On the Application of a Theory for Mobile Systems to Business Process Management

Frank Puhlmann

Business Process Technology Group
Hasso Plattner Institut, University of Potsdam
Potsdam, Germany

—Doctoral Thesis—

July, 2007

Zusammenfassung

Diese Arbeit untersucht die Anwendung einer Theorie für mobile Systeme – das π-Kalkül – auf den Bereich Geschäftsprozessmanagement. Dieser stellt Konzepte und Technologien zur Erfassung, Analyse, Ausrollung, Überwachung und Auswertung von Geschäftsprozessen zur Verfügung. Mit der fortschreitenden Verbreitung von dienstbasierten Architekturen als eine zentrale Realisierungsstrategie für Geschäftsprozessmanagement verschiebt sich der Fokus von statischen Prozessbeschreibungen, welche durch einen zentralen Abwickler in geschlossenen Umgebungen ausgeführt werden, hin zu agilen Interaktionen welche in verteilten Umgebungen wie dem Internet ausgeführt werden. Das π-Kalkül stellt eine Theorie zur Beschreibung solcher Systeme zur Verfügung.

Im Kontrast zu etablierten, formalen Grundlagen des Geschäftsprozessmanagements bietet das π-Kalkül eine direkte Unterstützung von Verbindungsübergabemobilität. Verbindungsübergabemobilität stellt die Bewegung von Verbindungen in einem abstrakten Raum von verbundenen Prozessen dar. Angewandt auf das Internet repräsentieren Verbindungen einheitliche Quellenanzeiger welche zwischen verschiedenen Entitäten übergeben werden. Aufgrund dieser Fähigkeit kann eine Kernfunktion von dienstbasierten Architekturen, dynamisches Binden, formal dargestellt werden. Dynamisches Binden ist ein Schlüsselkonzept welches zur Darstellung von agilen Interaktionen, in denen Geschäftsprozesse dynamisch aus gegebenen Diensten komponiert werden, benötigt wird. Neben der Unterstützung von dynamischem Binden muss eine formale Grundlage für Geschäftsprozessmanagement Möglichkeiten zur Unterstützung von existierenden Techniken bieten. Dazu werden die Fähigkeiten des π-Kalküls zur Darstellung von Daten, Prozessen und Interaktionen basierend auf bekannten Mustern untersucht. Durch die Bereitstellung einer formalen Interpretation dieser Muster können Modelle von Prozessen und Interaktionen zwischen diesen erstellt werden. Aufgrund der eindeutigen Beschreibungen der Modelle können diese zur Spezifikation und Analyse benutzt werden. Im Rahmen der Analyse werden Techniken zur Korrektheitsprüfung der erstellten Modelle entwickelt. Weiterhin wird eine Verbindung zu grafischen Darstellungen gegeben, wobei eine Notation zur Darstellung von dynamischem Binden eingeführt wird.

Abstract

This thesis investigates the application of a theory for mobile systems—the π-calculus—to business process management (BPM). BPM provides concepts and technologies for capturing, analyzing, deploying, running, monitoring, and mining business processes. With the arrival of service-oriented architectures (SOA), a core realization strategy for BPM, the focus shifts from static process descriptions enacted by central engines within closed environments to agile interactions that are executed in distributed environments like the Internet. The π-calculus provides a theory for describing these kinds of systems.

In contrast to established formal foundations for BPM, the π-calculus inherently supports link passing mobility. Link passing mobility denotes the movement of links in an abstract space of linked processes. Brought forward to the Internet, links denote uniform resource locators (URL) that are passed between different entities. Due to this capability, a core feature of SOAs, dynamic binding, can be represented formally. Dynamic binding is a key concept required to represent agile interactions, where business processes are dynamically composed out of given services. Besides supporting dynamic binding, a formal foundation for BPM has to provide means to support state-of-the-art techniques of BPM. Therefore we investigate the capabilities of the π-calculus for representing data, processes, and interactions based on common patterns. By providing formal interpretations of these patterns, models of processes and interactions among them can be created. Since the models provide an unambiguous semantics, they can be used for specification and analysis. Regarding analysis, we develop techniques using bisimulation equivalences for proving the correctness of the models. Furthermore, a link to graphical representations is given, where a notation for representing dynamic binding in a graphical manner is introduced.

Publications based on this Thesis

Early ideas have been published and presented at national and international conferences during the writing of this thesis. A starting point was a conference paper at the third conference on business process management (BPM) in Nancy (France).[1] It showed how the π-calculus might be used to represent the workflow patterns. With this paper, a first draft of the pattern formalizations contained in chapter 5 (Processes) has been brought to a larger audience. While the formalizations in most cases did not required advanced features of the π-calculus, such as link passing mobility, a subsequent paper revealed the strengths of the π-calculus for representing dynamic binding and correlation handling in service-oriented architectures.[2] It laid the foundations for section 6.1.1 (Correlations and Dynamic Binding). This paper has been presented at a workshop covering dynamic web processes alongside the third international conference on service-oriented computing (ICSOC) held in Amsterdam (The Netherlands). The investigation continued with a conference paper that discussed shifting requirements for BPM.[3] Beside the investigation of state-of-the-art, new requirements regarding technical and theoretical foundations have been found. Refined versions of these requirements are used to motivate the thesis in chapter 1 (The Shifting Focus). The results have been presented at the ninth conference on business information systems (BIS) in Klagenfurt (Austria). Thereafter the research focused on soundness properties of business processes formalized in the π-calculus. In contrast to existing properties, the application of bisimulation equivalence for reasoning on deadlock and livelock freedom has been investigated. Based on an extensive study of the workflow pattern formalizations, it turned out that several of them constitute problems regarding soundness. The problems have been overcome by a new soundness property that was named *lazy soundness*. A refined version is contained in section 5.3 (Properties) of chapter 5 (Processes). The new soundness property has been presented to the scientific community at the fourth conference on business process management (BPM) in Vienna (Austria), where it has been published as part

[1] **Frank Puhlmann**, Mathias Weske: *Using the Pi-Calculus for Formalizing Workflow Patterns.* In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera (Eds.): Business Process Management, volume 3649 of LNCS, Nancy, France, Springer-Verlag (2005) 153–168

[2] Hagen Overdick, **Frank Puhlmann**, Mathias Weske: *Towards a Formal Model for Agile Service Discovery and Integration.* In K. Verma, A. Sheth, M. Zaremba, and C. Bussler (Eds.): Proceedings of the International Workshop in Dynamic Web Processes (DWP 2005), Amsterdam, The Netherlands, IBM technical report RC23822 (2005)

[3] **Frank Puhlmann**: *Why do we actually need the Pi-Calculus for Business Process Management?* In W. Abramowicz and H. Mayr (Eds.): BIS 2006—Business Information Systems, volume P-85 of LNI, Klagenfurt, Austria, Gesellschaft fuer Informatik (2006) 77–89

of the conference proceedings.[4] The practical feasibility of lazy soundness has been shown in an additional presentation, published as part of the demo session proceedings.[5] At the same conference, a short paper, written together with a student of mine, gave an insight on how the service interaction patterns might be formalized in the π-calculus.[6] While chapter 6 (Interactions) contains a different approach for representing these patterns, the paper nevertheless provided valuable ideas. The publication series continued with a paper presented at a national conference on service-oriented information systems (EMISA) that took place in Hamburg (Germany).[7] It covered the unification of data, processes, and interactions to provide a unified formal representation of service-oriented architectures. The discussion has been based on an example that can be found in an extended version in chapter 7 (Unification). In the meantime, a book chapter on the suitability of the π-calculus for BPM has been published.[8] It basically contains an extended and updated version of the BIS paper published earlier. Another publication covers an extension of lazy soundness to prove compatibility in interactions.[9] The new compatibility property, denoted as *interaction soundness*, supports dynamic binding. To the knowledge of the author, this was the first paper that introduced compatibility with dynamic binding. The updated results can be found in section 6.3 (Interaction Soundness) of chapter 6 (Interactions). The paper has been presented at the fourth international conference on service-oriented computing (ICSOC) in Chicago (USA). Furthermore, during the writing of this thesis, the author supervised a master thesis where a graphical environment for the simulation of business processes with dynamic binding—based on the ideas found in the second and third part of this work—has been implemented. The corresponding tool has been presented at the open.BPM workshop in Hamburg (Germany).[10]

[4] **Frank Puhlmann**, Mathias Weske: *Investigations on Soundness Regarding Lazy Activities.* In S. Dustdar, J.L. Fiadeiro and A. Sheth (Eds.): Business Process Management, volume 4102 of LNCS, Vienna, Austria, Springer-Verlag (2006) 145–160

[5] **Frank Puhlmann**: *A Tool Chain for Lazy Soundness.* Demo Session of the 4th International Conference on Business Process Management, CEUR Workshop Proceedings Vol. 203, Vienna, Austria (2006) 9–16

[6] Gero Decker, **Frank Puhlmann**, Mathias Weske: *Formalizing Service Interactions.* In S. Dustdar, J.L. Fiadeiro and A. Sheth (Eds.): Business Process Management, volume 4102 of LNCS, Vienna, Austria, Springer-Verlag (2006) 414–419

[7] **Frank Puhlmann**: *A Unified Formal Foundation for Service Oriented Architectures.* In M. Weske and M. Nuettgens (Eds.): EMISA 2006, volume P-95 of LNI, Hamburg, Germany (2006) 7–19

[8] **Frank Puhlmann**: *On the Suitability of the Pi-Calculus for Business Process Management.* In Technologies for Business Information Systems. Springer-Verlag (2007) 51–62

[9] **Frank Puhlmann**, Mathias Weske: *Interaction Soundness for Service Orchestrations.* In A. Dan and W. Lamersdorf (Eds.): Service-Oriented Computing, volume 4294 of LNCS, Chicago, USA, Springer-Verlag (2006) 302–313

[10] Anja Bog, **Frank Puhlmann**: A Tool for the Simulation of Pi-Calculus Systems. In 1. GI-Workshop OpenBPM 2006: Geschäftsprozessmanagement mit Open Source-Technologien, Hamburg, Germany (2006)

Acknowledgements

This thesis would not have been written without the support of many people—I'd like to thank them all. Mathias Weske for being my doctoral adviser and giving me the freedom for my research. Uwe Nestmann for spending many friday afternoons discussing the technical foundations of this work. Wil van der Aalst for providing the seven challenges as well as tons of related work—without him I would still seek for my topic. Anja Bog for writing a Master thesis about—and implementing—the PiVizTool. Gero Decker and Hagen Overdick for countless hours of discussion about the π-calculus, BPM, and all the REST. Arnd Schnieders for being my office mate who always listened to my ideas and critically questioned them. My colleagues Jens Huendling, Dominik Kuropka, Guido Laures, Harald Meyer, and Hilmar Schuschel for always excitingly listening and commenting my talks. All the people that provided interesting papers, discussions, or talks at conferences and mailing lists—way too many to name them all. The anonymous reviewers who always rejected my submissions but provided excellent comments. And finally—my family. My parents for supporting years of study. My wife for giving me the love and support for realizing this work. And—my little daughter who always reminds me that there is something else to live for...

Potsdam, July 2007

Contents

Part I

Foundations

Introduction to Part I

Part I introduces the thesis by motivating the problem, summarizing the theoretical background, and discussing the state-of-the-art. It starts with the observation of fundamental shifts in the studied areas. With the arrival of service-oriented architectures (SOA)—a central realization strategy for business process management (BPM)—static process descriptions, enacted by central engines within closed environments, have come to their limitations. Instead, dynamic interactions, based on dynamic binding of services found in open environments, come into play. Existing theoretical treatments based on parallel system theory, however, elide dynamic binding and are thus left behind recent practical developments. Nevertheless, advancements in theoretical computer science gave rise to theories of mobile systems such as the π-calculus. The π-calculus is a calculus of parallel components that communicate and change their structure. Due to its support of changing structures, this calculus is well suited to represent dynamic binding required for today's BPM architectures. The preliminaries for the application of the π-calculus to the area of BPM will be settled in the first part.

Structure of Part I Part I is composed of three chapters. The first chapter introduces the shifting requirements for BPM and motivates the thesis. The second chapter introduces a variant of the π-calculus that is used as the formal foundation. The third chapter introduces the state-of-the-art in business process management.

Chapter 1

The Shifting Focus

This thesis discusses the application of a theory for the description of mobile systems into the domain of business process management (BPM). BPM focuses on designing, enacting, managing, analyzing, adapting, and mining business processes [14]. The investigated theory—the π-calculus [99]—has been developed during the last two decades based on observations on the limitations of existing formal theories for sequential and parallel systems. Since sequential and parallel systems are widely agreed on for the implementation and description of workflows, a special kind of business processes, we discuss why these do not match the shifting requirements for the wider area of business process management, introduce arising theories, and finally settle the scope and scientific contribution.

1.1 Shifting Requirements

Nowadays, we can observe a fundamental shift in the requirements for computer aided business processes. Those new requirements arise from the evolution of workflow management (WfM) to business process management. Current state-of-the-art in workflow research focuses on static process structures for designing and enacting business processes. BPM, in contrast, discusses agile orchestrations and choreographies resulting from service-oriented architectures (SOA) [41] as the central realization strategy for BPM. This leads to distribution instead of centralized engines, dynamic process structures instead of static workflows, and agile interactions between distributed services instead of pre-defined interactions. Why? Because the environments in which today's business processes are executed shift from closed to open. These sketched shifts raise interesting questions regarding the formal representation and verification of interacting business processes. In the following subsections we motivate the shifts and discuss the issues in detail.

1.1.1 From Static to Dynamic Systems

Current state-of-the-art research in workflow management focuses on static system theory for designing and enacting business processes. Examples are workflow nets [9], the YAWL system [11], workflow modules [85], or production workflows [83]. Analysis of business processes is

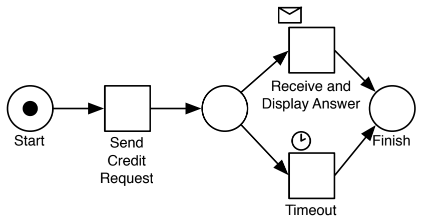


Figure 1.1: Sample business process in workflow net notation [9].

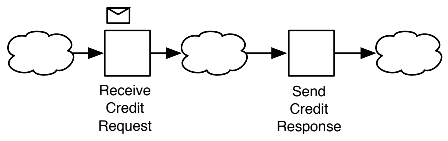


Figure 1.2: An abstract process for interaction with the process from figure 1.1.

focused on Petri nets [110], such as given by different variants of soundness [1, 51]. However, as BPM broadens to inter-organizational business processes between departments, companies, and corporations, static process descriptions have come to their limitations. This especially holds since the arrival of service-oriented architectures as a central realization strategy for BPM.

To underpin these assumptions we provide an example shown in figure 1.1. We used the workflow net notation [9], since this notation is state-of-the-art in (theoretical) WfM. The business process consists of a task that sends a credit request and afterwards waits for either the response or a timeout if no response has been received within a given timeframe. This process is executed in isolation in a workflow management system (WfMS). Each task appears at the work list of an employee who executes it. The first task consists of writing and sending a letter. Afterwards, two exclusive tasks appear at the work list. If an answer is received by mail within a given timeframe, the answer is processed, whereas otherwise the timeout task is selected (that contains some fallback actions).

Most business process management systems (BPMS) incorporate the service-oriented computing (SOC) paradigm. Using SOC, a business process can be *wrapped* into a service. A service can interact with other services to fulfill the goals of the contained business process. For these interactions to take place, a corresponding service is required. Let's assume this service to have an abstract process, meaning that we only know the parts that can be used for interaction, as shown in figure 1.2. We use clouds to denote the hidden parts. All we know is the interface description (receive a request, send response with the corresponding parameter format not shown in the visualization), as well as the interaction behavior (first receive a request, then send a response). To denote the interaction between the services in a static way, we need to introduce additional states that describe incoming requests and outgoing responses. The result contains two workflow nets, which interact by shared places, shown in figure 1.3.

However, converting business processes to services by defining their static interaction points is only half the truth of a service-oriented architecture. Beside a *service requester* and a *service*

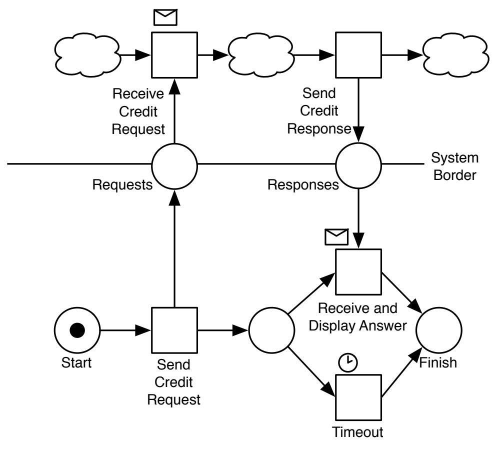


Figure 1.3: Static interaction between the business processes from figure 1.1 and 1.2.

provider, as given by the examples, a third role, called a *service broker* is employed inside a service-oriented architecture. The task of the service broker is to discover matching services based on a request from the service requester and a list of registered service providers. Matching services can then dynamically incorporated for usage within the business process of the service requester (denoted as *dynamic binding*). Notable, new service providers can register at the service broker even after the business process of the service requester has been deployed. Possible interaction partners cannot be anticipated in advance, but furthermore are discovered and dynamically integrated during runtime. Another scenario for dynamic binding is given by *callbacks*, either via a single or multiple other services. In this case, the service requester hands some kind of address to the service he invokes. The service is free to give this address to other services as needed. These other services, as well as the original service, can use the address for asynchronous responses. Therefore, the services need to be able to dynamically bind themselves to the original requester.

Requirement One. A theory for BPM based on service-oriented architectures requires support for dynamic binding of services that was not needed in static WfM theory.

1.1.2 From Central Engines to Distributed Services

Service-oriented architectures as the primary realization for BPM enforce another shift. Loose coupling between activities of business processes becomes important. Loose coupling is realized by making single activities available as services. Figure 1.4 shows the interaction from figure 1.3 by representing all tasks as individual services. A circle with a short name inside represents a service. Lines denote dependencies between services. Each line connects a postcondition of one service with the precondition of another one, where the precondition end is marked with a filled circle. Dependencies are given between $P1$ and $S1$, $P2$, $P3$ as well as between $S2$ and $P2$. The services $S1$, $P2$, and $P3$ can only be activated after $P1$ has been executed, meaning

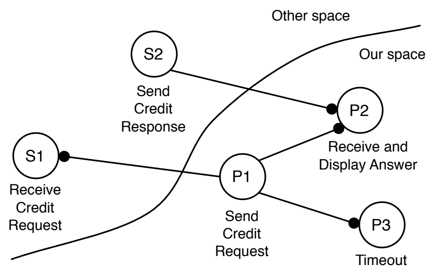


Figure 1.4: Dynamic routing and interaction view of figure 1.3.

the preconditions of $S1$, $P2$, and $P3$ depend on the postconditions of $P1$. The service $S2$ has some preconditions linked to $S1$ that are not known to us.

The loose coupling of different activities that are wrapped into services allows for highly distributed BPM systems. Instead of having a single engine controlling every aspect of a workflow, the dependencies are now spread across services representing parts of collaborating business processes. Distributed services wait for messages to arrive that trigger their activation and produce new messages to trigger other services. Still, there are some distinctions to be made. They start with different spaces in the environment where the distributed services live in. In figure 1.4, this is denoted with *our space* and *other space*. Our space is usually something like an intranet, where we control things like access conditions, availability, implementation issues, and so on. We make some of our services available to the outer world, acting as interaction points, without providing knowledge or access to our internal structures. Indeed, we are free to restructure our internals as wanted. Our processes incorporate other services that are available in the other space, typically in other intranets or the Internet. These other services are parts of systems such as ours, and represent interaction interfaces. However, we have only limited control over them, mostly by legal agreements. We cannot enforce their availability, functionality, or implementation. Still, we are free to drop them as interaction partners and bound to others. This high flexibility requires the shift from closed, central engines for enacting workflows to open, distributed services for representing business interactions.

Requirement Two. A theory for BPM should support composition and visibility of different components, since the focus shifts from centralized workflow engines to collaborating and distributed business processes, where integration becomes a core activity.

1.1.3 From Closed to Open Environments

In workflow theory, the execution environments are static and predictable. We denote this kind of environment as closed, since the level of external influences is rather low. However, with shifts from traditional departments and companies up to virtual organizations and agile, customer specific collaborations, the execution environment is shifting too. This new kind of environment is called open and is represented by large intranets as well as the Internet.

Closed environments are usually accessible, deterministic, static, and have a limited number of possible actions. Accessibility describes the level of knowledge we have about the execution environment regarding completeness, accuracy, and up-to-dateness of the information. In a single company or department we should be able to exactly capture this knowledge. If we expand the environment to the whole Internet, there is much left in the dark that could be useful or crucial for the business, however we are simply unable to find and incorporate it. Executing a task in an open environment is more uncertain then in a closed one. This is denoted as the determinism of the actions. In an open environment they are way more possibilities to foresee and handle. However, if the environment is complex enough, as e.g. the Internet, we cannot enforce everything. While closed environments are most static, open environments tend to be constantly changing in large parts, regardless of our actions. Interaction partners appear, disappear, are prevented, or something else happens that we have to take into account for the business to run. Furthermore, the number of interaction partners that can be invoked to perform a certain task is rising fast as the environment opens to the world. So the decision-making process of whom to incorporate into the business is getting more complex.

Requirement Three. A theory for BPM should support change, since the environment where business processes are executed is shifting from closed to open.

1.2 Advancing Theories

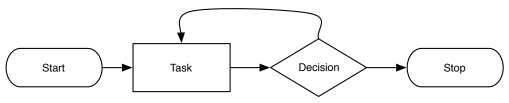
After having introduced the shifting requirements for business process management, we discuss how theories of computer science can pace up with them. We start with sequential systems, advance to parallel systems, and conclude with mobile system.

1.2.1 Sequential Systems

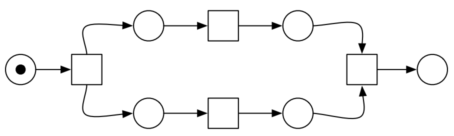
Sequential systems can be formally described by the λ-calculus [23]. The λ-calculus is a theory designed to investigate the definition of functions that are used for sequential computing. It brought the ideas of recursion and the precise definition of a computable function into discussion even before the first computers were constructed. In the view of computer science, the λ-calculus can be seen as the smallest universal programming language as any computable function can be expressed and evaluated using this formalism. The λ-calculus can be used to describe compositional systems, i.e. system where terms can be replaced and reused. Computational equal to the λ-calculus are Turing machines [122]. Both had and have a large impact on today's programming languages, where the former grounds functional programming and the latter imperative languages. A common graphical representation of sequential systems is given by flow charts as shown in figure 1.5(a).

1.2.2 Parallel Systems

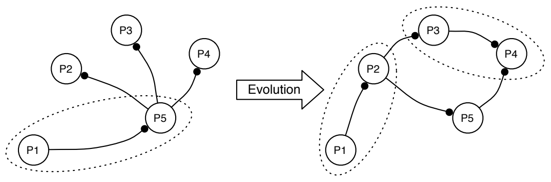
While the λ-calculus and Turing machines build the foundation for many computer science related topics, the formal description of business processes requires a different approach. In



(a) Sequential System.



(b) Parallel System.



(c) Mobile System.

typical business processes tasks are not only executed in sequential order, furthermore tasks are executed in parallel by different employees to speed up the processing. These different—then again sequential—processing paths have to be created and joined at some points in the business processes. Even further, parallel processing tasks could depend on each other. The optimization of business processes usually adds parallelism as well as dependencies as this is an effective way to reduce the throughput time for requests. These kinds of parallel processes are difficult to describe in terms of the λ-calculus. To overcome the limitations of sequential systems, an approach to represent parallel systems, called Petri nets [110], has been proposed. Petri nets have a powerful mathematical foundation as well as a strong visual representation. An example is shown in figure 1.5(b). Petri nets use the concept of an explicit state representation for parallel systems. Each Petri net is always in a precisely defined state denoted by the distribution of tokens over places contained in the net. The state of the system can be changed by firing transitions that relocate the token distribution over the places. Petri nets have been adapted by many systems that are used in the workflow management domain to describe business processes, e.g. in [55, 2]. Beside the advantages of Petri nets that include strong visualization capabilities, mathematical foundations, as well as their main purpose, the description of parallel systems, Petri nets also have serious drawbacks regarding the shifting requirements for BPM. The main drawbacks are the static structure of the nets as well as the missing capabilities for advanced composition as for instance recursion. A broad research on the capabilities of Petri nets regarding common patterns of behavior found in business processes showed that they fulfill basic tasks like splitting and merging process paths easily, while they fail at advanced patterns like multiple instances of a task with dynamic boundaries [12]. Whereas there exist approaches to overcome some or all of the limitations regarding the behavior [9, 11], the static structure and limited composition capabilities of Petri nets remains.[1]

[1] Petri have been extended with support for dynamic structure, like self-modifying Petri nets [124], recursion [69], and objects [100]. However, these enhancements also complicate the theory of the nets and thus have reached limited usage in the area of BPM.

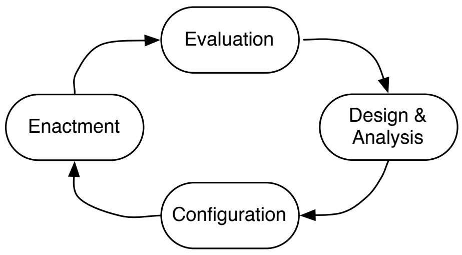


Figure 1.5: BPM lifecycle.

1.2.3 Mobile Systems

A theory for mobile systems, the π-calculus [99], overcomes the limitations of Petri nets regarding the static structure and limited composition capabilities at the cost of a more complex representation. The π-calculus represents mobility in mobile systems by directly expressing movements of links in an abstract space of linked processes. An example is shown in figure 1.5(c). Due to the fact that a mobile system's structure is evolving all the time, only snapshots can be given. Practical examples are hypertext links that can be created, passed around, and disappear. The π-calculus does not, however, support another kind of mobility that represents the movement of processes. An example is code that is sent across a network and executed at its destination. The π-calculus focuses on interactions as first class citizens. Interactions take place between different parallel processes. The processes use names for interaction, where names are a collective term for concepts like channels, links, pointers, and so on. As the mobile system evolves, names are communicated between processes and extrude or intrude their scope regarding certain processes. As synchronization between processes is based on interactions and received names can also be used as communication channels, the link structure is changing all the time the mobile system evolves. Another main difference to Petri net theory is given by the focus on observation of the external visible behavior of systems instead of their internal computations. Observational theory gives rise to bisimulation equivalence, which can be used to manifest invariants of the system under investigation. Link passing mobility, composition, and support for bisimulation equivalence of dynamic systems make the π-calculus a promising candidate for providing a theoretical foundation for business process management that has yet been neglected in scientific research.

1.3 Scope and Scientific Contribution

After having motivated the shifting requirements for business process management and discussed theories for supporting them, we determine the scope, highlight the scientific contribution, and introduce the structure of this thesis.

1.3.1 Scoping

Business process management can be seen as a circle of activities, shown in figure 1.5. The initial activities are placed inside *Design and Analysis*. Here, business processes are modeled from scratch or existing business processes are re-engineered. The business process models can then be analyzed regarding three different criteria: (1) Are they doing what they are supposed to do from a semantic viewpoint? This sub-activity is called validation. (2) Do they fulfill performance requirements? This one is called simulation. (3) Are they free of structural errors such as deadlocks? The last sub-activity is called verification. In the *Configuration* activity, systems are selected and the business processes are implemented, tested, and deployed. *Enactment* refers to the actual operation and enactment of the business processes, also including monitoring and maintenance. The *Evaluation* activity finally includes process mining and business activity mining as an input to business process re-engineering found again in *Design and Analysis*.

In the course of this thesis we investigate the interplay between business process management and a formal theory for mobile systems. As can already concluded from the description of the BPM lifecycle, formal theories play an important role in the design and analysis activity. Business processes are modeled formally for two major reasons. First of all, a formalized model allows collaborating people—e.g. business analysts, process modelers, or software engineers—to settle upon a common understanding. Each of the collaborators might have an own view atop of the formal model, focusing on certain aspects. The formal model defines concepts like activities, dependencies between them, and execution constrains as well as providing a unique semantics. Second, with the help of a theory, a formal model can be verified regarding certain properties. As shown in figure 1.6, this thesis focuses on the formal representation of business processes in the π-calculus. Closely related to the formal representation are graphical notations that will be used to derive formal models. Graphical notations are optimized for creating business processes and discuss their semantic meaning. Furthermore, the formal model can be used as a foundation for an executable representation that has to be enriched with organizational and system specific properties.

The scope of this thesis is set on the design and verification of formal models of data, processes, and interactions between processes as found in the design and analysis activity of BPM. The theory investigated is a variant of the π-calculus closely related to the original publication by Milner, Parrow, and Walker [99]. Out of scope are actual functional implementations, organizational and operational aspects, deployment, enactment, and evaluation.

1.3.2 Contribution

The scientific contribution provides a unified, sound, and formal foundation for the investigated areas of business process management. First of all, a sound formalization of data, workflow, and service interaction patterns will be given. Due to the nature of the original pattern descriptions, a number of implicit assumptions have to be made explicit in the course of this thesis. As a second step, algorithms for mapping a graphical notation to π-calculus expressions will be given. We focus on a subset of the Business Process Modeling Notation (BPMN). Furthermore, we enhance the subset of the BPMN to directly support dynamic interactions based on the idea of π-calculus names. To abstract the formal expressions of the π-calculus from a certain graphical notation for

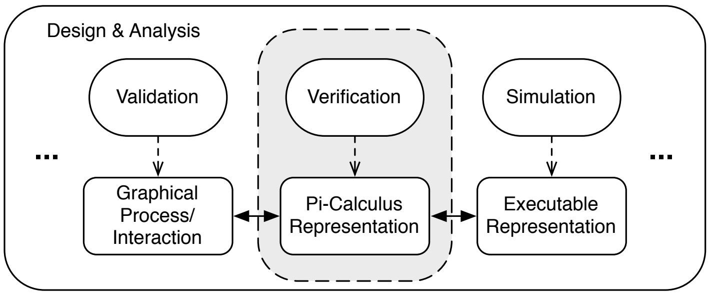

Design & Analysis
Validation
Verification
Simulation
...
...
Graphical Process/ Interaction
Pi-Calculus Representation
Executable Representation

Figure 1.6: The scope of this thesis classified inside the BPM lifecycle.

BPM, we introduce an intermediate layer given by process and interaction graphs. These graphs provide an abstract view of processes and interactions. Basing on the formal models, we derive two new kinds of soundness that will be denoted as *lazy soundness* and *interaction soundness*. While the former guarantees a temporal deadlock freedom for processes, the latter provides a compatibility notion for a set of interacting processes with a special focus on dynamic binding. The soundness properties will be complemented by a behavioral conformance property that considers observable interactions of different services. The property will be called *interaction equivalence*. Since interaction equivalence is too strong regarding several applications, a weaker version that will be based on simulation will conclude the different kinds of verification. Since the actual reasoning will be done in the π-calculus, algorithms for proving lazy soundness, interaction soundness, and interaction equivalence using bisimulation equivalences will be provided. Finally, a prototypical tool chain for showing the practical feasibility will be discussed.

1.3.3 Structure

This thesis is divided into three parts. The first part introduces the π-calculus and the domain of business process management including key concepts. The second part investigates how the π-calculus can be applied to describe and reason about models of business processes including data, processes, and interactions. The third part discusses the results leading to a unified formal foundation for the investigated areas.

Part I: Foundations. Chapter 2 introduces the π-calculus. It starts by classifying existing process calculi. A syntax and semantics for the π-calculus is given afterwards. The chapter is concluded by providing a graphical representation of π-calculus systems as well as discussing bisimulation as a way of reasoning. Chapter 3 introduces the domain of business process management starting with key concepts used throughout this thesis. It then discusses workflow as state-of-the-art including different perspectives and formal foundations given by set theory and Petri nets. Workflow is complemented by expansion to business process management and service-oriented architectures including orchestrations, choreographies, and existing formal foundations. The chapter concludes by introducing a notation for the graphical representation of business processes.

Part II: Investigations. Chapter 4 discusses how structured data can be represented in the π-calculus. Based on formal definitions of basic types like booleans and integers, complex structures like tuples, stacks, queues, and lists are defined. The chapter concludes by giving examples of how the workflow data patterns can be formalized. Chapter 5 investigates how business processes can be represented formally. It starts by introducing an abstract structure called process graph that builds an intermediate layer between a graphical notation and the π-calculus. It introduces an algorithm to map process graphs to π-calculus expressions based on the workflow patterns. Furthermore, reasoning on process graphs is introduced, leading to the definition of a new kind of soundness called *lazy soundness*, as well as adapting existing soundness definitions to process graphs. Reasoning is based on bisimulation equivalences between different invariants and π-calculus formalizations of the process graphs. Chapter 6 goes one step further and discusses interacting business processes. It discusses concepts like correlations and dynamic binding typically found in interacting systems. Interactions between business processes are represented using interaction graphs, that again represent an intermediate layer between a graphical notation and the π-calculus. This time, an algorithm for mapping interaction graphs to π-calculus expressions is given. Furthermore, possible realizations of the service interaction patterns inside an interaction graph are discussed. The chapter concludes by introducing reasoning on interaction graphs, deriving two new properties, namely *interaction soundness* and *interaction equivalence*. While the former focuses on deadlock freedom of a given process and a set of services, the latter ensures behavioral equivalence of different service realizations.

Part III: Results. In chapter 7, the unification of the three investigated areas is illustrated by example. Further concepts like data flow graphs are discovered, investigated, and described. The chapter concludes by applying simulation and reasoning on the example. Chapter 8 discusses the results by first returning to the fulfillment of the shifting requirements of BPM. Further topics include the discussion of drawbacks of the π-calculus for the investigated areas as well as related work. The thesis is concluded in chapter 9, where the results are summarized and ideas for future work are drawn.

Chapter 2

The π-calculus

This chapter describes the π-calculus as it is applied throughout this thesis. The π-calculus is a *process calculus*, which is a theoretical framework for the study of concurrent processes. Since there exist different variants of the π-calculus, e.g. [99, 108, 96, 97, 118], necessary and matching concepts are distilled and presented in this chapter. These are the syntactical rules to derive processes, formal semantics, and bisimulations between processes. The chapter starts by introducing different process calculi in general and classifies the π-calculus among them.

2.1 Classification

The history of process calculi can be traced back to the seventies, when in 1978 Hoare [71] proposed a language called *Communicating Sequential Processes* (CSP). It aimed at the description of parallel systems that are composed out of sequential components. The key concepts of CSP include guarded commands to control non-determinism and parallel commands to execute sequential processes concurrently. In contrast to existing approaches at that time, the sequential processes do not communicate using global variables. Instead, they denote a system of interacting automata with local variables that use communication with input and output commands to synchronize their execution. CSP has been extended later on with a formal semantics [40, 72] and is still a major foundation for the description and reasoning on parallel systems [111].

Since CSP lacked a formal semantics in the beginning, a competing approach called a *Calculus of Communicating Systems* (CCS) has been proposed in 1980 by Milner [92]. Milner based his calculus on rigid formal foundations required for the investigation of *observation equivalence* between systems made up of concurrent processes. Observation equivalence between two systems is given if their behavior is indistinguishable by observation in all possible environments (contexts). In his work, he also denoted what a useful process calculus is about:

> ”It should be possible to describe existing systems, to specify and program new systems, and to argue mathematically about them, all without leaving the notational framework of the calculus.” [92]

While Hoare’s CSP at this time mainly aimed at description and specification of systems, Milner’s CCS had a strong mathematical focus on reasoning about concurrent processes. Like CSP,

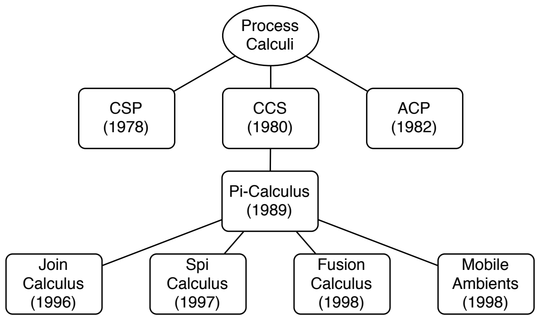


Figure 2.1: A classification of selected process calculi.

also CCS has been refined later on [93, 94].

In 1982 another approach, this time called an *Algebra of Communicating Processes* (ACP), was published by Baeten and Weijland [21]. The purpose of this approach was to provide an axiomatic investigation of concurrent processes. Baeten and Weijland furthermore coined the term *process algebra* in their work. Today, process algebra and process calculus are often used synonymously.

In the mid-eighties, all three major approaches (CSP, CCS, and ACP) reached a level of maturity making them well suited to fulfill Milner's requirements on a useful process calculus. In 1986 a technical report by Engberg and Nielsen introduced an approach to extend CCS with label passing [56]. In its core, it allowed for the transmission of communication links between concurrent processes. The processes, in turn, can use the newly received communication links to establish communication with processes prior unknown to them. Thus, the concurrent processes of Engberg and Nielsen are not based on static communication structures as in existing process calculi at that time. Their label passing approach is nowadays known as *link passing mobility* (see also page 53). Three years later, in 1989, the π-calculus was introduced by Milner, Parrow, and Walker based on CCS and the work of Engberg and Nielsen [99]. They described a calculus of communicating systems that is able to express processes with changing structures. The π-calculus merges variables, constants, and channels, by unifying them into one single concept called *name*. Names are used as input and output ports of processes as well as the values that are communicated. Based on or inspired by the π-calculus, several subsequent process calculi targeting more specific topics have been developed. Examples are the *Join Calculus* by Fournet and Gonthier [61], *Spi-Calculus* by Abadi and Gordon [16], *Mobile Ambients* by Cardelli et al. [44], and *Fusion Calculus* by Parrow [109]. In 2001 Milner introduced bigraphical reactive systems [98] as a model of mobile interactions.

A graphical classification of the different process calculi discussed is shown in figure 2.1. While CSP and ACP also motivated subsequent developments, only a selected subset of extensions to the concepts of the π-calculus are included. An extended discussion on the history of process algebra by Baeten can be found in [20]. A more recent discussion of mobile calculi by Nestmann can be found in [103].

2.2 Syntax and Semantics

The process calculus under consideration, the *polyadic* π-calculus, consists of an infinite set of names and another infinite set of *agent identifiers*.[1] As stated, names are a collective term for concepts like links, pointers, references, identifiers, channels, and so on. They are used for interaction among concurrent agents, as well as representing data that is communicated in these interactions. $\mathcal{N}$ denotes the set of names ranged over lowercase letters such as a, b, c and $\mathcal{K}$ denotes the set of agent identifiers ranged over uppercase letters such as R, S, T.

The agents *evolve* by performing *actions*. The *capabilities* for action are divided into four kinds. The first capability of an agent is sending a tuple of names, denoted as $\tilde{y}$, synchronously via another name used as a channel. The second capability represents the opposed functionality of receiving a tuple of names synchronously via another name, again used as a channel. To avoid confusion, the names of $\tilde{z}$ have to be pairwise distinct. The third capability is the execution of an unobservable action, a so called *silent step*. The last capability is performing a match between two names. Capabilities of agents are represented as *prefixes* given by:

$$\pi ::= \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi \ . \tag{2.1}$$

The output prefix $\overline{x}\langle\tilde{y}\rangle$ consists of a *subject* $\overline{x}$ and *objects* $\tilde{y}$. A name used as the subject of an output prefix is called a co-name, it is represented with a bar above the name. The subject can be thought of as an output port of an agent that contains it and it is able to send the objects. The input prefix $x(\tilde{z})$ consists of the subject x and the objects $\tilde{z}$. Here, the subject x can be thought of as an input port. The input prefix is able to receive arbitrary names and replace each further occurrence of the names of $\tilde{z}$ in the agent containing the input prefix with the received names. The unobservable prefix is denoted as τ, it represents an internal or silent step. The match prefix is denoted as $[x = y]\pi$, it behaves like the prefix π if x and y are equal. The agents of the π-calculus are given by:

$$\begin{aligned} P &::= M \mid P|P \mid \nu z\, P \mid A(x_1, \ldots, x_n) \\ M &::= \mathbf{0} \mid \pi.P \mid M + M \ . \end{aligned} \tag{2.2}$$

The termination symbol is $\mathbf{0}$, denoting an agent that can do nothing. The current capabilities of an agent are given by $\pi.P$. They state that the agent behaves as P after the action represented by π has been done. For instance, $\overline{a}\langle x\rangle.x(z).\mathbf{0}$ first sends the name x via a and thereafter receives a name via x. $M + M$ denotes a summation, where the agent continues as either the left or the right hand side. For instance, $a(x).\mathbf{0} + b(y).\mathbf{0}$ can receive a name either via a or b. $P|P$ represents parallel composition. The left and the right hand side are called *components* and are executed independently of each other. Two components can interact via shared names on matching input and output prefixes. For instance, in $\overline{a}\langle x\rangle.\mathbf{0} \mid a(y).\mathbf{0}$ the left hand component can send x via a and the right hand component can receive x via a. The restriction operator $\nu z\ P$ restricts the scope of the name z to P. Components of P can interact via z. For instance, in $(\nu a\ (\overline{a}\langle x\rangle.\mathbf{0} \mid a(y).\mathbf{0}) \mid Q)$, the left hand component can interact with the middle component,

[1] We use the term *agent* to denote a π-calculus process for avoiding a semantic mismatch with a (business) process as introduced in chapter 3.

whereas any instance of Q is unable to interact with the other components. $A(x_1, \ldots, x_n)$ represents a defined agent identifier:

$$A(x_1, \ldots, x_n) \overset{def}{=} P \text{ with } i \neq j \Rightarrow x_i \neq x_j \; . \tag{2.3}$$

Each agent identifier has a definition as above, where all names used as parameters are pairwise distinct. It can be seen as a process declaration with $x_1, \ldots, x_n$ as formal parameters found in P. The formal parameters are replaced by actual names as the agent evolves. An instance is

$$R(x) \overset{def}{=} \nu y \, \overline{x}\langle y\rangle.R(y) + \tau.\mathbf{0} \; ,$$

that either transmits a restricted name via the name given by the parameter or stops execution.

We use parentheses to resolve ambiguity or ease the understanding. Prefixes and restriction bind more tightly than composition. For instance $\pi.P \mid Q$ is $(\pi.P) \mid Q$. Furthermore, product and summation operators are used to denote multiple agents. For instance, $\prod_{i=1}^{3} P_i$ means $P_1 \mid P_2 \mid P_3$ and $\sum_{i=1}^{3} Q_i$ means $Q_1 + Q_2 + Q_3$. A sequence of identical prefixes is denoted by curly brackets, for instance $\{x(a)\}_1^3$ means $x(a).x(a).x(a)$. The length of a tuple $\tilde{z}$ is denoted as $|\tilde{z}|$. If the length of the object is zero, $\overline{x}\langle\rangle$ and $x()$ are denoted with elided brackets, i.e. $\overline{x}$ and x. Finally, we sometimes denote a sequence of restrictions such as $\nu z_1 \ldots \nu z_n$ in a short way by $\nu z_1, \ldots, z_n$ or put brackets around $(\nu z_1, \ldots, z_n)$.

2.2.1 Bindings

The π-calculus has two operators for name binding, i.e. they restrict the scope of a name:

Definition 2.1 (Binding) The input prefix $x(z).P$ and the restriction operator $\nu z \; P$ are *binding* the occurrence of z within the scope of P. $\square$

The occurrence of a name that is not bound is called *free*. The set of names that occur free in an agent P is denoted as $fn(P) \subseteq \mathcal{N}$, whereas the names that occur bound are denoted as $bn(P) \subseteq \mathcal{N}$. Examples are:

$$fn(x(y).(\overline{a}\langle y\rangle.\mathbf{0} + \overline{b}\langle y\rangle.\mathbf{0}) \mid (\nu d)\overline{x}\langle d\rangle.\mathbf{0}) = \{a, b, x\} \; ,$$

and

$$bn(x(y).(\overline{a}\langle y\rangle.\mathbf{0} + \overline{b}\langle y\rangle.\mathbf{0}) \mid (\nu d)\overline{x}\langle d\rangle.\mathbf{0}) = \{d, y\} \; .$$

A bound name in an agent can be changed to another name:

Definition 2.2 (α-conversion) Let $x \in bn(P)$. The syntactical replacement of x inside its scope with another name y, $y \notin fn(P) \cup bn(P)$, is denoted as *$\alpha$-conversion*. $\square$

Two agents P and Q are *α-convertible*, denoted as $P = Q$, if Q can be derived from P by a finite number of changes of bound names (α-conversions). An instance is given by

$$a(x).\overline{b}\langle x\rangle.\mathbf{0} = a(y).\overline{b}\langle y\rangle.\mathbf{0} \; .$$

Names can occur free and bound in the same agent. Thus, for

$$A \overset{def}{=} y(z).x(y).\overline{y}\langle b\rangle.\mathbf{0}$$

it holds that $y \in fn(A)$ and $y \in bn(A)$. To avoid such homonyms, the condition $fn(P) \cap bn(P) = \emptyset$ should be used whenever applicable. Furthermore, we consider all free names of an agent to be its parameters and omit them except those changing in parametric recursion. Consider for instance

$$A(x) \overset{def}{=} \overline{a}\langle x\rangle.A(x) + b(y).A(y) \ ,$$

where a and b are omitted from the parameter list. The name x has been kept because it is required for parametric recursion (a and b are invariant in A).

Agents can interact via free names, e.g. if $x \in fn(P)$ then agent P can use x for interaction with another agent. The evolution of an agent with an input prefix $x(\tilde{z}).P$ applies a substitution of the names of $\tilde{z}$ with the received names in P.

Definition 2.3 (Substitution) A *substitution* is a function that maps names to names: $\sigma : \mathcal{N} \rightarrow \mathcal{N}$. □

A substitution σ with $\sigma(x) = y$ that maps x to y and $\sigma(z) = z, z \neq x$ representing identity for all other names is written as $\{^y/_x\}$. A number of names that are substituted instantly is denoted $\{^{\tilde{y}}/_{\tilde{x}}\}$, where the names of $\tilde{x}$ have to be pairwise distinct and $|\tilde{x}| = |\tilde{y}|$. The application of a substitution σ to an agent A is denoted as $A\sigma$. To avoid unintended captures of bound names (called *scope intrusion*), an α-conversion of bound names has to take place wherever required. If $P\{^w/_y\}$ should take place and w is bound in P so that the *new* w disturbs the scope of the *old* w, then w has to be changed to z beforehand ($z \notin fn(P) \cup bn(P)$). Since names are used as subjects and objects, substitution triggered by an input prefix is what actually implements link passing mobility in π-calculus. An example is given by

$$\overline{x}\langle a\rangle.P' \mid x(b).\overline{b}\langle c\rangle.Q' \ ,$$

where the left hand component has knowledge of a name a that is sent via x to the right hand component. In the second prefix of the right hand component, the received name is used as the subject of an output prefix. Hence, the right hand component has gained a new interaction link.

