overline{\mathcal{P}_A}}, p, f)$*, and* t'' *contains* (f, ε, f) *that has replaced* $(I_{\overline{\mathcal{P}_A}}, p, f)$*. We abstain from noting this generalized form to avoid confusion here and in the following cases of the proof.*

Moreover, it has to be claimed, that all other process constants $(\bar{v}, \bar{p}, \bar{\bar{v}}) \in Q_{\Pi_S^C}$ *of* t*,* t' *and* t'' *do not perform action rules, while being translated to* ε*:* $(\bar{v}, \bar{p}, \bar{\bar{v}}) \overset{\lambda}{\Rightarrow} \varepsilon$*. This claim has to be considered. It is clear: If* t *contains just the term* $(I_{\overline{\mathcal{P}_A}}, p, f).\hat{t}$*, then the proof of [HU79] on the intersection of a regular language with a CFG can be used (cf. Section 3.2.3). If* t *has the form wlog* $((I_{\overline{\mathcal{P}_A}}, p, f).\hat{t})||\hat{\hat{t}}$*, where* $\hat{t}, \hat{\hat{t}} \in PEX(Q_{\Pi_S^C})$*, then* $\hat{t}$ *and* $\hat{\hat{t}}$ *have to be translatable into* ε *without performing action rules.*

Only λ-rules are allowed. As known, the term $(p.\bar{t})||\bar{\bar{t}} \in PEX(Q_{\Pi_{S,C}})$ is translatable into ε, where $\bar{t}, \bar{\bar{t}} \in Q_{\Pi_{S,C}}$, so that it is valid that $(p.\bar{t})||\bar{\bar{t}} \overset{a}{\Rightarrow}_{\overline{\mathcal{P}_A}} \varepsilon$. Thus, for each $(\bar{v}, \bar{p}, \bar{\bar{v}})$, that is contained in $\bar{t}$ or $\bar{\bar{t}}$, it is possible to translate it into ε (if $\bar{v} = \bar{\bar{v}}$) or to $(\bar{v}, \varepsilon, \bar{\bar{v}})$ (if $\bar{v} \neq \bar{\bar{v}}$). The latter case need a transition rule of $\mathcal{R}^0$, as they are the only one changing the (first) protocol state in the triple, without performing an action rule. Because a transition rule $\bar{v} \overset{a}{\rightarrow} \bar{\bar{v}}$ exist if a counterexample is constructed performing action a, the required transition rule $(\bar{v}, \varepsilon, \bar{\bar{v}}) \overset{\lambda}{\rightarrow}_{\Pi_S^C} (\bar{\bar{v}}, \varepsilon, \bar{\bar{v}})$ is constructable. This step is called protocol-state-synchronization. Thereafter, the following derivation is possible: $(\bar{v}, \varepsilon, \bar{\bar{v}}) \overset{\lambda}{\Rightarrow}_{\Pi_S^C} \varepsilon$ (using a transition rule of $\mathcal{R}^0$ in addition). Thus, every counterexample w described by $\mathcal{P}_A$ and $\Pi_{S,C}$ can be found using the constructed transition rules of Π_S^C if $w = 1$.

$|w| = n + 1$: *We will prove the statement containing $n + 1$ interactions.*

$$(I_{\overline{\mathcal{P}_A}}, I_{\Pi_{S,C}}, f) \underset{Ⓖ}{\overset{w'}{\Rightarrow}_{\Pi_S^C}} t \underset{Ⓗ}{\overset{a}{\Rightarrow}_{\Pi_S^C}} t' \underset{Ⓘ}{\overset{\lambda}{\Rightarrow}_{\Pi_S^C}} \varepsilon$$

where $|w'| = n$ and $a \in \Sigma_{\Pi_S^C}$.
By induction hypothesis is clear that Ⓖ can be performed. This step results in $t \in PEX(Q_{\Pi_S^C})$, wlog we request that the protocol states are synchronized (in case of parallel behavior while using transition rules of $\mathcal{R}^0$ to perform protocol-state-synchronizations). Thereafter, the step Ⓗ show the same semantics as Ⓑ at GeneralCase. The same is valid for Ⓘ and Ⓒ. □

A.5 Used Tools

We use the following tools for the generation of the graphics:

- websequencediagrams by Steve Hanov [Han07]
- Openoffice Draw
- LaTeX and several packages

The used icons are part of the packages:

- "Crystal Project Icons" by Everaldo Coelho, `http://www.everaldo.com`
- "Oxygen for KDE3" by "The Oxygen Team"

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich diese Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Um einen Doktorgrad habe ich mich bisher nicht beworben.

Andreas Both

Halle (Saale), den 27. November 2009

Lebenslauf

Persönliche Daten

Name:	Andreas Both
E-Mail-Adresse:	andreas.both@informatik.uni-halle.de
Geburtsdatum und -ort:	29.06.1979 in Halle (Saale), Deutschland
Familienstand:	verheirat, ein Kind

Hochschulausbildung

seit März 2005	Mitarbeiter am Lehrstuhl für "Softwaretechnik und Programmiersprachen" des Instituts für Informatik an der Martin-Luther-Universität Halle-Wittenberg (Prof. Dr. Wolf Zimmermann)
1999 – 2005	Studium im Fach Informatik an der Martin-Luther-Universität Halle-Wittenberg Abschluss als Diplominformatiker (Dipl.-Inform.) mit der Vertiefungsrichtung "Softwaretechnik und Programmiersprachen" (Wahlpflichtfach Designinformatik), Gesamtnote: Sehr gut

Grundwehrdienst

1998 – 1999	Grundwehrdienst im 3. Panzerartilleriebataillion 55, Homberg (Efze)

Schulausbildung

1992 – 1998	Trotha-Gymnasium "Hanns Eisler" in Halle (Saale), Abschluss: Abitur
1986 – 1992	Christian-Wolff-Gymnasium in Halle (Saale)