Bound occurrences of names introduced by the restriction operator νzP have two interesting properties. First, components of P can interact with each other via z. Second, components of P can *extrude* the scope of z to another agent by sending z via some name. For instance, in

$$\nu a \ (\overline{a}\langle b\rangle.a(x).\mathbf{0} \mid a(w).\overline{w}\langle a\rangle.\mathbf{0}) \mid b(y).\overline{y}\langle z\rangle.\mathbf{0}$$

the first and the second component can interact via a due to $fn(\overline{a}\langle b\rangle.a(x).\mathbf{0}) = \{a, b\}$ and $fn(a(w).\overline{w}\langle a\rangle.\mathbf{0}) = \{a\}$. The third component has no interaction with the other components yet. After an interaction of the first and the second component, the agent is given by:

$$\nu a \ (a(x).\mathbf{0} \mid \overline{b}\langle a\rangle.\mathbf{0}) \mid b(y).\overline{y}\langle z\rangle.\mathbf{0} \ .$$

Still, the occurrence of the name a is bound to the first and second component. However, the second component can communicate a via the free name b to the second component. The scope of a is extruded:

$$\nu a \ (a(x).\mathbf{0} \mid \mathbf{0} \mid \overline{a}\langle z\rangle.\mathbf{0}) \ .$$

Finally, also the third component can interact with the first component (omitted).

SC-ALPHA	P_1	$\equiv$	P_2, if $P_1 = P_2$
SC-MAT	$[x = x]\pi.P$	$\equiv$	$\pi.P$
SC-SUM-ASSOC	$M_1 + (M_2 + M_3)$	$\equiv$	$(M_1 + M_2) + M_3$
SC-SUM-COMM	$M_1 + M_2$	$\equiv$	$M_2 + M_1$
SC-SUM-INACT	$M + \mathbf{0}$	$\equiv$	M
SC-COMP-ASSOC	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$
SC-COMP-COMM	$P_1 \mid P_2$	$\equiv$	$P_2 \mid P_1$
SC-COMP-INACT	$P \mid \mathbf{0}$	$\equiv$	P
SC-RES	$\nu z\, \nu w\, P$	$\equiv$	$\nu w\, \nu z\, P$
SC-RES-INACT	$\nu z\, \mathbf{0}$	$\equiv$	$\mathbf{0}$
SC-RES-COMP	$\nu z\, (P_1 \mid P_2)$	$\equiv$	$P_1 \mid \nu z\, P_2$, if $z \notin fn(P_1)$
SC-UNFOLD	$A(\tilde{y})$	$\equiv$	$P\{^{\tilde{y}}/_{\tilde{x}}\}$, if $A(\tilde{x}) \overset{def}{=} P$

Table 2.1: The axioms of structural congruence.

2.2.2 Structural Congruence

Before we continue with a formal semantics for the evolution of agent terms, we require a definition of *structural congruence* between them. Structural congruence requires two prerequisites, namely *context* and *congruence*. Informally, a context is an agent with an expansion slot (hole) to add additional behavior given by another agent:

Definition 2.4 (Context) A *context* is an agent term with exactly one occurrence of a hole, denoted as $[\cdot]$, instead of a non-degenerated occurrence of $\mathbf{0}$. □

An occurrence of **0** is non-degenerated if it is not the left or right hand term in a sum. $C[P]$ denotes a context C with $[\cdot]$ replaced by agent P. The replacement is literal, which means that occurrence of names free in P may be bound in $C[P]$. For instance, let $C \overset{def}{=} \nu x\ (\overline{x}\langle a\rangle.\mathbf{0} \mid [\cdot])$, then $C[x(y).\mathbf{0}] = \nu x\ (\overline{x}\langle a\rangle.\mathbf{0} \mid x(y).\mathbf{0})$. Congruence is then given by:

Definition 2.5 (Congruence) An equivalence relation S on agents is a *congruence* if $(P, Q) \in S$ implies $(C[P], C[Q]) \in S$ for every context C. □

Structural congruence is a certain type of congruence:

Definition 2.6 (Structural Congruence) *Structural Congruence*, denoted as $\equiv$, is the smallest congruence on agents that obey to the axioms in table 2.1. The axioms of structural congruence allow the recasting of agent terms. □

SC-ALPHA relates α-convertible agents, whereas SC-MAT simply saves the introduction of a transition rule for match in the formal semantics given later on. For instance, using SC-MAT, SC-SUM-INACT, and SC-COMP-COMM, the following agents are structurally congruent:

$$a(b).\mathbf{0} \mid ([z = z]\overline{a}\langle z\rangle.\mathbf{0} + \mathbf{0}) \equiv (\overline{a}\langle z\rangle.\mathbf{0} \mid a(b).\mathbf{0})\ .$$

Structural congruence keeps commutativity and associativity for products and summations, as well as making **0** the identity element. More interestingly is the application of the axioms for restriction:

$$\nu x\ ((\overline{a}\langle x\rangle.\mathbf{0} + \overline{b}\langle x\rangle.\mathbf{0}) \mid \nu z\ \overline{y}\langle z\rangle.z(c).\mathbf{0}) \equiv \nu z\ (\nu x\ (\overline{a}\langle x\rangle.\mathbf{0} + \overline{b}\langle x\rangle.\mathbf{0}) \mid \overline{y}\langle z\rangle.z(c).\mathbf{0})\ .$$

Using SC-RES, SC-COMP-COMM, and SC-RES-COMP, the restriction operator νz has been brought to the top of the term, whereas νx moved inside. SC-RES-COMP realizes scope extrusion, as in

$$(\nu x\ \overline{a}\langle x\rangle.\mathbf{0}) \mid a(y).\mathbf{0} \equiv \nu x\ (\overline{a}\langle x\rangle.\mathbf{0} \mid a(y).\mathbf{0})\ ,$$

and in combination with α-conversion also scope intrusion, as in

$$(\nu x\ a(b).\overline{b}\langle x\rangle.\mathbf{0}) \mid \overline{a}\langle x\rangle.\mathbf{0} \equiv \nu x'\ (a(b).\overline{b}\langle x'\rangle.\mathbf{0} \mid \overline{a}\langle x\rangle.\mathbf{0})\ .$$

Finally, SC-UNFOLD can unfold terms realizing parametric recursion. For instance, an agent given by $A(x) \overset{def}{=} \nu a\ \overline{x}\langle a\rangle.A(x) + \tau.\mathbf{0}$ will unfold as a term in $x(y).A(y)$ as follows:

$$x(y).A(y) \equiv x(y).(\nu a\ \overline{y}\langle a\rangle.A(y) + \tau.\mathbf{0})\ .$$

2.2.3 Reduction Semantics

We now make the informal semantics of the π-calculus explicit by introducing a *reduction* relation on agent terms. Formally, the semantics is based on a transition system:

Definition 2.7 (Transition System) A *transition system* is defined as a pair $(S, \longrightarrow)$ with:

- S is a set of states and
- $\longrightarrow \subseteq S \times S$ is a transition relation. □

The set of states is given by the grammar according to equation 2.2. The idea behind the transition relation $\longrightarrow$ is that an agent P can evolve to P', denoted as $P \longrightarrow P'$, as a result of an *intraaction* between components of P. Thus, we only cover internal actions of an agent. The reduction relation is given by a set of inference rules (equations 2.4–2.8) that make use of structural congruence. Inference rules are composed out of *premises* and a *conclusion*. If the premises are fulfilled, the conclusion is also valid. Inference rules are written in the form:

$$\frac{\text{Premises}}{\text{Conclusion}}\ .$$

If the set of premises is empty, the conclusion is denoted as *axiom*. Regarding reduction, the key rule is an axiom:

$$(\overline{x}\langle\tilde{y}\rangle.P + M) \mid (x(\tilde{z}).Q + N) \longrightarrow P \mid Q\{^{\tilde{y}}/_{\tilde{z}}\} \text{ with } |\tilde{y}| = |\tilde{z}|\ . \qquad (2.4)$$

The axiom states that two components made up of sums can interact via a name x. If the intraaction takes place, M and N are discarded, the prefixes before P and Q are removed, and the names $\tilde{z}$ in Q are substituted with $\tilde{y}$. Interestingly, if an intraaction with an agent is possible, it can always be brought in a form resembling axiom 2.4 via the axioms of structural congruence from table 2.1. The corresponding inference rule is given by

$$\frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'} \; . \tag{2.5}$$

Consider for instance an agent

$$a(x).A \mid \overline{a}\langle b\rangle.B$$

that has to be brought into a form corresponding to axiom 2.4 to derive a reduction. By using SC-COMP-COMM the order of the components can be flipped and via SC-SUM-INACT the required sums can be added:

$$a(x).A \mid \overline{a}\langle b\rangle.B \equiv \overline{a}\langle b\rangle.B \mid a(x).A \equiv (\overline{a}\langle b\rangle.B + \mathbf{0}) \mid (a(x).A + \mathbf{0}) \; .$$

Since the form matches the axiom, a reduction is possible:

$$(\overline{a}\langle b\rangle.B + \mathbf{0}) \mid (a(x).A + \mathbf{0}) \longrightarrow B \mid A\{^b/_x\} \; .$$

By applying SC-COMP-COMM again, the expected result can also be denoted as

$$a(x).A \mid \overline{a}\langle b\rangle.B \longrightarrow A\{^b/_x\} \mid B \; ,$$

giving the intuitive expected behavior.

Beside intraactions inside an agent, also internal actions denoted by τ as given in equation 2.1 are possible. The formal behavior is captured in a second axiom:

$$\tau.P + M \longrightarrow P \; . \tag{2.6}$$

The axiom states that a sum with a τ prefix at the left term can reduce to P and discard M. Due to rule 2.5, SC-SUM-COMM can be used to flip the terms of the sum:

$$x(y).A + \tau.B \longrightarrow B \; .$$

A second inference rule considers the parallel composition of agent terms:

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \; . \tag{2.7}$$

The rule states that a component P can evolve independently of another component Q if P has a reduction. Consider

$$(\overline{u}\langle w\rangle.A \mid u(x).B) \mid \tau.C \longrightarrow (A \mid B\{^w/_x\}) \mid \tau.C \; ,$$

that can be reduced because the left hand term has an intraaction according to axiom 2.4 and rule 2.5.

The reduction semantics is completed with a third inference rule, covering restrictions:

$$\frac{P \longrightarrow P'}{\nu z\; P \longrightarrow \nu z\; P'} \; . \tag{2.8}$$

The rule states that a restriction of a name above an agent term does not inhibit a reduction. The restricted name can even be used as subject of an intraaction:

$$\nu a\; (\overline{a}\langle b\rangle.A + \mathbf{0}) \mid (a(x).B + \mathbf{0}) \longrightarrow \nu a\; (A \mid B\{^b/_x\}) \; .$$

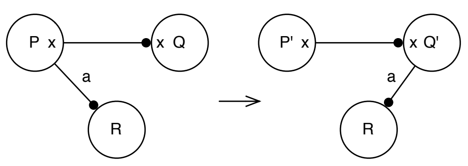


Figure 2.2: Flow graph example.

2.2.4 Flow Graphs

A graphical representation of π-calculus agents is given by an informal usage of Milner's flow graphs [91] as introduced in [99]. A flow graph is a certain kind of graph, where nodes represent agents and arcs represent communication links between them. Figure 2.2 shows a system composed of three agents before and after a reduction. The corresponding agents are given by

$$\nu x\ (P \mid Q) \mid R \text{ with } P \overset{def}{=} b(z).\mathbf{0} + \overline{x}\langle a\rangle.P', Q \overset{def}{=} x(y).Q', \text{ and } R \overset{def}{=} \overline{a}.R'\ ,$$

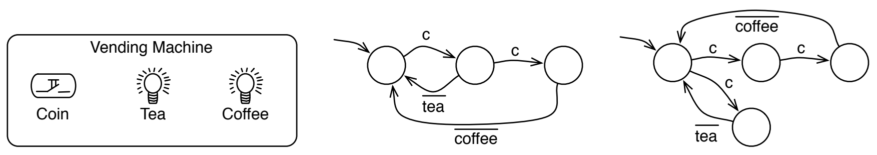
where only P and Q are intraacting and evolve to P' and Q', with $a \notin fn(P') \cup fn(Q)$. Nodes are denoted as circles with the name of the agent inside, where a hierarchical order might be kept (i.e. agents consisting of more than one component might be collapsed or expanded). Circles representing agents are connected using lines, where a dotted end denotes the target node. A line is drawn from each node representing an agent with an output prefix to another node representing an agent with a matching input prefix. Bound names are written inside the circle that represents the corresponding agent, as near as possible to the connecting edge. Free names are written as labels along the edges. In any case, it is possible to only show important names and agents. For instance, P can behave as shown, but additionally includes the name b as an input prefix that is not contained in the flow graph.

2.3 Bisimulation

In this section, equivalences between agents based on their external observable behavior are introduced. These are denoted as *bisimulation equivalences* or *bisimilarities*. If two agents are related by a bisimulation, they match each others transitions in a way that cannot be distinguished by an external observer. The informal meaning of bisimulation equivalence can be given as follows:

> Let P and Q be two related agents. If P can evolve to P', then also Q must be able to evolve to Q' such that P' and Q' are again related. If the same holds for the opposite direction, starting from Q, the two agents are called *bisimilar* or *bisimulation equivalent*.

Bisimulation was first mentioned by Park in [107], based on Milner's work on simulation [90]. According to Milner's extended work [94], bisimulation has its root in standard automata theory. See for instance figure 2.3(a), which shows a tea and coffee vending machine. The user



(a) External View. (b) Internal Automaton A. (c) Internal Automaton B.

Figure 2.3: A vending machine.

can insert coins (represented by the pushbutton) and receive either tea or coffee (represented by the bulbs). Furthermore, two different versions of the internal automaton are shown. The alphabet of both automata is made up of the transitions $Act_V = \{c, \overline{tea}, \overline{coffee}\}$. C represents an external input to the vending machine, i.e. the insertion of a coin, whereas $\overline{tea}$ and $\overline{coffee}$ represent an external output of the vending machine, i.e. the products. Both automata deliver tea for the insertion of one coin, and coffee for the insertion of two coins. Regarding automata theory, both accept the same language and are thus behavioral equivalent (omitted, see [97]). Regarding a thirsty user, both are quite different. While variant A shows a deterministic behavior, variant B acts non-deterministic! Instead of analyzing traces as in standard automata theory, bisimulation contains a stronger equivalence criterion, since the current actions are taken into account. Concerning the example, both automata for the vending machine can be proven not to be bisimulation equivalent according to the informal definition stated above.

Example 2.1 (Vending machines) Variant A of the vending machine is not bisimulation equivalent to variant B. Proof by counterexample:

1. Let $A \overset{def}{=} c.(\overline{tea}.A + c.\overline{coffee}.A)$ and $B \overset{def}{=} c.\overline{tea}.B + c.c.\overline{coffee}.B$ according to figure 2.3.
2. Now $A \xrightarrow{c} \overline{tea}.A + c.\overline{coffee}.A$, while B has a non-deterministic choice when mimicking the interaction, e.g. $B \xrightarrow{c} \overline{tea}.B$.
3. Finally, the remainder of A accepts another interaction $\overline{tea}.A + c.\overline{coffee}.A \xrightarrow{c} \overline{coffee}.A$ that the remainder of B is unable to mimic. □

2.3.1 LTS Semantics

The reduction semantics given in the previous section does not describe the external observable behavior of the π-calculus agents. Axiom 2.4 only describes internal actions (i.e. intraactions) of an agent. Since these actions are internal, the only external observation that can be made at most is the fact that something has happened. This something corresponds to an internal action denoted as τ. If we want to express that an agent has the capability to receive an input from an (arbitrary) environment, work on the input, and finally provide a result back to the environment, reduction semantics is not sufficient. An environment can be thought of as some kind of context where the agent is placed within. Consider for instance

$$\nu y\ i(x).\tau.\overline{o}\langle y\rangle.\mathbf{0}\ .$$

According to reduction semantics, this agent cannot evolve. Nevertheless, it contains possible communications with an environment via i and $\overline{o}$. Adding support for these kinds of interactions requires a differentiation between the actions that can occur. Beside internal actions, also input and output actions should be observable. The observation of different actions is made possible with a labeled transition system (LTS) semantics that bears the actions as labels:

Definition 2.8 (Labeled Transition System) A *labeled transition system* is defined as a three-tuple $(S, T, \xrightarrow{t})$ with:

- S is a set of states,
- T is a set of transition labels, and
- $\xrightarrow{t} \subseteq S \times S$ is a family of binary transition relations for each $t \in T$. $\square$

The set of states is given by the grammar according to equation 2.2. The set of transition labels, called actions, is derived from the prefixes.

Definition 2.9 (Actions) The *actions* α of the π-calculus are given by:

$$\alpha ::= \overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle \mid x(\tilde{y}) \mid \tau \; ,$$

where Act denotes the set of actions and $\tilde{z} \subseteq \tilde{y}$. $\square$

The first action corresponds to the output prefix, where the objects $\tilde{y}$ are sent via the subject x. The objects can be restricted names, denoted as $\nu\tilde{z}$, inside the tuple $\tilde{y}$. In this case scope extrusion takes place. The second action corresponds to the input prefix, where the objects $\tilde{y}$ are received via the subject x. The third action denotes an internal, unobservable action. The names contained in an action are given by $n(\alpha)$ and the bound names by $bn(\alpha)$:

$$n(\alpha) = \begin{cases} \alpha = \overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle & : \; \{x, \tilde{y}, \tilde{z}\} \\ \alpha = x(\tilde{y}) & : \; \{x, \tilde{y}\} \\ \alpha = \tau & : \; \emptyset \end{cases} \quad \text{and } bn(\alpha) = \begin{cases} \alpha = \overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle & : \; \{z\} \\ \alpha = x(\tilde{y}) & : \; \{\tilde{y}\} \\ \alpha = \tau & : \; \emptyset \end{cases} .$$

The semantics for the agents, i.e. how they can evolve, is given by the transition relations.

Definition 2.10 (Transition Relations) The *transition relations* $\xrightarrow{\alpha}$ of the π-calculus, with $\alpha \in Act$, are given by the rules in figure 2.4. $\square$

Rule STRUCT explicitly includes the axioms of structural congruence into the semantics, since they simplify the transition rules. PREFIX requires a special treatment of input transitions such as

$$P \xrightarrow{a(x)} P' \; .$$

Due to the *late* semantics, x does not denote the value received, but rather locates the places in P' where x will appear. An alternative rule with explicit substitution, such as

$$\frac{}{a(x).P \xrightarrow{au} P'\{^u/_x\}} \; ,$$

$$\textsc{Struct}\ \frac{Q \equiv P \qquad P \longrightarrow P' \qquad P' \equiv Q'}{Q \longrightarrow Q'} \qquad \textsc{Prefix}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \textsc{Sum}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\textsc{Par}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ (bn(\alpha)\cap fn(Q)=\emptyset) \qquad \textsc{Comm}\ \frac{P \xrightarrow{\overline{x}\langle\tilde{y}\rangle} P' \qquad Q \xrightarrow{x(\tilde{z})} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{^{\tilde{y}}/_{\tilde{z}}\}}\ (|\tilde{y}|=|\tilde{z}|)$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P'}{\nu z\, P \xrightarrow{\alpha} \nu z\, P'}\ (z \notin n(\alpha)) \qquad \textsc{Open}\ \frac{P \xrightarrow{\overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle} P'}{\nu a\, P \xrightarrow{\overline{x}\langle(\nu a\tilde{z})\tilde{y}\rangle} P'}\ (x\neq a \wedge a\notin\tilde{z} \wedge a\in\tilde{y})$$

Figure 2.4: The π-calculus transition rules.

would give a slightly different semantics that will not be discussed further. Using Prefix,

$$a(x).\overline{x}\langle y\rangle.\mathbf{0} \xrightarrow{a(u)} \overline{u}\langle y\rangle.\mathbf{0} \xrightarrow{\overline{u}\langle y\rangle} \mathbf{0}$$

evolves to inaction. Furthermore, using Struct, Prefix, and Sum,

$$\overline{b}\langle u\rangle.\mathbf{0} + \tau.a(x).\mathbf{0} \xrightarrow{\tau} a(x).\mathbf{0}$$

evolves to the right hand side of the summation. The corresponding derivation tree of the preceding transition is:

$$\textsc{Struct}\ \frac{\textsc{Sum}\ \dfrac{\tau.a(x).\mathbf{0} \xrightarrow{\tau} a(x).\mathbf{0}}{\tau.a(x).\mathbf{0} + \overline{b}\langle u\rangle.\mathbf{0} \xrightarrow{\tau} a(x).\mathbf{0}}}{\overline{b}\langle u\rangle.\mathbf{0} + \tau.a(x).\mathbf{0} \xrightarrow{\tau} a(x).\mathbf{0}}\ .$$

Rule Par has the side condition that Q does not contain a name that is bound in α. In $P \mid Q \xrightarrow{\alpha} P' \mid Q$ the action should not refer to any occurrence of a name in Q. Hence, an inference

$$\textsc{Par}\ \frac{a(x).P \xrightarrow{a(x)} P}{a(x).P \mid Q \xrightarrow{a(x)} P \mid Q}$$

combined with an output $\overline{a}\langle u\rangle.R \xrightarrow{\overline{a}\langle u\rangle} R$ using Comm

$$\textsc{Comm}\ \frac{\overline{a}\langle u\rangle.R \xrightarrow{\overline{a}\langle u\rangle} R \qquad (a(x).P \mid Q) \xrightarrow{a(x)} (P \mid Q)}{\overline{a}\langle u\rangle.R \mid (a(x).P \mid Q) \xrightarrow{\tau} R \mid (P \mid Q)\{^{u}/_{x}\}}$$

is only valid if $x \notin fn(Q)$ (according to Par), because otherwise the substitution $\{^{u}/_{x}\}$ might affect a free x in Q. To make an interaction possible if $x \in fn(Q)$, the bound name x of P has to be α-converted first. Still, COMM does not support scope extrusion directly. A possible solution is using STRUCT to bring the restriction to the top of the term before applying RES and COMM:

$$\text{STRUCT}\ \dfrac{\text{RES}\ \dfrac{\text{COMM}\ \dfrac{\overline{a}\langle u\rangle.P \xrightarrow{\overline{a}\langle u\rangle} P \qquad a(x).Q \xrightarrow{a(x)} Q}{\overline{a}\langle u\rangle.P \mid a(x).Q \xrightarrow{\tau} P\{^u/_x\} \mid Q}}{\nu u\ (\overline{a}\langle u\rangle.P \mid a(x).Q) \xrightarrow{\tau} \nu u\ (P \mid Q\{^u/_x\})}}{(\nu u\ \overline{a}\langle u\rangle.P) \mid a(x).Q \xrightarrow{\tau} \nu u\ (P \mid Q\{^u/_x\})} \quad .$$

The derivation tree above shows how bound output actions can be bypassed, while scope extrusion by communicating bound names is still possible. This is due to RES, where all interactions using COMM are valid, since $n(\tau)$ is always empty. Rule OPEN describes the output of bound names. An application is given by:

$$\text{OPEN}\ \dfrac{\overline{a}\langle u\rangle.P \xrightarrow{\overline{a}\langle u\rangle} P}{\nu u\ \overline{a}\langle u\rangle.P \xrightarrow{\overline{a}\langle \nu u\rangle} P} \ .$$

OPEN is required to capture the special case of exporting bound names. This case is required for open d-bisimulation introduced later on.

2.3.2 Ground Bisimulation

We are now prepared to give a formal definition of bisimulation for agent terms. To distinguish different kinds, we denote the basic bisimulation as ground bisimulation:

Definition 2.11 (Ground Bisimulation) A *ground bisimulation* is a symmetric, binary relation $\mathcal{R}$ on agents such that $\forall \alpha \in Act$:

$$P\mathcal{R}Q \wedge P \xrightarrow{\alpha} P' \Rightarrow \exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q' \text{ with } bn(\alpha) \cap (fn(P) \cup fn(Q)) = \emptyset \ .$$

P and Q are ground bisimilar, denoted as $P \sim Q$, if they are related by a ground bisimulation. □

Furthermore, bisimulation is an equivalence relation, hence $P \sim P$, $P \sim Q \Rightarrow Q \sim P$, and $P \sim Q \wedge Q \sim R \Rightarrow P \sim R$ hold. A proof can be found in [97]. Ground bisimulation considers a *strong* relation between interactions and unobservable actions. Two agents

$$P \overset{def}{=} a(x).\tau.\tau.\overline{b}\langle z\rangle.\mathbf{0} \text{ and } Q \overset{def}{=} a(x).\tau.\overline{b}\langle z\rangle.\mathbf{0}$$

are not bisimulation equivalent, since they differ in the number of their unobservable actions (τ transitions). A bisimulation that abstracts from these unobservable actions is called *weak* bisimulation. Weak bisimulations are of particular interest, since they abstract from the internal behavior of agents and instead only consider the external visible behavior. A weak bisimulation is obtained by defining $\Longrightarrow$ to represent zero or more τ transitions, i.e. $\xrightarrow{\tau}^*$, $\overset{\alpha}{\Longrightarrow}$ as

$\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and $\overset{\hat{\alpha}}{\Longrightarrow}$ as $\overset{\alpha}{\Longrightarrow}$ if $\alpha \neq \tau$ and $\Longrightarrow$ if $\alpha = \tau$.

Definition 2.12 (Weak Ground Bisimulation) A *weak ground bisimulation* is a symmetric, binary relation $\mathcal{R}$ on agents such that $\forall \alpha \in Act$:

$$P\mathcal{R}Q \wedge P \xrightarrow{\alpha} P' \Rightarrow \exists Q' : Q \overset{\hat{\alpha}}{\Longrightarrow} Q' \wedge P'\mathcal{R}Q' \text{ with } bn(\alpha) \cap (fn(P) \cup fn(Q)) = \emptyset\ .$$

P and Q are weak ground bisimilar, denoted as $P \approx Q$, if they are related by a weak ground bisimulation. □

The differences between the *strong* and weak kinds of bisimulation are shown by example. Consider for instance the agents

$$A \overset{def}{=} i.\overline{o}.\mathbf{0} \text{ and } B \overset{def}{=} \nu y\ (i.\overline{y}.\mathbf{0} \mid y.\overline{o}.\mathbf{0})\ .$$

Both agents are not (strong) ground bisimilar since B has an additional τ transition. They are, however, weak ground bisimilar due to the abstraction from τ transitions. To show that two agents are not bisimilar, we need to find a counterexample that considers all possibilities:

Proof 2.1 ($A \not\sim B$) By counterexample.

1. $A \xrightarrow{i} \overline{o}.\mathbf{0}$, mimicked by $B \xrightarrow{i} \nu y\ (\overline{y}.\mathbf{0} \mid y.\overline{o}.\mathbf{0})$.
2. The remainder of B continues with $\nu y\ (\overline{y}.\mathbf{0} \mid y.\overline{o}.\mathbf{0}) \xrightarrow{\tau} \nu y\ \overline{o}.\mathbf{0}$.

Since the remainder of A, $\overline{o}.\mathbf{0}$, is unable to mimic this transition, $A \not\sim B$ holds. □

To show that A and B are related according to weak ground bisimulation, we need to to find a relation $\mathcal{R}$.

Proof 2.2 ($A \approx B$) By enumeration of $\mathcal{R}$ with $(A, B) \in \mathcal{R}$.

$$\mathcal{R} = \{(\mathbf{0},\mathbf{0}), (i.\overline{o}.\mathbf{0}, \nu y\ (i.\overline{y}.\mathbf{0} \mid y.\overline{o}.\mathbf{0})), (\overline{o}.\mathbf{0}, \overline{o}.\mathbf{0}), (\overline{o}.\mathbf{0}, \nu y\ (\overline{y}.\mathbf{0} \mid y.\overline{o}.\mathbf{0}))\}$$

Since $\mathcal{R}$ is symmetric, $A \approx B$ holds. □

In certain situations, we only require one direction of a bisimulation, called *simulation*. A simulation is a one-way investigation of two agents P and Q. If Q is able to match all transitions of P, then Q simulates P. By removing the property of symmetry from the weak ground bisimulation definition, a weak ground simulation is given by:

Definition 2.13 (Weak Ground Simulation) A *weak ground simulation* is a binary relation $\mathcal{R}$ on agents such that $\forall \alpha \in Act$:

$$P\mathcal{R}Q \wedge P \xrightarrow{\alpha} P' \Rightarrow \exists Q' : Q \overset{\hat{\alpha}}{\Longrightarrow} Q' \wedge P'\mathcal{R}Q' \text{ with } bn(\alpha) \cap (fn(P) \cup fn(Q)) = \emptyset\ .$$

Q is weak ground similar to P, denoted as $P \precapprox Q$, if they are related by a weak ground simulation. □

We use the term *ground* to denote that only the subjects of the agents are covered correctly. Since an input action denotes a placeholder for the objects to be received, a substitution has to take place in the bisimulation game. For instance, two agents given by

$$P \overset{def}{=} a(x).P + a(x).\mathbf{0} \text{ and } Q \overset{def}{=} a(x).Q + a(x).[x = u]\tau.Q \;, \tag{2.9}$$

are bisimulation equivalent, $P \sim Q$, since x in the match prefix will never be substituted. To overcome this problem, different variants of bisimulation for the π-calculus have been developed. The most recent of them is called *open bisimulation* and will be discussed in the next subsection.

2.3.3 Open Bisimulation

Open bisimulation was introduced by Sangiorgi in [116]. It includes a quantification over all substitutions in the bisimulation definition to provide a congruence, i.e. make it work in arbitrary contexts. For a π-calculus variant without restriction, it is defined as follows:

Definition 2.14 (Open Bisimulation) An *open bisimulation* for a π-calculus variant without restriction is a symmetric, binary relation $\mathcal{R}$ on agents such that $\forall \alpha \in \mathit{Act}$ and $\forall \sigma$:

$$P \mathcal{R} Q \wedge P\sigma \xrightarrow{\alpha} P' \Rightarrow \exists Q' : Q\sigma \xrightarrow{\alpha} Q' \wedge P' \mathcal{R} Q' \;.$$

P and Q are open bisimilar, denoted as $P \sim_O Q$, if they are related by an open bisimulation. □

Open bisimulation does not contain a special treatment of input actions, since quantification over substitutions occurs for every transition. If $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\alpha} Q'$, the requirement above already states that $P'\sigma$ must be simulated by $Q'\sigma$ for all substitutions σ in the next step of the bisimulation game. Hence, the agents given in equation 2.9 are not open bisimilar. After $\xrightarrow{a(x)}$ occurred (and the right side of the sum has been chosen in both agents), a substitution $\{^u/_x\}$ enables further transitions in the remainder of B that cannot be mimicked by the remainder of A.

Open bisimulation is defined for a calculus without restriction, because a bound output action causes problems. For instance,

$$P \overset{def}{=} \nu x\, \overline{a}\langle x\rangle.[x = y]\tau.\mathbf{0} \text{ and } Q \overset{def}{=} \nu x\, \overline{a}\langle x\rangle.\mathbf{0}$$

should be open bisimilar, since x is distinct from all free names of P. However, they evolve to $P \xrightarrow{\overline{a}\langle \nu x\rangle} [x = y]\tau.\mathbf{0}$ and $Q \xrightarrow{\overline{a}\langle \nu x\rangle} \mathbf{0}$. Obviously, for a substitution $\{^x/_y\}$, both are not equivalent. The substitution is possible, since the bound output action removed the restriction νx. Anyhow, the name x is local to P, so it should never be equal to y. Therefore a list of names that will never be equal is required. This list is kept in the form of a *distinction* that relates names that will always be distinct.

Definition 2.15 (Distinction) A *distinction* is a finite, symmetric, irreflexive, and binary relation on names. □

Distinctions are ranged over by $\mathcal{D}$. A substitution σ respects a distinction $\mathcal{D}$ if $(a,b) \in \mathcal{D} \Rightarrow \sigma(a) \neq \sigma(b)$. If a substitution σ respects a distinction $\mathcal{D}$, then $D\sigma$ is the relation $\{(\sigma(a), \sigma(b)) : (a,b) \in \mathcal{D}\}$. Using distinctions, open bisimulation including the restriction operator can be defined:

Definition 2.16 (Open D-Bisimulation) An *open d-bisimulation* is a distinction-indexed family of a set of symmetric, binary relations $\mathcal{R}_{\mathcal{D}}$ on agents such that $\forall \alpha \in Act$ and $\forall \sigma$ respecting $\mathcal{D} : P\mathcal{R}_D Q \wedge P\sigma \xrightarrow{\alpha} P'$ and $bn(\alpha) \cap (fn(P\sigma) \cup fn(Q\sigma)) = \emptyset \Rightarrow$

1. If $\alpha = \overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle$ then $\exists Q' : Q\sigma \xrightarrow{\overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle} Q' \wedge P'\mathcal{R}_{D'}Q'$
where $D' = \mathcal{D}\sigma \cup \{\{z\} \times (fn(P\sigma) \cup fn(Q\sigma))\} \cup \{(fn(P\sigma) \cup fn(Q\sigma)) \times \{z\}\}$ for all z of $\tilde{z}$

2. else $\exists Q' : Q\sigma \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}_{\mathcal{D}\sigma}Q'$.

P and Q are open d-bisimilar, denoted as $P \sim_O^D Q$, if they are related by an open d-bisimulation. □

D' represents an extension of $\mathcal{D}$ by making the bound names $\tilde{z}$ of the output prefix $\overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle$ distinct to all free names of $P\sigma$ and $Q\sigma$ (clause (1) of the definition). Using open d-bisimulation, the agents

$$P \overset{def}{=} \nu x\ \overline{a}\langle x\rangle.[x=y]\tau.\mathbf{0} \text{ and } Q \overset{def}{=} \nu x\ \overline{a}\langle x\rangle.\mathbf{0}$$

are equivalent, since after $P \xrightarrow{\overline{a}\langle\nu x\rangle} [x=y]\tau.\mathbf{0}$ and $Q \xrightarrow{\overline{a}\langle\nu x\rangle} \mathbf{0}$, the remainder of $P \equiv \mathbf{0}$. This is due to distinction D that has become $D = \{(x,y),(y,x)\}$ in the transition $\xrightarrow{\overline{a}\langle\nu x\rangle}$ (regarding clause (1) of definition 2.16). Since open d-bisimulation states that all substitutions σ respect D, a substitution $\{^x/_y\}$ is not possible and hence $[x=y]\tau.\mathbf{0}$ cannot evolve further. Also, two agents

$$A \overset{def}{=} \nu x\ \overline{a}\langle x\rangle.b(y).[x=y]\tau.\mathbf{0} \text{ and } B \overset{def}{=} \nu x\ \overline{a}\langle x\rangle.b(y).\mathbf{0}$$

are open d-bisimilar ($A \sim_O^D B$). While it might be excepted that the bound name x sent via a can possible received again via b and the match prefix will become true, this can never happen due to the addition of x to the distinction. Thus, special care has to be taken in object-based evaluation inside agents.

A weak version of open d-bisimulation as well as weak open d-simulation are acquired accordingly:

Definition 2.17 (Weak Open D-Bisimulation) An *weak open d-bisimulation* is a distinction-indexed family of a set of symmetric, binary relations $\mathcal{R}_{\mathcal{D}}$ on agents such that $\forall \alpha \in Act$ and $\forall \sigma$ respecting $\mathcal{D} : P\mathcal{R}_D Q \wedge P\sigma \xrightarrow{\alpha} P'$ and $bn(\alpha) \cap (fn(P\sigma) \cup fn(Q\sigma)) = \emptyset \Rightarrow$

1. If $\alpha = \overline{x}\langle(\nu\tilde{z})\tilde{y}\rangle$ then $\exists Q' : Q\sigma \overset{\hat{\alpha}}{\Longrightarrow} Q' \wedge P'\mathcal{R}_{D'}Q'$
where $D' = \mathcal{D}\sigma \cup \{\{z\} \times (fn(P\sigma) \cup fn(Q\sigma))\} \cup \{(fn(P\sigma) \cup fn(Q\sigma)) \times \{z\}\}$ for all z of $\tilde{z}$

2. else $\exists Q' : Q\sigma \overset{\hat{\alpha}}{\Longrightarrow} Q' \wedge P'\mathcal{R}_{\mathcal{D}\sigma}Q'$.

P and Q are open d-bisimilar, denoted as $P \approx^D_O Q$, if they are related by a weak open d-bisimulation. $\square$

Definition 2.18 (Weak Open D-Simulation) A *weak open d-simulation* is acquired by removing the property of symmetry from definition 2.17 (Weak Open D-Bisimulation). An agent Q is open d-similar to P, denoted as $P \precapprox^D_O Q$, if they are related by a weak open d-simulation. $\square$

Chapter 3

Business Process Management

This chapter introduces business process management. It starts with discussing key concepts that are further on related to workflow and service-oriented architectures. Workflow is the traditional term for business processes executed and managed by computers and service-oriented architectures are a central realization technology for business process management.

Definition 3.1 (Business Process Management) *Business process management* (BPM) refers to an integrated set of activities for designing, enacting, managing, analyzing, optimizing, and adapting computerized business processes. □

Business process management sets the focus on business processes. As stated in chapter 1, this thesis focuses on design and verification of business processes. When presuming a process for now as a completely closed, timely and logical sequence of activities, we can define a business process.

Definition 3.2 (Business Process) A *business process* is a process that creates a value or result for a customer. It is directed by the business objectives of a company and by the business environment. □

Business objectives and the environments of business processes require the addition of business related attributes. Examples are: *Who* executes certain activities? *How* are certain activities executed? A detailed discussion follows along the lines of this chapter starting with the introduction of key concepts. The first concept is given by the activities of a business process.

Definition 3.3 (Activity) An *activity* is a piece of work to be done. An activity is also denoted as a *task*. □

It can be, for instance, a manual activity like phoning someone, writing a letter, etc., or an automated activity, like invoking a script or computer program. An activity can also be a decision, e.g. between two further activities, or another situation like waiting for previous activities to finish, e.g. a bus driver waiting for at least three passenger to enter the bus.

Definition 3.4 (Activity Instance) An *activity instance* is a concrete realization of an activity. □

Examples of an activity instance are actually phoning Mr. Smith, actually waiting for three

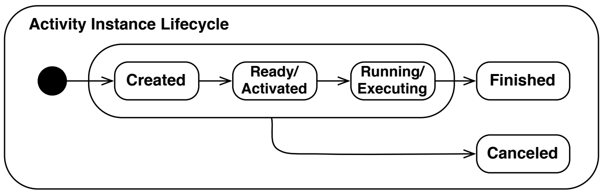


Figure 3.1: The lifecycle of an activity instance.

passengers, etc.

Definition 3.5 (Activity Instance Lifecycle) An *activity instance lifecycle* defines the states an activity instance can have. The possible states and transitions are shown in figure 3.1. □

Exemplary, this means for an activity instance of phoning someone: Discover the idea to phone Mr. Smith (created), fulfill all preconditions as finding phone number, get relevant documents on your desk (ready or activated), make the call (running/executing), and finally you're done (finished). At all stages you have the possibility to cancel your activity instance (cancel). When phoning Mr. Smith is on your to-do-list (created), you can remove this item, e.g. if you have not all documents at hand. After you have prepared everything (ready), you can decide to cancel the call. Even while phoning with Mr. Smith (running) you can cancel by simply hanging up in the middle of the call.

Definition 3.6 (Control Flow) *Control flow* defines temporal execution dependencies between activities. □

An example is writing a letter and thereafter sending it. Control flow relations are written as tuples of activities, e.g. (*Write Letter*, *Send Letter*). We assume transitivity of control flow relations, but not symmetry and reflexivity. After having defined a "sequence" of activities by control flow as well as activity itself, we can refine the definition of a process.

Definition 3.7 (Process) A *process* is a set of activities related by control flow. □

Example 3.1 (Credit Broker Process) An example is a credit broker process that finds the lowest interests for a given credit request. It might consist of the activities (A) *Receive Credit Request*, (B) *Process Credit Request*, and (C) *Show Results*. The dependencies are straightforward: A has to happen before B and B has to be finished before C. Accordingly, the control flow relations are given by (A, B) and (B, C). To denote that activity B can be executed several times (e.g. querying different banks), we add the control flow relation (B, B). Note that (A, C) is given by the transitivity of control flow.

Definition 3.8 (Process Instance) A *process instance* is the concrete realization of a business process. A process instance is also denoted as a *case*. □

Examples of a process instance are the actual processing of an insurance claim from Mr. Smith or buying a house including several steps.

Definition 3.9 (Process Instance Lifecylce) A *process instances lifecycle* defines the states a

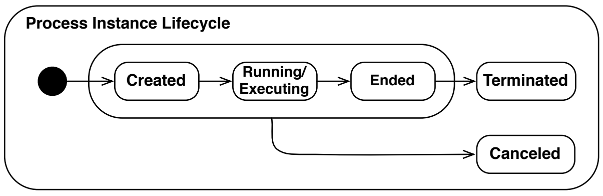


Figure 3.2: The lifecycle of a process instance.

process instance can have. The possible states and transitions are shown in figure 3.2. □

In the first state (created), all activities of a business process are converted to activity instances with the activity instance lifecycle state created. In the second state (running/executing), activity instances are activated, executed, finished, or canceled. The third state (ended) is reached if the result of the business process is provided by a certain activity instance. Other activity instances, doing clean-up or additional work, can still be activated, executed, finished, or canceled. If all activity instances of a process instance are in the state finished or canceled, the process process instance lifecycle is in the state terminated. Furthermore, a process instance can be canceled while it is not being terminated. In this case, all activity instances of the process instance are canceled immediately if they have not been finished already. The distinction between the states *ended* and *terminated* is sometimes blurred if the result of the business process is provided the moment the last activity instance finishes. In this case, the state terminated is reached immediately.

Definition 3.10 (Complex Activity) A *complex activity* is an activity consisting of a process. □

Throughout this thesis we further on abstract from complex activities by expanding them syntactically into the surrounding process.

Definition 3.11 (Interaction Flow) *Interaction flow* defines temporal dependencies between activities of different processes. □

An example is sending and receiving a letter. Interaction flow relations are written as tuples of activities from different processes (*Send Letter*, *Receive Letter*).

Definition 3.12 (Interaction) An *interaction* is given by a set of processes related by interaction flow. □

Example 3.2 (Credit Broker and Customer Interaction) An example interaction is given by the credit broker process from example 3.1 and a customer process. The customer has the activities (D) *Ask for Credit Offer* and (E) *Read Credit Offer* with the single relation D before E, formally: (D, E). The credit broker and the customer need to synchronize their processes using interaction flow from activity D to A, and from C to E. Thus, the interaction flows are given by the tuples (D, A) and (C, E) and the complete interaction is given by the processes of the credit broker and the customer as well as the interaction flows.

Definition 3.13 (Abstract Process) An *abstract process* is a process that contains only the

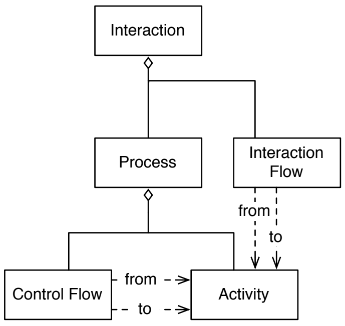


Figure 3.3: Relations between the key concepts.

activities and (combined) control flow relations that are required for an interaction. □

Considering example 3.2, the abstract process of the credit broker contains only the activities A and C with a combined control flow relation (A, C) when engaged inside an interaction with the customer.

Figure 3.3 shows the relations between the key concepts. A side condition is given by forbidding interaction flows between activities of the same process. The formal representation of processes and control flow will be discussed in detail in chapter 5 (Processes). The formal representation of interactions and interaction flows will be discussed afterward in chapter 6 (Interactions). Beforehand, we introduce existing work from the areas of workflow and service-oriented architectures.

3.1 Workflow

The traditional term for business processes enacted by computers is workflow [63]. A workflow describes business processes at a conceptual level required for understanding, communicating, and re-designing but also captures requirements for information systems and humans enacting the workflow. Hollingsworth of the Workflow Management Coalitions (WfMC) defines a workflow as follows:

Definition 3.14 (Workflow) *Workflow* defines *"the computerised facilitation or automation of a business process, in whole or part"* [73]. □

Closely related to workflow is the term workflow management system:

Definition 3.15 (Workflow Management System) A *Workflow Management System* (WfMS) is *"a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic"* [73]. □

A workflow is composed of different aspects, called workflow perspectives. These follow the divide and conquer approach to support separation of concerns.

Basic Control Flow Patterns	**Advanced Control Flow Patterns**	**Multiple Instance Patterns**
Sequence	Multi-choice	MI without Synchronization
Parallel Split	Synchronizing Merge	MI with a priori Design Time Knowledge
Synchronization	Multi-merge	MI with a priori Runtime Knowledge
Exclusive Choice	Discriminator (N-out-of-M-Join)	MI without a priori Runtime Knowledge
Simple Merge		

Structural Patterns	**State-based Patterns**	**Cancellation Patterns**
Arbitrary Cycles	Deferred Choice	Cancel Activity
Implicit Termination	Interleaved Parallel Routing	Cancel Case
	Milestone	

Figure 3.4: The workflow patterns, according to [12].

3.1.1 Workflow Perspectives

The workflow perspectives have been introduced by Curtis et al. in [48] and since then been refined, e.g. in [131, 130]. The most basic perspective is the functional one, defining activities inside workflows.

Definition 3.16 (Functional Perspective) The *functional perspective* covers activities required in a workflow. □

Activities in a workflow are in alignment with definition 3.3 (Activity). Activities can be composed of other activities as given by definition 3.10 (Complex Activity). Usually, this requires execution constraints between them, leading to a *sub-workflow*. The behavioral perspective gives the execution order of activities in a workflow.

Definition 3.17 (Behavioral Perspective) The *behavioral perspective* describes dependencies between activities required in a workflow. □

The behavioral perspective is the most important perspective, since it distinguishes workflows from traditional software engineering approaches (see definition 3.15). It defines dependencies between activities by the use of control flow. Common patterns have been collected as *workflow patterns* [12]. These are divided into six categories, depicted in figure 3.4. Workflow patterns will be investigated in detail in chapter 5 (Processes).

Definition 3.18 (Information Perspective) The *information perspective* describes workflow relevant application data. □

The information perspective models data in workflows. It can be distinguished between control flow, case, and environment relevant data. *Control flow data* is required for the correct routing of the workflow, e.g. if a given value is smaller than a threshold, execute activity A, otherwise B. *Case data* is required for the execution of the activities regarding a specific process instance, e.g. documents or pictures. *Environmental data* is available for a set of process instances, such as the tax rate to apply. Common patterns of data have been collected as *data patterns* [113]. These will be discussed in chapter 4 (Data).

Since workflows are executed in complex organizational and technical environments, resource assignment, either by humans or software systems, is another central aspect. Resource assignment is divided into the *organizational perspective* that assigns human labor to activities and the *operational perspective* that assign computer power to activities.

Definition 3.19 (Organizational Perspective) The *organizational perspective* describes the assignment of human resources to workflows. □

Since a direct mapping between activities found in a workflow and specific people working in a company is often to restrictive, role assignment and resolution is used. Instead of defining that activity A is executed by a certain employee, e.g. John, we say activity A is executed by a role, e.g. *scientific assistant*. Since John is a scientific assistant, we can assign the execution of activity A to John for a given process instance. If John is unavailable, we can look up the staff index for other members of the role scientific assistant.

Definition 3.20 (Operational Perspective) The *operational perspective* describes the integration of tools and applications into workflow management systems. □

The tools and applications are executed either fully automatic, i.e. without user interaction, or represent the activation of standard office software like word processors including default templates. The organizational and operational perspectives are often interleaved. For instance, an employee manually executes an activity and additionally an application program like a word processor is required. Common patterns for workflow resources have been collected as *resource patterns* [114]. The organizational and the operational perspective are out of scope for this thesis.

3.1.2 Formal Foundations

Since workflows describe the dependencies between activities, people, and other resources involved in companies or departments executed by a workflow management system, a precise and formal definition of the concepts is required. Two major approaches are based on set theory and Petri nets.

Set Theoretic Approaches

Set theory [54] uses logic operations on sets to denote activities, processes, and data. Leymann and Roller discuss a common approach in [83]. Further approaches based on set theory can be found for instance from Weske in [130]. Set theoretic approaches use directed, (a)cyclic graphs to denote workflows. A directed graph is a tuple consisting of nodes and edges.

Definition 3.21 (Directed Graph) A *directed graph* is a two-tuple (N, E) with

- N as a non-empty, finite set of nodes, and
- $E \subseteq N \times N$ as a set of directed edges between nodes. □

Edges represent relations and nodes represent activities in workflows. It can be distinguished between control and data flow graphs. The behavioral aspects of a workflow can be depicted as a workflow graph shown in figure 3.5. Each node represents a workflow activity and the edges

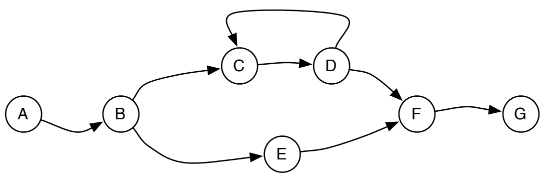


Figure 3.5: A workflow graph.

denote relations between them. As can be seen, activity B depends on activity A, whereas activities C end E depend on B. What is not contained in the graphical representation is how C and E depend on B. Will C and E be executed after B, or either C or E only? These properties of workflow graphs have to be added in a formal representation.

Definition 3.22 (Simple Process Graph) The example can be formalized using set theory by defining a simple process graph consisting of a four-tuple (N, E, S, J) with

- N as a non-empty, finite set of nodes,
- $E \subseteq N \times N$ as a set of directed edges between nodes,
- $S : N \nrightarrow \{\text{AND},\text{XOR}\}$ assigns each node a split condition if more than one edge is leaving this node, and
- $J : N \nrightarrow \{\text{AND},\text{XOR}\}$ assigns each node a join condition if more than one edge is targeting this node. □

In contrast to the graph-based visualization of the example, the formalization solves the ambiguity by providing a join and split behavior.

Example 3.3 (Set Theoretic Workflow Graph) The workflow graph from figure 3.5 is formalized according to definition 3.22 as follows:

1. $N = \{A, B, C, D, E, F, G\}$
2. $E = \{(A, B), (B, C), (C, D), (D, C), (B, E), (D, F), (E, F), (F, G)\}$
3. $S = \{(B,\text{AND}), (D,\text{XOR})\}$
4. $J = \{(C,\text{XOR}), (F,\text{AND})\}$ □

A set theoretic formalization of workflows does not only allow an unambiguous enactment of the contained processes but furthermore opens the door for analysis. Analysis of workflows, and business processes in a wider scope, includes *validation*, *simulation*, and *verification*. Validation investigates if a workflow does what it should do. Since the semantics of the activities has to be taken into account, this is almost ever a human task in workflow management. Simulation executes workflows and measures relevant data such as average throughput times, bottleneck activities, etc. Verification checks formal properties of workflows regarding the behavioral perspective. In particular, it is of special interest if a workflow contains deadlocks or livelocks. A deadlock for an arbitrary process is defined as follows:

Definition 3.23 (Deadlock) A *deadlock* is given if a process instance has no possibilities of further executing activity instances while not being terminated. □

Deadlocks can occur if one activity instance of a process instance is waiting for another one and vice versa or by structural errors. The former can occur by using shared resources and the latter is given for instance if an exclusive control flow split, where one path of execution is selected, is followed by a synchronizing merge that waits on all incoming paths. A livelock is given by:

Definition 3.24 (Livelock) A *livelock* is a situation during the execution of a process instance where the process instance can never terminate, but still enable and execute certain activity instances. □

Livelocks occur usually if shared resources are allocated unfair or by structural errors. The former is given if if a resource is used by one activity A that is required by activity B running in parallel. However, if A deallocates the resource, it is allocated to another activity different to B and so on. The latter is given for instance by misaligned splits and joins, where a cyclic path of the process can occur ever again. A detailed discussion on deadlocks and livelocks can be found in [129].

One approach to prove workflows formalized as simple process graphs to be deadlock free is by creating all possible *traces*. A trace is defined according to Hoare [72] as follows:

Definition 3.25 (Trace) A *trace* is a finite sequence of actions that occurred inside a process instance up to a specific moment in time. □

While we do not have defined actions on simple process graphs, we can consider them to be the nodes traversed so far. If all traces end up with nodes that have no outgoing edges, the simple process graph is deadlock free. While complete, this approach has the drawback of requiring infinite memory even for limited inputs. Already example 3.3 (Set Theoretic Workflow Graph) creates an infinite state space because of the contained loop.

Petri net based Approaches

Petri nets as invented by Carl Adam Petri are also based on graphs using set theory [110]. However, they form a special subclass as they have a widely acknowledged formal semantics.

Definition 3.26 (Petri net) A *Petri* net is given as a three-tuple (P, T, F):

- P is a finite set of places,
- T is a finite set of transitions ($P \cap T = \emptyset$), and
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs called flow relations. □

A Petri net is a directed graph with two types of alternating nodes (places and transitions). A place p directly connected by an arc to a transition t is called an *input place* of t. Accordingly,

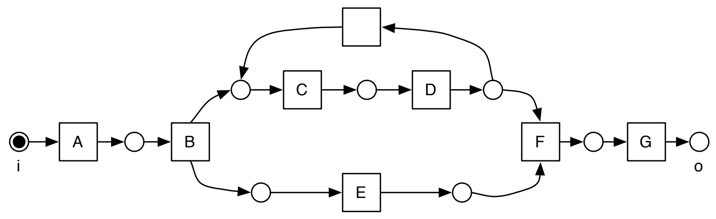


Figure 3.6: A Petri net.

an *output place* p denotes a place directly after a transition t, such that there exists a directed arc from t to p. The sets of output and input places for a transition t are denoted by $t\bullet$ for the former and $\bullet t$ for the latter. Corresponding, the set of transitions directly before and after a place p are denoted by $\bullet p$ and $p\bullet$. Places can contain *tokens* used for representing the current state. Petri nets have a strong graphical notation, shown in figure 3.6. Places are depicted as circles, transitions are drawn as rectangles, and tokens are represented as black dots inside places.

The state of a Petri net is defined as the distribution of tokens over places called *marking*, formally: $M \in P \rightarrow \mathbb{N}$. States are represented as summations of places multiplied by the contained tokens, e.g. $p_1 + 3p_2 + 2p_4$. The example from figure 3.6 has only one token in the leftmost place i and zero in all others, i.e. its current state is represented by i. States are partially ordered: $M_1 \leq M_2$ if $\forall p \in P : M_1(p) \leq M_2(p)$, with $M(p)$ denoting the number of tokens in place p in state M. The marking of a Petri net changes according to the following firing rules:

1. A transition t is enabled if each input place of t contains at least one token, and
2. An enabled transition t may fire. When firing, t removes one token from each of its input places and produces one token for each of its output places.

A transition is denoted as $M_1 \xrightarrow{t} M_2$ if transition t is enabled in M_1 and after firing t, state M_2 is reached. A firing sequence of transitions $\sigma = t_1, t_2, \ldots, t_n$ leading from state M_1 to state M_n is formally denoted as $M_1 \xrightarrow{\sigma} M_n$. Two important definitions for Petri nets are *reachability* and *path*.

Definition 3.27 (Reachable [Petri net]) A state M_n of a Petri net is *reachable* from another state M_1, denoted as $M_1 \xrightarrow{*} M_n$, if and only if there exists a firing sequence σ such that $M_1 \xrightarrow{\sigma} M_n$. □

Definition 3.28 (Path [Petri net]) A *path* in a Petri net (P, T, F) is a non-empty sequence $n_1, \ldots, n_k$ with $n_i \in (P \cup T)$, $n_i \in N$ for $1 \leq i \leq k$, such that $(n_1, n_2), \ldots, (n_{k-1}, n_k) \in F$. □

Petri nets have been refined for representing workflows [3, 10]. Transitions correspond to activities, while places and arcs are used for describing relations. A Petri net modeling the behavioral perspective of workflow is called a *workflow net* [2, 9].

Definition 3.29 (Workflow net) A *workflow net* is given by a Petri net (P, T, F) with the following properties:

1. There is exactly one initial place $i \in P$ with $\bullet i = \emptyset$;
2. There is exactly one final place $o \in P$ with $o\bullet = \emptyset$; and
3. Every node x with $x \in (P \cup T)$ is on a path from i to o. □

The properties of a workflow net can be checked statically, thus it can be decided if a given Petri net is a workflow net. However, there exist additional requirements for workflow nets regarding verification. These are called *soundness* properties [10].

Definition 3.30 (Sound) A workflow net $WF = (P, T, F)$ with an initial place i and a final place o is *sound* if and only if:

1. WF has the option to always complete: $\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$;
2. WF has a proper termination: $\forall_M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o)$; and
3. WF has no dead transitions: $\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{t} M'$. □

A less restrictive soundness property called *relaxed soundness* has been introduced in [51]. Relaxed soundness does not consider deadlocks or livelocks and allows token to remain in a workflow net after the final place is marked. It is based on the assumption that each transition should participate in the workflow at least once, i.e. there exist no dead activities.

Definition 3.31 (Relaxed Sound) A workflow net $WF = (P, T, F)$ with an initial place i and a final place o is *relaxed sound* if and only if WF has no dead transitions and each transition is on a path from i to o: $\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{t} M' \xrightarrow{*} o$. □

Beyond Workflow nets

While workflow nets dominated workflow theory for over a decade, serious drawbacks have been investigated leading to an extended specification called *Yet Another Workflow Language* (YAWL) [11]. Workflow nets are based on state/transition Petri nets and thus inherit all drawbacks of Petri nets as a control flow language for workflow. In particular, several kinds of workflow patterns involving multiple instances, advanced synchronization, and cancellation are not directly supported. High-level Petri nets as described in [77] overcome the limitations regarding the workflow pattern [112] but are less expressive. *Expressiveness* is used informal, denoting the modeling effort required to describe a business process. An extended discussion can be found in the YAWL documentation [11]. YAWL extends workflow nets graphically by making them look like an extension to Petri nets but is actually a mixture of Petri nets and a proprietary transition system. Formally, additional information are added to a workflow net for representing all workflow patterns.

Definition 3.32 (Extended Workflow net) An *extended workflow net* is a tuple $(C, \mathbf{i}, \mathbf{o}, F, split, join, rem, nofi)$ with:

- C is a set of conditions,

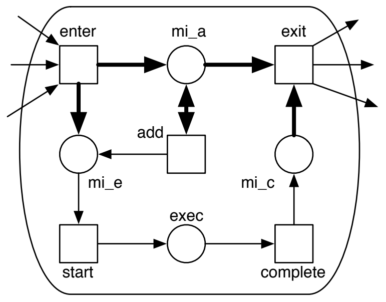
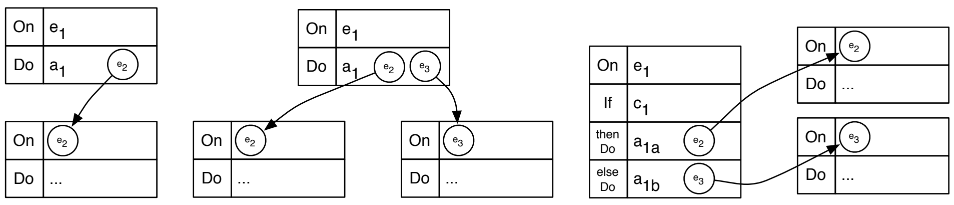


Figure 3.7: Illustrated YAWL task semantics, according to [11].

- $\mathbf{i} \in C$ is the input condition,
- $\mathbf{o} \in C$ is the output condition,
- T is a set of tasks (activities),
- $F \subseteq (C\backslash\{\mathbf{o}\} \times T) \cup (T \times C\backslash\{\mathbf{i}\}) \cup (T \times T)$ is the flow relation, such that every node in the graph $(C \cup T, F)$ is on a path from **i** to **o**,
- $split : T \rightarrow$ {AND,XOR,OR} specifies the split behavior of each task,
- $join : T \rightarrow$ {AND,XOR,OR} specifies the join behavior of each task,
- $rem : T \nrightarrow \mathcal{P}(T \cup C)\backslash\{\mathbf{i}, \mathbf{o}\}$ specifies additional tokens to be removed by emptying a part of the extended workflow net, and
- $nofi : T \nrightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances). □

The tuple (C^{ext}, T, F^{ext}) corresponds to a Petri with $C^{ext} = C \cup \{c_{(t_1,t_2)} | (t_1, t_2) \in F \cap (T \times T)\}$ representing all places including implicit ones, and $F^{ext} = (F\backslash(T \times T)) \cup \{(t_1, c_{(t_1,t_2)}) | (t_1, t_2) \in F \cap (T \times T)\} \cup \{(c_{(t_1,t_2)}, t_2) | (t_1, t_2) \in F \cap (T \times T)\}$ representing additional flow relations for implicit places. The $split$ and $join$ functions correspond to definition 3.22 (Simple Process Graph) adding OR-split/join behavior. Rem is a partial function able to remove tokens from a part of the extended workflow net if a certain task is executed. $Nofi$ is again a partial function used for multiple instance patterns.

The semantic of YAWL is based on a transition system focusing on tasks. An illustration depicting the semantics of a single YAWL task is shown in figure 3.7. While it looks like a Petri net, the behavior differs by the thick arcs, representing the generation/consumption of multiple tokens. Transition $enter$ for a task t occurs if $\bullet t$ contains sufficient tokens for the $join(t)$ function. $Enter$ consumes all tokens from $\bullet t$ and produces tokens inside the task depending on the number of instances to be created from $nofi(t)$. If only one instance should be produced, one token for mi_a and one token for mi_e are produced. If more instances are required, more pairs of



(a) Sequence. (b) Parallel. (c) Choice.

Figure 3.8: ECA business rules representing control flow.

tokens for mi_a and mi_e are produced. *Start* occurs once for each token placed in mi_e and produces a token for *exec* representing the executing of an instance of the task. *Complete* occurs if an instance is completed; the corresponding token from *exec* is consumed and another for mi_c produced. Transition *exit* occurs when all instances have finished, consuming corresponding tokens from mi_a and mi_c. *Exit* furthermore removes tokens from selected parts of the extended workflow net as given by $rem(t)$. The number of tokens produced depends on $split(t)$. Transition *add* comes into play if a task represents multiple instances with dynamic instance creation. As long as the maximum number of instances as defined by $nofi(t)$ has not been reached, *add* can create new instances by adding tokens to mi_e and mi_a. Each transition is formally described in [11].

Other Approaches

Another approach for the formal representation of workflows is based on business rules [79]. It originates from ECA rules found in active database systems, denoting event, condition, and action [50]. Basically, a workflow activity routed by business rules is activated if certain *events* have occurred and defined *conditions* hold. After the activity is finished, new events can be generated as part of the *action*. Business rules can have different triggers, for instance:

- OR-trigger event E_1 or E_2 trigger the rule ($E_1 \vee E_2$);
- AND-trigger, event E_1 and E_2 together ($E_1 \wedge E_2$);
- Sequence-trigger, event E_1 followed by E_2 (E_1, E_2);
- Counter-trigger, n times event E ($n * E$);
- M-out-of-n-trigger, m events out of a set of n;
- Periodical-trigger, every n-th event; or
- Interval-trigger, where every event E within an interval of events triggers the rule.

Figure 3.8 shows how business rules can be used to specify control flow. An extended discussion can be found in [79]. Several other approaches for representing workflow exists, e.g. by logic [49], agents [74], graph-grammars [19], and extensions of Petri nets [101]. These will not be discussed here.

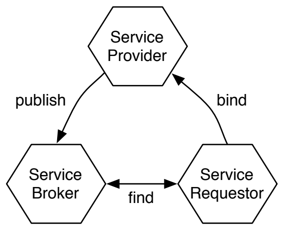


Figure 3.9: The service-oriented architecture, according to [41].

3.2 Service-oriented Architectures

Service-oriented architectures (SOA), as introduced in [41], provide the state-of-the-art architecture for realizing BPM solutions [104, 136]. SOA replaces architectures with tightly coupled components by a loose coupling approach where parts of the system are integrated just in time. These parts are called *services*. Since no common definition of a service exists, we give the following for the purpose of this thesis:

Definition 3.33 (Service) A *service* is an (abstract) process with interaction flows that represent the external visible interface. □

While activities representing purely computing tasks are referred to as *e-business services* [41], the definition of a service refers to a wider scope. In contrast to activities, services are an offer to perform work for someone external. Therefore each service has a well-defined specification containing its functionality and interaction behavior. Services are loosely coupled in a meaning that they are dynamically bound and easily exchangeable. They furthermore constitute the basic buildings blocks of a service-oriented architecture.

Definition 3.34 (Service-oriented Architecture) A *service-oriented architecture* is a software architecture style focusing *"on how services are described and organized to support their dynamic, automated discovery and use"* [41]. □

While the initial approach to SOA was based on web services [66, 75, 18], it can be abstracted from concrete realization strategies and focused on the core architecture. A service-oriented architecture is based on three key entities, *service providers*, *service brokers*, and *service requestors*, depicted in figure 3.9. Service providers *publish* the availability of their services at a service broker. This includes the functional description, the required interaction behavior, and how the services can be reached. Service brokers register and categorize published services and offer search capabilities. Service requesters utilize service brokers to *find* specific services and thereafter are able to *bind* to them.

Service-oriented architectures issue a number of questions. First of all, what should a service provider publish about its service? The functional description can be split into two parts, static interfaces and semantic descriptions. Whereas the former is already implemented by existing standards like WSDL [46], the latter has not yet been solved completely. Regarding required interaction behavior, the published information will in most cases be minimized to cover business secrets and allow updating the internal processes without notification; i.e. abstract processes are used. Binding is based on information contained in the description of how a certain service can be reached. While practically solved as done using `assign from PartnerLink` in BPEL4WS [28] the theoretical treatment of dynamic binding is still under investigation. The find operation takes most attention in research, focusing on semantic matching and behavioral compatibility. Current practical approaches as UDDI [105] only allow static interface matching. By using service-oriented architectures for business process management, highly flexible business processes are supported. Instead of predefined business processes, service can be discovered and integrated during runtime.

3.2.1 Orchestrations and Choreographies

Web services, a special kind of service that use standardized protocols, brought the terms *orchestration* and *choreography* into BPM related computer science:

> *"An orchestration defines the sequence and conditions in which one web service invokes other web services in order to realize some useful function. [...]"* [126].

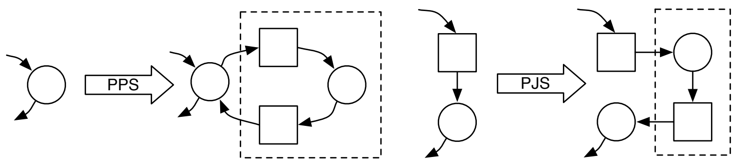
An orchestration corresponds to definition 3.7 (Process). Activities are represented by (web) services that are invoked following a given control flow, i.e. the orchestration. The complete orchestration is then itself a service:

> *"Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web Services, applications or human beings. Transactions among Web Services and their clients must clearly be well defined at the time of their execution, and may consist of multiple separate interactions whose composition constitutes a complete transaction. This composition, its message protocols, interfaces, sequencing, and associated logic, is considered to be a choreography."* [126]

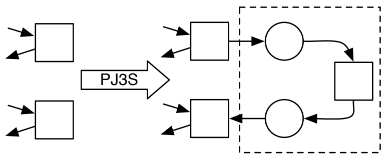
In the context of this thesis, choreography corresponds to definition 3.12 (Interaction). A choreography describes how multiple business processes work together regarding message protocols, interfaces, sequencing, and associated logic, whereas an interaction focuses on the sequences of messages given by the contained processes. Common patterns have been collected as *service interaction patterns* [24]. These will be discussed in detail in chapter 6 (Interactions).

3.2.2 Formal Foundations

Service-oriented architectures do not have a common formal foundation until now. Existing work can be parted into extensions to workflow, i.e. Petri net based, and other approaches including process algebra [21]. Most existing work focuses on orchestrations and choreographies, where service discovery and dynamic binding are elided.



(a) PPS rule. (b) PJS rule.



(c) PJ3S rule.

Figure 3.10: Inheritance-preserving transformation rules.

Petri net based Approaches

In [15] Weske and van der Aalst introduced an approach for interorganizational workflows (P2P approach, public-to-private) based on inheritance-preserving transformation rules for Petri nets [6, 5, 27]. While not directly related to service-oriented architectures, they nevertheless provide a formal representation of choreographies and describe how workflow nets used as orchestrations can be derived thereof.

The P2P approach is divided into three steps. In the first step the choreography (called public workflow) is modeled using workflow nets. In the second step, the workflow net is partitioned into domains representing different participants. Each transition belongs to exactly one participant, whereas places can be shared between two participants, denoting interactions. The resulting workflow net is called an *interorganizational workflow net*. In the third step, the orchestrations of the participants (called private workflows) are refined using the corresponding public part of the interorganizational workflow. To ensure conformance regarding the public workflow, certain rules have to be followed [6, 27], shown in figure 3.10. Figure 3.10(a) denotes the addition of a loop (PPS rule), figure 3.10(b) shows the insertion of transitions in-between existing transitions (PJS rule), and figure 3.10(c) shows how to add transitions in parallel to existing ones (PJ3S rule). These rules are based on projection inheritance for labeled Petri nets, informally defined as:

> *"If it is not possible to distinguish the behaviors of x and y when arbitrary tasks of x are executed, but only the effects of tasks that are also present in y are considered, then x is a subclass of y."* [6].

To formalize projection inheritance, an *abstraction operator* for *labeled Petri nets* is introduced. This operator is based on an unobservable action or *silent step* known from process algebra [21].

Definition 3.35 (Labeled Petri net) A *labeled Petri net* is given as a four-tuple (P, T, F, l) with P, T, F representing places, transitions, and arcs as given in definition 3.26 (Petri net) and

$l : T \rightarrow L$ is a labeling function with L being a set of labels. □

A labeled Petri net can contain markings:

Definition 3.36 (Marked, labeled Petri net) A *marked, labeled Petri net* is a tuple (N, s) with $N = (P, T, F, l)$ as a labeled Petri net and s is a bag over P denoting the marking of the net. □

An abstraction operator for Petri nets renames all transitions of a certain subset of the net to silent steps τ:

Definition 3.37 (Abstraction Operator) $N = (P, T, F, l_0 \cup \tau)$ is a labeled Petri net. The *abstraction operator* τ is a function that renames all transition labels for a certain subset $I \subseteq T$ to the silent step τ. Formally: $\tau_1(N) = (P, T, F, l_1)$, so that for any $t \in T : l_0(t) \in I \Rightarrow l_1(t) = \tau$ and $l_0(t) \notin I \Rightarrow l_1(t) = l_0(t)$. □

The formal definition of projection inheritance requires *branching bisimilarity* [64], an equivalence relating two processes regarding their observable runtime behavior. In contrast to equivalences based on traces, that only consider past actions, branching bisimulation considers the current actions. Hence branching bisimulation is stronger (i.e. it relates fewer processes) than equivalences based on traces [30]. Branching bisimilarity is rooted in process algebra and has been adapted to Petri nets by Basten [27]. It is denoted as $p \sim^b q$ for p and q being marked, labeled Petri nets.

Definition 3.38 (Projection Inheritance) *Projection inheritance* is given if two marked, labeled Petri nets N_0 and N_1 representing sound workflow nets are in a super-/subclass relationship. Formally: $N_1 \leq_{pj} N_0$ if and only if $I \subseteq T$ such that $(\tau_1(N_1), [i]) \sim^b (N_0, [i])$. □.

Projection inheritance thus relates any two nets N_0 and N_1 if N_1 is a subclass of N_0. Regarding the P2P approach, projection inheritance ensures that the private workflows of the participants (orchestrations) are a subclass of the public workflow (choreography). Accordingly, the private extensions do not disturb the behavior of the public workflow.

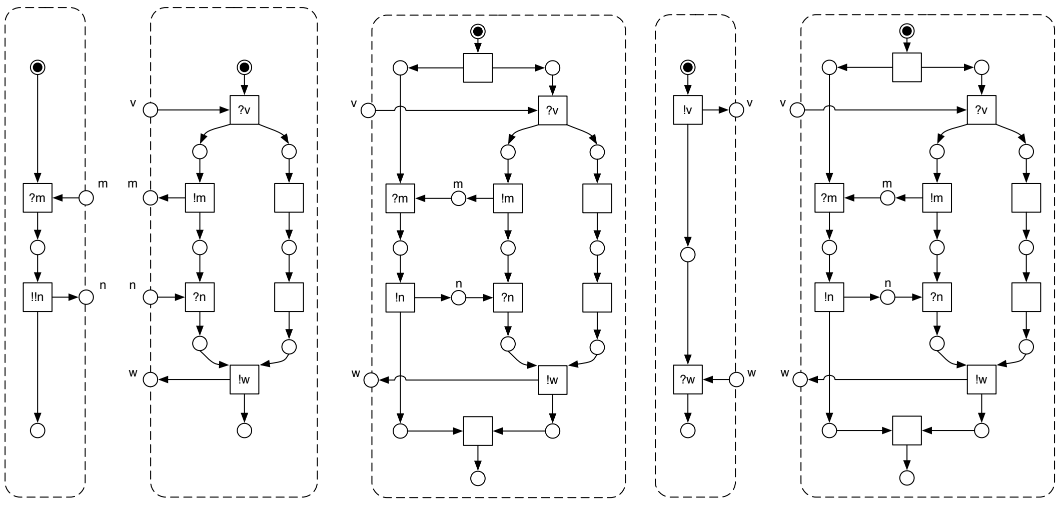
Martens proposed in [84, 85] a different approach for formalizing web services using workflow nets. The approach focuses on compatibility analysis of different services. Workflow nets that represent services should have a certain quality regarding their behavior denoted as *weak soundness*:

Definition 3.39 (Weak Sound) A workflow net $WF = (P, T, F)$ with an initial place i and a final place o is *weak sound* if and only if:

1. WF has the option to always complete: $\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$; and
2. WF has a proper termination: $\forall_M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o)$. □

Weak soundness is a subset of soundness (see definition 3.30) by omitting dead transitions. Martens argues that composed systems might include workflows where not all functionality is required. However, since the functionality is contained it should not disturb soundness. A web service (called *workflow module*) is modeled by an internal process represented by a workflow net and an interface able to communicate with the environment.

Definition 3.40 (Workflow Module) A *workflow module* is given by a Petri net $N = (P, T, F)$ if and only if



(a) Syntactical compatible modules.

(b) Composed module.

(c) Module and environment.

Figure 3.11: Services represented by workflow modules.

1. The set of places is divided into three disjoint sets: $P = P^N \cup P^I \cup P^O$ with P^N denoting internal places, P^I denoting input places, and P^O denoting output places.

2. The flow relation is divided into *internal flow*: $F^N \subseteq (P^N \times T) \cup (T \times P^N)$ and *communication flow*: $F^C \subseteq (P^I \times T) \cup (T \times P^O)$.

3. The internal process (P^N, T, F^N) is a workflow net.

4. No transition is connected to both an input and an output place at the same time: $\nexists t \in T$ such that $| \bullet t \cap P^I| > 0 \wedge |t \bullet \cap P^O| > 0$. $\square$

A distributed business process is then made up of composed workflow modules that are *syntactical compatible*. Syntactical compatibility for two workflow modules is given if the internal processes are disjoint and for each common place there is one output place in one module and one input place in the other module. Thus two modules are syntactical compatible if certain input and output places match (see figure 3.11(a)).

Two workflow modules A and B are composed by merging common places and specifying the remaining places as new interface. Furthermore, the initial states of A and B are preceded by a new transition and a new initial state. The same holds for the final states.

Definition 3.41 (Composed System) Let $A = (P_a, T_a, F_a)$ and $B = (P_b, T_b, F_b)$ be two syntactically compatible workflow modules. Furthermore, $i_s, o_s \notin (P_a \cup P_b)$ denote two additional places and $t_i, t_o \notin (T_a \cup T_b)$ denotes two new transitions. A *composed system* $\Pi = A \oplus B$ is given by a workflow module (P_s, T_s, F_s) with:

- $P_s = P_a \cup P_b \cup \{i_s, o_s\}$,

- $T_s = T_a \cup T_b \cup \{t_s, t_o\}$, and
- $F_s = F_a \cup F_b \cup \{(i_s, t_s), (t_i, i_a), (t_i, i_b), (o_a, t_o), (o_b, t_o), (t_o, o_s)\}$. □

An example of a composed workflow module is shown in figure 3.11(b). If two workflow modules A and B are composed such that $A \oplus B$ is a workflow net (i.e. the composed system has an empty interface), A is called an environment of B and vice versa (see figure 3.11(c)).

Definition 3.42 (Environment [Workflow Module]) Let $A \oplus B$ be a composed system. A is called an *environment* of B, if each output place of A is an input place of B, and each output place of B is an input place of A. □

Another concept introduced by Martens is the *usability* of workflow modules. Usability denotes the quality of a workflow module regarding possible environments:

Definition 3.43 (Usable [Workflow Module]) A workflow module A is called *usable* if there exists at least one environment B such that $A \oplus B$ is weak sound. Furthermore, if $A \oplus B$ is weak sound, the environment B utilizes workflow module A.

Based on the definition of usability, *simulation* between two different workflow modules A and B is given if each utilizing environment E of workflow module A is an utilizing environment of workflow module B. Equivalence between workflow modules is then given by:

Definition 3.44 (Equivalence [Workflow Modules]) Two workflow modules A and B are called *equivalent* if A simulates B and B simulates A. □

Two workflow modules are equivalent if an observer (the environment) cannot detect any differences between the workflow models regarding their external visible behavior. A service broker can use this equivalence relation to find behavioral appropriate services for a certain service requester.

A different view on describing the service behavior has been made by Massuthe and Schmidt based on the work of Martens [86, 87]. They propose *operating guidelines* containing communication structures for service requesters to be published at service brokers. Operating guidelines describe the wanted behavior of requesters in contrast to abstract processes of service providers. This is done to reduce the complexity for a service broker to select appropriate service providers for a specific service requester. Instead of deciding whether an environment containing the process of the requester utilizes each workflow module representing a possible interaction partner, with operating guidelines, the service broker has to decide if the requester's process matches the guidelines. The complexity of deciding weak soundness for each possible combination of interaction partners is in the order of the product of the sizes of the requester and provider, whereas matching is basically in the size of the requester's process.

The argumentation of Massuthe at al. is based on open workflow nets extending workflow modules with an initial and one or more final markings:

Definition 3.45 (Open Workflow net) An *open workflow net* is given by a Petri net $N = (P, T, F)$ and the following three additions:

- $in, out \subseteq P$ with $\forall p \in in : (t, p) \notin F$ and $\forall p \in out : (p, t) \notin F$ and $\forall p \in in \cup out : |\bullet p| + |p \bullet| \leq 1$,

- m_0 as the initial marking, and
- Ω as a set of final markings. □

Accordingly, the component-wise union of two open workflow nets M and N, denoted as $M \oplus N$, yields a composed net. A *deadlock* for an open workflow net is given if a certain marking m of the net enables no transitions at all. Based on these preconditions, weak termination is defined:

Definition 3.46 (Weak Termination) A *weak termination* of an open workflow net is given if all deadlocks are final markings of the set Ω. □

An open workflow net M is called a *strategy* for another open workflow net N if $M \oplus N$ has weak termination. Regarding service-oriented architectures, a service broker must decide whether a requester's process R is a strategy for a given service S as otherwise unexpected behavior might occur. Accordingly, an abstract process of S has to be transmitted to the service provider for decision-making. Operating guidelines use a different approach as introduced above:

Definition 3.47 (Operating Guideline) An *operating guideline* is the description of the behavior of all strategies for a certain requester's process. □

Thus an operating guideline describes how a service requester successfully interacts with a service. The *behavior* for a certain strategy is given by constructing a reachability tree of the inner places, i.e. $p \in P \backslash (in \cup out)$. Each edge of the reachability tree is is annotated with $!x$ if the corresponding transition in N is connected to an output place $p \in out$ and with $?x$ if the the corresponding transition is connected to an input place $p \in in$. Thereby each x represents a certain label. All other edges are marked as silent steps τ. The reachability trees can then be merged to provide a common behavior for all strategies to be published at the service broker.

Process Algebra and Calculi

Process algebra and calculi have been proposed as foundations for service-oriented architectures, e.g. by Bordeaux and Salaün in [32]. They propose to select a certain process algebra regarding the topic of investigation. This might include, but is not limited to, formal descriptions, composition, discovery, or correctness analysis of services.

In [115, 33] Bordeaux et al. give examples using CCS [94]. CCS, as the predecessor of the π-calculus, uses a set of names for representing actions and messages inside a system, given by $Act = \mathcal{A} \cup \overline{\mathcal{A}} \cup \tau$. $\mathcal{A}$ represents names given by lowercase letters such as $a, b, c, \ldots$, $\overline{\mathcal{A}}$ represents corresponding co-names given by overlined lowercase letters such as $\overline{a}, \overline{b}, \overline{c}, \ldots$, and τ represents a silent step. The basic capabilities of processes are receiving a message (simply denoted by writing the name), emitting a message (denoted by writing the co-name), or perform an unobservable action (denoted by τ). The processes of CCS are given by the following grammar:

$$P ::= \alpha.P \mid \mathbf{0} \mid P|P' \mid P + P' \mid P \backslash L \mid P[f] \ . \tag{3.1}$$

Informally, $\alpha.P$ executes the action α and thereafter behaves as P; $\mathbf{0}$ denotes inaction, a process that can do nothing anymore; $P|P'$ denotes parallel execution of P and P'; $P + P'$ is the

$$\textsc{Act}\ \frac{}{\alpha.A \xrightarrow{\alpha} A} \qquad \textsc{Sum-L}\ \frac{A \xrightarrow{\alpha} A'}{A+B \xrightarrow{\alpha} A'} \qquad \textsc{Com-L}\ \frac{A \xrightarrow{\alpha} A'}{A|B \xrightarrow{\alpha} A'|B}$$

$$\textsc{Com-I}\ \frac{A \xrightarrow{a} A' \quad B \xrightarrow{\overline{a}} B'}{A \mid B \xrightarrow{\tau} A' \mid B'} \qquad \textsc{Res}\ \frac{A \xrightarrow{\alpha} A'}{A\backslash L \xrightarrow{\alpha} A'\backslash L}\ (\alpha,\overline{\alpha}\notin L) \qquad \textsc{Rel}\ \frac{A \xrightarrow{\alpha} A'}{A[f] \xrightarrow{f(\alpha)} A'[f]}$$

Figure 3.12: CCS transition rules.

exclusive choice between P or P', $P\backslash L$ restricts the scope of the set of names L to P; and $P[f]$ relabels names in P given by f as a relabeling function. Uppercase letters are used to range over process identifiers derived by P, such as A, B, C. The semantics of CCS is defined using a labeled transition system: The states are given by P, i.e. the process definitions, and Act represents the set of transitions labels, i.e. the actions and messages. The set of rules for CCS is given in figure 3.12. Elided from the rules are the symmetric forms SUM-R and COM-R that simply swap the left and right components as well as congruence (CON). The rules correspond to the informal description of the grammar from equation 3.1.

According to Bordeaux et al., CCS processes can be used to describe orchestrations and choreographies in service-oriented architectures. For instance, the required behavior of a citizen making a request can be formally described as follows:

$$C = \overline{req}.question.\overline{answer}.(refuse.C + accept.C)\ .$$

A citizen C sends a request (*req*), receives some $questions$, sends the $answer$, and finally receives either a $refuse$ or $accept$ of the request. Thereafter the citizen process is reset using recursion. This behavior can now be complemented by several systems supporting the citizen, i.e. wait for requests, process them, and finally send the result. Several kinds of bisimilarity can then be used to reason on equivalences. However, Bordeaux et al. only give examples instead of a concrete methodology using process algebra for service-oriented architectures. Especially, they do not consider more complex control flow relations as for instance given by workflow patterns.

Approaches using proprietary transition systems for formalizing service choreography and orchestration have also been proposed, e.g. in [34, 88]. Exemplarily, we investigate the approach from Busi et al. [42]. They introduced a formal model of choreographies including roles, variables, and operations. Conversions between roles are defined by using the following grammar:

$$C_P ::= \mathbf{0} \mid C_P;C_P \mid \mu \mid C_P|C_P \mid C_P + C_P\ . \tag{3.2}$$

Informally, $\mathbf{0}$ denotes a null conversation, μ is an interaction, $C_P;C_P$ sequential composition, $C_P|C_P$ parallel composition, and C_P+C_P exclusive choice. Interaction μ is further specified by $(\rho_A, \rho_B, o, \tilde{x}, \tilde{y}, dir)$ denoting an interaction from a role ρ_A to another role ρ_B. The operation to be performed is denoted by o, whereas $\tilde{x}$ and $\tilde{y}$ denote sequences of variables used by the sender and receiver. The direction is denoted using $dir \in \{\downarrow, \uparrow\}$ representing request ($\uparrow$) or response ($\downarrow$). The semantics is given using labeled transition systems, where sequence, parallel, and choice are analogue to CSS, and the interaction rule is given by:

$$\mu \xrightarrow{\mu} \mathbf{0}, \mu = (\rho_A, \rho_B, o, \tilde{x}, \tilde{y}, dir) \ .$$

The behavior of the interaction rule depends on the direction. If $dir = \uparrow$ the information from $\tilde{x}$ of ρ_A are passed to $\tilde{y}$ of ρ_B and the operation o is enacted at ρ_B. For $dir = \downarrow$, the reverse holds. Processes are defined using the following grammar:

$$P ::= \mathbf{0} \mid o \mid \overline{o} \mid o(\tilde{x}) \mid \overline{o}(\tilde{y}) \mid o(\tilde{x}, \tilde{y}, P) \mid \overline{o}(\tilde{x}, \tilde{y}) \mid P; P \mid P + P \mid P|P \ .$$

Most parts are analogue to CSS with the notable exception of actions for external synchronization. $\overline{o}$ and $o(\tilde{x})$ denote simple input and output. A request-response operation is denoted by $o(\tilde{x}, \tilde{y}, P)$, meaning the process receives messages, stores the received messages in $\tilde{x}$, executes a process P, and finally sends the information contained in $\tilde{y}$ back to the requester. An invocation on an operation o is denoted by $\overline{o}(\tilde{x}, \tilde{y})$ with $\tilde{x}$ representing the request and $\tilde{y}$ the response. Processes are executed at different locations inside an orchestrated system E given by:

$$E ::= [P]_{id} \mid E \| E \ . \tag{3.3}$$

$[P]_{id}$ is called an orchestrator, representing a process identified by id. Processes inside an orchestrated systems can only be composed in parallel using $\|$. The formal semantics is given by thirteen transition rules for P and six more complex ones for E, to be found in [42]. Based on the given formalizations, branching bisimilarity can then be used to reason on conformance between an orchestration given by E and a choreography given by C_P. However, the approach by Busi et al. does not consider existing patterns for choreographies and interactions and thus might have a restricted applicability. Furthermore, it is based on a proprietary transition system instead of standard proposals like CCS, meaning a lack of tool support, academic acceptance and investigations.

Recent research on choreography patterns also emphasizes the importance of *mobility* for the formal representation of routing and dynamic binding in choreographies [24]. The concept of mobility referred to denotes *link passing mobility* capabilities for processes. Examples are hypertext links that can be passed along participants of choreographies allowing them to gain access to prior unknown services. A different approach of mobility is denoted as *process passing mobility*. An example is code send across at network and executed at the receiver's site. Guidi and Lucci differentiate in [67] four mobility types described in a proprietary service-based language. *Internal state mobility* describes message exchange between a sender and receiver, i.e. the message is mobile. *Location mobility* describes a location exchange between a sender and receiver, where the receiver afterwards can access a service at the location transmitted (i.e. it resembles link passing mobility). *Interface mobility* allows services to acquire operations at run-time and exhibit them in their interfaces, i.e. the interface changes dynamically. *Functional mobility* refers to processes that can be received and executed inside the receiver's processes (i.e. it resembles process passing mobility).

Recent Standards

While not formal in a sense of mathematical sound, existing standards give substantial grounding to BPM and SOA. XML-based orchestration languages for the description of composed web

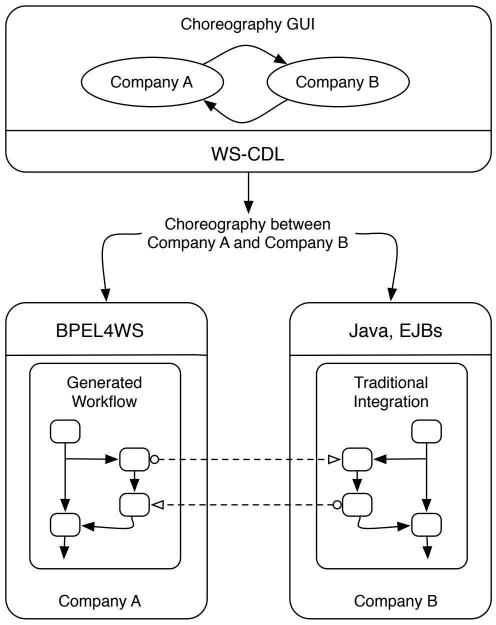


Figure 3.13: WS-CDL based web service integration, according to [128].

services are for instance WSFL, XLang, or BPML [76, 89, 35]. Today the *Business Process Execution Language for Web Service* (BPEL4WS) [28] supersedes these standards. As investigated by Wohed et al. in [8], BPEL4WS supports most, but not all workflow patterns. In particular, BPEL4WS might cause problems regarding application areas requiring arbitrary loops, milestone, or advanced multiple instances pattern. Formal verification for BPEL4WS has been investigated using different directions as for instance state machines by Farabod et al., Fisteus et al., or Fu et al. [57, 60, 62], process algebra by Ferrara [58], or Petri nets by Stahl et al., or Schlingloff et al. [70, 119]. Each of these approaches gives a formal semantics to BPEL4WS.

XML-based choreography languages are the *Web Services Choreography Interface* (WSCI) and the *Web Service Choreography Description Language* (WS-CDL) [127, 128]. WSCI focuses on the description of the observable behavior of web services and uses this knowledge to describe collective message exchange among a set of interacting web services. Thus, WSCI provides a message oriented view of the choreography. WS-CDL, in contrast, focuses on describing a global viewpoint on all interacting business partners. Figure 3.13 shows the application area of WS-CDL. A choreography between a number of companies is specified using WS-CDL. The abstract processes for each company are then generated out of the WS-CDL specification and can be implemented using different technologies as depicted in the figure. A discussion of the advantages and disadvantages of WS-CDL by Barros et al. can be found in [26]. A formalization of WSCI in CSS has been provided by Brogi et al. [38], whereas Gorrieri et al. discuss a proprietary process algebra for the formalization of WS-CDL [65].

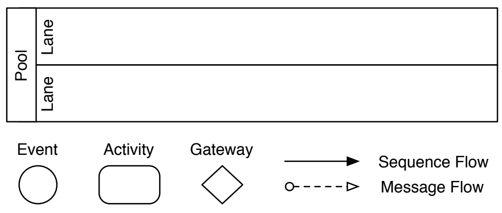


Figure 3.14: BPMN core elements.

3.3 Graphical Notation

Graphical notations for BPM are manifold, ranging from event-driven processes chains over activity diagrams up to Petri nets [78, 106], including many vendor specific ones. To represent the concepts introduced beforehand graphically, a subset of the *Business Process Modeling Notation* [36], short BPMN, is used. The BPMN allows for modeling processes and interactions in so-called *business process diagrams* (BPD). A BPD represents either a single process or a multiple of processes with corresponding interactions. An introduction to BPMN can be found in [132]. Since the BPMN does not support all workflow patterns directly [133, 134], we introduce some additions for a direct representation of patterns like n-out-of-m-join or multiple instances without synchronization.

3.3.1 Business Process Diagrams

The BPMN was designed as a modeling notation capable of communicating a wide variety of information to different audiences ranging from business analysts to IT experts. For these different needs, three types of business processes diagrams have been defined:

- Private (internal) business processes,
- Abstract (public) business processes, and
- Collaboration (global) business processes.

Private business processes represent the internal processes of an organization and conform with definition 3.7 (Process). Abstract business processes represent the interaction between a private process and another (undefined) process. Only the activities relevant for communication are contained inside. They conform to definition 3.33 (Service). Collaboration business processes represent the interaction between two or more abstract business processes. Accordingly, they conform to definition 3.12 (Interaction).

Core Elements

A business process diagram is composed out of core elements shown in figure 3.14. These elements are further on specialized while keeping their outside shape. The primary modeling elements termed as *flow objects* are *events*, *activities*, and *gateways*. An events is something that

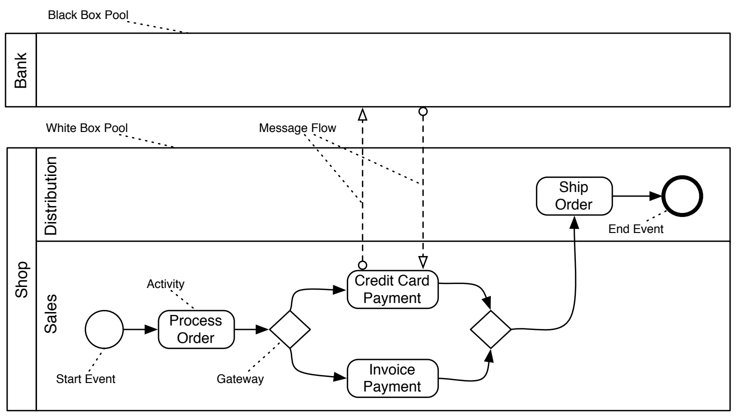


Figure 3.15: Example BDP using core elements.

happens in the course of a business process. It affects the flow of a process and can have a trigger or result. An activity is work a company performs; it can be atomic or complex. Gateways are used for routing sequence flows. In this thesis, all flow objects conform to definition 3.3 (Activity).

Sequence flows connect events, activities, and gateways and therefore are termed *connectivity elements*. BPMN sequence flow represents the control flow concept (definition 3.6). Message flow shows the flow of messages between different business processes. Thus, it represents the interaction flow concept (definition 3.11).

All flow objects are placed inside pools. A pool is a container for grouping a set of activities, and the relations between them, for a particular organization. To allow further decomposition, lanes inside a pool can be used. These can represent the organizational workflow perspective. Pools can be black or white boxed. A black box pool hides its inside details, so message flows can only occur to the outside rectangle of the pool. A white box pool shows internal details, so message flows connect to internal elements.

An example of a BPD is shown in figure 3.15. The example is a mixture of a private and abstract business process. It consists of two participants (organizations), denoted as *Bank* and *Shop*. The former is shown as a black box pool, whereas the latter is a white box pool. The business process starts at the *Shop's* sales lane by receiving a start event, i.e. an order. The order is thereafter processed in the activity *Process Order*. The next activity depends on the routing decision of the gateway after *Process Order*. A default gateway, as shown in the example, makes an exclusive decision between the two downstream activities. Thus, either *Credit Card Payment* or *Invoice Payment* is executed next. While the latter is a simple activity, the former interacts with the *Bank's* pool by using message flow. Afterward, the sequence flow is joined and another activity, *Shop Order* is executed by the department *Distribution*. The business process is concluded with an end event, denoted using a bold outlined circle.

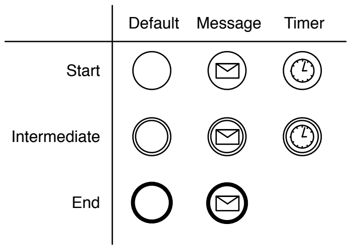
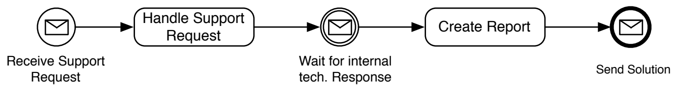
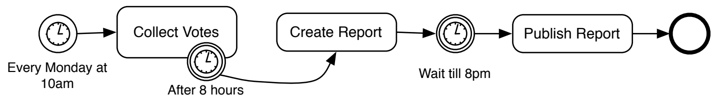


Figure 3.16: BPMN events.



(a) Message example.



(b) Timer example.

Figure 3.17: BPMN event examples.

Events

As mentioned earlier, the core elements can be modified to achieve more complex behavior. An example has already been given in figure 3.15, where the core element event has been named start event, and a derived shape, with a bolder outline, end event. A third kind of event is called intermediate event, denoted using a double-lined outline. An intermediate event affects the flow of the process, but does not start or end it.

Based on start, intermediate, and end events, different types of events have been specified. A subset required for this thesis is shown in figure 3.16. In the BPMN notation, all start events produce a *token*, which follows the sequence flow of the process. All end events consume tokens. Thus, some informal kind of Petri net semantics is used to denote process execution in BPMN.

A default event has no specific trigger or result beside starting and ending the process. A *message start event* starts the process the moment a message is received. A message intermediate event holds the process flow until either a message is received or sends a message. A message end event ends a process by sending a message. An example is shown in figure 3.17(a). A timer start event triggers the start of a process at a specific time. A timer intermediate event holds the process flow until a given time constraint is fulfilled. Furthermore, intermediate events can be placed at the border of an activity, as shown in figure 3.17(b). If such an event occurs, i.e. a message is received or a time constraint reached, the outgoing sequence flow of the event is activated immediately, while the default sequence flow is canceled.

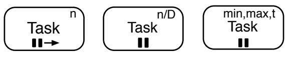
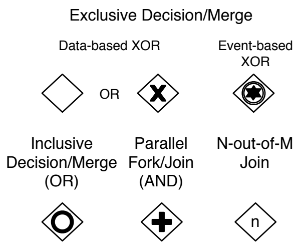
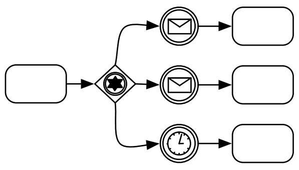


Figure 3.18: BPMN extension for multiple instances.



(a) BPMN gateways.



(b) Event-based gateway example.

Figure 3.19: BPMN gateways with example.

Activities

Activities of the BPMN can be divided into processes, sub-processes, and tasks. Sub-processes can contain other sub-processes. In BPMN, a process is work performed within a company or organization. The term business process refers to one or more of these processes. Each process is contained within a pool. A sub-process defines a compound activity. It can be shown collapsed, hiding its inner details, or expanded, showing its inside details. A collapsed sub-process is marked with a plus sign at the bottom. Processes can either embed sub-process or reference them, where the semantics changes accordingly, i.e. inline vs. call behavior. A task represents an atomic activity within a process. While the BPMN defines several types of tasks, they are not sufficient to support multiple instance workflow patterns graphically. To overcome these limitations, an extension is shown in figure 3.18. The left task denotes multiple instances without synchronization, i.e. n instances of the task are created and the sequence flow is passed on immediately. The middle task represents synchronized multiple instances, either with a dynamic number of instance calculated during runtime of the process (denoted by *D* at the upper right corner) or with a static number of instances known at design time (denoted with a natural number instead of *D*). The right task denotes multiple instances with limited priori knowledge, where a minimum and maximum number of instances to be created can be given by min and max. Furthermore, a threshold can be given via t.

Gateways

Gateways are used as routing constructs for sequence flow. They can decide, split, or merge the flow of the process. The possible gateways are shown in figure 3.19(a). The interesting types are the event-based exclusive choice and the n-out-of-m-join. The former is contained in the official specification, whereas the latter has been added to support the discriminator and n-out-of-m-join workflow pattern. The event-based based exclusive choice is used to model decisions based on events rather than process data. The process flow will continue if one of the specified events occurs. In figure 3.19(b), this might be a *yes* message, a *no* message, or a timeout. The n-out-of-m-join gateway waits for n incoming sequence flows and activates the outgoing ones thereafter.

3.3.2 Formal Foundations

The BPMN has no formal foundation yet. The specification defines a mapping to BPEL4WS as an executable language. However, this mapping is not to be meant as a semantics for BPMN and will hence be removed from subsequent version of the BPMN specification according to IBM sources. During this thesis we will refer to business process diagrams for visualizing formal processes and interactions. Basically, we utilize *process* and *interaction graphs* that will be introduced in chapter 5 (Processes) and 6 (Interactions). The former is used to represent private business process diagrams, whereas the latter denotes abstract or collaboration business process diagrams.

Part II

Investigations

Introduction to Part II

Part II discusses the application of the π-calculus to derive formal models of interacting business processes. It starts with the representation of data as processes. Data is required for internal routing decisions, to represent cases, and environmental values. Since the representation of data values and structures in terms of agents is uncommon in the BPM domain, a detailed derivation of cells, stacks, queues, booleans, and natural numbers as well as functions working on them is given. Building atop of data, the formal representation of processes as process graphs is introduced. Each instance of a process graphs gets a formal semantics by a π-calculus mapping, which describes the actual execution behavior. The semantics of the activities is given by a catalogue of formalized process patterns, covering a broad range of possible application scenarios. The formalized business processes are then analyzed according to several soundness properties. The π-calculus representation of business processes reveals its strengths in describing interactions among a set of business processes. Due to the direct support of dynamic binding and correlations, agile interactions can be given. Once again, a pattern catalogue is investigated, serving as a link to the practical applicability. Finally, formal analysis regarding compatibility and conformance of interactions is introduced.

Structure of Part II Part II is composed of three chapters. The first chapter develops the representation of data in the π-calculus. The second chapter discusses the formal representation and verification of business processes based on process patterns. The third chapter extends the discussion to agile interactions between business processes based on interaction patterns.

Chapter 4

Data

In this chapter we discuss how data can be represented in the π-calculus. The chapter starts by introducing how names can be made persistent using a kind of memory *cell* agent. The cell agent is further on enhanced to support *stack* and *queue* like behavior for subsequent interactions. Afterward, the representation of *boolean values* is discussed. A short introduction to *types* for names is given and the representation of *functions* working on booleans is shown. Based on booleans, *natural numbers* are introduced and syntactical extensions to the π-calculus regarding their handling are defined. Natural numbers are used to represent advanced structures such as *lists*. Finally, common data patterns found in business processes are investigated.

4.1 Structures

This section describes the representation of data structures in the π-calculus. Since the λ-calculus can be encoded in the π-calculus (see e.g. [118]), it is possible to represent all kinds of functions and their corresponding data in the π-calculus (e.g. Milner [95] or Sangiorgi [117]). However, since the functional representation of data has a high computational effort, we investigate a more straightforward representation. Our approach is inspired by the examples given in the original paper on the π-calculus [99] as well as the PICT language [123]. We use agents to represent data, make it persistent using recursion, and apply names as pointers to agents representing a certain data type. As also our representation requires computational efforts, it is only applied if necessary for simulation or verification, whereas otherwise the representation of data is assumed to be implemented natively.

For the integration of the structures, values, and functions introduced later on, we first have to make a convention regarding their free names that can be used by other agents:

Convention 4.1 (Unique Availability) Let P be the composition of all agents representing structures, values, and functions. Let Q be the top-level composition of all other agents in a system given by $P \mid Q$:

$$P \overset{def}{=} \prod_{i=1}^{n} Pi \quad Q \overset{def}{=} \prod_{i=1}^{m} Qi$$

To allow a *unique availability* of all components of P to all components of Q, i.e. they can interact conflict-free, we assume the following properties to hold:

1. The free names of all components of P are unique, formally $fn(Pi) \cap fn(Pj) = \emptyset$ for all i, j with $0 < i < n \wedge 0 < j < n \wedge i \neq j$.
2. Free names of P can occur as either (1) subjects of input prefixes in Q or (2) as arbitrary objects of output prefixes in Q, i.e. no component of Q provides a functionality via the same free names as P. □

The free names of an agent representing a structure, value, or function are then used to access its functionality.

4.1.1 Basic Structures

Basic structures provide elementary grouping and accessing features to names. Each basic structure has a simple interface consisting of an *accessor* name for adding and removing names. Any count of names can be sent to an accessor name by using it as an output prefix and retrieved afterward by using it as an input prefix. We distinguish three types of return possibilities: (1) only the last name, or the last sequence of names ($\tilde{n}$) is returned infinite often (e.g. cell, pair), (2) the last name sent is the first name returned (stack), or (3) the first name sent is the first name returned (queue). For b and c we require an additional name that is triggered when the structure is empty.

The basic structures are defined in the following paragraphs. Each basic structure is globally available to other agents inside a system and can produce a copy of itself via recursion.

Definition 4.1 (Cell) A *cell* holds a name and allows *read* and *write* operations to retrieve or change the content:

$$\mathit{CELL} \overset{def}{=} \nu c\, \overline{cell}\langle c\rangle.(\mathit{CELL_1}(\bot) \mid \mathit{CELL})$$
$$\mathit{CELL_1}(n) \overset{def}{=} \overline{c}\langle n\rangle.\mathit{CELL_1}(n) + c(x).\mathit{CELL_1}(x)\ .$$

A new cell is initialized with the default name $\bot$ (false). The restricted name retrieved by reading via the name *cell* is then used as read and write accessor to the cell's content. □

For instance, consider the agents

$$A \overset{def}{=} \nu d\ cell(c).\overline{c}\langle d\rangle.\overline{b}\langle c\rangle.\mathbf{0} \text{ and } B \overset{def}{=} b(p).p(x).\tau.\mathbf{0}$$

inside a system

$$S \overset{def}{=} \nu cell\ \nu\bot\ \nu b\ (A \mid B \mid \mathit{CELL})\ .$$

Agent A first creates a restricted name d and retrieves a fresh cell c. Afterward the name d is stored inside the cell via c, and the name c is sent via b. Agent B receives the name of the cell via

b and afterward retrieves the content. A cell can be easily extended to a pair, storing a sequence of two names:

Definition 4.2 (Pair) A *pair* holds a sequence of two names and allows *read* and *write* operations to retrieve or change the content:

$$PAIR \stackrel{def}{=} \nu t\, \overline{pair}\langle t\rangle.(PAIR_1(\bot,\bot) \mid PAIR)$$
$$PAIR_1(m,n) \stackrel{def}{=} \bar{t}\langle m,n\rangle.PAIR_1(m,n) + t(x,y).PAIR_1(x,y)\;.$$

□

A new pair is initialized and accessed similar to a cell. Furthermore, we define an agent $TRIPLE$ holding a sequence of three names according to $PAIR$ (omitted). By employing pairs and triples, more advanced data structures can be created. We investigate stacks and queues.

Definition 4.3 (Stack) A *stack* stores names that can be removed in reverse order; i.e. first in, last out. Names can be contained in the stack several times. The stack consists of two operations, *push* to add names to and *pop* to remove names from the stack. The stack presented here is based on two assumptions. (1) The *push* operation can be called infinite often; i.e. there is no upper limit on the size of the stack, and (2) the *pop* operation can be called as long as there are elements on the stack. If the stack size is zero, the special name $empty$ can be read infinite often instead. These assumptions simplify the definition of the stack without restricting its expressive power. The stack is given by:

$$STACK \stackrel{def}{=} \nu s\, \nu empty\; \overline{stack}\langle s, empty\rangle.(STACK_0 \mid STACK)\;.$$

$STACK$ first creates two restricted names: s, used as an accessor name for *push* and *pop* operations, and $empty$, used to represent the empty stack. It then behaves as follows:

$$\begin{aligned} STACK_0 \stackrel{def}{=} & \overline{empty}.STACK_0 + s(newvalue).triple(next). \\ & \overline{next}\langle\bot,\bot,newvalue\rangle.STACK_1(next)\;, \end{aligned}$$

where $STACK_0$ either returns $empty$ or receives a name $newvalue$ via s to push on the stack. In the last case, a new triple is created and initialized with $(prev, test, value)$, where $prev$ represents the previous triple ($\bot$ as this is the first triple on the stack), $test$ is a flag if there are more elements on the stack (also $\bot$), and $value$ is the received value.[1] If a name has been pushed on the stack, the agent continues as $STACK_1$ with the current triple as a parameter:

$$\begin{aligned} STACK_1(curr) \stackrel{def}{=} & curr(prev, test, value).(\bar{s}\langle value\rangle. \\ & ([test = \top]STACK_1(prev) + [test = \bot]STACK_0) + \\ & s(newvalue).triple(next).\overline{next}\langle curr, \top, newvalue\rangle. \\ & STACK_1(next))\;. \end{aligned}$$

[1] We explicitly have to denote a name for testing if there are more elements on the stack, as a mismatch operator (e.g. $prev \neq \bot$) is not contained in the considered π-calculus grammar.

The agent $STACK_1$ first retrieves the values $(prev, test, value)$ from the current triple to have them prepared for immediate response in the case a *pop* interaction on s occurs as the next transition. In this case, the $value$ is sent via s. If there are more elements on the stack ($test = \top$) the agent behaves as $STACK_1$ with $prev$ as a parameter and otherwise as $STACK_0$. If an element is added to the stack by using s as a *push* interaction, a new triple is created and initialized with $(curr, \top, newvalue)$, where $curr$ represents the current triple (now acting as the predecessor), $\top$ for signaling that there are more elements on the stack, and $newvalue$ as the pushed value. The agent then behaves as $STACK_1$ with the newly allocated triple as parameter. □

Definition 4.4 (Queue) A *queue* stores names that can be removed in order; i.e. first in, first out. Names can be contained in the queue several times. The queue consists of two operations, *enqueue* to add names to and *dequeue* to remove names from the queue. The queue presented here is based on two assumptions. (1) The *enqueue* operation can be called infinite often; i.e. there is no upper limit on the size of the queue, and (2) the *dequeue* operation can be called as long as there are elements in the queue. If the queue is empty, the special name $empty$ can be read infinite often. The queue is given by:

$$QUEUE \overset{def}{=} \nu q\, \nu empty\; \overline{queue}\langle q, empty\rangle.(QUEUE_0 \mid QUEUE)\ .$$

The queue creates, equal to the stack, two fresh names: q used as an accessor for *enqueue* and *dequeue* operations, and $empty$, used to represent the empty queue. It then behaves as follows:

$$\begin{aligned} QUEUE_0 \overset{def}{=} & \overline{empty}.QUEUE_0 + q(newvalue).triple(newtriple).\\ & \overline{newtriple}\langle\bot, \bot, newvalue\rangle.QUEUE_1(newtriple, newtriple)\ , \end{aligned}$$

where $QUEUE_0$ either returns $empty$ infinite often or receives a name via q to *enqueue* to the queue. In the last case, a new triple is created and initialized with $(next, test, value)$, where $next$ represents the next triple ($\bot$ as this is the only triple in the queue), $test$ is a flag if there are more elements in the queue after this one (also $\bot$), and $value$ is the received value. If a name has been enqueued, the agent continues as $QUEUE_1$ with the current triple as an explicit parameter representing the first and last triple of the queue:

$$\begin{aligned} QUEUE_1(first, last) \overset{def}{=} & first(next, test, value).(\overline{q}\langle value\rangle.\\ & ([test = \top]QUEUE_1(next, last) + [test = \bot]QUEUE_0)+\\ & q(newvalue).triple(newtriple).\overline{newtriple}\langle\bot, \bot, newvalue\rangle.\\ & last(oldnext, oldtest, oldvalue).\overline{last}\langle newtriple, \top, oldvalue\rangle.\\ & QUEUE_1(first, newtriple)\ . \end{aligned}$$

The agent $QUEUE_1$ works analog to the stack with the exception that the queue needs to update the next pointer of the triple previous to the newly added triple. □

4.1.2 Iterators

An iterator iterates through a data structure. We distinguish two types of iterators, destructive and non-destructive. Destructive operators remove the elements from the structure, whereas non-destructive iterators keep the elements in the structure.

Definition 4.5 (Iterator) An destructive *iterator* that works on stacks and queues is defined by:

$$I \overset{def}{=} s(x).\tau_I.I + empty.I' \ .$$

The iterator works on a structure s. While there are elements available in the structure, the left hand side of the iterator is chosen. The work done with the current element is here denoted as τ_I. If the basic structure returns $empty$, the iterator finishes. □

A non-destructive iterator needs to have knowledge about the data structure it iterates. Since this might cause problems related to concurrent access, special care has to be taken when employing these iterators. A trivial non-destructive iterator for a stack uses a temporary stack to store the values:

$$\begin{aligned} IS &\overset{def}{=} stack(tmpstack, tmpempty).IS_0 \\ IS_0 &\overset{def}{=} s(x).\overline{tmpstack}\langle x\rangle.\tau_{IS_0}.IS_0 + empty.IS_1 \\ IS_1 &\overset{def}{=} tmpstack(x).\overline{s}\langle x\rangle.IS_1 + tmpempty.IS' \ . \end{aligned}$$

In agent IS a new temporary stack $tmpstack$ is allocated first. Thereafter, each element from the original stack s is read and written to the temporary stack. Afterward the content of the current stack's value is evaluated insie τ_{IS_0}. Once the original stack is empty, agent IS_1 restores the content of the original stack s by iterating over the temporary stack $tmpstack$. A non-destructive iterator for queue works accordingly. However, the proposed non-destructive iterator is not safe in concurrent environments, where the data structure can be accessed in parallel.

Example 4.1 (Bank Counters) An example illustrating the problems is a given by a bank which has several counters that serve incoming customers according to a first in, first serve principle. The formal representation consists of a waiting queue, where new names (i.e. customers) are enqueued using an agent $FILL$ (i.e. the customers enter the bank building). The waiting queue is processed by several agents $SERVE$ representing the bank's counters. A sample system is then given as

$$WQ \overset{def}{=} queue(wq, we).(FILL \mid SERVE \mid SERVE) \ ,$$

where two agents $SERVE$ work on elements of the queue added by $FILL$. Possible implementations are

$$FILL \overset{def}{=} \nu t\ \tau.\overline{wq}\langle t\rangle.FILL \text{ and } SERVE \overset{def}{=} wq(x).\tau.SERVE \ .$$

By adding a fourth component $INSPECT$, that searches the waiting queue for a specific name (i.e. a premium customer), unwanted behavior can occur. The implementation has been adapted from the non-destructive stack iterator to a queue:

$$\begin{aligned} \mathit{INSPECT} &\stackrel{\mathit{def}}{=} \mathit{queue}(\mathit{tmpqueue}, \mathit{tmpempty}).\mathit{INSPECT_0} \\ \mathit{INSPECT_0} &\stackrel{\mathit{def}}{=} \mathit{wq}(x).\overline{\mathit{tmpqueue}}\langle x\rangle.\tau.\mathit{INSPECT_0} + \mathit{we}.\mathit{INSPECT_1} \\ \mathit{INSPECT_1} &\stackrel{\mathit{def}}{=} \mathit{tmpqueue}(x).\overline{\mathit{wq}}\langle x\rangle.\mathit{INSPECT_1} + \mathit{tmpempty}.\mathbf{0}\ . \end{aligned}$$

The agent $\mathit{INSPECT}$ declares a non-destructive iterator that uses $\mathit{tmpqueue}$ as a temporary queue. In $\mathit{INSPECT_0}$, each name read from the waiting queue wq is stored in $\mathit{tmpqueue}$. Afterward the name is evaluated by some unobservable action represented by τ (i.e. inspected for a specific customer). If the waiting queue emits the name we, the queue is empty and $\mathit{INSPECT_1}$ is executed. $\mathit{INSPECT_1}$ iterates the temporary queue in a destructive manner. Each name read is enqueued in the original waiting queue wq. This approach causes two problems. First, the agent FILL is able to enqueue new names to the waiting queue while $\mathit{INSPECT_1}$ is restoring the original state. Thus, new names can be added in arbitrary positions. Second, since $\mathit{INSPECT_0}$ dequeues names, parallel transitions of QUEUE might consume names that are contained a later positions in the waiting queue. Thus, the processing behavior of the waiting queue is random instead as first in, first serve, which in turn would anger the bank's customers.

To overcome the problems, we show how a queue with a non-destructive iterator is constructed. A corresponding stack is defined accordingly.

Definition 4.6 (Iterator Queue) A queue with a non-destructive iterator, denoted as *iterator queue*, is derived from pattern 4.4 (Queue). While the iterator queue is iterated, no enqueue or dequeue operation can take place. Technically, the iterator queue employees an additional name i, called the *iterator accessor*. Via i, two restricted names it and $\mathit{itempty}$ for iterating the queue can be acquired. These names can be used according to definition 4.5 (Iterator) without destroying the queue's content. Subsequent interactions via it return the names contained inside the queue, wheras $\mathit{itempty}$ signal that no more names are available for iteration. In contrast to empty, the iterator name $\mathit{itempty}$ is only available for interaction once. Thereafter, the iterator queue returns to its normal operation, meaning that names can now be enqueued and dequeued again. The iterator queue is given by:

$$\mathit{IQUEUE} \stackrel{\mathit{def}}{=} \nu q\ \nu \mathit{empty}\ \nu i\ \overline{\mathit{iqueue}}\langle q, \mathit{empty}, i\rangle.(\mathit{IQUEUE_0} \mid \mathit{IQUEUE})\ .$$

The iterator queue creates and returns, in addition to a queue, another restricted name i. It then behaves as follows:

$$\begin{aligned} \mathit{IQUEUE_0} \stackrel{\mathit{def}}{=} &\nu \mathit{it}\ \nu \mathit{itempty}\ (\overline{\mathit{empty}}.\mathit{IQUEUE_0} + q(\mathit{newvalue}).\mathit{triple}(\mathit{newtriple}). \\ &\overline{\mathit{newtriple}}\langle \bot, \bot, \mathit{newvalue}\rangle.\mathit{IQUEUE_1}(\mathit{newtriple}, \mathit{newtriple}) + \\ &\bar{i}\langle \mathit{it}, \mathit{itempty}\rangle.\overline{\mathit{itempty}}.\mathit{IQUEUE_0})\ . \end{aligned}$$

$\mathit{IQUEUE_0}$ has an additional summation accessed via i that returns two names for iterating the queue. However, only $\mathit{itempty}$ can be used for interaction, since the iterator queue is empty at this point. If a name has been enqueued, the iterator queue behaves as $IQUEUE_1$:

$$\begin{aligned}
&IQUEUE_1(first, last) \overset{def}{=} \\
&\qquad \nu it\ \nu itempty\ first(next, test, value).(\overline{q}\langle value\rangle. \\
&\qquad ([test = \top]IQUEUE_1(next, last) + [test = \bot]IQUEUE_0)+ \\
&\qquad q(newvalue).triple(newtriple).\overline{newtriple}\langle\bot, \bot, newvalue\rangle. \\
&\qquad last(oldnext, oldtest, oldvalue).\overline{last}\langle newtriple, \top, oldvalue\rangle. \\
&\qquad IQUEUE_1(first, newtriple) + \overline{i}\langle it, itempty\rangle.IQUEUE_2(next))\ .
\end{aligned}$$

$IQUEUE_1$ works as $QUEUE_1$, except for the restricted names it and $itempty$ and a possible interaction via i. If an interaction via i occurred, the iterator queue behaves as $IQUEUE_2$:

$$\begin{aligned}
&IQUEUE_2(curr) \overset{def}{=} \\
&\qquad [test = \top]curr(next, test, value).\overline{it}\langle value\rangle.IQUEUE_2(next)+ \\
&\qquad [test = \bot]\overline{itempty}.IQUEUE_1(first, last)\ .
\end{aligned}$$

When agent $IQUEUE_2$ is active, all remaining names inside the queue have to be read via it before $itempty$ can be read (according to definition 4.5 (Iterator)). Afterward, the iterator queue behaves as $IQUEUE_1$ again. □

The iterator queue provides a concurrent safe implementation of a queue, that has the property of blocking all enqueue and dequeue interactions while it is being iterated. Due to the blocking semantics, special care has to be taken inside potential iterator agents.

We are now able to define a safe variant of example 4.1 (Bank Counters) by using an iterator queue. The agent WQ_{SAFE} represents a correctly working waiting queue that can be inspected:

$$WQ_{SAFE} \overset{def}{=} iqueue(wq, we, i).(FILL \mid SERVE \mid SERVE \mid INSPECT_{SAFE})\ ,$$

where $FILL$ and $SERVE$ remain the same. The agent $INSPECT_{SAFE}$ is given by:

$$\begin{aligned}
INSPECT_{SAFE} &\overset{def}{=} i(it, itempty).INSPECT_{SAFE0} \\
INSPECT_{SAFE0} &\overset{def}{=} it(x).\tau.INSPECT_{SAFE0} + itempty.\mathbf{0}\ .
\end{aligned}$$

Since an iterator queue is used to inspect the waiting queue, names cannot be enqueued or dequeued while an iteration takes place. Thus, the problems found in example 4.1 (Bank Counters) do not exist any longer.

4.2 Values, Types, and Functions

This section introduces the representation of data values, data types, and functions in the π-calculus.

4.2.1 Booleans and Bytes

The basic unit of data is a *bit* that is represented as a boolean value.

Definition 4.7 (Boolean) A *boolean* represents a truth value inside a system of agents. It is given by

$$\nu\top\ \nu\bot\ S\ ,$$

where $\top$ represents true, $\bot$ represents false, and S represents the system of agents. □

For instance, a system S composed of two agents A and B that use boolean values is given by:

$$S \overset{def}{=} \nu\top\ \nu\bot\ \nu ch\ (A \mid B)\ ,$$
$$A \overset{def}{=} \tau.(\overline{ch}\langle\top\rangle.A + \overline{ch}\langle\bot\rangle.A)\ \text{, and}$$
$$B \overset{def}{=} ch(x).([x = \top]\tau.B' + [x = \bot]\tau.\mathbf{0})\ .$$

The agent S defines two restricted names representing true and false values as well as an interaction channel. The components A and B can then evolve concurrently. However, only A can evolve immediately, since B has no counterpart for interaction. A does some internal calculation that is abstracted from by τ and afterward sends either $\top$ or $\bot$ via the name ch. In both cases, A evolves by recursion as originally defined. In the second step of A, an interaction between A and B is possible. Thus, a possible interaction for B is given by:

$$ch(x).([x = \top]\tau.B' + [x = \bot]\tau.\mathbf{0}) \xrightarrow{ch(\top)} [\top = \top]\tau.B' + [\top = \bot]\tau.\mathbf{0}\ .$$

Since $\top \neq \bot$, only one active transition of the sum remains for B, making it deterministic (B can execute the left hand side of the sum). If A had sent $\bot$ via ch instead, the right hand side of B would have been enabled for execution. By regarding ch as a pointer, it clearly points to an agent A, that is able to return either $\top$ or $\bot$ an infinite number of times. Consequently, the *type* of the name ch can be said to be boolean, since it always points to an agent representing boolean values in S.

Definition 4.8 (Type) The *type* of a name n is given by the kind of data an agent able to interact via n represents. If more complex data can be accessed via multiple names, the names are subscripted with their corresponding part. □

The type of a name can be denoted with a colon behind the name, e.g. $raining : boolean$ or $patients : queue_{iterator}$ to make the terms more readable. In contrast to theoretical treatments such as given in [118], we consider types as purely additional information without any formal meaning. Thus, the type of a name only denotes what can be expected by using the name as the object of an input or output prefix. While different agents can interact via the same name, and a type overloading is also possible, we prohibit this for typed names. In other words, the type of n defines the codomain of a *function* that is pointed to by n. An example of a function represented by an agent is already given by A. This agent is able to emit boolean values in a non-deterministic manner; it represents a function that returns random boolean values. Since A considered as a function does not take any input, its signature is simply given by:

$$A :\rightarrow \mathit{boolean} \; .$$

Instead of providing a random boolean generator, two more elaborate agents provide constants for true and false values:

$$\mathit{TRUE} = \overline{\mathit{true}}\langle\top\rangle.\mathit{TRUE} \qquad \mathit{FALSE} = \overline{\mathit{false}}\langle\bot\rangle.\mathit{FALSE} \; .$$

These agents are assumed to be placed inside a system which restricts true and false as well as $\top$ and $\bot$ globally. An agent representing a function with parameters requires a two-way interaction. First the parameters and a response channel are transmitted and afterward the response is read via the response channel. A function AND representing a boolean disjunction with the signature

$$\mathit{AND} : \mathit{boolean} \times \mathit{boolean} \rightarrow \mathit{boolean}$$

that compares two booleans is given by the agent

$$\begin{aligned}\mathit{AND} \overset{\mathit{def}}{=} \mathit{and}(\mathit{b1}, \mathit{b2}, \mathit{resp}).\mathit{b1}(x).\mathit{b2}(y).([x = \top][y = \top]\overline{\mathit{resp}}\langle\top\rangle.\mathit{AND}+ \\ [x = \bot]\overline{\mathit{resp}}\langle\bot\rangle.\mathit{AND}+ \\ [y = \bot]\overline{\mathit{resp}}\langle\bot\rangle.\mathit{AND}) \; .\end{aligned}$$

The agent AND is made globally available inside a system using the restricted name and. When interacting via and, the subject is expected to consist of three parts: two names $\mathit{b1}$ and $\mathit{b2}$ representing pointers to booleans, and a third name resp used as a response channel. First, AND fetches the actual values of the pointers to the booleans. Second, it returns $\top$ via resp if both names $\mathit{b1}$ and $\mathit{b2}$ equal $\top$, and $\bot$ otherwise. Another system T composed out of

$$T \overset{\mathit{def}}{=} \nu\top \; \nu\bot \; \nu\mathit{true} \; \nu\mathit{false} \; \nu\mathit{and} \; (\mathit{TRUE} \mid \mathit{FALSE} \mid \mathit{AND} \mid C) \text{ , and}$$

$$C \overset{\mathit{def}}{=} \nu r \; \mathit{and}(\mathit{true}, \mathit{true}, r).r(x).([x = \top]\tau.C' + [x = \bot]\tau.C'') \; ,$$

with AND, TRUE, and FALSE given as above, uses the concepts introduced so far. However, the right hand side of agent C's sum will never be enabled due to the interaction with agent AND, where two true values are compared. Furthermore, agent AND only provides a one-time interaction via resp. A better solution for agent AND incorporates the return of a variable containing the result instead of directly providing it. A variable is represented by a *cell*.

The modified agent AND is given as follows, where we assume it to be placed inside a composition with CELL and the restricted names $\top$ and $\bot$:

Definition 4.9 (Boolean Conjunction) The agent AND compares two names typed as booleans for *boolean conjunction*.

$$\begin{aligned}\mathit{AND} &\overset{\mathit{def}}{=} \mathit{cell}(v).\mathit{and}(\mathit{b1}, \mathit{b2}, \mathit{resp}).\mathit{b1}(x).\mathit{b2}(y).([x = \top][y = \top]\overline{v}\langle\top\rangle.\mathit{AND_1}+ \\ &\quad [x = \bot]\overline{v}\langle\bot\rangle.\mathit{AND_1} + [y = \bot]\overline{v}\langle\bot\rangle.\mathit{AND_1}) \\ \mathit{AND_1} &\overset{\mathit{def}}{=} (\overline{\mathit{resp}}\langle v\rangle.\mathbf{0} \mid \mathit{AND}) \; .\end{aligned}$$

□

In addition to the agent AND introduced earlier, the new variant is not blocking until the response has been collected, since the modified AND is activated again using recursion placed in parallel with the response sent via $resp$. A boolean disjunction is given by:

Definition 4.10 (Boolean Disjunction) The agent OR compares two names typed as booleans for *boolean disjunction*:

$$\begin{aligned} OR &\overset{def}{=} cell(v).or(b1, b2, resp).b1(x).b2(y).([x = \bot][y = \bot]\overline{v}\langle\bot\rangle.OR_1 + \\ &\quad [x = \top]\overline{v}\langle\top\rangle.OR_1 + [y = \top]\overline{v}\langle\top\rangle.OR_1) \\ OR_1 &\overset{def}{=} (\overline{resp}\langle v\rangle.\mathbf{0} \mid OR) \ . \end{aligned}$$

□

Finally, a boolean negation is given by the agent NEG:

Definition 4.11 (Boolean Negation) The agent NEG applies *boolean negation* to a name typed as boolean.

$$\begin{aligned} NEG &\overset{def}{=} neg(b, resp).true(t).false(f).b(x).(\\ &\quad ([b = t]\overline{resp}\langle false\rangle.\mathbf{0} + [b = f]\overline{resp}\langle true\rangle.\mathbf{0}) \mid NEG) \ . \end{aligned}$$

□

The boolean negation incorporates the $TRUE$ and $FALSE$ agents to first fetch the actual names for true and false and furthermore returns the result as a constant. Agents 4.9 (Boolean Conjunction) and 4.10 (Boolean Disjunction) can be adapted to work the same way. We showed both variants to provide a choice for the application. Usage of the fixed names $\top$ and $\bot$ provides less overhead, whereas the agents $TRUE$ and $FALSE$ provide more flexibility regarding the actual names for true and false, as well as providing constants for them. In the remainder, we use the agents $TRUE$ and $FALSE$ as defined, e.g. providing the names $\top$ for true and $\bot$ for false. Thus, a fetching of the actual values for true and false is omitted.

A second unit of data is a *byte* that is represented by a tuple of eight bits:

Definition 4.12 (Byte) A *byte* is given by a tuple of eight boolean values used as subjects of input and outputs prefixes. The type of a byte is $byte$, e.g. $byte_{42} : byte$. □

For instance,

$$\langle\bot, \bot, \top, \bot, \top, \bot, \top, \bot\rangle$$

represents the decimal value 42. An agent returning a constant with this value is given by:

$$BYTE_{42} \overset{def}{=} \overline{byte42}\langle\bot, \bot, \top, \bot, \top, \bot, \top, \bot\rangle.BYTE_{42} \ ,$$

and accordingly for for each $i \in \{0 \ldots 255\}$ in $BYTE_i$. However, since a byte has only a fixed capacity and basic functions like addition and comparison can only be implemented using rather complex agents, they will not be discussed further. Instead, a representation of natural numbers as a generalization of bytes will be discussed.

4.2.2 Natural Numbers

Any natural number can be represented in a binary form as a sequence of true and false values. Sequences of true and false values can be represented using $QUEUE$ agents in π-calculus. Since queues work first in, first out, we define the lowest significant bit of natural number to be the first element of the queue:

Definition 4.13 (Natural Number) A *natural number* $n \in \mathbb{N}$ is represented as an iterator queue containing the binary representation of n constructed of true and false, where the first name in the queue represents the lowest significant bit. The type of a natural number is $number$, e.g. $num_9 : number$. Since numbers are almost ever used as constants, only the iterator accessor of an iterator queue is used if not stated otherwise. $\square$

An agent NUM_9 for constructing an iterator queue representing the decimal value 9 is given by:

$$NUM_9 \stackrel{def}{=} iqueue(n9, e9, i9).\overline{n9}\langle\top\rangle.\overline{n9}\langle\bot\rangle.\overline{n9}\langle\bot\rangle.\overline{n9}\langle\top\rangle.\overline{num_9}\langle i9\rangle.NUM_9 \ .$$

Since the size of an iterator queue is unbounded, there is no theoretical upper limit on the value of any natural number to be represented. We returned only the iterator accessor to avoid an unintended change of a natural number. When the value of a natural number is evaluated, two iterator names are received from the iterator accessor of the iterator queue.

Before two natural numbers can be further processed, e.g. added, they have to have the same size of their binary representation. That is, if the binary representation of a natural number $n2$ is shorter than another natural number $n1$, $n2$ has to be filled with false values. For instance, with

$$n1 = \langle\top, \top, \bot\rangle \text{ and } n2 = \langle\top, \bot\rangle \ ,$$

$n2$ has a shorter binary representation than $n1$ and thus has to be filled with an additional false value. The filling is called *normalization* of two queues. Technically, it simplifies the processing:

Definition 4.14 (Normalize) Two (iterator) queues representing natural numbers are *normalized*, i.e. the length of the queues is adjusted to the same, by an agent $NORM$:

$$\begin{aligned} NORM \stackrel{def}{=} & norm(n1, n2, resp).n1(q1, e1).n2(q2, e2). \\ & iqueue(q3, e3, i3).iqueue(q4, e4, i4). \\ & (NORM_1 \mid NORM) \ . \end{aligned}$$

The agent $NORM$ first receives two iterator accessors $n1$ and $n2$ as well as a response channel $resp$. Via $n1$ and $n2$, the iterator names $q1$ and $q2$, with their corresponding empty queue names $e1$ and $e2$ for accessing the values of the two numbers, are received. Afterward, two new iterator queues $q3$ and $q4$ used as resulting queues are allocated. Most of the work is done in the nested terms of agent $NORM_1$:

$$\begin{aligned} NORM_1 \stackrel{def}{=} & q1(x).\overline{q3}\langle x\rangle.(q2(y).\overline{q4}\langle y\rangle.NORM_1 + e2.FILL_2)+ \\ & e1.(q2(y).\overline{q4}\langle y\rangle.FILL_1 + e2.DONE) \ . \end{aligned}$$

$NORM_1$ starts by either reading a name x via $q1$, if one is available, or receive via $e1$. If x has been received, it is enqueued in $q3$ and thereafter a name y is read via $q2$, if one is available, or an $e2$ is received. If y has been received, it is enqueued in $q4$ and $NORM_1$ recursively behaves as $NORM_1$ again, which means that the processed parts of both queues have the same length. If $e2$ is received instead of y via $q2$, it means that the length of $q2$ is less than the length of $q1$ and thus $q4$ has to be filled to the same length as $q1$ by agent $FILL_2$. If an $e1$ is received at the top-level summation, it is tested if a name y can be read via $q2$, or the second queue is also empty by signaling $e2$. In the former case, y is enqueued in $q4$ and $NORM_1$ behaves as $FILL_1$ to fill $q3$ to the same length as $q2$, since $q1$ is shorter than $q2$. In the latter case, both queues already have the same size and $NORM_1$ behaves as agent $DONE$:

$$\begin{aligned}
FILL_1 &\overset{def}{=} \overline{q3}\langle\bot\rangle.(q2(y).\overline{q4}\langle y\rangle.FILL_1 + e2.DONE)\\
FILL_2 &\overset{def}{=} \overline{q4}\langle\bot\rangle.(q1(x).\overline{q3}\langle x\rangle.FILL_2 + e1.DONE)\\
DONE &\overset{def}{=} \overline{resp}\langle i3, i4\rangle.\mathbf{0}\ .
\end{aligned}$$

The $FILL$ agents first insert a false value ($\bot$) into the queue to be filled and thereafter tries to read another name via the longer queue's name. If this succeeds, the name is enqueued in the corresponding result queue and $FILL$ behaves as $FILL$ recursively again. If it fails, no more truth values are available and thus both result queues $q3$ and $q4$ now have the same length. $DONE$ simply returns the iterator accessors $i3$ and $i4$ representing normalized versions of $n1$ and $n2$. □

By employing normalization on two natural numbers $n1$ and $n2$, we can compare them boolean-wise using the iterator queues as shift registers:

Definition 4.15 (Compare) Two natural numbers can be *compared* on equivalence by a function with the signature

$$CMP : number \times number \rightarrow boolean\ ,$$

represented by the agent CMP:

$$\begin{aligned}
CMP &\overset{def}{=} \nu r\ cmp(n1, n2, resp).\overline{norm}\langle n1, n2, r\rangle.r(n3, n4).\\
&\quad n3(q1, e1).n4(q2, e2).(CMP_1 \mid CMP)\\
CMP_1 &\overset{def}{=} q1(a).q2(b).([a = b]CMP_1 + [a = \bot][b = \top]\overline{resp}\langle false\rangle.\mathbf{0}+\\
&\quad [a = \top][b = \bot]\overline{resp}\langle false\rangle.\mathbf{0}) + e1.\overline{resp}\langle true\rangle.\mathbf{0}\ .
\end{aligned}$$

Analog to $NORM$, agent CMP starts by receiving two iterator accessors $n1$ and $n2$ as well as a response channel *resp* via *cmp*. Thereafter agent CMP receives the iterator names via $n1$ and $n2$, normalizes them, and behaves as agent CMP_1. CMP_1 tries to fetch the first boolean value of the first natural number via $q1$ and then the first boolean value of the second natural number via $q2$. The boolean values are evaluated afterward. If both are the same, CMP_1 is recursively enabled again. If both are different, the constant *false* is returned, since both natural numbers are different. If in the top-level summation $e1$ is received, the queues representing the natural numbers are empty and thus no more boolean values are available for comparison. Hence, both natural numbers are the same and the constant $true$ is returned. Notable, it is not necessary to empty the queues $n3$ and $n4$, since these are local to a certain occurrence of CMP. □

Agent CMP gives an idea of how normalized iterator queues representing natural numbers can be used as shift registers for boolean-wise processing. Another common operation is testing if a natural number is less than another natural number:

Definition 4.16 (Less) For two natural numbers, $n1$ and $n2$, it can be tested if $n1$ is *less* than $n2$. Hence, an ordering criterion for natural numbers is provided. The signature of the function is given by

$$LESS : number \times number \rightarrow boolean\ ,$$

and the corresponding implementation by $LESS$:

$$\begin{aligned} LESS \overset{def}{=} & \nu r\ less(n1, n2, resp).\overline{norm}\langle n1, n2, r\rangle.r(n3, n4). \\ & n3(q1, e1).n4(q2, e2).(LESS_1(\bot) \mid LESS)\ . \end{aligned}$$

$LESS$ receives two names $n1$ and $n2$ typed as numbers and a response channel $resp$ via $less$. After normalizing and receiving the iterator names, it behaves as $LESS_1$ with the parameter $\bot$:

$$\begin{aligned} LESS_1(c) \overset{def}{=} q1(b1).q2(b2).(&[b1 = \bot][b2 = \bot]LESS_1(c)+ \\ &[b1 = \top][b2 = \top]LESS_1(c)+ \\ &[b1 = \bot][b2 = \top]LESS_1(\top)+ \\ &[b1 = \top][b2 = \bot]LESS_1(\bot))+ \\ &e1.([c = \top]\overline{resp}\langle true\rangle.\mathbf{0} + [c = \bot]\overline{resp}\langle false\rangle.\mathbf{0})\ . \end{aligned}$$

The parameter c of $LESS_1$ is used to denote if the first natural number represented by $q1$ is currently less than the second natural number represented by $q2$. $LESS_1$ fetches the current boolean values $b1$ and $b2$ of the queues $q1$ and $q2$. The following summation considers all possibilities. If the boolean values $b1$ and $b2$ are equal, the remainder of the agent behaves again as $LESS_1$ with the current result c as a parameter. If $b1$ is true and $b2$ is false, the agent behaves as $LESS_1$ with true as a parameter. If $b1$ is false and $b2$ is true, the agent evolves to $LESS_1$ with false as a parameter. If, however, the queue $q1$ is empty by signaling $e1$, a constant $true$ or $false$ for the current result is returned via $resp$. □

We can also construct an agent ADD for the boolean-wise addition of two natural numbers. The result is a natural number representing the result of the addition:

Definition 4.17 (Add) Two natural numbers can be *added* by a function with the signature

$$ADD : number \times number \rightarrow number\ ,$$

represented by the agent ADD:

$$\begin{aligned} ADD \overset{def}{=} & \nu r\ add(n1, n2, resp).\overline{norm}\langle n1, n2, r\rangle.r(n3, n4). \\ & n3(q1, e1).n4(q2, e2).iqueue(q3, e3, i3).(ADD_1(\bot) \mid ADD)\ . \end{aligned}$$

In addition to CMP and LESS, ADD creates a new iterator queue $q3$ used as return value. ADD_1 is given by:

$$\begin{aligned}
\mathit{ADD}_1(c) \overset{def}{=} q1(b1).q2(b2).([b1 = \bot][b2 = \bot][c = \bot]\overline{q3}\langle\bot\rangle.\mathit{ADD}_1(\bot)+ \\
[b1 = \bot][b2 = \bot][c = \top]\overline{q3}\langle\top\rangle.\mathit{ADD}_1(\bot)+ \\
[b1 = \top][b2 = \bot][c = \bot]\overline{q3}\langle\top\rangle.\mathit{ADD}_1(\bot)+ \\
[b1 = \top][b2 = \bot][c = \top]\overline{q3}\langle\bot\rangle.\mathit{ADD}_1(\top)+ \\
[b1 = \bot][b2 = \top][c = \bot]\overline{q3}\langle\top\rangle.\mathit{ADD}_1(\bot)+ \\
[b1 = \bot][b2 = \top][c = \top]\overline{q3}\langle\bot\rangle.\mathit{ADD}_1(\top)+ \\
[b1 = \top][b2 = \top][c = \bot]\overline{q3}\langle\bot\rangle.\mathit{ADD}_1(\top)+ \\
[b1 = \top][b2 = \top][c = \top]\overline{q3}\langle\top\rangle.\mathit{ADD}_1(\top))+ \\
e1.\overline{q3}\langle c\rangle.\overline{resp}\langle i3\rangle.\mathbf{0}\ .
\end{aligned}$$

The agent ADD_1 has a parameter c that is used to represent a *carry flag* that denotes if an overflow occurred. The evolution of ADD_1 starts by fetching the current boolean values $b1$ and $b2$ of the queues representing the natural numbers used as input. The following summation considers all eight possibilities regarding $b1$, $b2$, and c. Accordingly, a new boolean value is added to the queue $q3$ and then the agent behaves again as ADD_1 with the new value of the carry flag as parameter. If $q1$ is empty, i.e. $e1$ is signaled, the carry flag is enqueued at $q3$ and the iterator name of $q3$ are returned via $resp$. □

An agent SUB for subtraction can be given accordingly to ADD. Since this is straightforward, we omit the definition.

4.2.3 Syntactical Extensions

Since the representation of natural numbers is a common task, we introduce a syntactical extension that eases the use. In particular, we introduce constant names and according agents for all natural numbers $n \in \mathbb{N}$ and extended match constructs for evaluating them. Technically, the set of names is extended by all natural numbers: $\mathcal{N}_\mathbb{N} = \mathcal{N} \cup \mathbb{N}$, the agent identifiers are extended by the corresponding agents that produce the natural numbers: $\mathcal{K}_\mathbb{N} = \mathcal{K} \cup \{\mathit{NUM}_x \mid x \in \mathbb{N}\}$, and the extended prefixes $\pi_\mathbb{N}$ are given by:

$$\begin{aligned}
\pi_\mathbb{N} ::= &\overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi_\mathbb{N} \mid \\
&\text{if } n1 : \mathit{number}\{==, \neq, <, >\} n2 : \mathit{number} \text{ then } P \text{ else } P'\ .
\end{aligned} \tag{4.1}$$

The names $n1$ and $n2$ in equation 4.1 are typed as numbers, thus the extension only applies to natural numbers. We provide infix operators for (1) the equality of two natural numbers, denoted as $n1 == n2$, (2) the inequality of two natural numbers, denoted as $n1 \neq n2$, (3) a test if $n1$ is less than $n2$, denoted as $n1 < n2$, and (4) a test if $n1$ is greater than $n1$, denoted as $n1 > n2$.

If the else part is omitted, inaction ($\mathbf{0}$) is assumed instead. Usage examples are given by

$$
\begin{aligned}
A &\overset{def}{=} \nu resp\ \overline{add}\langle 4, 5, resp\rangle.resp(n).\text{if } n == 9 \text{ then } A' \text{ ,}\\
B &\overset{def}{=} b(age : number).\text{if } age < 20 \text{ then } B' \text{ else } B'' \text{ , and}\\
C &\overset{def}{=} c(i : queue_{iterator}, size).i(q1, e1).C_1(0)\\
C_1(count) &\overset{def}{=} \nu resp\ q1(x).\overline{add}\langle count, 1, resp\rangle.resp(e).C_1(e) +\\
&\quad e1.\overline{size}\langle count\rangle.C \text{ .}
\end{aligned}
\tag{4.2}
$$

Agent A first adds two natural numbers given as the constants 4 and 5. The respective agents are assumed to be:

$$
\begin{aligned}
NUM_4 &\overset{def}{=} iqueue(n4, e4, i4).\overline{n4}\langle\bot\rangle.\overline{n4}\langle\bot\rangle.\overline{n4}\langle\top\rangle.\overline{num_4}\langle i4\rangle.NUM_4 \text{ and}\\
NUM_5 &\overset{def}{=} iqueue(n5, e5, i5).\overline{n5}\langle\top\rangle.\overline{n5}\langle\bot\rangle.\overline{n5}\langle\top\rangle.\overline{num_5}\langle i5\rangle.NUM_5 \text{ ,}
\end{aligned}
$$

where the constants are received via $num_4(4)$ and $num_5(5)$. The constant 9 used later on is acquired in the same way. The statement if $n == 9$ then A' is expanded using the agent CMP as follows:

$$\nu r\ cmp(n, 9, r).r(e).[e = true]\tau.A' \text{ .}$$

The complete expanded agent A is then given by:

$$
\begin{aligned}
A \overset{def}{=} &\nu resp\ \nu r\ num_4(4).num_5(5).num_9(9).\overline{add}\langle 4, 5, resp\rangle.resp(n).\\
&\overline{cmp}\langle n, 9, r\rangle.r(e).[e = true]\tau.A' \text{ .}
\end{aligned}
$$

The agent B first receives a name age typed as natural number via b and then behaves as B' if $age < 20$ and otherwise as B''. The expanded agent B is given by:

$$
\begin{aligned}
B \overset{def}{=} &\nu r\ num_{20}(20).b(age : number).\overline{less}\langle age, 20, r\rangle.r(e).\\
&([e = true]\tau.B' + [e = false]\tau.B'') \text{ .}
\end{aligned}
$$

Agent C counts the number of elements contained inside an iterator queue. Since no new extensions are contained, the expanded agent is omitted. The formal syntactical enhancements for natural numbers are given by:

$$
\begin{aligned}
n \mid n \in \mathbb{N} &\longmapsto num_n(n) \text{ before the first use of the constant,}\\
\text{if } n1 == n2 \text{ then } P \text{ else } P' &\longmapsto \nu r\ \overline{cmp}\langle n1, n2, r\rangle.r(e).([e = true]\tau.P + [e = false]\tau.P')\\
\text{if } n1 \neq n2 \text{ then } P \text{ else } P' &\longmapsto \nu r\ \overline{cmp}\langle n1, n2, r\rangle.r(e).([e = true]\tau.P' + [e = false]\tau.P)\\
\text{if } n1 < n2 \text{ then } P \text{ else } P' &\longmapsto \nu r\ \overline{less}\langle n1, n2, r\rangle.r(e).([e = true]\tau.P + [e = false]\tau.P')\\
\text{if } n1 > n2 \text{ then } P \text{ else } P' &\longmapsto \nu r\ \overline{less}\langle n2, n1, r\rangle.r(e).([e = true]\tau.P + [e = false]\tau.P')
\end{aligned}
$$

If the else part is omitted, the right hand side of the summation in the syntactical expansion is left out. Furthermore, we require an infinite number of agents $NUM_n \mid n \in \mathbb{N}$ that incorporate an iterator queue to generate new queues containing the boolean representation of n, e.g.

$$\begin{aligned}
\mathit{NUM_0} &\stackrel{def}{=} \mathit{iqueue}(\mathit{n0}, \mathit{e0}, \mathit{i0}).\overline{\mathit{n0}}\langle\bot\rangle.\overline{\mathit{num_0}}\langle \mathit{i0}\rangle.\mathit{NUM_0} \ , \\
\mathit{NUM_1} &\stackrel{def}{=} \mathit{iqueue}(\mathit{n1}, \mathit{e1}, \mathit{i1}).\overline{\mathit{n1}}\langle\top\rangle.\overline{\mathit{num_1}}\langle \mathit{i1}\rangle.\mathit{NUM_1} \ , \\
\mathit{NUM_2} &\stackrel{def}{=} \mathit{iqueue}(\mathit{n2}, \mathit{e2}, \mathit{i2}).\overline{\mathit{n1}}\langle\bot\rangle.\overline{\mathit{n1}}\langle\top\rangle.\overline{\mathit{num_2}}\langle \mathit{i2}\rangle.\mathit{NUM_2} \ , \\
\mathit{NUM_3} &\stackrel{def}{=} \dots \ .
\end{aligned}$$

Note that each agent $\mathit{NUM_n}$ creates a fresh iterator queue instead of acting as a singleton on one queue. This is required for the concurrent evaluation of two iterator queues representing the same natural number.

By using the natural number extension, we can provide language constructs like while loops known from imperative languages: `while (i=0; i++; i<3) BLOCK` . An instance is given as follows, where the block is represented by τ:

$$\begin{aligned}
A &\stackrel{def}{=} A_0(0) \\
A_0(i) &\stackrel{def}{=} \nu \mathit{resp} \text{ if } i < 3 \text{ then } \tau.\overline{\mathit{add}}\langle i, 1, \mathit{resp}\rangle.\mathit{resp}(r).A_0(r) \text{ else } P \ .
\end{aligned}$$

Similar constructs like until or for loops are also possible. Furthermore, we can define an agent that interacts with another agent a given times via a name also given:

$$\begin{aligned}
\mathit{EXEC_W} &\stackrel{def}{=} \mathit{exec_{while}}(\mathit{times} : \mathit{number}, n).(\mathit{EXEC_{W0}}(0) \mid \mathit{EXEC_W}) \\
\mathit{EXEC_{W0}}(\mathit{count}) &\stackrel{def}{=} \nu r \text{ if } \mathit{count} < \mathit{times} \text{ then } (\overline{n}.\mathbf{0} \mid \\
&\qquad \mathit{add}(\mathit{count}, 1, r).r(c).\mathit{EXEC_{W0}}(c)) \ .
\end{aligned}$$

This agent is called *while executor*, since it interacts repeatedly with another agent in a while loop manner.

4.2.4 Derived Values and Structures

Using standard techniques, further data values can be derived. We give examples for character strings and lists. A character string is represented as a queue and the contained characters are given by natural numbers. Different encodings for characters as natural numbers can be used. For the ease of presentation we refer to ASCII [45].

Definition 4.18 (Character String) A *character string* stores characters in order. It is given by an iterator queue of natural numbers. Starting with the first element in the queue, each natural number encodes a character of the string. The type of a character string is string. If a character string is used as a constant, only the iterator name is used if not stated otherwise. □

For instance, a character string containing ”HELLO WORLD” using ASCII encoding is given by:

$$\begin{aligned}
S \stackrel{def}{=} &\mathit{iqueue}(q, e, i).\overline{q}\langle 72\rangle.\overline{q}\langle 69\rangle.\overline{q}\langle 76\rangle.\overline{q}\langle 76\rangle.\overline{q}\langle 79\rangle.\overline{q}\langle 32\rangle.\overline{q}\langle 87\rangle.\overline{q}\langle 79\rangle.\overline{q}\langle 82\rangle.\overline{q}\langle 76\rangle. \\
&\overline{q}\langle 68\rangle.\overline{s}\langle i\rangle.S \ .
\end{aligned}$$

The size of a character string is given by the length of the corresponding iterator queue. Hence, agent C from equation 4.2 can be used to count the characters in a given character string.

Definition 4.19 (String Comparison) A *string comparison* on equality of two character strings regarding their contents can be made using a function with the signature

$$CMPS : string \times string \rightarrow boolean \ ,$$

given by the agent $CMPS$:

$$\begin{aligned} CMPS &\overset{def}{=} cmps(s1, s2, resp).s1(q1, e1).s2(q2, s2).CMPS_1 \\ CMPS_1 &\overset{def}{=} q1(x).(q2(y).\text{if } x == y \text{ then } CMPS_1 \text{ else } \overline{resp}\langle false\rangle + \\ &\quad e2.CMPS_2(q1, e1)) + \\ &\quad e1.(q2(y).CMPS_2(q2, e2) + e2.\overline{resp}\langle true\rangle) \\ CMPS_2(q, e) &\overset{def}{=} q(z).CMPS_2(q, e) + e.\overline{resp}\langle false\rangle.\mathbf{0} \ . \end{aligned}$$

Agent $CMPS$ first fetches the queues containing the characters and uses nested summations in $CMPS_1$ to compare the natural numbers representing the characters in a recursive manner. If one of the queues is shorter than the other one, the shorter one has to be iterated completely to unlock the corresponding iterator. This is done in $CMPS_2$. □

We can introduce further functions and syntactical extensions to ease the handling of character strings in the π-calculus. However, the ideas should have become clear by now. An extended discussion on how to implement advanced structures in a concurrent programming language based on the π-calculus has been done by Turner in [123]. For the process and interaction patterns introduced later on, we still need one more definition that will conclude our presentation of values and structures. The last one is called *list*:

Definition 4.20 (List) A *list* stores names that can be removed or retrieved. Names can be contained in the list several times. The list consists of three operations, *append* to add names to the list, *remove* to remove names from the list if they are contained, and *iterate* to iterate over the content. Elements inside the list are identified using natural numbers. The list presented here is an ordered list given by:

$$\begin{aligned} LIST &\overset{def}{=} \nu app\ \nu rem\ \nu it\ iqueue(q, e, i).num_0(id).\overline{list}\langle app, rem, it\rangle. \\ &\quad (LIST_0 \mid LIST) \ . \end{aligned}$$

$LIST$ first creates three fresh names, app to append names to the list, rem to remove a name at a specific position from the list, and it to receive non-destructive iterator names for the list. Thereafter an iterator queue for storing the values of the list and a name representing the latest id used inside the list are created. The restricted names are thereafter sent via $list$. The agent then behaves as follows:

$$\begin{aligned} LIST_1 &\overset{def}{=} \nu r\ \nu li\ \nu ei\ (app(n, ch).pair(t).\bar{t}\langle id, n\rangle.\overline{ch}\langle id\rangle.\bar{q}\langle t\rangle.\overline{add}\langle id, 1, r\rangle.r(id).LIST_1 + \\ &\quad rem(idr : number).queue(qtmp, etmp).LIST_2 + \\ &\quad \overline{it}\langle li, ei\rangle.i(qi, qe).LIST_3) \ . \end{aligned}$$

$LIST_1$ first creates three fresh names, where r is used as a response channel for an add interaction inside the app operation, and li and ie are used as iterator names for the it operation. The operations are placed inside a sum. The append operation receives a name n to be appended via app as well as a channel ch, creates a new pair t, inserts the current id and the received name n into the pair, enqueues the pair in q, returns the id via ch, and finally increases the id by one. The remove operation receives the identification number idr of the name to be removed via rem, allocates a new queue with $qtmp$ and $etmp$, and then behaves as $LIST_2$. The iterator operation returns the fresh names li and ei via it, receives the two iterator names from i to iterate the list, and behaves as $LIST_3$. The agent $LIST_2$ removes an element identified by idr from the list if it is contained:

$$LIST_2 \stackrel{def}{=} q(v).(v(idt, nt).\text{if } idt \neq idr \text{ then } \overline{qtmp}\langle v\rangle.LIST_2 \text{ else } LIST_2) + e.LIST_{21}$$
$$LIST_{21} \stackrel{def}{=} qtmp(v).\overline{q}\langle v\rangle.LIST_{21} + etmp.LIST_1\ .$$

The agent $LIST_3$ encapsulates the iterator of q to provide a consistent behavior:

$$LIST_3 \stackrel{def}{=} qi(x).\overline{il}\langle x\rangle.LIST_3 + qe.\overline{ei}.LIST_1\ .$$

□

4.3 Data Patterns

After the last sections laid the cornerstones for representing data in the π-calculus, this section describes how data is represented in business processes based on the workflow data patterns [113]. We have to distinguish three different kinds of data: (1) data used inside activities for internal calculations and decision making, (2) process instance data, and (3) data provided by the environment. We represent each kind of data as π-calculus agents. For instance, we have agents that represent internal data, like a cell referencing a natural number, agents representing process instance data, like a list containing character strings that describe an insurance claim, and agents representing environmental data like external triggers, sensors, or constants. We implement the access restrictions for each kind of data by using π-calculus restrictions. Each activity of a business process is represented by an agent consisting of only τ as an abstraction for the functional perspective. Since possible control flow dependencies between activities will be discussed in detail in chapter 5 (Processes), we focus on the data aspects of activities in the remainder of this chapter.

Figure 4.1 shows a sample system of agents focusing on data. It contains six agents representing activities (A-F) and four additional ones that represent data as introduced before. The agent A has access to process instance data, represented by the agent X. During the execution of the business process composed out of the activities, the restricted name x will be forwarded to other agents representing activities. Agent B has access to data provided for all process instances by the business process management system here represented by agent Z. Since access occurs via the free name z, every agent representing an activity can incorporate this data. Agent C uses activity internal data via the restricted name y it shares with agent Y. The scope of the

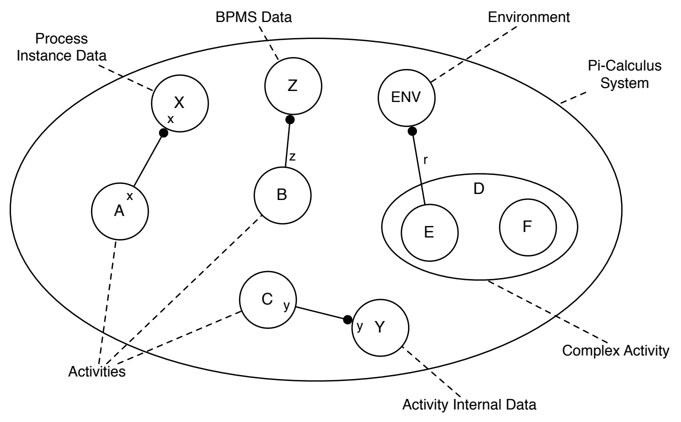


Figure 4.1: Flow graph of agents representing business process activities and data.

name y will not be extruded any further. The agent D consists of two components E and F and thus describes a complex activity. If the scope of a name is extruded to D, it should also include E and F. Finally, agent E uses data provided by the environment via the name r. Access to the environment can occur by either restricted names scoped to certain activities; i.e. external triggers, or by free names representing constants or functions. Examples for each of the different types of data found in business processes are contained in the data visibility patterns subsection.

Furthermore, we do not make a sharp distinction between activities and activity instances (accordingly for processes and process instances). An activity is given by an agent according to its definition; whereas an activity instance is given by an agent that already evolved at least once (see chapter 5.1.3 for details). To keep consistency with the terms introduced in chapter 3 (Business Process Management), we adapt the pattern names given in the data pattern documentation [113] to the introduced terminology. This regards tasks, that are denoted as activities, cases, that are denoted as process instances, workflows that are denoted as processes, sub processes that are denoted as complex activities, as well as workflow management systems that are denoted as business process management systems. Since the data pattern descriptions are complex, and only given in natural language, we focus on examples of the different implementation possibilities. Hence, in a pattern like style, we show one adequate solution for each pattern without assuming completeness.

4.3.1 Data Visibility Patterns

Data visibility patterns define different layers of accessibility for data elements. The layers are depicted in figure 4.2. Inner layers have access to shared data of all outer layers, wheras the converse does not hold. For instance, an activity can access shared data of a complex activity it is part of, incorporate process instance data, and data provided for all instances by the BPMS and the environment. A process, however, has no permission to access data that is restricted to a certain activity. The different data visibility patterns are discussed in this subsection.

Pattern 4.1 (Activity Data) *Description: Data elements can be defined by activities which are*

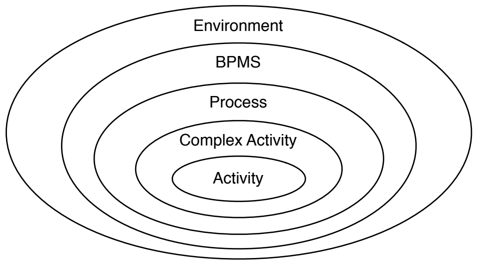


Figure 4.2: Different data layers.

accessible only within the context of individual execution instances of that activity. (According to [113, p.6])

Implementation: Each activity can use restricted names for internal calculations. These names can either be directly created using the ν operator or by creating new data structures such as a cell. For instance,

$$A \overset{def}{=} \nu x \; cell(c).\tau.\mathbf{0} \; ,$$

represents an activity that (1) creates a restricted name x used for internal calculation, and (2) acquires another restricted name c pointing to a cell. The scope of x is restricted to A, whereas c is restricted between $CELL$ and A.

Pattern 4.2 (Complex Activity Data) *Description: Complex activities are able to define data elements, which are accessible by each of their components. (According to [113, p.7])*

Implementation: A complex activity is represented by an agent consisting of several components, where each component represents an activity. Complex activity data is then created according to pattern 4.1 (Activity Data), with the distinction that the names are scoped to all components. For instance,

$$C \overset{def}{=} queue(q, e).(A \mid B) \; ,$$

represents a complex activity C with the activities A and B contained inside. C first creates a new queue q, that can afterward be accessed by A and B.

Pattern 4.3 (Scope Data) *Description: Data elements can be defined which are accessible by a subset of the activities in a process instance. (According to [113, p.9])*

Implementation: A process instance is given by an agent consisting of several components which represent activities and complex activities. Simple subsets can be defined by restricting the scope of a name to certain components. More complex scopes (i.e. overlapping ones) require the use of data interaction patterns introduced later on. For instance,

$$I \overset{def}{=} (A \mid B \mid \nu z \; (C \mid D)) \; ,$$

restricts the scope of the name z between the components C and D.

Pattern 4.4 (Multiple Instance Data) *Description: Activities which are able to execute multiple times within a single process instance can define data elements which are specific to an* *individual execution instance. (According to [113, p.10])*

Implementation: Pattern 4.1 (Activity Data) can be applied to provide each instance of an activity with its own restricted names. For instance,

$$M \overset{def}{=} \nu x\ \tau.M + \tau.\mathbf{0}\ ,$$

provides multiple executions of the functional part τ, each with its own restricted name x representing a data element.

Pattern 4.5 (Process Instance Data) *Description: Data elements are supported which are specific to a process instance. They can be accessed by all components of the process during the execution of the process instance. (According to [113, p.12])*

Implementation: Since a complex activity represents a process (see definition 3.10), the solution from pattern 4.2 (Complex Activity Data) is sufficient.

Pattern 4.6 (Business Process Management System Data) *Description: Data elements are supported which are accessible to all components in each and every process instance and are within the control of the business process management system (BPMS). (According to [113, p.13])*

Implementation: This pattern requires the definition of a BPMS in π-calculus. Basically, a BPMS is an agent consisting of a component representing a process that can be enacted several times. Data available to all components has then to be defined inside the BPMS agent. For instance,

$$BPMS \overset{def}{=} stack(s,e).(P_{enact}) \text{ and } P_{enact} \overset{def}{=} start.(P \mid P_{enact})\ ,$$

creates a new instance of a process represented by agent P each time the agent $BPMS$ receives the name $start$. Immediately, further instances can be created using recursion. All instances have access to the stack created first in $BPMS$.

Pattern 4.7 (Environment Data) *Description: Data elements, which exist in the external operating environment, are able to be accessed by components of the process during execution. (According to [113, p.14])*

Implementation: This pattern requires the definition of an environment. Basically, an environment is represented by an agent $\mathcal{E}$ enacted concurrently with a $BPMS$ agent. For instance,

$$SYS \overset{def}{=} \nu sensor\ (BPMS \mid \mathcal{E})\ ,$$

defines a system consisting of a BPMS and environment. The environment agent $\mathcal{E}$ can interact with the $BPMS$ agent via $sensor$, that is available to all components inside SYS.

4.3.2 Data Interaction Patterns

Data interaction patterns describe how activities of a business process can exchange data. The data interaction patterns are parted into internal and external ones. We only discuss internal data interaction, since external data interaction is closely related to the service interaction patterns that will be discussed in detail in chapter 6 (Interactions).

Pattern 4.8 (Data Interaction—Activity to Activity) *Description: The ability to communicate data elements between one activity instance and another within the same process instance. (According to [113, p.16])*

Implementation: Two activities can exchange data by the use of restricted names. The restrictions should only cover the agents representing the activities involved under consideration of SC-RES-COMP . For instance, in a process with two activities represented by the agent

$$P \overset{def}{=} \nu d\ (\mathit{cell}(a).\tau.\overline{d}\langle a\rangle.\mathbf{0} \mid d(x).\tau.\mathbf{0})\ ,$$

the left hand component (i.e. activity) passes the name a to the right hand component (i.e. activity) using the restricted name d. Furthermore, activity to activity data interaction can take place by adapting pattern 4.5 (Process Instance Data).

Pattern 4.9 (Data Interaction—Complex Activity Decomposition) *Description: The ability to pass data elements to a complex activity. (According to [113, p.18])*

Implementation: A complex activity receives data from preceding activities or other complex activities by receiving it via a restricted name according to pattern 4.8 (Data Interaction—Activity to Activity). For instance, a complex activity receiving a name available to all of its activities is given as

$$C \overset{def}{=} d(x).(A \mid B)\ .$$

Consequently, the name d has to be restricted between the agent representing the preceding activity and C.

Pattern 4.10 (Data Interaction—Complex Activity Finalization) *Description: The ability to pass data elements from a complex activity. (According to [113, p.20])*

Implementation: This pattern complements the preceding pattern. However, a substantial extension to complex activities is required, namely an explicit synchronization of the components. This is again done using restricted names. For instance,

$$C \overset{def}{=} \nu c1\ \nu c2\ (\mathit{cell}(u).\tau.\overline{c1}\langle u\rangle.\mathbf{0} \mid \nu v\ \tau.\overline{c2}\langle v\rangle.\mathbf{0} \mid c1(x).c2(y).\overline{d}\langle x, y\rangle.\mathbf{0})$$

shows an agent with three components representing a complex activity. The left component (i.e. activity) acquires a new cell u, whereas the middle component creates a restricted name v. Both names, u and v, are sent as subject in the complex activity synchronization component, represented by the right hand term. The agents representing the activities contained in the complex activity are synchronized via $c1$ and $c2$. The data is transmitted to an agent representing the subsequent activity via d.

Pattern 4.11 (Data Interaction—To Multiple Instance Activities) *Description: The ability to pass data elements from a preceding activity instance to a subsequent activity which is able to support multiple instances. This may involve passing the data elements to all instances of the multiple instances activity or distributing them on a selective basis. (According to [113, p.20])*

Implementation: This pattern distinguishes two possibilities: Either all activity instances work on the same, shared data or each instance receives a specific data element to work on. An example for the first approach is given by:

$$M \overset{def}{=} cell(c).N \text{ and } N \overset{def}{=} \tau.N + \tau.\mathbf{0}\ .$$

Agent M first creates a cell that is shared by all instances created in N. An example for the second approach is given by:

$$M \overset{def}{=} m(q, e).N \text{ and } N \overset{def}{=} (q(x).\tau.\mathbf{0} \mid N) + e.\mathbf{0}\ .$$

The second example uses a queue as input. For each entry of the queue, an instance is created that works on the specific entry.

Pattern 4.12 (Data Interaction—From Multiple Instance Activities) *Description: The ability to pass data elements from an activity which supports multiple execution instances to a subsequent activity. (According to [113, p.22])*

Implementation: Just like the preceding pattern, also this pattern distinguishes two possibilities. Either all activity instances return a shared data element or each instance returns a specific element. Since different types of multiple instance activities are discussed in detail in chapter 5 (Processes), we consider multiple instances without any synchronization. For the first approach (shared data), the current calculated value could be accessed any time, whereas for the second approach (individual data), access is possible as soon as the last instance has been created. We give an example for the latter case:

$$M \overset{def}{=} queue(q, e).N \text{ and } N \overset{def}{=} (\nu x\ \tau.\overline{q}\langle x\rangle.\mathbf{0} \mid N) + \tau.\overline{d}\langle q, e\rangle.\mathbf{0}\ .$$

The queue containing the results is created in agent M and filled by each recursive call of N that represents a multiple instance activity. After N decides that no more recursion should happen, i.e. the right hand term of the sum is chosen, the queue is sent via d to the subsequent activity.

Pattern 4.13 (Data Interaction—Process Instance to Process Instance) *Description: The passing of data elements from process instance during its execution to another process instance that is executing concurrently. (According to [113, p.23])*

Implementation: This pattern can be implemented by employing shared data at the BPMS or environment level. Thus, pattern 4.6 (Business Process Management System Data) or pattern 4.7 (Environment Data) can be applied.

4.3.3 Data Transfer Patterns

The data transfer patterns describe mechanisms for the actual transfer of data elements. They extend the patterns introduced in section 4.3.1. Since these patterns deal with technical details such as data passing by value or reference, not everything can be represented in the π-calculus.

Pattern 4.14 (Data Transfer by Value—Incoming) *Description: The ability of an activity to receive incoming data elements by value relieving it from the need to have shared names or common address space with the activities from which it receive them. (According to [113, p.34])*

Implementation: The π-calculus does not support any values directly; only references by names are available. Thus, the pattern is not supported.

Pattern 4.15 (Data Transfer by Value—Outgoing) *Description: The ability of an activity to pass data elements to subsequent activities as values relieving it from the need to have shared names or common address space with the activities to which it is passing them. (According to [113, p.35])*

Implementation: The π-calculus does not support any values directly, only references by names are available. Thus, the pattern is not supported.

Pattern 4.16 (Data Transfer—Copy In/Copy Out) *Description: The ability of an activity to copy the values of a set of data elements into its address space at the commencement of execution and to copy their final values back at completion. (According to [113, p.35])*

Implementation: This pattern can be supported by defining an agent $COPY$ that is able to return a copy of a given data type. If we assume such an agent,

$$A \overset{def}{=} \nu resp\ a(x).cell(c).\overline{copy}\langle x, c, resp\rangle.resp.\tau.\overline{copy}\langle c, x, resp\rangle.resp.\overline{b}\langle x\rangle.\mathbf{0} \tag{4.3}$$

represents an activity that receives data via a, creates a cell scoped to the activity, copies the data, executes its functional part, and finally copies back the value and transmits the result via b. However, especially copy out it seldom useful in concurrent environments.

Pattern 4.17 (Data Transfer by Reference—Unlocked) *Description: The ability to communicate data elements between activities by utilizing a reference to the location of the data element in some mutually accessible location. No concurrency restrictions apply to the shared data element. (According to [113, p.36])*

Implementation: Unlocked data transfer by reference is the default case in most patterns given beforehand. This is due to the fact that names represent references to data.

Pattern 4.18 (Data Transfer by Reference—With Lock) *Description: The ability to communicate data elements between activities by passing a reference to the location of the data element in some mutually accessible location. Concurrency restrictions are implied with the receiving activity receiving the privilege of read-only or dedicated access to the data element. (According to [113, p.37])*

Implementation: This pattern is implemented by different data structures, for instance in definition 4.6 (Iterator Queue). If an iterator is requested from the iterator queue and only the iterator is transmitted to an activity, the access to the queue is blocked for all concurrent activities. Read only access can be implemented by using distinct names for read and write operations, such as an extended memory cell:

$$\begin{aligned} CELL_{RW} &\overset{def}{=} \nu read\ \nu write\ \overline{cell_{RW}}\langle read, write\rangle.(CELL_{RW1}(\perp) \mid CELL_{RW}) \\ CELL_{RW1}(n) &\overset{def}{=} \overline{read}\langle n\rangle.CELL_{RW1}(n) + write(x).CELL_{RW1}(x)\ . \end{aligned} \tag{4.4}$$

The agent $CELL_{RW}$ uses distinct names $read$ and $write$. If the cell should be read (or write) only, the corresponding name has to be transmitted to an activity.

Pattern 4.19 (Data Transformation—Input) *Description: The ability to apply a transformation function to a data element prior to it being passed to an activity. (According to [113, p.38])*

Implementation: This pattern can be implemented by providing agents for transforming the data correspondingly.

Pattern 4.20 (Data Transformation—Output) *Description: The ability to apply a transformation function to a data element immediately prior to it being passed out of an activity. (According to [113, p.39])*

Implementation: This pattern can be implemented by providing agents for transforming the data correspondingly.

4.3.4 Data-based Routing Patterns

Data-based routing patterns describe how data can be used to define control flow between activities. They already anticipate the topic of processes that will be investigated in chapter 5 (Processes). Thus, we only provide short examples of how the patterns can be realized in the π-calculus.

Pattern 4.21 (Activity Precondition—Data Existence) *Description: Data-based preconditions can be specified for activities based on the presence of data elements at the time of execution. (According to [113, p.39])*

Implementation: One possible implementation is given by an agent representing an activity that is enacted each time data can be read from a queue:

$$A \overset{def}{=} q(d).(\tau.A' \mid A) \; ,$$

where q is queue. There are also implementations possible where an exception handling takes place if the data is not available. For instance, if an empty name is read from the queue, the current activity instance could be skipped or the whole process instance could be canceled.

Pattern 4.22 (Activity Precondition—Data Value) *Description: Data-based preconditions can be specified for activities based on the value of specific parameters at the time of execution. (According to [113, p.41])*

Implementation: For instance, an agent representing an activity is enacted if a cell c contains the value 3. The agent polls the values of the cell:

$$A \overset{def}{=} c(x).\text{if } x == 3 \text{ then } \tau.A' \text{ else } \tau.A \; .$$

Since polling produces a performance overhead, this is not a recommended implementation. Furthermore, an exception handling can take place if a data value is not met.

Pattern 4.23 (Activity Postcondition—Data Existence) *Description: Data-based postconditions can be specified for activities based on the existence of specific parameters at the time of execution. (According to [113, p.42])*

Implementation: This pattern can have two different implementations. The first alternative is to hold the execution of the activity, whereas the second repeats the activity until the postcondition is met. While the latter one can be implemented using a while loop, the former one can be given for instance as

$$A \stackrel{def}{=} \tau.A' \text{ and } A' \stackrel{def}{=} c(x).([x = \bot]A' + ([x = \top]A'') .$$

The activity represented by the agent above polls a cell via the name c after the functional part has been executed. As long as the value is $\bot$, the polling continues, whereas a subsequent part of the activity is activated in A'' if the value is $\top$.

Pattern 4.24 (Activity Postcondition—Data Value) *Description: Data-based postconditions can be specified for activities based on the value of specific parameters at the time of execution. (According to [113, p.43])*

Implementation: The implementation of this pattern is similar to activity postconditions—data existence (pattern 4.23).

Pattern 4.25 (Event-based Activity Trigger) *Description: The ability for an external event to initiate an activity. (According to [113, p.43])*

Implementation: This patterns triggers an activity if an external event occurs. Since events and data are represented as names in the π-calculus, a possible implementation is given by an environment $\mathcal{E}$ that is able to signal an event:

$$SYS \stackrel{def}{=} \nu evt\ (evt.\tau.A' \mid \mathcal{E}) .$$

The activity is represented as the left hand component. It is executed immediately after an interaction via evt occurs. This interaction is triggered by $\mathcal{E}$.

Pattern 4.26 (Data-based Task Trigger) *Description: The ability to trigger a specific activity when an expression based on process data elements evaluates to true. (According to [113, p.44])*

Implementation: This pattern can be implemented using polling or events generated by the data producing processes. If, for instance, the environment provides the data, the solution from pattern 4.25 (Event-based Activity Trigger) is sufficient.

Pattern 4.27 (Data-based Routing) *Description: The ability to alter the control flow within a process instance as a consequence of the value of data-based expressions. (According to [113, p.45])*

Implementation: This pattern resembles the process pattern 5.4 (Exclusive Choice) and will be discussed in chapter 5 (Processes).

Chapter 5

Processes

In this chapter we discuss how business processes can be represented by introducing *process graphs* as a static structure for defining dependencies between activities. Each node of a process graph represents an activity, while each edge defines a control flow constraint. The execution semantics of a process graph is given by π-calculus agents. The agent terms are based on the workflow patterns to cover a broad range of possible behavior. Moreover, we use (bi)-simulation equivalences to decide whether a process graph fulfills certain soundness properties. In particular, we investigate lazy, weak, and relaxed soundness.

5.1 Representation

In this section we describe the representation of business processes in the π-calculus. We introduce a graph structure for business processes, continue with a formal semantics, and conclude with a discussion of processes vs. instances.

5.1.1 Structure

We start with the definitions of a process graph, a data structure that represents a process as given by definition 3.7 (Process). Process graphs provide a uniform representation of business processes regardless of their actual notations:

Definition 5.1 (Process Graph) A *process graph* is a four-tuple consisting of nodes, directed edges, types, and attributes. Formally: $P = (N, E, T, A)$ with

- N as a finite, non-empty set of nodes,
- $E \subseteq (N \times N)$ as a set of directed edges between nodes,
- $T : N \rightarrow \mathit{TYPE}$ as a function from nodes to types, and
- $A \subseteq (N \times (\mathit{KEY} \times \mathit{VALUE}))$ as a relation from nodes to key/value pairs. $\Box$

The nodes N of a process graph define the activities of a process, and the directed edges E define dependencies between activities. Each node has exactly one type assigned by the function

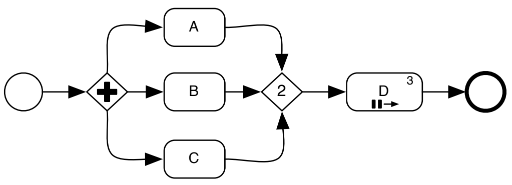


Figure 5.1: A simple business process.

T matching to one or more process patterns. An exception is given by the special type of a single activity. In this case, pattern 5.1 (Sequence) has to be applied. A type is given by an arbitrary text string. Furthermore, each node can hold optional attributes represented by key/values pairs denoted by A. Keys and values are given by arbitrary text strings. Complex activities are represented by a node N of the special type $\mathit{Reference}$, that references another process graph, i.e. $T(N) = \mathit{Reference}$. As such composed process graphs can always be flattened, we only consider flat process graphs. Some functions for accessing the sets of a process graph are given by:

- $\mathit{pre} : N \to \mathcal{P}(E)$ returns the set of edges having N as target.
- $\mathit{post} : N \to \mathcal{P}(E)$ returns the set of edges having N as source.
- $\mathit{type} : N \to T$ returns the type of a node.

To show the coherence between a process graph and a graphical notation, we give an example of how to map the structurally relevant parts of a business process diagram to a process graph.

Example 5.1 (Partly Mapping of a BPD to a Process Graph) A BPD is exemplary mapped to a process graph $P = (N, E, T, A)$ by the following steps:

1. N is given by all flow object of the BPD.
2. E is given by all sequence flows of the BPD.
3. T is given by the corresponding types of the flow objects.
4. A is given by additional attributes of flow objects, e.g.:
   (a) The number of incoming sequence flows for an n-out-of-m-join node;
   (b) The number of instances to be created for an activity;
   (c) The nodes to be canceled for a cancel event. □

An example of a business process modeled in BPMN is given in figure 5.1. The process contains an n-out-of-m-join pattern, modeled by a gateway with the number of required sequence flows inside, as well as a multiple instances without synchronization pattern, modeled by activity D. The complete business process diagram is mapped to a process graph according to the mapping rules given in example 5.1.

Example 5.2 (Simple Business Process) The process graph $P = (N, E, T, A)$ of the example from figure 5.1 is given by:

1. $N = \{N1, N2, N3, N4, N5, N6, N7, N8\}$
2. $E = \{\ (N1, N2), (N2, N3), (N2, N4), (N2, N5), (N3, N6), (N4, N6),$ $(N5, N6), (N6, N7), (N7, N8)\ \}$
3. $T = \{(N1, \mathit{StartEvent}), (N2, \mathit{ANDGateway}), (N3, \mathit{Task}), (N4, \mathit{Task}),$ $(N5, \mathit{Task}), (N6, \mathit{NoutofMJoin}), (N7, \mathit{MIwithoutSync}), (N8, \mathit{EndEvent})\}$
4. $A = \{(N6, (\mathit{continue}, 2)), (N7, (\mathit{count}, 3))\}$ □

5.1.2 Behavior

The definition of a process graph contains the types of the nodes, and thereby their behavioral semantics, only in a textual form. This causes no problems regarding ordinary activity nodes, since we abstract from their actual semantics. However, for node types that regard to control flow dependencies, a formal semantics has to be found. This topic has already been motivated in definition 3.22 (Simple Process Graph). In contrast to a simple process graph, a process graph can contain more advanced routing constructs making its formalization more complex. We give a formal execution semantics to a process graph by mapping it to π-calculus agents according to the following algorithm.

Algorithm 5.1 (Mapping Process Graphs to Agents) A process graph $P = (P_N, P_E, P_T, P_A)$ is mapped to π-calculus agents as follows:

1. All nodes of P are assigned a unique π-calculus agent identifier $N1 \ldots N|P_N|$.
2. All edges of P are assigned a unique π-calculus name $e1 \ldots e|P_E|$.
3. The π-calculus agents are defined according to the process patterns found in the next section as given by the type of the corresponding node. Special care has to be taken for supporting loop behavior
   (a) All agents representing a node with no incoming edges (i.e. initial nodes) do not support loop behavior, and
   (b) All other agents support loop behavior by recursion (can be omitted if the process graph does not contain cycles).
4. An agent $N \overset{def}{=} (\nu e1, \ldots, e|P_E|\ )(\prod_{i=1}^{|P_N|} Ni)$ representing a process instance is defined. This agent might contain further components or restricted names according to the contained patterns. □

The formalization of a process graph in the π-calculus starts with a mapping from nodes to agents. Hence, let every node be an independent agent. Each agent has pre- and postconditions. A precondition for an agent B could be that it should only be executed after an agent A has

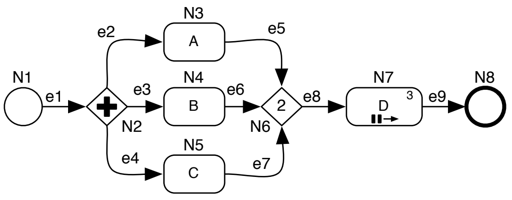


Figure 5.2: A simple business process annotated with agent identifiers and names.

finished executing the activity it represents. A postcondition for an agent B could state that B has finished execution and then signals this to other agents. The pre- and postconditions are represented in the second step of the algorithm by unique π-calculus names. A precondition is denoted by using a name as input prefix and a postcondition by using a name as an output prefix. In the third step, a pattern is applied to each agent for correctly consuming and generating the pre- and postconditions. During this step, α-conversions of the names given in the patterns to the corresponding names assigned to the edges have to be made. An agent representing a basic activity inside a sequence is given by:

$$A \stackrel{def}{=} x.\tau.\overline{y}.\mathbf{0} \ .$$

A waits for a single precondition (via x), executes the functional perspective of the activity it represents (abstracted from by τ), and finally fulfills a postcondition by emitting via y. If an agent representing a node of a process graph has no preconditions, it represents an initial node. Correspondingly, it represents a final node if it has no postcondition. In the fourth step, all agents representing nodes are placed as components inside a composition N. Furthermore, the names assigned to the edges are restricted to N.

To allow generic definitions of process patterns that can be further on extended, we refine the abstraction of the functional perspective given by τ with a *functional abstraction*:

Definition 5.2 (Functional Abstraction) A *functional abstraction*, denoted as $\langle\cdot\rangle$, represents a placeholder for a sequence of prefixes and restrictions. A functional abstraction may be placed anywhere inside an agent definition where a prefix could be placed. The grammar of the sequences is given by:

$$\begin{aligned} P &::= Q.\pi \mid \pi \mid \nu z\, P \\ Q &::= Q.\pi \mid \pi \\ \pi &::= \overline{x}\langle y\rangle \mid x(z) \mid \tau \end{aligned} \tag{5.1}$$

□

$A\langle\varphi\rangle$ denotes the replacement of all functional abstractions inside the agent A by the sequence φ. Equal to a context, the replacement is literal. An example is given by $A \stackrel{def}{=} a.\langle\cdot\rangle.\bar{b}.\mathbf{0}$, that can be resolved by $A\langle m.\tau\rangle$ to $A \stackrel{def}{=} a.m.\tau.\bar{b}.\mathbf{0}$.

Example 5.3 (Simple Business Process Formalization) We can now map the process graph from example 5.1 to π-calculus agents according to algorithm 5.1 by anticipating some of the pattern formalizations. Figure 5.2 shows an annotated version of the business process diagram where steps one and two of the algorithm have been applied. Each node is assigned a π-calculus identifier and each edge is assigned a unique π-calculus name. In the third step, the actual agent definitions are created. The first node of the business process is a start event given by the agent

$$N1 \overset{def}{=} \langle\cdot\rangle.\overline{e1}.\mathbf{0}\ .$$

Since the node representing the start event has no incoming edges, also the agent formalizing it has no precondition. $N1$ simply emits via $e1$ the completion of the functional part of the start event represented by $\langle\cdot\rangle$. The next node is of the type BPMN and gateway. Since this gateway combines two different process patterns, namely parallel split (pattern 5.2) and synchronization (pattern 5.3), it has to be evaluated which one—or even both—have to be applied. Since $|pre(N2)| = 1$ the node does not represent a synchronization pattern. However, with $|post(N2)| = 3$, the parallel split pattern applies. The corresponding formalization of node $N2$ is given as

$$N2 \overset{def}{=} e1.\langle\cdot\rangle.(\overline{e2}.\mathbf{0} \mid \overline{e3}.\mathbf{0} \mid \overline{e4}.\mathbf{0})\ .$$

According to algorithm 5.1, we omitted loop behavior since the process graph is acyclic. After the functional part represented by $\langle\cdot\rangle$ has been executed, the agent provides all necessary postconditions via the names $e2$, $e3$, and $e4$. A detailed discussion of the applied pattern (and also for the nodes to follow) can be found in section 5.2. We omit the discussion here and continue with the agents representing the tasks A, B, and C:

$$N3 \overset{def}{=} e2.\langle\cdot\rangle.\overline{e5}.\mathbf{0}\ ,\quad N4 \overset{def}{=} e3.\langle\cdot\rangle.\overline{e6}.\mathbf{0}\ , \text{and } N5 \overset{def}{=} e4.\langle\cdot\rangle.\overline{e7}.\mathbf{0}\ .$$

All tasks are placed inside a sequence as explained earlier. More interesting is the next node, representing a 2-out-of-3 join as a special kind of the discriminator (pattern 5.9). The formalization is given by

$$N6 \overset{def}{=} \nu h\, \nu e\, (N6_1 \mid N6_2)\ ,$$

with the components

$$N6_1 \overset{def}{=} e5.\overline{h}.\mathbf{0} \mid e6.\overline{h}.\mathbf{0} \mid e7.\overline{h}.\mathbf{0} \text{ and } N6_2 \overset{def}{=} h.h.\overline{e}.h.N6 \mid e.\langle\cdot\rangle.\overline{e8}.\mathbf{0}\ .$$

The agent $N6$ uses the preconditions in such a way, that after two arbitrary names of the set $\{e5, e6, e7\}$ have been received, the functional part is executed and the postcondition is provided via $e8$. Again, a detailed description of the pattern follows. The multiple instances task represents a multiple instances without synchronization (pattern 5.13). It is given by the following agent:

$$N7 \overset{def}{=} e8.(\langle\cdot\rangle.\mathbf{0} \mid \langle\cdot\rangle.\mathbf{0} \mid \langle\cdot\rangle.\mathbf{0} \mid \overline{e9}.\mathbf{0})\ .$$

In this agent, the functional part represent by $\langle\cdot\rangle$ is executed three times in parallel without any synchronization afterward. The postcondition is provided immediately via $e9$. The last node is of the type end event and given by:

$$N8 \overset{def}{=} e9.\langle\cdot\rangle.\mathbf{0}\ .$$

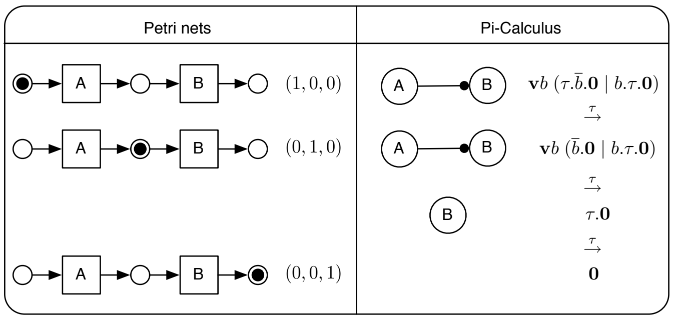


Figure 5.3: Process instances.

Finally, the last step of algorithm 5.1 is applied by defining

$$N \overset{def}{=} (\nu e1, \ldots, e9)(\prod_{i=1}^{8} Ni)\ .$$

The agent N is composed out of the agents representing the nodes of the process graph. However, due to the control flow dependencies that have been implemented, only component $N1$ can start evolving immediately.

5.1.3 Processes and Instances

In this subsection we discuss how the concepts process (definition 3.7) and process instance (definition 3.8) are distinguished in terms of process graphs and π-calculus mappings. The π-calculus by itself does not differentiate between processes and process instances (vice versa for activities). At each step of the evolution of a system made of agents, the state is directly represented by the agent terms. In other formalizations, as for instance Petri nets, there is a differentiation between the definition of a business process (e.g. given by a Petri net), and the state it is currently in (e.g. the token distribution).

Figure 5.3 depicts the issue. The left hand side contains different states of a Petri net that represents a sequence between two activities A and B. The states are formally denoted by the markings such as $(1, 0, 0)$ and graphically by tokens distributed over the places. A marking always belongs to a specific Petri net given by a three tuple (P, T, F) (see definition 3.26). The Petri net represents the static structure of a process, whereas the markings represent a process instance. However, as stated above, in the π-calculus both concepts are merged. This is depicted at the right hand side of the figure. A graphical representation is given by a flow graph, however this is only for illustrating purposes. The agent definitions represent the current state the system is in. As the system, and thereby the process it represents, evolves, the agents change their structure by using term rewriting. Thus, the structure as well as the current state are described only in terms of π-calculus agents.

It is not our aim to discuss these different paradigms here. They can be mapped partly by stating that a π-calculus agent corresponds to the concept of a process (definition 3.7) if it is in the state given by its definition since it contains the complete behavior of the corresponding process graph. A π-calculus agent might corresponds to the concept of a process instance (definition 3.8) if its current state represents an evolution of a defined agent. An example for the former case is given by

$$S \overset{def}{=} \nu b\,(\tau.\bar{b}.\mathbf{0} \mid b.\tau.\mathbf{0})\ .$$

An example for the latter case is the first evolution of the agent given above:

$$S \xrightarrow{\tau} \nu b\,(\bar{b}.\mathbf{0} \mid b.\tau.\mathbf{0})\ .$$

Both definitions only hold for a system of agents representing a business process. However, due to recursion inside agents, an agent representing currently a process instance can represent a process again. Consider for instance,

$$A \overset{def}{=} \tau.(\bar{b}.\mathbf{0} \mid A)\ ,$$

that corresponds to an activity. After $\xrightarrow{\tau}$ it corresponds to an activity instance, but after $\xrightarrow{\bar{b}}$ and SC-COMP-INACT it corresponds again to an activity following the above definitions. Since this subtle problem makes the distinction between processes and process instances as well as between activities and activity instances in the π-calculus difficult, we avoid using these terms. Instead, we talk about a *prototypical* representation that merges both concepts. If the structural definition of a process according to definition 3.7 (Process) is required, we revert to a process graph (definition 5.1). Each process graph gets a formal semantics according to algorithm 5.1 (Mapping Process Graphs to Agents). Due to the property that the agent representing the initial node of the process graph can only be executed once, each π-calculus mapping of a process graph is seen as a process instance according to definition 3.8 (Process Instance).

5.2 Process Patterns

This section introduces different process patterns as required by algorithm 5.1. The patterns proposed are based on the workflow patterns, as well as an additional one that is especially suited for interactions introduced in chapter 6 (Interactions). As done in chapter 4 (Data), we adapt the description of the patterns to the terminology used throughout this thesis.

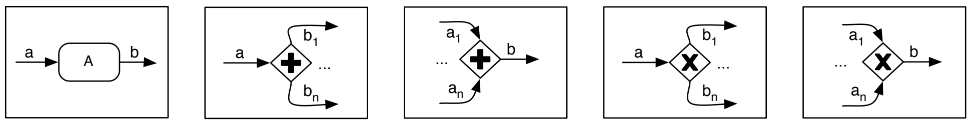
5.2.1 Basic Control Flow Patterns

The basic control flow patterns capture elementary aspects of control flow. A graphical representation of these patterns is given in figure 5.4.

Pattern 5.1 (Sequence) *Description: An activity in a business process is enabled after the completion of another activity in the same process. (According to [12, p.6])*

Implementation: A sequence is represented by an agent A waiting for a precondition via a, thereafter executing the functional perspective of the activity the agent represents (i.e. $\langle\cdot\rangle$), and finally provides a postcondition via b:

$$A \overset{def}{=} a.\langle\cdot\rangle.\bar{b}.\mathbf{0}\ .$$



(a) Sequence. (b) Parall. Split. (c) Synchroniz. (d) Excl. Choice. (e) Simple Merge.

Figure 5.4: Basic control flow patterns.

This pattern applies to a node N of a process graph that has at most one incoming and at most one outgoing edge: $|pre(N)| \leq 1$ and $|post(N)| \leq 1$. If $pre(N) = \emptyset$, the name a is omitted, and if $post(N) = \emptyset$, the name b is omitted from the pattern.

Pattern 5.2 (Parallel Split) *Description: A point in the business process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order. (According to [12, p.7])*

Implementation: To achieve a parallel split from an agent A, representing a node of a process graph, n names b are emitted as a postcondition:

$$A \stackrel{def}{=} a.\langle\cdot\rangle.(\prod_{i=1}^{n} \overline{b_i}.\mathbf{0}) \; .$$

This pattern applies to a node N of a process graph that has at most one incoming edge and at least two outgoing edges: $|pre(N)| \leq 1$ and $|post(N)| \geq 2$. If $pre(N) = \emptyset$, the name a is omitted from the pattern.

Pattern 5.3 (Synchronization) *Description: A point in the business process where multiple parallel (complex) activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed once. (According to [12, p.7])*

Implementation: To achieve synchronization at an agent A, representing a node of a process graph, n names are received as a precondition:

$$A \stackrel{def}{=} \{a_i\}_{i=1}^{n}.\langle\cdot\rangle.\overline{b}.\mathbf{0} \; .$$

The sequential ordering of the names a_i, representing preconditions, causes no problems, since the π-calculus semantics applied is synchronous. The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $pre(N) \geq 2$ and $post(N) \leq 1$. If $post(N) = \emptyset$, the name b is omitted from the pattern.

The parallel split and the synchronization patterns can be combined into one node of a process graph. The pattern is then given accordingly:

$$A \stackrel{def}{=} \{a_i\}_{i=1}^{n}.\langle\cdot\rangle.(\prod_{i=1}^{m} \overline{b_i}.\mathbf{0}) \; .$$

Pattern 5.4 (Exclusive Choice) *Description: A point in the business process where, based on a decision or data, one of several branches is chosen. (According to [12, p.8])*

Implementation: An exclusive choice from an agent A, representing a node of a process graph, is achieved by emitting one name b_i out of a set with size n:

$$A \overset{def}{=} a.\langle\cdot\rangle.(\sum_{i=1}^{n} \overline{b_i}.\mathbf{0}) .$$

The pattern given makes a non-deterministic choice. It applies to a node N of a process graph that has at most one incoming edge and at least two outgoing edges: $|pre(N)| \leq 1$ and $|post(N)| \geq 2$. If $pre(N) = \emptyset$, the name a is omitted from the pattern. A data-based choice according to pattern 4.27 (Data-based Routing) is represented by using either the match operator of the π-calculus (for comparing π-calculus names) or higher level abstractions like natural number comparators. Consider for instance

$$A \overset{def}{=} a.\langle\cdot\rangle.\text{if } value < \mathit{100} \text{ then } \overline{b_{\mathit{1}}}.\mathbf{0} \text{ else } \overline{b_{\mathit{2}}}.\mathbf{0} ,$$

where the name $value$ represents a natural number generated in $\langle\cdot\rangle$.

Pattern 5.5 (Simple Merge) *Description: A point in the business process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel. (According to [12, p.9])*

Implementation: A simple merge at an agent A, representing a node of a process graph, is achieved by receiving one name a_i out of a set with size n:

$$A \overset{def}{=} \sum_{i=1}^{n} a_i.\langle\cdot\rangle.\overline{b}.\mathbf{0} .$$

The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $pre(N) \geq 2$ and $post(N) \leq 1$. If $post(N) = \emptyset$, the name b is omitted from the pattern. If more than one name should be used as a precondition, pattern 5.7 (Synchronizing Merge) applies.

Just as the parallel split and synchronization patterns can be combined into one node of a process graph, the same holds for the exclusive choice and simple merge patterns. The corresponding pattern is given by:

$$A \overset{def}{=} \sum_{i=1}^{n} a_i.\langle\cdot\rangle.(\sum_{i=1}^{m} \overline{b_i}.\mathbf{0}) .$$

Example 5.4 (Basic Control Flow Patters) We illustrate the application of the basic control flow patterns by a process graph containing them all. Figure 5.5 depicts the process graph. We already annotated the corresponding π-calculus agent identifiers inside the nodes as well as the π-calculus names beside the edges. The types of the nodes are attached next to the nodes and correspond directly to the patterns introduced so far. The nodes that are executed sequentially are given by

$$N\mathit{2} \overset{def}{=} e\mathit{1}.\langle\cdot\rangle.\overline{e\mathit{3}}.\mathbf{0}, N\mathit{3} \overset{def}{=} e\mathit{2}.\langle\cdot\rangle.\overline{e\mathit{4}}.\mathbf{0}, N\mathit{6} \overset{def}{=} e\mathit{6}.\langle\cdot\rangle.\overline{e\mathit{8}}.\mathbf{0}, \text{ and } N\mathit{7} \overset{def}{=} e\mathit{7}.\langle\cdot\rangle.\overline{e\mathit{9}}.\mathbf{0} .$$

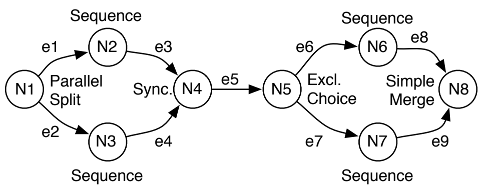


Figure 5.5: Basic control flow pattern example.

The parallel split and synchronization patterns are implemented as

$$N1 \overset{def}{=} \langle\cdot\rangle.(\overline{e1}.\mathbf{0} \mid \overline{e2}.\mathbf{0}) \text{ and } N4 \overset{def}{=} e3.e4.\langle\cdot\rangle.\overline{e5}.\mathbf{0} .$$

Finally, the exclusive choice and simple merge patterns are implemented by

$$N5 \overset{def}{=} e5.\langle\cdot\rangle.(\overline{e6}.\mathbf{0} + \overline{e7}.\mathbf{0}) \text{ and } N8 \overset{def}{=} e8.\langle\cdot\rangle.\mathbf{0} + e9.\langle\cdot\rangle.\mathbf{0} .$$

We did not use recursion inside the agent definitions, since the process graph is acyclic. The global agent representing the complete process graph is given by:

$$N \overset{def}{=} \nu e1 \ldots e9\ (\prod_{i=1}^{8} Ni) .$$

5.2.2 Advanced Branching and Synchronization Patterns

The advanced branching and synchronization patterns cover more elaborate control flow splits and merges. A graphical representation of these patterns is given in figure 5.6.

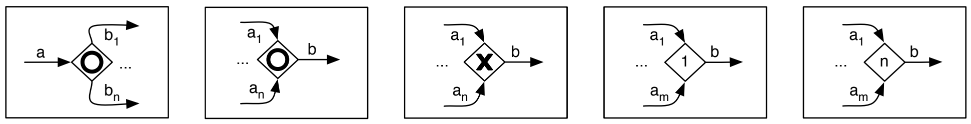
Pattern 5.6 (Multiple Choice) *Description: A point in the workflow process where, based on a decision or data, a number of branches are chosen. (According to [12, p.9])*

Implementation: A multiple choice from an agent A, representing a node of a process graph, is achieved by emitting a number of names b_i out of a set with size n:

$$A \overset{def}{=} \nu c\, a.\langle\cdot\rangle.(\prod_{i=1}^{n}(\overbrace{\overline{b_i}.\mathbf{0}}^{enable} + \overbrace{c.\mathbf{0}}^{\text{cancel}}) \mid \{\overline{c}\}_{i=1}^{n-1}.\mathbf{0}) .$$

The pattern given makes a non-deterministic choice where at least one name out of b_i is emitted. The last constraint is achieved by the right hand component of A, that only emits $n-1$ restricted names c. It applies to a node N of a process graph that has at most one incoming edge and at least two outgoing edges: $|pre(N)| \leq 1$ and $|post(N)| \geq 2$. If $pre(N) = \emptyset$, the name a is omitted from the pattern. A data-based choice according to pattern 4.27 (Data-based Routing) is represented by using either the match operator of the π-calculus (for comparing π-calculus names) or higher level abstractions like natural number comparators. Consider for instance

$$A \overset{def}{=} a.\langle\cdot\rangle.([x = y]\overline{b_1}.\mathbf{0} \mid [x = z]\overline{b_2}.\mathbf{0}) ,$$



(a) Mult. Choice. (b) Synch. Merge. (c) Multi. Merge. (d) Discriminator. (e) N-out-of-M.

Figure 5.6: Advanced control flow patterns.

where the name b_1 and b_2 are sent based on the evaluation of native π-calculus names. Care has to be taken that SC-MAT can be applied at least once.

Pattern 5.7 (Synchronizing Merge) *Description: A point in the business process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete. (According to [12, p.11])*

Implementation: A synchronizing merge at an agent A, representing a node of a process graph, is achieved by receiving a number of names out of a set with size n:

$$A \overset{def}{=} \nu c\, \nu w\, \nu d\, (\prod_{i=1}^{n} \overbrace{(a_i.(\underbrace{\bar{d}.\mathbf{0}}_{\text{final}} + \underbrace{w.c.\mathbf{0}}_{\text{more}})}^{\text{accept}} + \overbrace{c.\mathbf{0}}^{\text{cancel}}) \mid d.\{\bar{c}\}_{i=1}^{n-1}.\langle\cdot\rangle.\bar{b}.\mathbf{0} \mid \{\bar{w}\}_{i=1}^{n-1}.\mathbf{0})\,.$$

The pattern implementation makes a non-deterministic choice between executing the functional abstraction $\langle\cdot\rangle$ or waiting for further names. The three restricted names c, w, and d represent either cancel, wait, or done triggers. After a name has been received via a_i, the component can decide between waiting for further names if this is possible, i.e. an interaction via w can occur, or signaling d, which leads to the cancellation of all remaining parallel components via c. Only after all components waiting for further names are canceled, the functional part is executed. The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $pre(N) \geq 2$ and $post(N) \leq 1$. If $post(N) = \emptyset$, the name b is omitted from the pattern.

Pattern 5.8 (Multiple Merge) *Description: A point in a business process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch. (According to [12, p.13])*

Implementation: A multiple merge at an agent A, representing a node of a process graph, is achieved by receiving arbitrary names out of a set with size n:

$$A \overset{def}{=} \sum_{i=1}^{n} a_i.(\langle\cdot\rangle.\bar{b}.\mathbf{0} \mid A)\,.$$

The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $pre(N) \geq 2$ and $post(N) \leq 1$. If $post(N) = \emptyset$, the name b is omitted

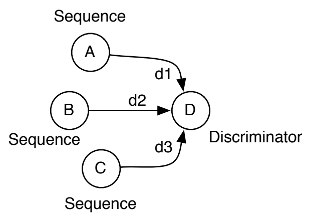


Figure 5.7: Discriminator example.

from the pattern. Note that subsequent nodes of the process graph must support pattern 5.11 (Arbitrary Cycles).

Pattern 5.9 (Discriminator) *Description: The discriminator is a point in a business process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and "ignores" them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again. (According to [12, p.14])*

Implementation: A discriminator at an agent A, representing a node of a process graph, is achieved by receiving a name out of a set with size m and thereafter executing the functional abstraction $\langle\cdot\rangle$, while waiting for the remaining names of the set:

$$A \stackrel{def}{=} \nu h\, \nu e\, (A_1 \mid A_2), \quad A_1 \stackrel{def}{=} \prod_{i=1}^{m} a_i.\overline{h}.\mathbf{0}, \text{ and } A_2 \stackrel{def}{=} h.\overline{e}.\{h\}_1^{m-1}.A \mid e.\langle\cdot\rangle.\overline{b}.\mathbf{0}\,.$$

The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $pre(N) \geq 2$ and $post(N) \leq 1$. If $post(N) = \emptyset$, the name b is omitted from the pattern. Note that subsequent nodes of the process graph must support pattern 5.11 (Arbitrary Cycles).

Example 5.5 (Discriminator) We illustrate a possible evolution of the discriminator by an example consisting of four agents A, B, C, and D. The first three agents represent nodes of a process graph prior to the discriminator that is represented as D:

$$DISC \stackrel{def}{=} \nu d1\, \nu d2\, \nu d3\, (A \mid B \mid C \mid \nu h\, \nu e\, (D_1 \mid D_2))\,.$$

The agents A, B, and C are defined according to pattern 5.1 (Sequence), whereas D is given by pattern 5.9 (Discriminator). The names used as pre- and postconditions between the agents can be found in figure 5.7. The sequential nodes are given by:

$$A \stackrel{def}{=} \tau.\overline{d1}.\mathbf{0}, \quad B \stackrel{def}{=} \tau.\overline{d2}.\mathbf{0}, \text{ and } C \stackrel{def}{=} \tau.\overline{d3}.\mathbf{0}\,.$$

Since we would like to evolve through the system, we replaced $\langle\cdot\rangle$ of the pattern definitions by τ. The agent representing a discriminator with the matching names is given by:

$$D \stackrel{def}{=} \nu h\, \nu e\, (D_1 \mid D_2), \quad D_1 \stackrel{def}{=} \prod_{i=1}^{3} di.\overline{h}.\mathbf{0}, \text{ and } D_2 \stackrel{def}{=} h.\overline{e}.h.h.D \mid e.\tau.\mathbf{0}\,.$$

The evolution of DISC begins with either A, B, or C emitting a name after the corresponding τ transition (omitted). We assume A to emit $\mathit{d1}$ first:

$$\mathit{DISC} \xrightarrow{\tau} \mathit{DISC}_1 \overset{def}{=} \nu d2\ \nu d3\ (B \mid C \mid \nu h\ \nu e\ (D_{11} \mid D_2))\ .$$

The agent A has vanished since no more prefixes exist after emitting the name $\mathit{d1}$. Agent D_1 has evolved to D_{11} and is defined by $D_{11} \overset{def}{=} \overline{h}.\mathbf{0} \mid d2.\overline{h}.\mathbf{0} \mid d3.\overline{h}.\mathbf{0}$. Immediately after, a communication between D_{11} and D_2 is possible:

$$\mathit{DISC}_1 \xrightarrow{\tau} \mathit{DISC}_2 \overset{def}{=} \nu d2\ \nu d3\ (B \mid C \mid \nu h\ \nu e\ (D_{12} \mid D_{21}))\ .$$

D_{11} communicates the name h to D_2 and evolves to $D_{12} \overset{def}{=} d2.\overline{h}.\mathbf{0} \mid d3.\overline{h}.\mathbf{0}$. The left hand component of D_{12} has vanished as it reached inaction. The agent D_2 evolves to $D_{21} \overset{def}{=} \overline{e}.h.h.D \mid e.\tau.\mathbf{0}$. Now e can be communicated inside D_{21}:

$$\mathit{DISC}_2 \xrightarrow{\tau} \mathit{DISC}_3 \overset{def}{=} \nu d2\ \nu d3\ (B \mid C \mid \nu h\ (D_{12} \mid D_{22}))\ .$$

D_{22} is given by $D_{22} \overset{def}{=} h.h.D \mid \tau.\mathbf{0}$. Note that the right hand side of D_{22} now only consists of $\tau.\mathbf{0}$. After a τ transition, the right hand side of D_{22} vanishes. Now agent B can communicate $\mathit{d2}$ and D_{12} can communicate h in turn:

$$\mathit{DISC}_2 \xrightarrow{\tau} \mathit{DISC}_3 \overset{def}{=} \nu d3\ (C \mid \nu h\ (D_{13} \mid D_{23}))\ .$$

Agent B has vanished after communicating $\mathit{d2}$. D_{12} evolves to $D_{13} \overset{def}{=} d3.\overline{h}.\mathbf{0}$. Agent D_{23} is given by $D_{23} \overset{def}{=} h.D$. Finally agent C can communicate $\mathit{d3}$ to D_{13} and D_{13} can communicate h to D_{23}:

$$\mathit{DISC}_3 \xrightarrow{\tau} \mathit{DISC}_4 \equiv D\ .$$

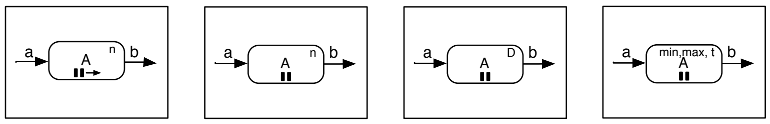
Since no more transitions inside DISC_4 are possible, the example is concluded.

Pattern 5.10 (N-out-of-M-Join) *Description: An n-out-of-m-join is a generic variant of a discriminator that waits for n out of m incoming branches.*

Implementation: An n-out-of-m-join at an agent A, representing a node of a process graph, is achieved by receiving n names out of a set with size m and thereafter executing the functional abstraction $\langle\cdot\rangle$, while waiting for the remaining names of the set:

$$A \overset{def}{=} \nu h\ \nu e\ (A_1 \mid A_2), \quad A_1 \overset{def}{=} \prod_{i=1}^{m} a_i.\overline{h}.\mathbf{0}, \text{ and } A_2 \overset{def}{=} \{h\}_1^n.\overline{e}.\{h\}_n^{m-1}.A \mid e.\langle\cdot\rangle.\overline{b}.\mathbf{0}\ .$$

The patterns applies to a node N of a process graph that has at least two incoming edges and at most one outgoing edge: $\mathit{pre}(N) \geq 2$ and $\mathit{post}(N) \leq 1$. If $\mathit{post}(N) = \emptyset$, the name b is omitted from the pattern. Note that subsequent nodes of the process graph must support pattern 5.11 (Arbitrary Cycles).



(a) Without Sync. (b) Design Knowl. (c) Runtime K. (d) No priori K.

Figure 5.8: Multiple instance patterns.

5.2.3 Structural Patterns

Structural patterns describe routing situations regarding the structure of a process. These patterns do not have an explicit graphical representation.

Pattern 5.11 (Arbitrary Cycles) *Description: A point in a business process where one or more activities can be done repeatedly. (According to [12, p.17])*

Implementation: Arbitrary cycles are inherently given by the pre- and postcondition based approach. A postcondition, i.e. the name generated by an agent representing a node, can fulfill the precondition of and trigger arbitrary other agents. However, agents that represent nodes contained inside a cycle must support multiple executions by recursion. This is achieved by introducing a recursion in parallel to the functional part represented by $\langle\cdot\rangle$. We show the principle for agents representing a node of a process graph matching pattern 5.1 (Sequence):

$$A \overset{def}{=} a.\langle\cdot\rangle.\bar{b}.\mathbf{0} \text{ becomes } A \overset{def}{=} a.(\langle\cdot\rangle.\bar{b}.\mathbf{0} \mid A)\ .$$

This pattern applies to all nodes of a process graph P that are contained inside a cycle of P.

Pattern 5.12 (Implicit Termination) *Description: A given complex activity should be terminated when there is nothing else to be done. In other words, there are no active activities in the business process and no other activity can be made active (and at the same time the business process is not in deadlock). (According to [12, p.19])*

Implementation: The implicit termination pattern terminates a complex activity if no other activities can be made active. The π-calculus contains the special symbol $\mathbf{0}$ for this purpose. This pattern applies to all nodes N of a process graph that have zero outgoing edges: $post(N) = 0$.

5.2.4 Multiple Instance Patterns

Multiple instance patterns create multiple activity instances. A graphical representation of these patterns is given in figure 5.8.

Pattern 5.13 (Multiple Instances without Synchronization) *Description: Within the context of a single process instance multiple instances of an activity are created, i.e., there is a facility to spawn off new threads of control. Each of these threads is independent of other threads. Moreover, there is no need to synchronize these threads. (According to [12, p.20])*

Implementation: An agent, representing a node of a process graph, that can spawn of n static instances without synchronization is given by:

$$A \overset{def}{=} a.(\prod_{i=1}^{n}(\langle\cdot\rangle.\mathbf{0}) \mid \bar{b}.\mathbf{0})\ .$$

This pattern applies to all nodes N of a process graph that have a type matching to the pattern description. This pattern incorporates pattern 5.12 (Implicit Termination). Furthermore, the same conditions as given by pattern 5.1 (Sequence) apply.

Pattern 5.14 (Multiple Instances with a priori Design Time Knowledge) *Description: For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are finished some other activity needs to be started. (According to [12, p.21])*

Implementation: An agent, representing a node of a process graph, that can spawn of n static instances with synchronization is given by:

$$A \overset{def}{=} \nu h\ a.(\prod_{i=1}^{n}(\langle\cdot\rangle.\bar{h}.\mathbf{0}) \mid \{h\}_1^n.\bar{b}.\mathbf{0})\ .$$

This pattern applies to all nodes N of a process graph that have a type matching to the pattern description. The same conditions as given by pattern 5.1 (Sequence) apply.

Pattern 5.15 (Multiple Instances with a priori Runtime Knowledge) *Description: For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance varies and may depend on characteristics of the process instance or availability of resources, but is known at some stage during runtime, before the instances of that activity have to be created. Once all instances are finished some other activity needs to be started. (According to [12, p.22])*

Implementation: An agent, representing a node of a process graph, that can spawn of n instances with synchronization, where n is known before the execution of the first instance, is given by:

$$\begin{aligned}
A &\overset{def}{=} \nu run\ \nu start\ a.(A_1(b) \mid run.A_2 \mid A_3)\\
A_1(prev) &\overset{def}{=} \nu next\ \overline{create\_i}\langle next, prev\rangle.A_1(next) + \overline{run}.\overline{prev}.\mathbf{0}\\
A_2 &\overset{def}{=} \overline{start}.A_2\\
A_3 &\overset{def}{=} create\_i(next, prev).(start.\langle\cdot\rangle.next.\overline{prev}.\mathbf{0} \mid A_3)\ .
\end{aligned}$$

The pattern given creates a non-determistic number of instances before the first instance is executed. Agent A creates two restricted names run and $start$. The former is emitted after all instances have been created, which in turn triggers the emission of a unbound number of $start$ names at A_2. The name $start$ is used as a shared precondition for all instances. Instances are created in agent A_1, where $next$ and $prev$ represent names to the subsequent and preceding agents. These names are used in A_3 to synchronize the finalization of the instances. Since $prev$ equals initially b, the agent representing the subsequent node of the process graph is triggered via

b after all instances have finished. A deterministic, data-based implementation using structures and values from chapter 4 (Data) is given by

$$A \overset{def}{=} a.queue(q, e).(A_1(0, b) \mid A_3) \ ,$$

where q represents a queue for storing the names used as a precondition for each instance. The preconditions, as well as the instances are created in the agents

$$\begin{aligned} A_1(i, prev) &\overset{def}{=} \nu resp\ \nu start\ \nu next \text{ if } i < n \text{ then } \overline{create\_i}\langle start, next, prev\rangle. \\ &\quad \overline{q}\langle start\rangle.\overline{add}\langle i, 1, resp\rangle.resp(x).A_1(x, next) \text{ else } A_2 \\ A_2 &\overset{def}{=} q(start).(\overline{start}.\mathbf{0} \mid A_2) + e.\overline{prev}.\mathbf{0} \ . \end{aligned}$$

The parameters i and $prev$ of agent A_1 represent the number of instances already created and the name pointing to the agent previously created. Agent A_1 uses a while loop to create n instances via $create\_i$ and thereafter sends all names stored in q so far, thus starting all instances. If the queue signals empty, the name of the previously created agent, i.e. $prev$ is sent. The instances are created in then agent

$$A_3 \overset{def}{=} create\_i(start, next, prev).(start.\langle\cdot\rangle.next.\overline{prev}.\mathbf{0} \mid A_3) \ .$$

A new instance represented by agent A_3 waits for its specific precondition represented by $start$, executes the functional part, waits for the completion of the agent representing the instance created after the current instance via $next$, and thereafter communicates the name $prev$ triggering the preceding instance. If the current instance is the first instance created, $prev$ matches b. Hence, the postcondition of the node represented by A is fulfilled. If the pattern formalization should be used for verification of the process graph, pattern 5.1 (Sequence) has to be applied instead, representing the case $n = 1$. This is due to the fact that n is unbound, i.e. if $n = \infty$, the agent will never finish. The pattern applies to all nodes N of a process graph that have a type matching to the pattern description. The same conditions as given by pattern 5.1 (Sequence) apply.

Pattern 5.16 (Multiple Instances without a priori Runtime Knowledge) *Description: For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are finished, some other activity needs to be started. The difference with pattern 5.15 (Multiple Instances with a priori Runtime Knowledge) is that even while some of the instances are running or already finished, new ones can be created. (According to [12, p.25])*

Implementation: An agent, representing a node of a process graph, that can spawn of instances with synchronization, where the number of instances is unknown until all instances have finished, is given by:

$$\begin{aligned} A &\overset{def}{=} a.(A_1(b) \mid A_2) \\ A_1(prev) &\overset{def}{=} \nu next\ \overline{create\_i}\langle next, prev\rangle.A_1(next) + \overline{prev}.\mathbf{0} \\ A_2 &\overset{def}{=} create\_i(next, prev).(\langle\cdot\rangle.next.\overline{prev}.\mathbf{0} \mid A_2) \ . \end{aligned}$$

The implementation of this patterns closely resembles the previous one, with the difference that instances can be created all the time and start immediately. This has been realized by removing the $start$ preconditions for each instance created inside A_2 as well as the corresponding agent. A data-based implementation can use an if .. then .. else statement to make the summation deterministic based on minimum, maximum, and threshold values. A corresponding agent A contains an agent A_1, which is responsible for creating the minimum number of instances as well as collecting the required threshold values for continuation:

$$A \overset{def}{=} \nu done\ \nu mindone\ a.(A_1(0) \mid mindone.\{done\}_1^t.\overline{b}.\mathbf{0}) ,$$

and

$$A_1(i) \overset{def}{=} \text{if } i < min \text{ then } \nu r\ \overline{create\_i}.add(i,1,r).r(x).A_1(x) \text{ else } \overline{mindone}.A_2(i) .$$

The minimum value min represents the number of instances that are created at least, max represents the number of instances that are created at most, and t denotes the number of instances that need to be finished (i.e. the threshold value) for the pattern to complete. After the minimum number of instances has been created by A_1, it signals $mindone$, which in turn activates the threshold counter given by $\{done\}_1^t$ in the nested right hand component of A. Thereafter A_2 is activated, which simply resembles A_1 for creating the remaining number of instances:

$$A_2(i) \overset{def}{=} \text{if } i < max \text{ then } \nu r\ \overline{create\_i}.add(i,1,r).r(x).A_2(x) .$$

Agent A_3 is responsible for creating the instances via $create\_i$:

$$A_3 \overset{def}{=} create\_i.(\langle\cdot\rangle.\overline{done}.\mathbf{0} \mid A_3) .$$

If the pattern formalization should be used for verification of the process graph, pattern 5.1 (Sequence) has to be applied instead. This is due to the fact that the number of instances can be unbound, thus the agent might never finish. The pattern applies to all nodes N of a process graph that have a type matching to the pattern description. The same conditions as given by pattern 5.1 (Sequence) apply.

Example 5.6 (Multiple Instances without a priori Runtime Knowledge) We derive a possible evolution of an agent representing a multiple instances without a priori runtime knowledge pattern. The example shows how the recursive structure of the agents is build up while creating instances and how it is broken down while finalizing. Agent A, representing a node resembling a multiple instances without a priori runtime knowledge pattern, is given by:

$$A \overset{def}{=} a.(A_1(b) \mid A_2) .$$

The agent initializes A_1 with the name representing the postcondition of the node, i.e. b in the example. Agent A_1 in turn has the choice between creating a new instance by interacting via $create\_i$ with agent A_2 or emitting b, thus fulfilling the postcondition. We suppose agent A_1 to create a new instance. Hence, the evolution of agent A after receiving a is given as follows:

$$(\nu n_1\ \overline{create\_i}\langle n_1, b\rangle.A_1(n_1) + \overline{b}.\mathbf{0}) \mid A_2 \xrightarrow{\tau} A_1(n_1) \mid \underbrace{(\tau.n_1.\overline{b}.\mathbf{0})}_{\text{1st instance}} \mid A_2 .$$

The agents A_1 and A_2 are defined as given in pattern 5.16 (Multiple Instances without a priori Runtime Knowledge). After an interaction between the components A_1 and A_2 occurred, a first instance as marked above is created. As before, we inserted τ into the functional abstraction found in A_2. Thereafter, the resulting agent can be unfolded to:

$$(\nu n_2\ \overline{create\_i}\langle n_2, n_1\rangle.A_1(n_2) + \overline{n_1}.\mathbf{0}) \mid \underbrace{(\tau.n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid A_2 \ .$$

If we assume again an interaction between the left hand side of the component resembling A_1 and the agent A_2, thus creating a second instance, the system evolves as follows:

$$A_1(n_1) \mid \underbrace{(\tau.n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid A_2 \xrightarrow{\tau} A_1(n_2) \mid \underbrace{(\tau.n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(\tau.n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \ .$$

We can continue creating new instances as long as we select the left hand side of the summation found in A_1. Meanwhile, components representing instances can already evolve further. For instance, with

$$A_1(n_2) \mid \underbrace{(\tau.n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(\tau.n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \xrightarrow{\tau} A_1(n_2) \mid \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(\tau.n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2$$

the agent representing the first instance evolved. The same holds for

$$A_1(n_2) \mid \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(\tau.n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \xrightarrow{\tau} A_1(n_2) \mid \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \ ,$$

where now the agent representing the second instance evolved. While we are still able to create further instances, we conclude the example by synchronizing the existing ones. This is done by selecting the right hand side of the summation contained in agent $A_1(n_2)$:

$$(\overline{n_2}.\mathbf{0}) \mid \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \ .$$

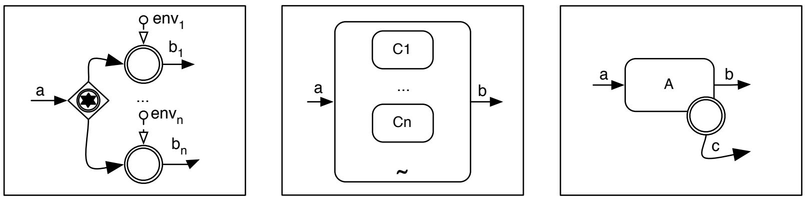
Now, the components can interact multiple times resulting in a component providing the post-condition given by b:

$$(\overline{n_2}.\mathbf{0}) \mid \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(n_2.\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \xrightarrow{\tau} \underbrace{(n_1.\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid \underbrace{(\overline{n_1}.\mathbf{0})}_{\text{2nd instance}} \mid A_2 \xrightarrow{\tau} \underbrace{(\bar{b}.\mathbf{0})}_{\text{1st instance}} \mid A_2 \ .$$

Agent A_2 remains, but has no possibilities for interaction. This agent may also be cleaned-up by an extended version of the pattern that transmits a restricted name to agent A_2 evolving it to inaction by a summation (omitted).

5.2.5 State Based Patterns

State based patterns capture implicit behavior of processes that is based on the environment or other parts of the process. A graphical representation of pattern 5.17 (Deferred Choice) and



(a) Deferred Choice. (b) Int. Parall. Routing. (c) Ev.-bas. Rerouting.

Figure 5.9: State based and additional patterns.

pattern 5.18 (Interleaved Parallel Routing) is given in figure 5.9(a) and 5.9(b). Pattern 5.19 (Milestone) has no graphical representation in the BPMN.

Pattern 5.17 (Deferred Choice) *Description: A point in the business process where one of several branches is chosen. In contrast to pattern 5.4 (Exclusive Choice), the choice is not made explicitly (e.g. base on data or a decision) but several alternatives are offered to the environment. However, in contrast to pattern 5.2 (Parallel Split), only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible. (According to [12, p.28])*

Implementation: A deferred choice after an agent A, representing a node of a process graph, is achieved by guarding the summations of pattern 5.4 (Exclusive Choice) with names generated by an agent representing the environment ($\mathcal{E}$):

$$A \overset{def}{=} a.\langle\cdot\rangle.(\sum_{i=1}^{n} env_i.\overline{b_i}.\mathbf{0}) \ .$$

The pattern given makes a deterministic choice based on external names env_i from the environment $\mathcal{E}$. In the BPMN representation, these names are gathered in subsequent nodes of the process graph. Hence, the implementation of the pattern is not in alignment with algorithm 5.1 (Mapping Process Graphs to Agents) by requiring knowledge from other nodes of the process graph. This is due to a limitation of the π-calculus semantics, which does not allow a cancellation-based implementation as suggested in the pattern description. A detailed discussion follows in chapter 8.2.5. The pattern applies to a node N of a process graph that has at most one incoming edge and at least two outgoing edges: $|pre(N)| \leq 1$ and $|post(N)| \geq 2$. If $pre(N) = \emptyset$, the name a is omitted from the pattern.

Pattern 5.18 (Interleaved Parallel Routing) *Description: A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at runtime, and no two activities are executed at the same moment (i.e. no two activities are running for the same process instance at the same time). (According to [12, p.31])*

Implementation: An agent A, representing a node of a process graph, that executes a set with size n of other nodes with the precondition names c_i and the postcondition names d_i in an

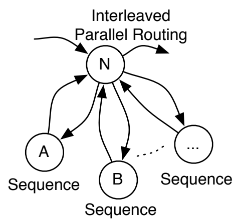


Figure 5.10: Process graph structure for interleaved parallel routing pattern.

interleaved parallel routing manner is given by:

$$A \overset{def}{=} \nu next\, \nu done\; a.(\{\overline{next}.done\}_1^n.\bar{b}.\mathbf{0} \mid \prod_{i=1}^{n}(next.\overline{c_i}.d_i.\overline{done}.\mathbf{0}))\ .$$

This pattern applies to a subset of a process graph with a structure as shown in figure 5.10, where the node N is represented by agent A. Each attached activity contained in the routing sequence is represented and connected via c_i and d_i by a node as shown. The nodes of the set are implemented according to pattern 5.1 (Sequence). Furthermore, N should have at most one incoming and at most one outgoing edge: $|pre(N)| \leq 1$ and $|post(N)| \leq 1$. If $pre(N) = \emptyset$, the name a is omitted, and if $post(N) = \emptyset$, the name b is omitted from the pattern.

Pattern 5.19 (Milestone) *Description: The enabling of an activity depends on the process instance being in a specific state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. (According to [12, p.34])*

Implementation: An agent A, representing a node of a process graph, that should only be executed if a certain milestone has been reached and not yet expired is given by:

$$A \overset{def}{=} a.notice.\langle\cdot\rangle.\bar{b}.\mathbf{0}\ .$$

The milestone is represented by an asynchronous emission of the name $notice$, e.g. by

$$P \overset{def}{=} (\overline{notice}.\mathbf{0} + withdraw.\mathbf{0}) \mid P'\ .$$

The milestone can be withdrawn by P' sending on $\overline{withdraw}$. This pattern applies to a node N of a process graph that has at most one incoming and at most one outgoing edge: $|pre(N)| \leq 1$ and $|post(N)| \leq 1$. If $pre(N) = \emptyset$, the name a is omitted, and if $post(N) = \emptyset$, the name b is omitted from the pattern.

5.2.6 Cancellation Patterns

The cancellation pattern describe the withdrawal of one or more activities. These patterns do not have a graphical representation in the BPMN. However, pattern 5.22 (Event-based Rerouting) can be used to denote that activities can be canceled in a graphical manner.

Pattern 5.20 (Cancel Activity) *Description: An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed. (According to [12, p.37])*

Implementation: An agent A, representing a node of a process graph, that can be canceled is given by:

$$A \stackrel{def}{=} a.\langle\cdot\rangle.\bar{b}.\mathbf{0} + cancel.\mathbf{0}\ .$$

The cancel notification can be signaled from arbitrary other agents, where the name $cancel$ is shared with. The activity cannot be canceled anymore once it has been activated by receiving a. The pattern applies to a node N of a process graph that has at most one incoming and at most one outgoing edge: $|pre(N)| \leq 1$ and $|post(N)| \leq 1$. If $pre(N) = \emptyset$, the name a is omitted, and if $post(N) = \emptyset$, the name b is omitted from the pattern.

Pattern 5.21 (Cancel Process Instance) *Description: A process instance is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed). (According to [12, p.37])*

Implementation: Cancel process instance equals pattern 5.20 (Cancel Activity) with the difference that all remaining agents representing nodes of a process graph receive a cancel name. To implement this pattern, all agents representing nodes of a process graph have to be enhanced with an according sum.

5.2.7 Additional Pattern

This subsection introduces an additional pattern typically found in interacting processes that will be introduced in the next chapter. A graphical representation is given in figure 5.9(c).

Pattern 5.22 (Event-based Rerouting) *Description: The event-based rerouting pattern represents the change of the control flow based on an event (e.g. a message) that occurs during the execution of an activity instance. The moment the event occurs, the control flow is passed immediately to another activity. However, the event is only considered if it occurs during the execution of the activity instance.*

Implementation: An agent A, representing a node of a process graph, that can be interrupted leading to a rerouting of control flow is given by

$$A \stackrel{def}{=} \nu check\ a.(A_1 \mid A_2)$$

with

$$A_1 = \langle\cdot\rangle.check(v).[v = \top]\bar{b}.\mathbf{0} \text{ and } A_2 \stackrel{def}{=} ir.\bar{c}.\overline{check}\langle\bot\rangle + \overline{check}\langle\top\rangle.\mathbf{0}\ .$$

In case the interrupt via ir does not occur, the postcondition of A is b, emitted after the functional part has been executed. If the interrupt occurs, the postcondition of A is c, emitted immediately after the reception of ir. The pattern applies to a node N of a process graph that has at most one incoming and two outgoing edge representing the possible control flows: $|pre(N)| \leq 1$ and $|post(N)| = 2$. If $pre(N) = \emptyset$, the name a is omitted from the pattern.

5.3 Properties

After having introduced a formal semantics for process graphs in the previous section, we investigate how bisimulation equivalence can be used for verification. We start by analyzing the static structure of a process graph, followed by the *black box* behavior of the corresponding π-calculus mapping. The results provide a property that is weaker than existing soundness properties. However, it has the advantages of being computational less expensive and supporting all process patterns. The new property is denoted as *lazy soundness*. The characterization of lazy soundness is further on extended to allow reasoning on existing soundness properties. In particular, we investigate weak soundness (definition 3.39) and relaxed soundness (definition 3.31). A business process that fulfills weak and relaxed soundness is also sound (see definition 3.30).

5.3.1 Structural Soundness

A first soundness property resembles definition 3.29 (Workflow net). The property is called *structural soundness* since it applies to the structure of a process graph. Informally, structural soundness is given by:

> A process graph is *structural sound* if it has exactly one initial node, exactly one final node, and all other nodes lie on a path between the initial and the final node.

The property of structural soundness for a process graph is desired, since only by having distinguished initial and final nodes, we can state when the execution of a process instances is started and ended. If there exist other nodes that are not on a path between the initial and the final node, they might resemble other initial and final nodes, which contradicts the exclusivity of the initial and final nodes. Structural soundness is based on the concepts introduced in the following definitions. We first need to define a *path* in a process graph:

Definition 5.3 (Path) A *path* in a process graph $P = (N, E, T, A)$ is a sequence of nodes $\langle n_1, \ldots, n_k \rangle$ with $n_i \in N$ for $1 \leq i \leq k$ such that $(n_1, n_2), \ldots, (n_{k-1}, n_k) \in E$. □

Based on the definition of a path, the reachability of nodes contained in the process graph can be given:

Definition 5.4 (Reachable) A node $n_k \in N$ of a process graph $P = (N, E, T, A)$ is *reachable* from another node $n_1 \in N$, denoted as $n_1 \xrightarrow{*} n_k$, if and only if there exist a path leading from the first to the second node. □

Furthermore, we define two special kinds of nodes, denoted as *initial node* and *final node*:

Definition 5.5 (Initial Node) A node $n \in N$ of a process graph $P = (N, E, T, A)$ is an *initial node* if and only if n is not the target of any edge. Formally: $pre(n) = \emptyset$. □

Definition 5.6 (Final Node) A node $n \in N$ of a process graph $P = (N, E, T, A)$ is an *final node* if and only if n is not the source of any edge. Formally: $post(n) = \emptyset$. □

We define a subset of the possible process graphs that have exactly one initial node and exactly one final node:

Definition 5.7 (Defined Process Graph) A process graph $P = (N, E, T, A)$ is *defined* if and only if there is exactly one initial node, denoted as N_i, that is not the target of any edge and exactly one final node, denoted as N_o, that is not the source of any edge. Formally: $\exists n \in N : pre(n) = \emptyset \wedge \forall n_1, n_2 \in N : pre(n_1) = \emptyset \wedge pre(n_2) = \emptyset \Rightarrow n_1 = n_2$ and $\exists n \in N : post(n) = \emptyset \wedge \forall n_1, n_2 \in N : post(n_1) = \emptyset \wedge post(n_2) = \emptyset \Rightarrow n_1 = n_2$. □

The subset of defined process graphs can be restricted further by requiring that all nodes are on a path between the initial and final node:

Definition 5.8 (Strongly Connected Process Graph) A defined process graph $P = (N, E, T, A)$ is *strongly connected*, if and only if all nodes lie on a path from the initial to the final node. Formally: $\forall n \in N : N_i \xrightarrow{*} n \Rightarrow n \xrightarrow{*} N_o$ □

This definition is in contrast to common definitions of a strongly connected directed graph, e.g. by Knuth [80]. This is due to the fact that we do not require a graph to be short circuited for verification. Since we have introduced all prerequisites, we can define structural soundness:

Definition 5.9 (Structural Sound Process Graph) A process graph $P = (N, E, T, A)$ is *structural sound* if and only if:

1. There is exactly one initial node $N_i \in N$.
2. There is exactly one final node $N_o \in N$.
3. Every node is on a path from N_i to N_o. □

We use definition 5.8 (Strongly Connected Process Graph) to provide a criterion for structural soundness:

Lemma 5.1 *A strongly connected process graph is structural sound.*

Proof 5.1 (Lemma 5.1) Direct proof. Criterion 1 and 2 from definition 5.9 are fulfilled, as a strongly connected process graph is defined. Criterion 3 follows directly from definition 5.8. □

Finally, we provide an algorithm for deciding structural soundness for process graphs.

Algorithm 5.2 (Deciding Structural Soundness) We describe an algorithm for deciding structural soundness of a process graph $P(N, E, T, A)$:

1. Check if P is defined, i.e. has exactly one initial and exactly one final node (see definition 5.7).
2. Check if P is strongly connected, i.e. if every node is on a path from the initial to the final node (see definition 5.8). □

A structural sound process graph builds the foundation for verification that will be introduced in the following.

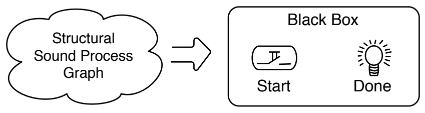


Figure 5.11: Black box investigation of a structural sound process graph.

5.3.2 Lazy Soundness

In a first verification of the execution semantics of a given structural sound process graph we would like to investigate its black box behavior. The black box behavior is given by an external observer that watches the execution of the initial and the final node. The approach is depicted in figure 5.11. A structural sound process graph, representing a business process, is placed inside a black box with a pushbutton and a bulb. The pushbutton is used to start a new process instance, whereas the bulb denotes the successful end of the process instance. The pushbutton corresponds to the execution of the initial node of the contained process graph, whereas the bulb denotes the execution of the final node. Each time the initial node is executed by pressing the pushbutton, the observer should see the execution of the final node exactly once at a later point in time by a flash of the bulb. If the observer cannot always observe the execution of the final node, the process graph must have serious errors leading to deadlocks or livelocks. If the final node is executed more than once, the observer is unable to detect when the process instance has ended. Both observations are a desired correctness property for business processes. They guarantee that once a business process is started it will always deliver a result.

The black box verification closely resembles the first criterion of definition 3.30 (Sound). It states that a workflow net has the option to always complete:

$$\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o) \; .$$

The main difference is given by the fact that the Petri net based soundness definition is based on states, whereas we would like to observe the occurrence of nodes. Similar to the given criterion is our aim of capturing all possible states that can occur in between the start and the end of a business process. However, the black box verification does not consider the second criterion of soundness:

$$\forall_M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o) \; .$$

This is due to the fact that the external observer does not have any knowledge about the nodes executed inside the black box. Hence, he cannot decide if further actions occur inside the black box. The same holds for the third criterion of soundness:

$$\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{t} M' \; .$$

Again, since the external observer has no knowledge about the nodes executed inside the black box, he cannot judge if all of them participate in the business process.

Due to the lack of supporting other observations beside the execution of the initial and final node, the black box verification provides a weaker soundness property than definition 3.30

(Sound) and the subset given by definition 3.39 (Weak Sound). It also misses definition 3.31 (Relaxed Sound), since equal to criterion three of soundness, observations regarding the executed nodes are required. In particular, the black box verification approach gives raise to dead nodes inside business processes (as forbidden by the third soundness criterion) as well as allowing nodes to be active after the final node has been reached (as forbidden by the second soundness criterion).

However, both criteria are dispensable under certain conditions. First of all, dead nodes can appear in interacting business processes, where a certain business process is defined in a generic manner with additional paths not used in all possible compositions. This argumentation has already been given by Martens for defining weak soundness (definition 3.39). Since we will investigate interacting business processes in chapter 6 (Interactions), a soundness property supporting arbitrary compositions will be useful. The second criterion, i.e. allowing nodes to be active after a final node has been reached, requires a differentiation between the *termination* and *end* of a process instance (see definition 3.8). A process instance is *ended* if it provides its outcome, i.e. the result it should provide. A process instance is *terminated* if no more activities can be executed. According to the soundness definition, the termination and the end of a business process are the same, given by the state o of a workflow net. However, in a black box verification approach, we can only observe the end of the business process. Since we cannot observe nodes other than the initial and the final, the actual termination of the business process is unobservable.

Indeed, a number of process patterns given in section 5.2 can leave nodes active after a distinguished final node has ended the business process. In particular, pattern 5.9 (Discriminator), pattern 5.10 (N-out-of-M-Join), and pattern 5.13 (Multiple Instances without Synchronization) show this behavior. These process patterns are called *critical patterns*. We denote the activities that can still be active after the business process has ended as *lazy activities*. Example 5.2 already gave a business process that contains two of the critical patterns. The first lazy activity is created after the n-out-of-m-join collected two incoming sequence flows and started the subsequent activity. For instance, after the activities A and B have finished, activity C might still be active. However, the n-out-of-m-join already triggered the subsequent activity D. Since D itself represents a multiple instances without synchronization pattern, three concurrent instances of D are created and the control flow is passed on immediately. Thus, in the worst case four activities are remaining active while the end event has already ended the business process.

Since the black box verification approach is different to existing soundness properties, we denote it as *lazy soundness* as it deals with business processes containing lazy activities. Informally, lazy soundness guarantees the following property of a business process:

> A structural sound process graph representing a business process is *lazy sound* if in any case a result is provided exactly once.

The result is provided through the execution of the final node. Thereafter, arbitrary actions, including those leading to livelocks and deadlocks, might happen. We are not interested in them, since from the viewpoint of an external observer, the business process has fulfilled its goal. The requirement of structural soundness for the underlying process graph provides distinguished initial and final nodes that can be observed.

The black box verification of lazy soundness requires some assumptions on the fairness of the execution semantics. While the observer is able to trigger the execution of the initial and see the execution of the final node, it remains unclear, from his point of view, if all possible paths inside the observed process graph have been taken after a number of executions. In particular, the observer is interested in an all quantification over the possible paths that can be traversed during the execution of a process graph. This knowledge would allow the generalization of the observed deadlock and livelock freedom. Furthermore, if the observer triggers the execution of the initial node, but cannot detect the execution of the final node at a later point in time, it remains unclear if he should wait any longer or if the process graph contains deadlocks or livelocks.

The above mentioned problems can be overcome by using bisimulation equivalences. Due to the π-calculus semantics of a process graph, we can compare the actual behavior of a process graph with a *wanted*, *invariant* behavior. Since a bisimulation contains an all quantification over all possible transitions, we can be sure that all possible paths inside the process graph have been traversed. Furthermore, the existence of a bisimulation renders the waiting problem void. The only thing that has to be added to the π-calculus mapping of a process graph are two free names i and $\overline{o}$ for observing the execution of initial and the final node.

Algorithm 5.3 (Lazy Soundness Annotated π-calculus Mapping) To annotate a π-calculus mapping D of a process graph $P = (N, E, T, M)$ according to algorithm 5.1 (Mapping Process Graphs to Agents) for reasoning on lazy soundness, we need to replace the functional abstractions of the agent definitions. Let n iterate over all elements of N and A_n be the agent representing the node n. Furthermore, $\{i, o\} \cap (fn(D) \cup bn(D)) = \emptyset$. The functional abstractions have to be replaced as follows:

- $A_n\langle\tau\rangle$, if n has incoming and outgoing edges (i.e. $|pre(n)| > 0 \wedge |post(n)| > 0$),
- $A_n\langle i.\tau\rangle$, if n has only outgoing edges (i.e. $|pre(n)| = 0 \wedge |post(n)| > 0$),
- $A_n\langle\tau.\overline{o}\rangle$, if n has only incoming edges (i.e. $|pre(n)| > 0 \wedge |post(n)| = 0$), and
- $A_n\langle i.\tau.\overline{o}\rangle$ if n has no incoming or outgoing edges (i.e. $|pre(n)| = |post(n)| = 0$). □

To annotate example 5.3 (Simple Business Process Formalization), all functional abstractions have to be replaced by τ, except for the agents representing the initial and the final node. These are modified as follows (the complete mapping is shown in appendix A.1.1):

$$N1 \overset{def}{=} i.\tau.\overline{e1}.\mathbf{0} \text{ and } N8 \overset{def}{=} e9.\tau.\overline{o}.N8 \; .$$

The invariant behavior, i.e. the expected one, is given by an agent

$$S_{LAZY} \overset{def}{=} i.\tau.\overline{o}.\mathbf{0} \; .$$

S_{LAZY} is composed of three prefixes representing the invariant behavior of a lazy sound process graph. First, the initial node is observed by i. Thereafter, arbitrary internal actions can happen (τ). In the end, the final node is always observed via $\overline{o}$. We added the τ-prefix to represent the black box placed in between the initial and final node. According to the weak bisimulation semantics, we can also omit it. In any case, we can now give a formal definition of a lazy sound process graph by comparing its actual behavior with the invariant one:

Definition 5.10 (Lazy Sound Process Graph) A structural sound process graph $P = (N, E, T, A)$ with a semantics given by the lazy soundness annotated π-calculus mapping D of P is *lazy sound* if $D \approx S_{LAZY}$ holds. $\square$

We conclude this section by providing an algorithm for deciding lazy soundness.

Algorithm 5.4 (Deciding Lazy Soundness) An algorithm for deciding lazy soundness of a structural sound process graph $P = (N, E, T, A)$ is given as follows:

1. Map the structural sound process graph to π-calculus, following algorithm 5.1.
2. Annotate the π-calculus mapping for lazy soundness, following algorithm 5.3.
3. Check the annotated mapping for weak bisimulation equivalence with S_{LAZY}.
4. If the equivalence holds, P is lazy sound. $\square$

Appendix A.1.1 shows how example 5.3 (Simple Business Process Formalization) is proven to be lazy sound using existing tools.

5.3.3 Weak Soundness

After the investigation of lazy soundness, which provides a soundness property closely related to the first criterion of soundness (definition 3.30), we would additionally like to mimic the second criterion using bisimulation:

$$\forall_M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o) \; .$$

The criterion states that the termination and the end of a process instance are the same. Therefore it enforces that after the state o no other state can follow. Hence, no lazy activities are allowed in a business process. As already motivated, this behavior can only be guaranteed by observing the execution of the nodes inside the black box. In contrast to the Petri net based definition given above, that enumerates all states, we reduce the investigation to the activities found in a business process. After the activity that is represented by the final node has been executed, no other activities should be or become active. Since the first and the second criterion of soundness are the same as weak soundness, we denote this property also as *weak soundness*. Informally, it guarantees the following properties of a business process:

> A structural sound process graph representing a business process is *weak sound* if in any case a result is provided and the process instance is terminated the moment the result is provided.

Due to the immediate termination of the business process no lazy activities can remain. Furthermore, the result can only be provided once. For proving weak soundness, we need to be able to observe the occurrence of nodes. If we can only observe the occurrence of nodes in between the

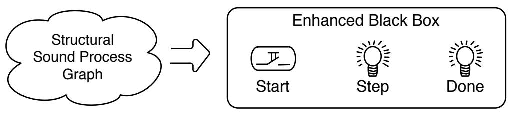


Figure 5.12: Enhanced black box investigation of a structural sound process graph.

observation of the initial and the final node, we can be sure that the business process is terminated at the moment the result is provided via the final node. An enhanced black box is shown in figure 5.12.

The external observer starts a new instance of the structural sound process graph given into the enhanced black box with a push of the start button. Thereafter he observes a flash of the step bulb for each execution of a node. Finally, he is able to observe a flash of the done bulb to denote the end of the process instance. If he is unable to observe a flash of the done bulb in all cases, deadlocks or livelocks are contained in the process graph rendering it unsound. If he is able to observe a flash of the step bulb after a flash of the done bulb, nodes of the process graph are still executed after the process instance has ended. Hence, the process graph contains lazy activities and thus is not weak sound.

However, bisimulation equivalences requires knowledge of how often the step bulb flashes in advance. Since we do not know this, we need a trick. We assume that each each agent of a process graph representing a node triggers a special agent. This special agent is placed as a component inside the global agent, so it is available to all other agents by a common name. The special agent is able to emit via a free name $\overline{s}$ exactly once, thus denoting the flash of the step bulb. The trick behind this agent is based on the idea that the emission via $\overline{s}$ is triggered non-deterministically. The special agent has the choice between emitting the name $\overline{s}$ or doing nothing. If he decided to do nothing, he has again the chance of emitting via $\overline{s}$ if he is triggered by another agent representing a node. However, after the emission via $\overline{s}$, he cannot emit via $\overline{s}$ anymore. Due to the non-deterministic behavior of the special agent, and the all quantification of bisimulation equivalence, we are able to observe the occurrence of $\overline{s}$ in between i and $\overline{o}$ either zero or one times. If the process graph contains lazy activities, we can observe $\overline{s}$ after $\overline{o}$. Furthermore, due to still observing $\overline{o}$, we are able to detect deadlocks and livelocks as before. The special agent is denoted as an *activity observation agent*.

Definition 5.11 (Activity Observation Agent) An *activity observation agent* is given by:

$$X(x,s) \stackrel{def}{=} x(ack).(\tau.\overline{ack}.\mathbf{0} \mid X(x,s)) + x(ack).(\tau.\overline{s}.\overline{ack}.\mathbf{0} \mid X_1(x))$$
$$X_1(x) \stackrel{def}{=} x(ack).(\tau.\overline{ack}.\mathbf{0} \mid X_1(x)) \; .$$

□

$X(x,s)$ receives a response channel ack via x and offers the non-deterministic choice between omitting the free name s and continue as $X_1(x)$ or do an unobservable action τ and behave as $X(x,s)$ again after triggering the received response channel. $X_1(x)$ has no observable behavior. An activity observation agent gives each instance of a process graph mapped to π-calculus agents the possibility of emitting via $\overline{s}$ once. The inclusion of the acknowledgment via ack is required, due to otherwise the final node can emit via $\overline{s}$ after $\overline{o}$. Furthermore, due to ack, the more natural way of placing the π-calculus mapping of a process graph in a context is blocked. The activity observation agent is included in a π-calculus mapping as follows:

Algorithm 5.5 (Weak Soundness Annotated π-calculus Mapping) To annotate the π-calculus mapping D of a process graph $P = (N, E, T, A)$ according to algorithm 5.1 (Mapping Process Graphs to Agents) for reasoning on weak soundness, we need to replace the functional abstractions of the agent definitions. Let n iterate over all elements of N and A_n be the agent representing the node n. Furthermore, $\{i, o, s, x\} \cap (fn(D) \cup bn(D)) = \emptyset$. The functional abstractions have to be replaced as follows:

- $A_n\langle \nu ack\ \overline{x}\langle ack\rangle.ack\rangle$, if n has incoming and outgoing edges (i.e. $|pre(n)| > 0 \wedge |post(n)| > 0$),
- $A_n\langle \nu ack\ i.\overline{x}\langle ack\rangle.ack\rangle$, if n has only outgoing edges (i.e. $|pre(n)| = 0 \wedge |post(n)| > 0$),
- $A_n\langle \nu ack\ \overline{x}\langle ack\rangle.ack.\overline{o}\rangle$, if n has only incoming edges (i.e. $|pre(n)| > 0 \wedge |post(n)| = 0$), and
- $A_n\langle \nu ack\ i.\overline{x}\langle ack\rangle.ack.\overline{o}\rangle$ if n has no incoming or outgoing edges (i.e. $|pre(n)| = |post(n)| = 0$).

Furthermore, we need to add the activity observation agent from definition 5.11 to the global agent D, providing a restricted name x to all components of D:

$$D \stackrel{def}{=} (\nu e1, \ldots, e|E|, x)(\prod_{i=1}^{|N|}(Di) \mid X)\ .$$

□

The introduction of the free names i, $\overline{o}$, and $\overline{s}$ ensures the external observability of the π-calculus mapping of a process graph regarding weak soundness. An example of a weak soundness annotated π-calculus mapping of a process graph is shown in appendix A.1.2.

The invariant behavior is given by an agent

$$S_{WEAK} \stackrel{def}{=} i.(\tau.\overline{o}.\mathbf{0} + \tau.\overline{s}.\overline{o}.\mathbf{0})\ .$$

S_{WEAK} is composed as S_{LAZY} regarding i and $\overline{o}$. After the observation of the initial node via i, a deterministic choice between observing $\overline{o}$ or $\overline{s}$ is made. If $\overline{o}$ is observed, no other observations are possible (due to S_{WEAK} becomes inaction). If $\overline{s}$ is observed, the next observation has to be $\overline{o}$. Thereafter, no other observations are possible. This behavior resembles the enhanced black box with the exception that the step bulb might flash only once before the done bulb flashes. A formal definition of weak soundness for a process graph is now given by:

Definition 5.12 (Weak Sound Process Graph) A structural sound process graph $P = (N, E, T, A)$ with a semantics given by the weak soundness annotated π-calculus mapping D of P is *weak sound* if $D \approx S_{WEAK}$ holds. □

An algorithm for deciding weak soundness is given accordingly to algorithm 5.4 (Deciding Lazy Soundness). Examples for π-calculus mappings regarding weak soundness can be found in appendix A.1.1.

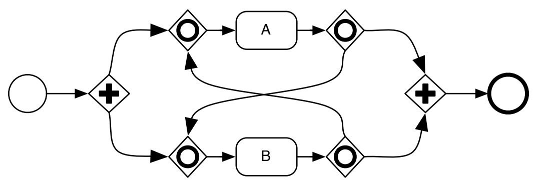


Figure 5.13: A vicious circle according to [7].

5.3.4 Relaxed Soundness

We conclude the investigations on soundness by a slightly modified version of the third criterion of definition 3.30 (Sound). Instead of analyzing the reachability of each node of a process graph as given accordingly for workflow nets by

$$\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{t} M' \, ,$$

we additionally would like to include the final node. This property has been given for workflow nets by definition 3.31 (Relaxed Sound):

$$\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{\tau} M' \xrightarrow{*} o \, .$$

The formula states that each task of a workflow net can participate in at least one transition sequence leading from the initial to the final state. According to process graphs, we can define *relaxed soundness* informally as:

> A structural sound process graph representing a business process is *relaxed sound* if each node of the process graph has the possibility of being executed in between the execution of the initial and the final node.

The definition of relaxed soundness does not cover deadlocks, livelocks, or lazy activities. A relaxed sound business process only guarantees that a minimum number of valid executions covering all activities are contained. However, it supports business processes containing pattern 5.7 (Synchronizing Merge). The semantics of this pattern is of high interest, since it renders simple merge and synchronization void, leaving only one merge pattern for a business process designer. However, a business process containing the synchronizing merge pattern cannot be lazy or weak sound. Our argumentation is based on two propositions.

Proposition 5.1 *A synchronizing merge consisting of more than one incoming edge has always non-deterministic local behavior.*

This proposition is given by the unguarded summations of pattern 5.7 (Synchronizing Merge). Interestingly, the pattern cannot be given in a deterministic formalization without loosing its universality. This has already been discussed extensively in other formalization approaches such as [51, 11, 137]. Hence, we derive a second proposition:

Proposition 5.2 *There exists no universal algorithm for deciding the number of incoming edges for a synchronizing merge consisting of more than one incoming edge.*

For the second proposition we refer to the vicious circle introduced in [7]. The proof starts by assuming that there exists a universal algorithm deciding the number of incoming edges for a synchronizing merge. However, there exist processes that have an ambiguous behavior regarding the synchronous merge. Figure 5.13 depicts an example. While a detailed discussion can be found in the cited paper, it is obvious that the problem occurs at the or-join gateways right after the and-gateway. Each gateway has to make a decision if it should resume the sequence flow further downstream or wait for more incoming sequence flows. Since the gateways in both parallel flows depend on the outcome of the complementary parallel flow, a non-deterministic decision has to be made. Thus, the universal algorithm has to solve the non-determinism, which in turn leads to a contradiction, since neither sequence flow is the *right* one.

According to propositions 5.1 and 5.2, business processes containing the synchronizing merge pattern always contain deadlocks due to the semantics of the pattern. Since lazy and weak soundness do not allow deadlocks they are not applicable. To support reasoning on business processes with synchronizing merges using these properties, we can use several workarounds ranging from introducing a local semantics (e.g. true/false token [83]) to global analysis (e.g. delay the synchronizing merge, while other activities can be executed [11]). We do not discuss these workarounds further, but focus on proving relaxed soundness using the enhanced black box verification approach.

Similar to weak soundness, we need to be able to observe the execution of nodes inside the black box for proving relaxed soundness. We can reuse the interface of the enhanced black from figure 5.12 by preparing the process graph according to the node that should be observed. To be able to observe the execution of all nodes, we need to prepare as much process graphs as there are nodes in it. In particular, we need to create a π-calculus mapping of the process graph for each node that should be observed. The agent representing the node under observation has to be enhanced with the ability of emitting $\overline{s}$. Thereafter we can place all prepared process graphs into the enhanced black box and investigate if for each process graph we can at least once see the step and done bulb flashing in sequence after a press of the pushbutton. However, we might have a problem if a node is contained in the loop that is always executed more than once. In this case we observe multiple flashes of the step bulb that in turn is difficult to formally analyze. To overcome this problem, we define a special agent that is triggered by each node.

Definition 5.13 (Activity Loop Observation Agent) An *activity loop observation agent* is given by:

$$Y(y,s) \overset{def}{=} y(ack).\overline{s}.\overline{ack}.Y_1(y) \text{ and } Y_1(y) \overset{def}{=} y(ack).\tau.\overline{ack}.Y_1(y)\ .$$

□

$Y(y,s)$ receives a response channel ack via y and emits the free name s one time before sending the acknowledgment and continuing as Y_1. Further interactions via y do not emit the free name s again. Hence, the free name s is only emitted once even if the agent interacting with Y represents a node contained inside an arbitrary cycle. The activity loop observation agent is included in the π-calculus mapping as follows:

Algorithm 5.6 (Relaxed Soundness Annotated π-calculus Mapping) To annotate a π-calculus mapping D of a process graph $P = (N,E,T,M)$ according to algorithm 5.1 (Mapping Process Graphs to Agents) for reasoning on relaxed soundness regarding a certain node $p \in N$, we need to replace the functional abstractions of the agent definitions. Let n iterate over all elements of N and A_n be the agent representing the node n. Furthermore, $\{i, o, s, y\} \cap (fn(D) \cup bn(D)) = \emptyset$. The functional abstractions have to be replaced as follows:

- If $n = p$,
  - $A_n\langle \nu ack\ \overline{y}\langle ack\rangle.ack\rangle$, if n has incoming and outgoing edges (i.e. $|pre(n)| > 0 \wedge |post(n)| > 0$),
  - $A_n\langle \nu ack\ i.\overline{y}\langle ack\rangle.ack\rangle$, if n has only outgoing edges (i.e. $|pre(n)| = 0 \wedge |post(n)| > 0$),
  - $A_n\langle \nu ack\ \overline{y}\langle ack\rangle.ack.\overline{o}\rangle$, if n has only incoming edges (i.e. $|pre(n)| > 0 \wedge |post(n)| = 0$), and
  - $A_n\langle \nu ack\ i.\overline{y}\langle ack\rangle.ack.\overline{o}\rangle$ if n has no incoming or outgoing edges (i.e. $|pre(n)| = |post(n)| = 0$).
- else
  - $A_n\langle \tau\rangle$, if n has incoming and outgoing edges (i.e. $|pre(n)| > 0 \wedge |post(n)| > 0$),
  - $A_n\langle i.\tau\rangle$, if n has only outgoing edges (i.e. $|pre(n)| = 0 \wedge |post(n)| > 0$),
  - $A_n\langle \tau.\overline{o}\rangle$, if n has only incoming edges (i.e. $|pre(n)| > 0 \wedge |post(n)| = 0$), and
  - $A_n\langle i.\tau.\overline{o}\rangle$ if n has no incoming or outgoing edges (i.e. $|pre(n)| = |post(n)| = 0$)

Furthermore, we need to add the activity loop observation agent from definition 5.13 to the global agent D, providing a restricted name y to all components of D:

$$D \overset{def}{=} (\nu e1, \ldots, e|E|, y)(\prod_{i=1}^{|N|}(Di) \mid Y)\ .$$

□

The introduction of the free names i, $\overline{o}$, and $\overline{s}$ ensures the external observability of the π-calculus mapping of a process graph regarding relaxed soundness. For a complete reasoning on relaxed soundness, a mapping for each node of the process graph has to be investigated. An example of a relaxed soundness annotated π-calculus mapping of a process graph is shown in appendix A.1.3.

The invariant behavior is given by an agent

$$S_{RELAXED} \overset{def}{=} i.\overline{s}.\overline{o}.\mathbf{0}\ .$$

$S_{RELAXED}$ gives a sequence of free names that must be observable from each π-calculus mapping annotated according to relaxed soundness. However, in contrast to S_{LAZY} and S_{WEAK}, not all possible instances of a process graph have to show this behavior. Instead, it is enough if there *exists* an instance of the process graph that fulfill the wanted behavior. The difference between an exists and an all quantification can be expressed by using simulation instead of bisimulation. Consider for instance an agent A that simulates an agent B: A *can* do anything B can. However, A can contain additional observable behavior. Additional observable behavior is not allowed in a bisimulation equivalence, that enforces that A has exactly the same observable behavior as B and vice versa. Since we want a statement if there *exists* an instance of a process graph, we have to use simulation instead of bisimulation for defining relaxed soundness for process graphs:

Definition 5.14 (Relaxed Sound Process Graph) A structural sound process graph $P = (N, E, T, A)$ is *relaxed sound* if for each relaxed soundness annotated π-calculus mapping D considering $n \in N$ it holds that $S_{RELAXED} \lesssim_{\approx} D$. □

An corresponding algorithm for proving relaxed soundness has to consider all nodes of a process graph:

Algorithm 5.7 (Deciding Relaxed Soundness) An algorithm for deciding relaxed soundness of a structural sound process graph $P = (N, E, T, A)$ is given as follows:

1. Map the structural sound process graph to π-calculus, following algorithm 5.1.
2. Annotate a new copy of the π-calculus mapping from the first step according to relaxed soundness for each $n \in N$ as given by algorithm 5.6.
3. Check all annotated mappings for weak similarity with $S_{RELAXED}$.
4. If all annotated mappings fulfill the simulation, P is relaxed sound.

The algorithm can be optimized by considering only the nodes of a process graph that fulfill $n \in N | type(n) = Task$. □

An example for deciding relaxed soundness can be found in appendix A.1.3.

Chapter 6

Interactions

In this chapter we discuss how a set of distributed business processes can synchronize and communicate based on interaction flows. Therefore all participating process graphs are placed inside an *interaction graph* that is complemented with interaction flow. Due to link passing mobility of the π-calculus, not all interaction flows have to be statically pre-defined but can furthermore be created dynamically. Possible patterns, given by the *service interaction patterns*, for realizing interactions between process graphs are discussed. Finally, we introduce reasoning on interaction compatibility for a given process graph and a set of services, as well as a conformance notion between services.

6.1 Representation

This section describes how distributed, interacting business processes are formally represented in the π-calculus.

6.1.1 Correlations and Dynamic Binding

A common task between processes invoking other processes is response matching. This matchmaking is done using *correlations* that relate a response with a request. Usually, some kind of correlation identifier is placed inside each request and response. The invoking as well as the responding processes have to take care of correlating the requests based on the identifiers. In the π-calculus, the unique identifier of a request is represented by a restricted name. Since names are unique and can be used as interaction channels, an unambiguous representation of the correlations is straightforward. Consider for instance the interacting business processes represented by the agents A and B:

$$A \overset{def}{=} \nu ch\, \bar{b}\langle ch\rangle.(ch(r).A' \mid A) \text{ and } B \overset{def}{=} \nu r\, b(ch).(\tau.\overline{ch}\langle r\rangle.\mathbf{0} \mid B)\ .$$

Agent A is able to invoke B several times via b, even before a first response is received. B in turn is able to process multiple request initiated via b at the same time. Hence, matching requests and responses have to be correlated. This is done by using ch in A as a correlation identifier. Since ch is unique for each recursive execution of A, the matchmaking is done implicitly via ch.

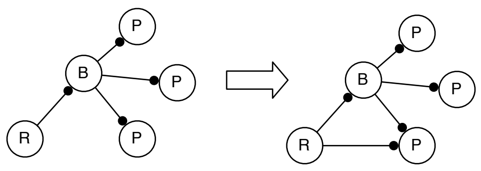


Figure 6.1: Dynamic binding in π-calculus.

Beside supporting correlations, the π-calculus directly expresses dynamic binding of interaction partners as found in service-oriented architectures. Figure 6.1 depicts how dynamic binding is realized using link passing mobility. The left hand side shows the three different roles of a SOA, denoted as circles. A service requester (R) knows a service broker (B). The service broker has knowledge about a number of service providers (P). The service broker evaluates the request of the service requester and returns a corresponding link to a service provider. The service requester then uses this link to dynamically bind to the service provider. Hence, the link structure changes over time as shown at the right hand side of the figure. A simple implementation of a broker having static knowledge of two providers reachable via $p1$ and $p2$ is given by the agent R:

$$B \overset{def}{=} b(ch).((\tau.\overline{ch}\langle p1\rangle.\mathbf{0} + \tau.\overline{ch}\langle p2\rangle.\mathbf{0}) \mid B) \; .$$

The agent B is able to emit either the name $p1$ or $p2$ based on an internal decision via the received name ch. A more elaborate implementation might use list structures, where possible providers can register and de-register during the runtime of the broker. However, we stuck to the simple variant for now. The service providers are given by the parameterized agent P:

$$P(p) \overset{def}{=} \nu resp\; p(req, ch).(\tau.\overline{ch}\langle resp\rangle.\mathbf{0} \mid P(p)) \; .$$

A service requester that is able to dynamically incorporate a service provider according to the interaction behavior of P is given by:

$$R \overset{def}{=} \nu req\; \nu ch1\; \nu ch2\; \overline{b}\langle ch1\rangle.ch1(p).\overline{p}\langle req, ch2\rangle.ch2(resp).\mathbf{0} \; .$$

In the first two transitions, R acquires a link to a specific service provider represented by p. Thereafter it uses p to dynamically bind to the service provider. The working system is given by:

$$SYS \overset{def}{=} \nu b\; \nu p1\; \nu p2\; (B \mid P1(p1) \mid P2(p2) \mid R) \; .$$

The system is composed out of the requester's agent R as well as others agents building an environment inside which R is running. This environment can now be changed, e.g. new service providers can be added or removed, all without modifying the service requester.

6.1.2 Structure

Interactions take place between processes represented by process graphs. A graph consisting of multiple connected process graphs representing an interaction structure is called an *interaction* *graph*. An interaction graph is a data structure that represents a set of process graphs with their respective interaction flows.

Definition 6.1 (Interaction Graph) An *interaction graph* is a three-tuple consisting of process graphs, directed interaction edges and a mapping of labels to interaction edges. Formally: $IG = (PS, C, L)$:

- PS is a finite, non-empty set of structural sound process graphs where all nodes are distinct.
- $C \subseteq (P1_N \times P2_N)$ with $P1(P1_N, \dots), P2(P2_N, \dots) \in PS$ is a set of directed interaction edges. Furthermore, $P1 \neq P2$.
- $L \subseteq (C \times LABEL)$ is a set of labels attached to directed interaction edges. □

The set PS defines the process graphs participating in the interaction. The directed interaction edges C connect different, interacting process graphs. Each directed interaction edge has a label assigned by the set L. The label type will be given in definition 6.2 (Interaction Flow Labels). We define functions for accessing components of interaction graphs:

- $source : C \rightarrow P_N$ returns the source node of a process graph $P(P_N, \dots)$ from the set of process graphs PS for a directed interaction edge.
- $target : C \rightarrow P_N$ returns the target node of a process graph $P(P_N, \dots)$ from the set of process graphs PS for a directed interaction edge.
- $in : N \rightarrow \mathcal{P}(C)$ returns the set of incoming interaction edges for a node N.
- $out : N \rightarrow \mathcal{P}(C)$ returns the set of outgoing interaction edges for a node N.
- $label : C \rightarrow LABEL$ returns the label of a directed interaction edge.

Using these functions, we are able to restrict the possible interaction graphs by stating that each node of a process graph contained in an interaction graph should have at most one interaction edge, either as a target or a source. The only exception is a *service node* that has exactly one in- and one outgoing interaction edge. However, a service node cannot be connected via an interaction edge to another service node. The restrictions are made to keep process behavior out of a node. If we allow an arbitrary number of interaction edges per node, process decisions like ordering and data dependencies would have to be solved inside the node. This would cause unwanted redundancy with the previous chapters. The restrictions are formally denoted as:

1. Generic Nodes: $\forall n \in P_N$ of $PS(P_N, \dots) : |in(n) \cup out(n)| \leq 1$, and
2. Service Nodes: $\forall n1 \in P_N$ of $PS(P_N, \dots) : |in(n1)| = 1 \wedge |out(n1)| = 1 \Rightarrow (\exists n2 \in P_N$ of $PS(P_N, \dots) : in(n1) = out(n2) \wedge |in(n2) \cup out(n2)| = 1) \wedge (\exists n3 \in P_N$ of $PS(P_N, \dots) : out(n1) = in(n3) \wedge |in(n3) \cup out(n3)| = 1)$.

In the remainder of this thesis we consider only interaction graphs satisfying the restrictions.

Definition 6.2 (Interaction Flow Labels) *Interaction flow labels* are derived from π-calculus names. A label consists of two parts: A name used as channel and optional names used as data, e.g. $channel(data1, data2, \dots)$. Formally:

$$\begin{aligned} LABEL &::= CH \mid [INT]\ CH\ \texttt{(}\ DATA\ \texttt{)} \\ CH &::= NAME \\ DATA &::= NAME \mid NAME\ \texttt{,}\ DATA \\ NAME &::= \pi\text{-calculus name} \end{aligned}$$

□

The following rules apply to interaction flow labels regarding the π-calculus:

- A name used as a channel represents the subject of an output prefix for the agent where the interaction flow originates from (source agent) and the subject of an input prefix for the agent where the interaction flow ends (target agent).
- If the name used as a channel has not been sent to the target agent before, it is restricted between all π-calculus agents interacting with the target agent.
- A name used as a data value that has not been received or restricted in the source agent before generates a new restricted name for the source agent.
- Furthermore, all scope extrusion and intrusion rules of the π-calculus apply.

We define functions for accessing elements of a label.

- $channel : LABEL \rightarrow CH$ returns the channel of a label.
- $data : LABEL \rightarrow \mathcal{P}(NAME)$ returns the set of data names of a label.

To show the coherence between an interaction graph and a graphical notation, we give an example of how to map the structurally relevant parts of a BPD, representing two abstract interacting business processes, to an interaction graph. We assume the messages flows to be labeled as stated in definition 6.2 (Interaction Flow Labels).

Example 6.1 (Partly Mapping of a BPD to an Interaction Graph) A BPD containing two or more private or abstract interacting business processes is mapped to an interaction graph $IG = (PS, C, L)$ as follows:

1. PS is given according to algorithm 5.1 (Partly Mapping of a BPD to a Process Graph).
2. C is given by all message flows of the BPD.
3. L is given by the labels of all messages flows of the BPD. □

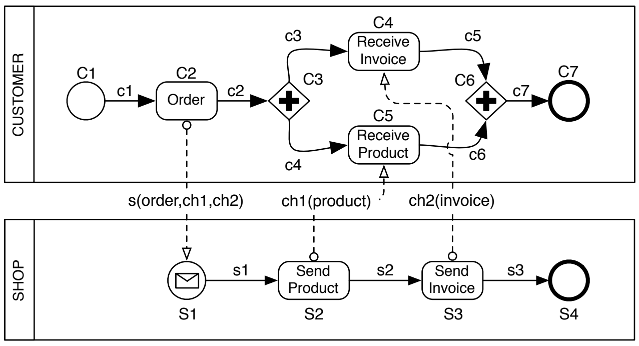


Figure 6.2: Two interacting business processes.

An example of two interacting business processes is given in figure 6.2. The upper pool represents a *Customer* process that sends a request to the lower pool representing a *Shop* process. The first interaction flow is labeled $s(order, ch1, ch2)$, where the cannel used is s and $\{order, ch1, ch2\}$ represent the corresponding payload. The *Shop* in turn uses the contained names $ch1$ and $ch2$ to ship a product and an invoice to the *Customer*. The complete interaction is mapped to an interaction graph according to the mapping rules given in example 6.1.

Example 6.2 (Two Interacting Business Processes) The interaction graph $IG = (PS, C, L)$ of the interaction from figure 6.2 is given by:

1. $PS = \{C, S\}$ with $C = (N_C, E_C, T_C, A_C)$ given by:

   (a) $N_C = \{C1, C2, C3, C4, C5, C6, C7\}$

   (b) $E_C = \{(C1, C2), (C2, C3), (C3, C4), (C3, C5), (C4, C6), (C5, C6), (C6, C7)\}$

   (c) $T_C = \{(C1, StartEvent), (C2, Task), (C3, ANDGateway), (C4, Task), (C5, Task), (C6, ANDGateway), (C7, EndEvent)\}$

   (d) $A_C = \emptyset$.

   and $S = (N_S, E_S, T_S, A_S)$ with:

   (a) $N_S = \{S1, S2, S3, S4\}$

   (b) $E_S = \{(S1, S2), (S2, S3), (S3, S4)\}$

   (c) $T_S = \{(S1, MessageStartEvent), (S2, Task), (S3, Task), (S4, EndEvent)\}$

   (d) $A_S = \emptyset$

2. $C = \{(C2, S1), (S2, C5), (S3, C4)\}$

3. $L = \{((C2, S1), s(order, ch1, ch2)), ((S2, C5), ch1(product)), ((S3, C4), ch2(invoice))\}$. □

We define a subtype of an interaction graph for reasoning on compatibility and conformance based on process graphs with an external visible behavior. The so-called *service graph* combines a process graph contained in an interaction graph with an external visible behavior:

Definition 6.3 (Service Graph) A *service graph* is a subset of an interaction graph containing in- or outbound interaction edges used as a behavioral interface. Formally, $SG = (PS, C, L)$:

- $PS = (N, E, T, A)$ is a structural sound process graph.
- $C \subseteq (N \times \bot) \cup (\bot \times N)$ is a set of directed interaction edges.
- $L \subseteq (C \times LABEL)$ is a set of labels attached to directed interaction edges. $\Box$

A service graph differs to an interaction graph by only considering one process graph with its in- and outgoing interaction edges. The symbol $\bot$ is used as a connector to an environment. The environment itself can either be the remaining part of an interaction graph without the service graph as well as an arbitrary process structure without any limitations as long as it is able to interact properly with the service graph. For a proper interaction between a service graph and an environment at least one *static interaction edge* between both has to exist.

Definition 6.4 (Static Interaction Edge) An interaction edge of a service graph $SG = (PS, C, L)$ is *static* if the channel of the label has not been acquired using dynamic binding. Hence, the channel is not found as the data of any interaction edge. Formally: An interaction edge $e1 \in C$ is static if $\forall e2 \in C : data(label(e2)) \neq channel(label(e1))$. The set of static interaction edges of SG is denoted as $C_{STATIC} \subseteq C$. $\Box$

Definition 6.5 (Environment) Let $SG = (PS, C, L)$ be a service graph. An *environment* E for SG is given if E uses at least one static interaction edge of SG. The behavioral interface E_i of E is given by a set consisting of tuples (DIR, L), where $DIR \rightarrow \{in, out\}$ and $L : LABEL$ as given in definition 6.2 (Interaction Flow Labels). E uses a static interaction edge of SG if:

- $\exists c \in C_{STATIC}$ with $target(c) = \bot : \exists (d, l) \in E_i$ such that $d = in \wedge channel(label(c)) = channel(l) \wedge |data(label(c))| = |data(l)|$, or
- $\exists c \in C_{STATIC}$ with $source(c) = \bot : \exists (d, l) \in E_i$ such that $d = out \wedge channel(label(c)) = channel(l) \wedge |data(label(c))| = |data(l)|$. $\Box$

A service graph SG unified with an environment E is denoted as $SG \uplus E$. Since an environment has no formal structure, we cannot give a semantics for $\uplus$ right now. However, after formalizing interaction and service graphs in the π-calculus in the next section, we will define an environment as an arbitrary agent being able to interact properly with the π-calculus representation of a service graph. By not limiting the structural properties of an environment already at this point, we gain freedom required to formally represent dynamic binding with participants unknown at design time.

6.1.3 Behavior

A formal semantics is given to an interaction graph by mapping it to π-calculus agents:

Algorithm 6.1 (Mapping Interaction Graphs to Agents) An interaction graph $IG = (PS, C, L)$ is mapped to π-calculus agents as follows. We denote the components of the process graphs from PS of IG with Pi_N, Pi_E, Pi_T, and Pi_A. Furthermore $\mathbb{G}$ represents a set of names known to all interacting processes.

1. Map all processes graphs of PS to agents denoted by Pi according to algorithm 5.1 (Mapping Process Graphs to Agents). Use α-conversion, if required, to ensure that $bn(Pi) \cap (channel(l) \cup data(l)) = \emptyset$ for each $l \in L$.

2. Execute for all nodes of all process graphs from PS that are the target of an interaction edge. i.e. $\forall c \in C : target(c) = n, n \in Pi_N$, the following sub-procedure:

   (a) Replace the functional abstraction of the π-calculus representation of the node n with an input prefix of the subject $channel(label(c))$ and the object(s) $data(label(c))$ followed by ".$\langle \cdot \rangle$". Exception for pattern 5.17 (Deferred Choice): If the node type of n is an intermediate message event, and another node $n2$ directly preceding n has the type event based gateway, the subject has already been inserted in the agent mapping of $n2$ before $\langle \cdot \rangle$. In this case, only additional objects are added to the prefix where the subject appears, whereas the agent mapping of n is untouched.

   (b) Take care that all received objects are passed to all further nodes of the process graph containing the node n (i.e. all nodes reachable from n). See algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents) for details.

3. Execute for all nodes of all process graphs from PS that are the source of an interaction edge, i.e. if $\forall c \in C : source(c) = n, n \in Pi_N$, the following sub-procedure:

   (a) Replace the functional abstraction of the π-calculus representation of the node n with "$\langle \cdot \rangle$." followed by an output prefix of the subject $channel(label(c))$ and the object(s) $data(label(c))$.

   (b) If the subject of the output prefix has not been received during an earlier interaction:

      - Default: Add the subject of the output prefix to $\mathbb{G}$ to restrict it between the two interacting agents.
      - Interaction edges targeting $\perp$ (Service Graphs): Don't restrict the output prefix.

   (c) For all objects contained in the output prefix: If the object has not been received during an earlier interaction, restrict the object before the output prefix and take care that the scope is extruded to all agents representing further nodes of the corresponding process graph. See algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents) for details.

4. Define an agent $I \overset{def}{=} \nu\mathbb{G} \ (\prod_{i=1}^{|PS|} Pi)$ representing the interaction graph IG. $\square$

The formalization of an interaction graph in the π-calculus starts with a mapping of each process graph contained in the interaction graph to agents. Thereafter, all agents representing nodes of an interaction graph with an an incoming interaction edge are modified with additional preconditions based on the interaction flow label. Finally, all agents representing nodes of an interaction graph with an outgoing interaction edge are modified with additional postconditions, again based on the interaction flows label. For instance, an agent representing a node of a process graph contained inside a sequence pattern is modified as follows:

$$A \overset{def}{=} a.\langle\cdot\rangle.\bar{b}.\mathbf{0}\ ,$$

becomes

$$A \overset{def}{=} a.ch(x).\langle\cdot\rangle.\bar{b}.\mathbf{0}\ ,$$

with an incoming interaction flow labeled $ch(x)$. If an additional outgoing interaction flow is connected to the node represented by A, such as $x(resp)$, the agent becomes

$$A \overset{def}{=} a.ch(x).\langle\cdot\rangle.\nu resp\ \overline{x}\langle resp\rangle.\bar{b}.\mathbf{0}\ .$$

The name $resp$ has been restricted due to the fact that it has not been received during an earlier interaction. To decide what an "earlier interaction" means, an implementation of algorithm 6.1 has to consider all interactions by pre-processing the interaction graph. A more complex example is introduced below:

Example 6.3 (Two Interacting Business Processes Formalization) The interaction graph from example 6.2 (Two Interacting Business Processes) is mapped to π-calculus agents according to algorithm 6.1 (Mapping Interaction Graphs to Agents). We start by mapping the process graphs C and S contained in PS:

$$C \overset{def}{=} (\nu c1, \ldots, c7\,) \prod_{i=1}^{7} Ci \text{ and } S \overset{def}{=} (\nu s1, \ldots, s3\,) \prod_{i=1}^{4} Si\ .$$

The agents Ci and Si are given accordingly as stated in algorithm 5.1 (Mapping Process Graphs to Agents). We only consider nodes with in- or outgoing interaction flow and omit recursion since the example contains no arbitrary cycles. For accuracy with figure 6.2 (Two Interacting Business Processes), we proceed in a logical order instead of first processing incoming and then outgoing interaction edges.

First of all, node $C2$ has an outgoing interaction flow labeled $s(order, ch1, ch2)$, hence the corresponding agent is given by:

$$C2 \overset{def}{=} \nu order, ch1, ch2\ c1.\langle\cdot\rangle.\overline{s}\langle order, ch1, ch2\rangle.\overline{c2}\langle order, ch1, ch2\rangle.\mathbf{0}\ .$$

The names $order$, $ch1$, and $ch2$ are restricted inside the agent $C2$ because they have not been received during an earlier interaction. The interaction itself takes places via s, where the restricted names are communicated. Furthermore, they are forwarded to agents representing nodes downstream in the process graph via $c2$. Additionally, the name s has to be restricted between C and S:

$$I \overset{def}{=} \nu s\ (C \mid S)\ .$$

The corresponding interaction partner for node $C2$ is node $S1$ of the process graph S. This process graph represents a service as can be conducted from the corresponding BPD shown in figure 6.2. The service waits for a request and sends two different response messages. The agent representing node $S1$ is given by:

$$S1 \stackrel{def}{=} s(order, ch1, ch2).\langle\cdot\rangle.\overline{s1}\langle order, ch1, ch2\rangle.\mathbf{0}\ .$$

The agent $S1$ has an interaction based precondition s. After receiving via s, the objects received are forwarded to agents represent subsequent nodes via $s1$. The next interaction is contained in the node $S2$, where a restricted name $product$ is returned to node $C5$ of process graph C:

$$S2 \stackrel{def}{=} \nu product\ s1(order, ch1, ch2).\langle\cdot\rangle.\overline{ch1}\langle product\rangle.\overline{s2}\langle order, ch1, ch2, product\rangle.\mathbf{0}\ .$$

The response channel is not bound statically, but rather derived from the objects received in agent $S1$. This ensures the correct routing of the responses using an asynchronous callback mechanisms. The agent representing the corresponding node $C5$ is given by:

$$C5 \stackrel{def}{=} c4(order, ch1, ch2).ch1(product).\langle\cdot\rangle.\overline{c6}\langle order, ch1, ch2, product\rangle.\mathbf{0}\ .$$

Agent $C5$ interacts via $ch1$, which in turn it acquired as an object of $c4$. The last interaction is given by the agent representing the node $S3$:

$$S3 \stackrel{def}{=} \nu invoice\ s2(order, ch1, ch2, product).\langle\cdot\rangle.\overline{ch2}\langle invoice\rangle.\overline{s3}\langle order, ch1, ch2, \ldots\rangle.\mathbf{0}\ ,$$

as well as the agent representing the node $C4$:

$$C4 \stackrel{def}{=} c3(order, ch1, ch2).ch2(invoice).\langle\cdot\rangle.\overline{c5}\langle order, ch1, ch2, invoice\rangle.\mathbf{0}\ .$$

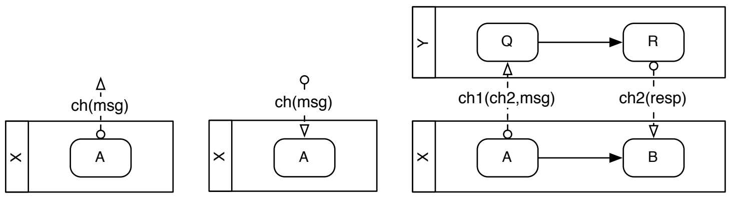
As can already be seen by this small example, algorithm 6.1 (Mapping Interaction Graphs to Agents) creates a large overhead of names to be forwarded to other agents. To overcome this problem, only the names required for interactions in agents representing nodes further downstream the process graphs can be forwarded. This can either be done by analyzing the interaction graph beforehand or by optimizing the derived agents afterwards.

To conclude this section, we can now give a formal description of an environment for a service graph mapped to agents:

Definition 6.6 (Environment Agent) Let S be a service graph mapped to agents (According to algorithm 6.1). A π-calculus agent E is called an *environment agent* for S if they share at least one free name, i.e. $fn(E) \cap fn(S) \neq \emptyset$. The cardinalities of the objects of all prefixes in E and S whose subjects match the intersection of the free names of E and S have to be the same. Furthermore, all free names used as subjects of input or output prefixes in S must have a corresponding input or output prefix in E. This means, that the subjects where the free names are used have to be inverse between S and E. □

A subject α is inverted by the following function:

$$inverse(\alpha) = \begin{cases} \alpha = x & : \ \overline{x} \\ \alpha = \overline{x} & : \ x \end{cases}\ .$$



(a) Send. (b) Receive. (c) Send/Receive.

Figure 6.3: Single transmission bilateral interaction patterns.

The definition of an environment agent E for a certain agent S representing a service graph states that S might have the possibility to interact with E. According to definition 6.5 (Environment), this means that at least one interaction edge of the service graph of S, represented by the set of free names of S, is used. We can now state how S is formally unified with E, i.e. $S \uplus E$:

$$SYS \stackrel{def}{=} \nu(fn(S) \cup fn(E))\ (S \mid E) \quad . \tag{6.1}$$

The unification of an agent S representing a service graph and an environment agent E is given by the parallel composition of S and E as well as restricting the free names of S and E.

6.2 Interaction Patterns

After having introduced the principles of interactions in the π-calculus, we investigate how common patterns of interaction can be represented in different process, interaction, or service graph structures. In particular, we investigate the service interaction patterns as described in [25]. To give a more elaborate presentation of the patterns, we use the BPMN notation as introduced in chapter 3.3.1 (Business Process Diagrams). Example 5.1 (Partly Mapping of a BPD to a Process Graph) shows how this notation can be mapped to process graphs. The description of the service interaction patterns has been adapted to match the terminology used throughout this thesis.

6.2.1 Single Transmission Bilateral Interaction Patterns

The single transmission bilateral interaction patterns represent basic interaction behavior. Graphical representations are shown in figure 6.3.

Pattern 6.1 (Send) *Description: A process sends a message to another process. (According to [25, p.4])*

Implementation: A graphical representation of this pattern is shown in figure 6.3(a). The π-calculus mapping implements a reliable delivery with a blocking semantics as follows:

$$A \stackrel{def}{=} \langle\cdot\rangle.\overline{ch}\langle msg\rangle.\mathbf{0} \ .$$

The implementation of pattern 6.1 (Send) does not show how A actually acquires the name ch. If an interaction between A and a composition of other agents E is defined as

$$I \overset{def}{=} \nu ch\ (A \mid E)\ ,$$

a static binding is described. If it is defined as

$$I \overset{def}{=} \nu lookup\ (lookup(ch).A \mid E)\ ,$$

with E being able to communicate a name used for interaction with a certain component of itself via $lookup$, a dynamic binding is described. If an unreliable message transmission should be modeled, an agent acting as a proxy between A and the environment has to be added (here with static binding):

$$I \overset{def}{=} \nu ch\ (A \mid B \mid E)\ ,$$

with B given by $B \overset{def}{=} ch(x).B$. Due to the non-determinisms contained in I, interactions via ch can now be *captured* by B, thus providing an unreliable delivery. These considerations on static vs. dynamic and reliable vs. unreliable message transmission hold for the remaining interaction patterns as well.

Pattern 6.2 (Receive) *Description: A process receives a message from another process. (According to [25, p.5])*

Implementation: A graphical representation of this pattern is shown in figure 6.3(b). The π-calculus mapping implements a reliable reception with a blocking semantics as follows:

$$A \overset{def}{=} ch(msg).\langle\cdot\rangle.\mathbf{0}\ .$$

Pattern 6.3 (Send/Receive) *Description: A process X engages in two causally related interactions. In the first interaction X sends a message to another process Y (the request), while in the second one X receives a message from Y (the response). (According to [25, p.7])*

Implementation: A graphical representation of this pattern is shown in figure 6.3(c). The π-calculus mapping implements a reliable interaction with a blocking semantics as follows:

$$I \overset{def}{=} \nu ch1\ (X \mid Y) \text{ with } X \overset{def}{=} \nu x1\ (A \mid B), \text{and } Y \overset{def}{=} \nu y1\ (Q \mid R)\ .$$

The components of X are given by:

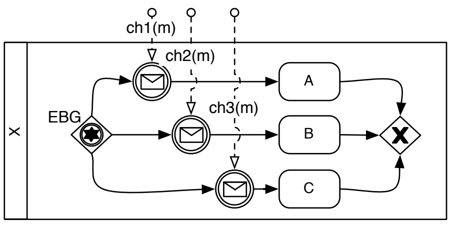
$$A \overset{def}{=} \nu ch2\ \nu msg\ \langle\cdot\rangle.\overline{ch1}\langle ch2, msg\rangle.\overline{x1}\langle ch2, msg\rangle.\mathbf{0}$$

and

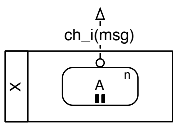
$$B \overset{def}{=} x1(ch2, msg).ch2(resp).\langle\cdot\rangle.\mathbf{0}\ .$$

The components of Y are given by:

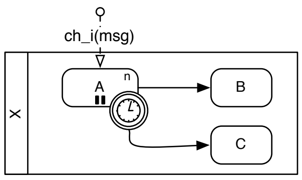
$$Q \overset{def}{=} ch1(ch2, msg).\langle\cdot\rangle.\overline{y1}\langle ch2, msg\rangle$$



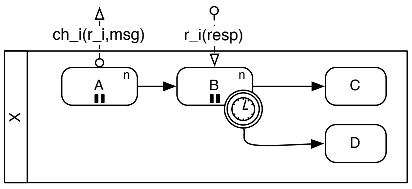
(a) Racing incoming messages.



(b) One-to-many send.



(c) One-from-many receive.



(d) One-to-many send/receive.

Figure 6.4: Single transmission multilateral interaction patterns.

and

$$R \overset{def}{=} \nu resp\ y1(ch2, msg).\langle\cdot\rangle.\overline{ch2}\langle resp\rangle.\mathbf{0}\ .$$

The send and receive interactions are correlated via the restricted name $ch2$ created inside agent A.

6.2.2 Single Transmission Multilateral Interaction Patterns

The single transmission multilateral interaction patterns represent one to many or many to one interactions. Graphical representations are shown in figure 6.4. We use multiple instance tasks to represent the transmission or reception of multiple names. The names are annotated with indices, where we assume that the index of each name is unique to an instance. Typically, the indices will be counted from one to n.

Pattern 6.4 (Racing Incoming Messages) *Description: A process expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of processes. The way a message is processed depends on its type and/or the category of processes from which it comes. (According to [25, p.8])*

Implementation: A graphical representation of this pattern is shown in figure 6.4(a). It resembles pattern 5.17 (Deferred Choice). The π-calculus mapping of the event-based gateway EBG implements a reliable interaction with a blocking semantics as follows:

$$EBG \overset{def}{=} \langle\cdot\rangle.(c1(m).\overline{x1}\langle m\rangle.\mathbf{0} + c2(m).\overline{x2}\langle m\rangle.\mathbf{0} + c3(m).\overline{x3}\langle m\rangle.\mathbf{0})\ ,$$

where $x1$, $x2$, and $x3$ represent names used as preconditions for the intermediate message events. This pattern requires a special processing for the mapping from interaction graphs to agents. The names $c1 \ldots c3$ are actually taken from the interaction flows of the nodes directly following the node representing the event-based gateway. This is required due to the fact the the π-calculus supports no transactional transitions.

Pattern 6.5 (One-to-many Send) *Description: A process sends messages to several other processes. The messages all have the same type (although their contents may differ). (According to [25, p.9])*

Implementation: A graphical representation of this pattern is shown in figure 6.4(b). The π-calculus mapping is given by pattern 6.1 (Send) executed n times using pattern 5.14 (Multiple Instances with a priori Design Time Knowledge) or pattern 5.15 (Multiple Instances with a priori Runtime Knowledge), depending on the point in time where the interaction partners are known.

Pattern 6.6 (One-from-many Receive) *Description: A process receives a number of logically related messages that arise from autonomous events occurring at different processes. The arrival of messages needs to be timely so that they can be correlated as a single logical request. The interaction may complete successfully or not depending on the set of messages gathered. (According to [25, p.11])*

Implementation: A graphical representation of this pattern is shown in figure 6.4(c). The π-calculus representation is given by pattern 6.2 (Receive) executed n times using pattern 5.14 (Multiple Instances with a priori Design Time Knowledge) or pattern 5.15 (Multiple Instances with a priori Runtime Knowledge), depending on the point in time where the interaction partners are known. If a timeout occurs, i.e. not all responses have been gathered within a certain interval, the control flow is rerouted using the intermediate message event.

Pattern 6.7 (One-to-many Send/Receive) *Description: A process sends a request to several other processes, which may all be identical or logical related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe and some processes may even not respond at all. The interaction may complete successfully or not depending on the set of responses gathered. (According to [25, p.14])*

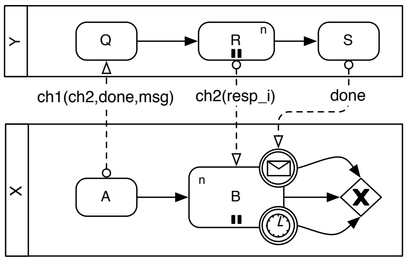
Implementation: This pattern combines the two preceding patterns (One-to-many Send and One-from-many Receive) into one pattern. A graphical representation is shown in figure 6.4(d). The forwarding of the names created, i.e. r_i, should be implemented according to pattern 4.12 (Data Interaction—From Multiple Instance Activities) and pattern 4.11 (Data Interaction—To Multiple Instance Activities).

6.2.3 Multi Transmission Interaction Patterns

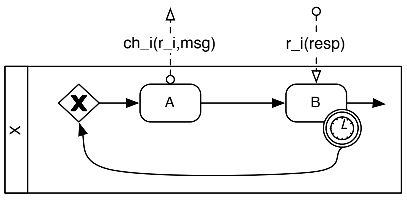
The multi transmission interaction patterns represent many to many interactions. Graphical representations are shown in figure 6.5.

Pattern 6.8 (Multi-responses) *Description: A process X sends a request to another process Y. Subsequently, X receives any number of responses from Y until no further responses are required. The trigger of no further responses can arise from a temporal condition or message content, and can arise from either X or Y's side. (According to [25, p.15])*

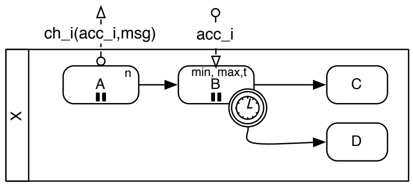
Implementation: A graphical representation of this pattern is shown in figure 6.5(a). Process



(a) Multi-responses.



(b) Contingent request.



(c) Atomic multicast notification.

Figure 6.5: Multi transmission interaction patterns.

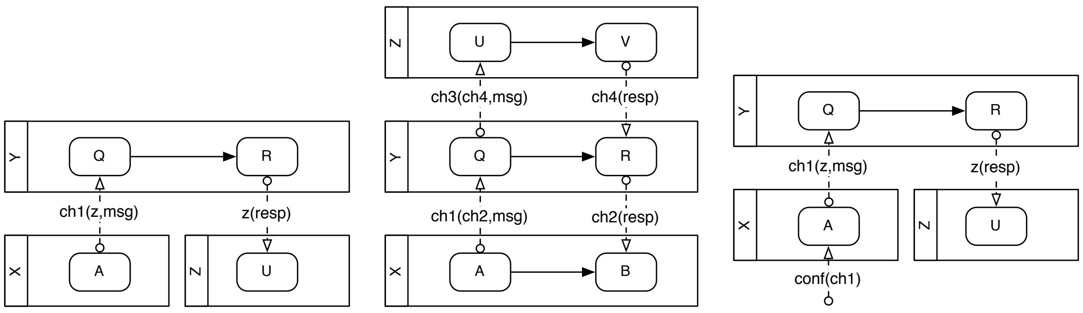
X as the initiator of the interaction sends two names $done$ and msg beside the response channel $ch2$ to Y. The name $done$ is used by Y to signal the stop condition to X, where the reception of new messages on $ch2$ is aborted. Furthermore, the reception of new messages by X can be aborted by a timeout attached to activity B.

Pattern 6.9 (Contingent Request) *Description: A process X makes a request to another party Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another process Z, and so on. (According to [25, p.17])*

Implementation: A graphical representation of this pattern is shown in figure 6.5(b). In the beginning of the pattern, a request is send on channel ch_i, where i enumerates the different interaction partners. The subsequent activity B has a timeout attached leading to another iteration of the pattern if the response is not received within time. According to the formalization of pattern 5.22 (Event-based Rerouting), each response is accepted event if the timeout has been activated. However, if a timeout occurred, the message is discarded and no further control flow is enabled.

Pattern 6.10 (Atomic Multicast Notification) *Description: A process sends notifications to several processes such that a certain number of processes are required to accept the notification within a certain timeframe. For example, all processes or just one process are required to accept the notification. (According to [25, p.18])*

Implementation: A graphical representation of this pattern is shown in figure 6.5(c). This pattern resembles pattern 6.7 (One-to-many Send/Receive). However, the minimum, maximum, and thresholds values of activity B have to be set according to the required notifications.



(a) Request with a Referral. (b) Relayed Request. (c) Dynamic Routing.

Figure 6.6: Routing patterns.

6.2.4 Routing Patterns

The routing patterns describe flexible interaction behavior between a set of processes. Graphical representations are shown in figure 6.6.

Pattern 6.11 (Request with a Referral) *Description: Process X sends a request to process Y indicating that any follow-up response should be sent to a number of other processes (Z1, Z2, . . . , Zn) depending on the evaluation of certain conditions. While faults are sent by default to these processes, they could alternatively be sent to another nominated process (which may be process A). (According to [25, p.20])*

Implementation: A graphical representation of this pattern is shown in figure 6.6(a). The referral is contained as z in the object of $ch1$. However, X and Z need to share the name z beforehand. Instead of incorporating a single interaction partner via z, also a number of interaction partners can be integrated via z_i.

Pattern 6.12 (Relayed Request) *Description: Process X makes a request to process Y which delegates the request to other processes (Z1, . . . , Zn). Processes Z1, . . . , Zn then continue interacting with process X while process Y observes a "view" of the interactions including faults. The interacting parties are aware of this "view". (According to [25, p.21])*

Implementation: A graphical representation of this pattern is shown in figure 6.6(b). Contained is a simple interaction. A relayed request is establish by using Y as a proxy for the interaction. This solutions ensures that Y receives all interactions between X and Z while being able to capture the important ones.

Pattern 6.13 (Dynamic Routing) *Description: A request is required to be routed to several processes based on a routing condition. The routing order is flexible and more than one process can be activated to receive a request. When the processes that were issued the request have completed, the next set of processes are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the intermediate steps. (According to [25, p.22])*

Implementation: A graphical representation of this pattern is shown in figure 6.6(c). The pattern contains two mechanisms for deciding the routing. From A to Q the corresponding channel is "injected" dynamically at runtime from the outside via $conf$. From R to U, the channel is contained inside a message received in an activity (here Q) before R.

6.3 Properties

In this section we develop correctness properties for interactions. First, we extend lazy soundness for a process graph to include the possible interactions with a certain environment made up of different services. Thus, we prove a process graph to be free of deadlocks and livelocks including the invoked behavior of its interaction partners. Since these can be bound dynamically, link passing mobility has to be considered. The soundness property developed is called accordingly *interaction soundness*. It provides a *compatibility* notion between a service and its environment. Second, the π-calculus representations of two service graphs can be matched for behavioral equivalence. The property is called *interaction equivalence* and defines a *conformance* relation. It can be used for two major purposes. On the one hand it allows for testing if a service can be replaced by another one. On the other hand, it allows checking if a certain implementation of a service follows an abstract process.

6.3.1 Interaction Soundness

In this subsection we investigate an extension of the black box verification approach of lazy soundness to interactions between a service graph and its environment. In particular, we are interested if a given service graph contains deadlocks. Lazy soundness allows to prove this property for the internal structure of the process graph contained in the service graph. The extended property should additionally consider the interaction edges as pre- and postconditions to the nodes. Notable, only a single interaction edge has to exists initially. All other interaction edges can be acquired using dynamic binding. To make this work, a service graph has to be unified with a given environment. Similar to lazy soundness, each time an external observer executes the initial node, she should be able to observe the execution of the final node at a later point in time. Informally, interaction soundness can be described as follows:

> A service graph SG is *interaction sound* regarding environment E if $SG \uplus E$ is lazy sound.

Example 6.4 (Stock Exchange Interaction) Interaction soundness is motivated by an example shown in figure 6.7. The example describes the internal process of a *Stock Broker* and its environment. The stock broker offers the ability of bidding at two different stock exchanges at the same time. The order is thereafter placed at the first stock exchange responding positive, i.e. where the order has been accepted. This functionality is realized inside the process using pattern 5.9 (Discriminator). Since there are many stock exchanges available, with different properties such as fees, rates, and business hours, a *Stock Exchange Repository* is contained as a service in the environment. It is invoked as the first activity of the stock broker, *Find & Bind Stock Exchanges*. The repository has knowledge about a number of *Stock Exchanges*, connected in the

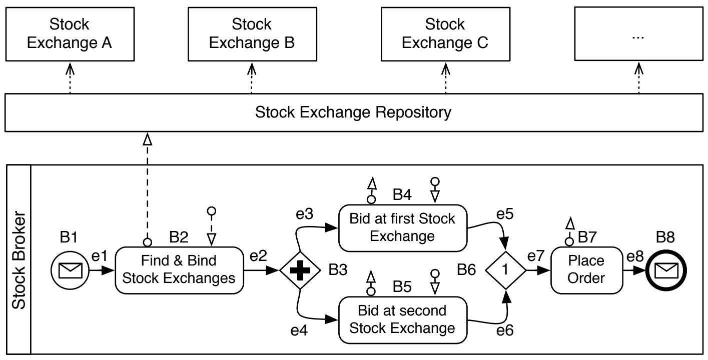

Stock Exchange A
Stock Exchange B
Stock Exchange C
...
Stock Exchange Repository
Stock Broker
B1
e1
B2
Find & Bind Stock Exchanges
e2
B3
e3
B4
Bid at first Stock Exchange
e4
B5
Bid at second Stock Exchange
e5
e6
B6
1
e7
B7
Place Order
e8
B8

Figure 6.7: Stock exchange interaction.

BPMN diagram using associations. Two of them matching the requested conditions are returned to the stock broker. The stock broker is now able to dynamically bind to the stock exchanges formerly unknown to him. This is denoted using in- and outgoing message flows at the activities *Bid at first Stock Exchange* and *Bid at second Stock Exchange*. Each stock exchange returns a special token if the bid has been accepted. This token is used inside *Place Order* to place the order at the corresponding stock exchange. Of course, only a successful bidder should be able to place the order.

We now argue why reasoning on the soundness of the example is not trivial. First, the stock broker's process contains a discriminator. As has already been shown, a discriminator leaves running (lazy) activities behind, i.e. one of the activities before the discriminator remains activated or running after the other one has already been finished. The activated or running activity might stay in this state even after the final node has been reached. Second, the process is contained inside an environment where the services are dynamically bound at runtime. Only the connection between the *Service Broker* and the *Stock Exchange Repository* is known at design time. Thus, reasoning on soundness includes an all quantification over the services that can be potentially bound at a given point in time. Third, a mechanisms for correlating selected stock exchanges with the activities *Bid at first Stock Exchange* and *Bid at second Stock Exchange* has to be provided.

As has already been suggested, lazy soundness (definition 5.10) provides a property dealing with lazy activities. The second and the third issue can be overcome as described in section 6.1.1 (Correlations and Dynamic Binding) by using service graphs mapped to agents (algorithm 6.1) and environment agents (definition 6.6). According to equation 6.1, we can define a system of agents that represent a service graph unified with an environment. Due to the fact of having a single representation of the system under investigation, we can use the black box verification approach without any modifications. The only thing that has to be done is a preparation of the agent mapping considering the unification and annotation according to lazy soundness.

Algorithm 6.2 (Interaction Soundness Annotated π-calculus Mapping) Let SYS be an agent consisting of the unification of a π-calculus mapping S of a service graph $SG = (PS, C, L)$

with an environment agent E as follows:

$$SYS \overset{def}{=} \nu(fn(S) \cup fn(E))\ (S \mid E) \text{ with } i, o \notin fn(S) \cup fn(E) \cup bn(S)\ .$$

The condition $i, o \notin fn(S) \cup fn(E) \cup bn(S)$ can always be fulfilled inside SYS by α-conversion if required. Furthermore, we need to replace the functional abstractions of the agent definitions. Let n iterate over all nodes of PS and A_n be the agent representing node n in SYS. The functional abstractions have to be replaced as follows:

- $A_n\langle\tau\rangle$, if the the corresponding service graph node has incoming and outgoing edges[1] (i.e. $|pre(n)| > 0 \land |post(n)| > 0$),
- $A_n\langle i.\tau\rangle$, if the corresponding service graph node has only outgoing edges (i.e. $|pre(n)| = 0 \land |post(n)| > 0$),
- $A_n\langle \tau.\bar{o}\rangle$, if the corresponding service graph node has only incoming edges (i.e. $|pre(n)| > 0 \land |post(n)| = 0$), and
- $A_n\langle i.\tau.\bar{o}\rangle$ if the corresponding service graph node has no incoming or outgoing edges (i.e. $|pre(n)| = |post(n)| = 0$). □

The invariant behavior of the agent mapping is given by S_{LAZY}. A formal definition of interaction soundness based on lazy soundness is then given by:

Definition 6.7 (Interaction Sound Service Graph) Let SYS be a π-calculus representation of a system consisting of (1) an interaction soundness annotated π-calculus mapping of a service graph SG unified with (2) an environment agent E according to algorithm 6.2. SG is *interaction sound* regarding the environment that E represents if $SYS \approx S_{LAZY}$ holds. □

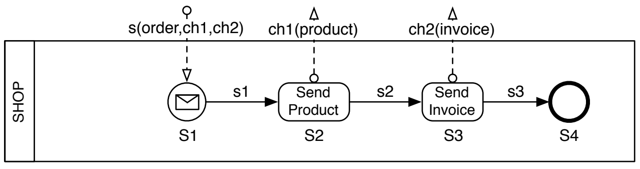
Appendix A.2.1 shows how example 6.4 (Stock Exchange Interaction) is proven to be interaction sound using existing tools.

6.3.2 Interaction Equivalence

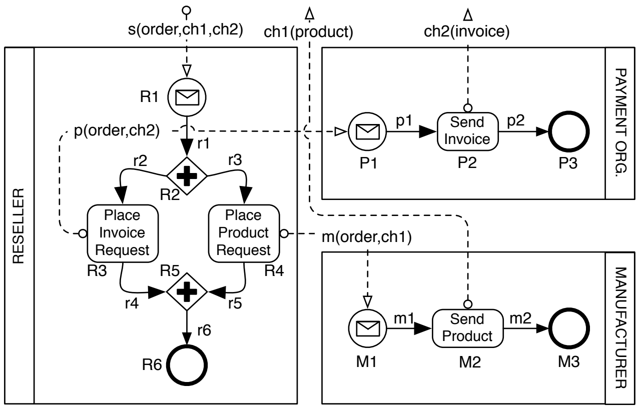
In this subsection we extend the idea behind interaction soundness. While interaction soundness investigates whether a service is able to interact properly with a given environment, i.e. both are compatible, we now abstract from certain services using an environment. Instead, we investigate if an environment behaves like another one, i.e. both are conforming. Two different directions can be distinguished. Either an environment A is able to behave like another environment B, meaning A simulates B, or arbitrary interactions of both environments can be mimicked in any direction by A or B. The former is denoted as *interaction simulation*, whereas the latter is denoted accordingly as *interaction equivalence*.

Example 6.5 (E-Business Solutions) Interaction equivalence is motivated by an example shown in figure 6.8. Contained are two different environments for the *Customer* process from figure 6.2. While the one depicted in figure 6.8(a) shows the original interaction partner, the one

[1] Edges refers to control flow edges here.



(a) Environment 1.



(b) Environment 2.

Figure 6.8: Two different environments for the customer process from figure 6.2.

from figure 6.8(b) shows a more advanced construction that should replace the first one. Instead of directly processing the customer's request by sending a product and an invoice, now a *Reseller* enters the scene. As the name suggests, the reseller only redirects the order to a *Manufacturer* selected from a set. Furthermore, the reseller supports different payment methods which are handled by different *Payment Organizations*. The reseller therefore also selects an applicable one.

Interaction equivalence deals with the replaceability of different environments in such a way, that any service using the different environments is not capable of detecting any differences regarding the interaction behavior. In a weaker scenario, interaction simulation enforces that an environment is able to behave like another one, but not necessarily the other way around. In case of the example this means, can we replace the *Shop* from figure 6.8(a) by the *Reseller* construct from figure 6.8(b) such that any service, representing arbitrary customers, is unable to detect a difference?

Interaction simulation and equivalence can only be given by taking the semantics of an environment into account. Hence, we have to consider environment agents. Since we already have a congruence for agents abstracting from all internal actions, the definition of interaction equivalence is straightforward. This time we have to use weak open bisimulation for supporting arbitrary in- and outgoing interaction edges, which might only be known at runtime due to dynamic binding:

Definition 6.8 (Interaction Equivalence) Two environment agents D_1 and D_2 are *interaction equivalent* if $D_1 \approx_O^D D_2$. $\square$

Regarding the definition of interaction simulation, we only have to consider one direction using weak open d-simulation:

Definition 6.9 (Interaction Simulation) An environment agent D_2 *simulates the interactions* of another environment agent D_1 if $D_1 \precapprox_O^D D_2$. $\square$

Appendix A.2.2 shows how the environments from figure 6.8 perform regarding interaction simulation and equivalence using existing tools.

Part III

Results

Introduction to Part III

Part III discusses the results of the investigations and concludes the thesis. It starts with an illustrating example that highlights how the formal models of data, processes, and interactions are able to provide a unified representation of the investigated areas of BPM. Thereafter, the investigations are recapitulated and advantages as well as disadvantages are discussed. Furthermore, the results are set into scene with related work. Finally, the thesis is concluded by a summary and an outlook on future work.

Structure of Part III Part III is composed of three chapters. The first chapter introduces the example. The second chapter recapitulates and discusses the investigations. The third chapter concludes the thesis by summing up and showing further work.

Chapter 7

Unification

This chapter discusses how data, processes, and interactions can be brought together in a unified way. The discussion is based on an example consisting of interacting business processes with data-based decisions. We showcase a simulation of the formalized system and discuss lazy soundness, interaction soundness, as well as interaction equivalence regarding the example. Special attention is paid on the representation of a service broker that supports dynamic registration and removal of services during runtime.

7.1 Formal Models

The initial view of the example is shown in figure 7.1. It describes a loan broker interaction consisting of three participants. The first participant is a *customer*, shown in the bottom pool. The pool contains an investment process. If the investment is below a certain threshold, it is made directly and the process finishes. If it is above the threshold, an interaction with other participants takes place. In the latter case, the activity *Find Bank* sends a request to a *loan broker*. The request of the customer consists of all relevant data to allow the loan broker to select a certain *bank* offering the lowest interest rate for the customer. The relation between the loan broker and the bank is represented by an association between their corresponding pools. It is assumed that the loan broker is always able to return a link to a certain bank. This is denoted via the interaction flow label $ch(bank)$ at the interaction flow between the nodes $B3$ and $C3$. The customer, in turn, is able to request the loan in the activity *Request Loan* by using the received link $bank$. Via this link, a request req and two links used for responses, acc for the successful grant of the loan and rej for the rejection, are sent. Afterward, the customer either receives an accept or reject message, leading to either the activity *Buy* or *Reject*. Thereafter, the process of the customer finishes.

Figure 7.1 only represents the static model of the system. However, in each instance of the system, several banks are available, which all should conform to the interaction behavior of the bank pool but are not initially bound to the customer. In the example, only the loan broker and the customer are initially connected, while different banks might register and remove their links from the loan broker during the evolution of the system. By investigating the example, it will be shown how data, processes, and interactions can be brought together in one unified formal

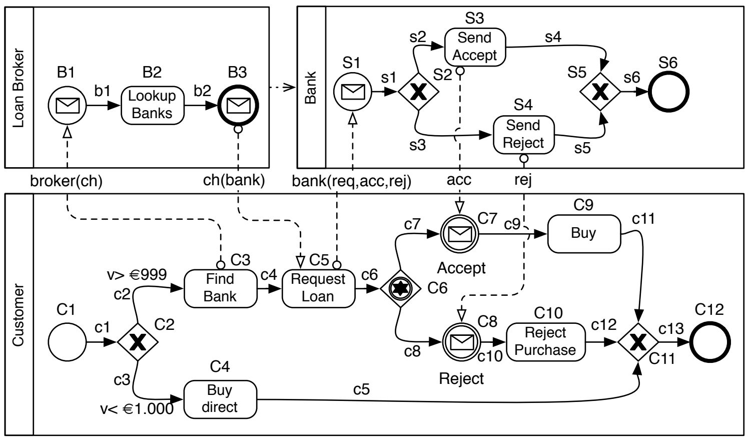


Figure 7.1: Loan broker interaction.

model.

7.1.1 The Customer

The formalization of the customer starts with a mapping of its business process diagram to a process graph according to example 5.1 (Partly Mapping of a BPD to a Process Graph).

Example 7.1 (Process Graph of the Customer) The process graph $P_C = (N, E, T, A)$ of the customer from figure 7.1 is given by:

1. $N = \{C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12\}$
2. $E = \{(C1, C2), (C2, C3), (C2, C4), (C3, C5), (C5, C6), (C6, C7), (C6, C8), (C7, C9), (C8, C10), (C9, C11), (C10, C11), (C11, C12)\}$
3. $T = \{(C1, StartEvent), (C2, XORGateway), (C3, Task), (C4, Task), (C5, Task), (C6, EventBasedGateway), (C7, IntermediateMessageEvent), (C8, IntermediateMessageEvent), (C9, Task), (C10, Task), (C11, XORGateway), (C12, EndEvent)\}$
4. $A = \emptyset$

Since the mapping algorithm does not consider data flow, the process graph is incomplete in a sense that the decision made in the node $C2$ (XOR Gateway) is not contained. To overcome this limitation, we provide a lightweight extension to a process graph that does not touch definition 5.1 (Process Graph). Data-based decisions made in a gateway are captured as attributes to the gateway node together with the corresponding edge. For instance, the attributes for example 7.1 (Process Graph of the Customer) are modified to:

$$A' = \{(C2, ((C2, C3), "v > 999")), (C2, ((C2, C4), "v < 1000"))\},$$

where we abstract from the currency operator for the sake of simplicity. Furthermore, we have to initialize the values used and define their types in the initial node. In case of the example this is $C1$:

$$A'' = A' \cup \{(C1, (variable, "v : number"))\} .$$

We use the key *variable* to denote external variables required for the process to operate. Where these variables are initialized from is out of scope for the formalization. For instance, they could be received during an earlier interaction with another process or read from a database.

While we worked around a modification of definition 5.1 (Process Graph), additional effort is required to describe the routing of data through a process. This is given by a *data flow graph*. A data flow graph does not only consider data used internally inside processes but also data received and transmitted during interactions with other processes. Hence, only by combining the concepts investigated in chapter 5 (Processes) and chapter 6 (Interactions) with chapter 4 (Data), a definition can be given:

Definition 7.1 (Data Flow Graph) A *data flow graph* is a three tuple consisting of nodes, directed data flow edges, and markings of the edges. Formally: $D = (N, E, M)$ with

- N as a finite, non-empty set of nodes according to a process graph,
- $E \subseteq (N \times N)$ as a set of directed data flow edges, and
- $M : E \rightarrow STRING$ as a function mapping directed data flow edges to markings given by text strings. □

Furthermore, each data flow graph D belongs to a certain process graph P. Hence, the nodes N of D correspond to nodes N of P. Different nodes of D can be connected by data flow edges, where we assume transitivity. The data types actually routed through the edges of D are given by M. M is a total function, so each data flow edges has exactly one marking. As a final constraint, data flow has to follow control flow, thus fulfilling the following requirement between a data flow graph D and a process graph P:

$$\forall (a, b) \in E \text{ of } D : \exists c, b \in N \text{ of } P \text{ such that } a = c \wedge b = d \wedge \exists \epsilon : c \xrightarrow{\epsilon} d . \quad (7.1)$$

The requirement states that for each two nodes a and b of a data flow graph that are connected by a data flow edge, two corresponding nodes c and d of a process graph exist. Furthermore, there has to exists a path between c and d inside the process graph.

To show the coherence between a data flow graph and a graphical notation, we give an example of how to map the relevant parts of a business process diagram to a data flow graph.

Example 7.2 (Partly Mapping of a BPD to a Data Flow Graph) A BPD with annotated message flows is mapped to a data flow graph $D = (N, E, M)$ by the following steps (sketch):

1. N is given by all flow objects of the BPD that produce or require data. We consider either:
   (a) Gateways with outgoing sequence flows with labels or
   (b) Flow objects with incoming or outgoing message flows.

2. E and M are given by constructing the set of data source nodes and data target nodes combined to tuples:
   (a) The set of data sources $\mathcal{S}$ is given by: Flow objects with incoming message flows (the source values are given by the data part of the message flow label), flow objects with outgoing message flows (the source values are given by a parts of the message flow label), and each start event (where the source values are given by all other kind of required data values).
   (b) The set of data targets $\mathcal{T}$ is given by: Flow objects with incoming message flows (where the channel part of the message flow label is the target value), flow objects with outgoing message flows (where all parts of the message flow label are the target value), and gateways which use the labels from outgoing sequence flows for routing decisions (where the labels construct the target values).
   (c) The elements of E are given by (s, t) with $s \in \mathcal{S}$ and $t \in \mathcal{T}$ where the source value matches the target value. During this step, also the set of M is extended by $((s, t), \mathit{sourcevalue})$ for each pair (s, t).

The mapping allows multiple targets for data source nodes as well as multiple sources for data target nodes. The mapping given is different to algorithm 6.1 (Mapping Interaction Graphs to Agents), since data values are created in the initial nodes instead in the node where they are required first. This is due to the generic approach of the mapping. We can now give the initial data flow graph for the process graph of example 7.1 (Process Graph of the Customer):

$$D_C(N, E, M) = (\{\mathit{C1}, \mathit{C2}\}, \{(\mathit{C1}, \mathit{C2})\}, \{((\mathit{C1}, \mathit{C2}), "v")\}\ .$$

The data flow graph D_C describes how the number v is routed from the initial activity to the first gateway. As stated, a complete data flow graph also considers data received and transmitted during interactions. Accordingly, the complete data flow graph of the example is given by:

Example 7.3 (Data Flow Graph of the Customer) The data flow graph $D_C = (N, E, M)$ of the customer from figure 7.1 is given by:

1. $N = \{\mathit{C1}, \mathit{C2}, \mathit{C3}, \mathit{C5}, \mathit{C7}, \mathit{C8}\}$
2. $E = \{(\mathit{C1}, \mathit{C2}), (\mathit{C3}, \mathit{C5}), (\mathit{C5}, \mathit{C7}), (\mathit{C5}, \mathit{C8})\}$
3. $M = \{((\mathit{C1}, \mathit{C2}), "v"), ((\mathit{C3}, \mathit{C5}), "\mathit{ch}"), ((\mathit{C5}, \mathit{C7}), "\mathit{acc}"), ((\mathit{C5}, \mathit{C8}), "\mathit{rej}")\}$

To derive the data flow graph, we used the knowledge from the labeled message flows of the BPD. These correspond to the interaction edges of the interaction graph that will be discussed later on. We can prove the consistency between the data flow and the process graph of the example:

Proof 7.1 (Consistency of the Data Flow and Process Graph of the Customer) Direct proof. To show the consistency between the data flow graph D_C from example 7.3 and the process graph P_C from example 7.1, we have to show the fulfillment of the requirement given in equation 7.1. Since D_C contains four data flow edges, we have to consider four cases for $\forall(a, b) \in E$ of D:

- Case 1: $a = \mathit{C1}, b = \mathit{C2} : c = \mathit{C1} \in N$ of $P, d = \mathit{C2} \in N$ of P. Since $a = c$, $b = d$, and $\langle \mathit{C1}, \mathit{C2} \rangle$ is a path found in P, the first case holds.
- Case 2: $a = \mathit{C3}, b = \mathit{C5} : c = \mathit{C3} \in N$ of $P, d = \mathit{C5} \in N$ of P. Since $a = c$, $b = d$, and $\langle \mathit{C3}, \mathit{C5} \rangle$ is a path found in P, the second case holds.
- Case 3: $a = \mathit{C5}, b = \mathit{C7} : c = \mathit{C5} \in N$ of $P, d = \mathit{C7} \in N$ of P. Since $a = c$, $b = d$, and $\langle \mathit{C5}, \mathit{C6}, \mathit{C7} \rangle$ is a path found in P, the third case holds.
- Case 4: $a = \mathit{C5}, b = \mathit{C8} : c = \mathit{C5} \in N$ of $P, d = \mathit{C8} \in N$ of P. Since $a = c$, $b = d$, and $\langle \mathit{C5}, \mathit{C6}, \mathit{C8} \rangle$ is a path found in P, the fourth case holds.

Since all cases hold, the consistency between D_C and P_C has been proved. □

To derive the formal model of the customer based on the process and data flow graph, we need to extend algorithm 5.1 (Mapping Process Graphs to Agents) to support data flow graphs. This algorithm also describes how to process data received or produced during an interaction as given by algorithm 6.1 (Mapping Interaction Graphs to Agents).

Algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents) A process graph $P = (P_N, P_E, P_T, P_A)$ with a corresponding data flow graph $D = (D_N, D_E, D_M)$ is mapped to a π-calculus agent N as follows:

1. Map the process graph to agents as given by algorithm 5.1 (Mapping Process Graphs to Agents). Use α-conversion, if required, to ensure that $bn(N) \cap D_M(e) = \emptyset$ for each $e \in D_E$.
2. Find the corresponding path ϵ in P for each data flow edge d of D. Extend the objects of the names representing pre- and postconditions in each agent corresponding to a node of ϵ with the marking of the data flow edge d.
3. For each node n of P with $A(n) = (\mathit{variable}, *)$ find the corresponding agent and restrict the name contained as the value before it. □

We can apply this algorithm to the process and data flow graph of the customer from figure 7.1.

Example 7.4 (Agent Formalization of the Customer) The process graph from example 7.1 (Process Graph of the Customer) and the data flow graph from example 7.3 (Data Flow Graph of the Customer) are mapped to π-calculus agents according to algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents). We start by formalizing the process graph, where we omit recursive definitions since the process graph is acyclic:

$$C \stackrel{def}{=} (\nu c1, \ldots, c13\,) \prod_{i=1}^{12} Ci\ .$$

The node $\mathit{C1}$ is a start event placed inside a sequence pattern with only a postcondition given by:

$$\mathit{C1} \stackrel{def}{=} \langle\cdot\rangle.\overline{c1}.\mathbf{0}\ .$$

Node $C2$ is more complex because it represents a data-based exclusive choice according to the patterns 4.27 (Data-based Routing) and 5.4 (Exclusive Choice). The pattern is implemented by taking the attributes of A' into account:

$$C2 \overset{def}{=} c1.\langle\cdot\rangle.\text{if } v < 1000 \text{ then } \overline{c2}.\mathbf{0} \text{ else } \overline{c3}.\mathbf{0}\ .$$

Note that we already optimized the conditions for an if then else statement, since both are mutually exclusive without a gap. The nodes of the type task are given by:

$$C3 \overset{def}{=} c2.\langle\cdot\rangle.\overline{c4}.\mathbf{0}\ ,\ \ C4 \overset{def}{=} c3.\langle\cdot\rangle.\overline{c5}.\mathbf{0}\ ,\ \ C5 \overset{def}{=} c4.\langle\cdot\rangle.\overline{c6}.\mathbf{0}\ ,$$

$$C9 \overset{def}{=} c9.\langle\cdot\rangle.\overline{c11}.\mathbf{0} \text{ and } C10 \overset{def}{=} c10.\langle\cdot\rangle.\overline{c12}.\mathbf{0}\ .$$

The nodes $C6$, $C7$, and $C8$ represent an event-based gateway matching to pattern 5.17 (Deferred Choice):

$$C6 \overset{def}{=} c6.\langle\cdot\rangle.(acc.\overline{c7}.\mathbf{0} + rej.\overline{c8}.\mathbf{0})\ ,\ \ C7 \overset{def}{=} c7.\langle\cdot\rangle.\overline{c9}.\mathbf{0} \text{ and } c8.\langle\cdot\rangle.\overline{c10}.\mathbf{0}\ .$$

The environment triggers acc and rej are already contained in agent $C6$, since otherwise a deterministic decision would not be possible. The nodes $C11$ of the type gateway and $C12$ of the type end event are implemented by:

$$C11 \overset{def}{=} c5.\langle\cdot\rangle.\overline{c13}.\mathbf{0} + c11.\langle\cdot\rangle.\overline{c13}.\mathbf{0} + c12.\langle\cdot\rangle.\overline{c13}.\mathbf{0} \text{ and } C12 \overset{def}{=} c13.\langle\cdot\rangle.\mathbf{0}\ .$$

We can now apply algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents) to the data flow graph of the *Customer*. The first step of the algorithm has already been shown by the mapping of the process graph to agents. In the second step we have to find the corresponding paths in the process graph to extend the names representing pre- and postconditions. The first element of the data flow edges set of the data flow graph is $(C1, C2)$, hence the agent extension is straightforward:

$$C1 \overset{def}{=} \nu v : number\ \langle\cdot\rangle.\overline{c1}\langle v\rangle.\mathbf{0} \text{ and } C2 \overset{def}{=} c1(v).\langle\cdot\rangle.\text{if } v < 1000 \text{ then } \overline{c2}.\mathbf{0} \text{ else } \overline{c3}.\mathbf{0}\ .$$

We already applied step three of the algorithm, since the attribute key of the node $C1$ matches $variable$. The path between the nodes $C3$ and $C5$ of the data flow edge $(C3, C5)$ is again a single edge, leading to the definition of

$$C3 \overset{def}{=} c2.\langle\cdot\rangle.\overline{c4}\langle ch\rangle.\mathbf{0}\ ,\quad \text{and } C5 \overset{def}{=} c4(ch).\langle\cdot\rangle.\overline{c6}.\mathbf{0}$$

The corresponding paths for the data flow edges $(C5, C7)$ and $(C5, C8)$ are $\epsilon_1 = \langle C5, C6, C7\rangle$ and $\epsilon_2 = \langle C5, C6, C8\rangle$. The corresponding extensions of the agents for ϵ_1 are given by:

$$C5 \overset{def}{=} c4(ch).\langle\cdot\rangle.\overline{c6}\langle acc\rangle.\mathbf{0}\ ,\ \ C6 \overset{def}{=} c6(acc).\langle\cdot\rangle.(acc.\overline{c7}\langle acc\rangle.\mathbf{0} + rej.\overline{c8}.\mathbf{0})\ ,$$

and

$$C7 \overset{def}{=} c7(acc).\langle\cdot\rangle.\overline{c9}.\mathbf{0}\ ,$$

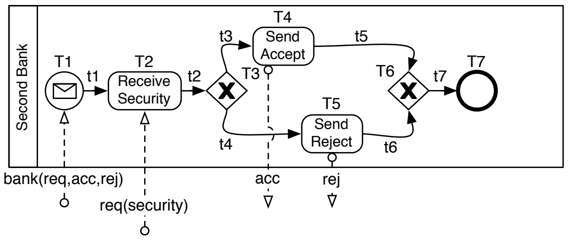


Figure 7.2: A second bank for the loan broker interaction.

where acc is technically not required in $C7$, since it is already implemented in $C6$. However, we kept it for completeness. The extensions for ϵ_2 conclude the mapping:

$$C5 \overset{def}{=} c4(ch).\langle\cdot\rangle.\overline{c6}\langle acc, rej\rangle.\mathbf{0}\ ,\ \ C6 \overset{def}{=} c6(acc, rej).\langle\cdot\rangle.(acc.\overline{c7}\langle acc\rangle.\mathbf{0} + rej.\overline{c8}\langle rej\rangle.\mathbf{0})\ ,$$

and

$$C8 \overset{def}{=} c8(rej).\langle\cdot\rangle.\overline{c10}.\mathbf{0}\ ,$$

As can be seen from the example, data flow edges with the same sub-path in the process graph can add additional values as the objects of the names used as pre- and postconditions.

7.1.2 The Bank

The formalization of the bank starts with a mapping of its business process diagram to a process graph:

Example 7.5 (Process Graph of the Bank) The process graph $P_B = (N, E, T, A)$ of the bank from figure 7.1 is given by:

1. $N = \{S1, S2, S3, S4, S5, S6\}$
2. $E = \{(S1, S2), (S2, S3), (S2, S4), (S3, S5), (S4, S5), (S5, S6)\}$
3. $T = \{(S1, MessageStartEvent), (S2, XORGateway), (S3, Task), (S4, Task),$ $(S5, XORGateway), (S6, EndEvent)\}$
4. $A = \emptyset$

To make the reasoning later on more interesting, we add another bank with a different interaction behavior. The difference is established by requiring an additional security right after the first request. To distinguish both later on, we talk about the bank, or first bank, for the previous given one and about the second bank for the one introduced below.

Example 7.6 (Process Graph of the Second Bank) The process graph $P_{B2} = (N, E, T, A)$ of the *Second Bank* from figure 7.2 is given by:

1. $N = \{T1, T2, T3, T4, T5, T6, T7\}$

2. $E = \{(T1, T2), (T2, T3), (T3, T4), (T3, T5), (T4, T6), (T5, T6), (T6, T7)\}$

3. $T = \{(T1, MessageStartEvent), (T2, Task)(T3, XORGateway), (T4, Task), (T5, Task), (T6, XORGateway), (T7, EndEvent)\}$

4. $A = \emptyset$

Since the processes of the banks are abstract, we do not know how their internal routing decision is calculated. Hence, we cannot provide a data-based gateway and have to assume a non-deterministic choice to cover all cases. Nevertheless, we can still provide the data flow graphs according to example 7.2 (Partly Mapping of a BPD to a Data Flow Graph) for the message flows of the BPD.

Example 7.7 (Data Flow Graph of the First Bank) The data flow graph $D_B = (N, E, M)$ of the bank from figure 7.1 is given by:

1. $N = \{S1, S3, S4\}$

2. $E = \{(S1, S3), (S1, S4)\}$

3. $M = \{((S1, S3), ”acc”), ((S1, S4), ”rej”)\}$

The data flow graph for the process graph from the second bank is given accordingly, since we do not know where the security influences the decision making:

Example 7.8 (Data Flow Graph of the Second Bank) The data flow graph $D_{B2} = (N, E, M)$ of the second bank from figure 7.2 is given by:

1. $N = \{T1, T4, T5\}$

2. $E = \{(T1, T4), (T1, T5), (T1, T2)\}$

3. $M = \{((T1, T4), ”acc”), ((T1, T5), ”rej”), ((T1, T2), ”req”)\}$

We can prove the consistency of the data flow and process graph of the *First Bank*:

Proof 7.2 (Consistency of the Data Flow and Process Graph of the First Bank) Direct proof. To show the consistency between the data flow graph D_B from example 7.7 and the process graph P_B from example 7.5, we have to show the fulfillment of the requirements given in equation 7.1. Since D_B contains two data flow edges, we have to consider two cases for $\forall(a, b) \in E$ of D_B:

- Case 1: $a = S1, b = S3 : c = S1 \in N$ of $P_D, d = S3 \in N$ of P_D. Since $a = c, b = d$, and $\langle S1, S2, S3\rangle$ is a path found in P_B, the first case holds.

- Case 2: $a = S1, b = S4 : c = S1 \in N$ of $P_D, d = S4 \in N$ of P_D. Since $a = c, b = d$, and $\langle S1, S2, S4\rangle$ is a path found in P_B, the second case holds.

Since all cases hold, the consistency between D_B and P_D has been proved. $\square$

We omit the proof for the second bank and continue with the agent formalization of the banks.

Example 7.9 (Agent Formalization of the First Bank) The process graph from example 7.5 (Process Graph of the Bank) and the data flow graph from example 7.7 (Data Flow Graph of the First Bank) are mapped to π-calculus agents according to algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents). We start by formalizing the process graph, where we omit recursive definitions since the process graph is acyclic:

$$S \overset{def}{=} (\nu s1, \ldots, s6\,) \prod_{i=1}^{6} Si\ .$$

The nodes $S1$, $S3$, $S4$, and $S6$ are placed inside sequence patterns:

$$S1 \overset{def}{=} \langle\cdot\rangle.\overline{s1}.\mathbf{0}\ ,\ S3 \overset{def}{=} s2.\langle\cdot\rangle.\overline{s4}.\mathbf{0}\ ,\ S4 \overset{def}{=} s3.\langle\cdot\rangle.\overline{s5}.\mathbf{0} \text{ and } S6 \overset{def}{=} s6.\langle\cdot\rangle.\mathbf{0}\ .$$

Node $S2$ represents an exclusive choice according to pattern 5.4 (Exclusive Choice) and $S5$ represents a merge according to pattern 5.5 (Simple Merge):

$$S2 \overset{def}{=} s1.\langle\cdot\rangle.(\overline{s2}.\mathbf{0} + \overline{s3}.\mathbf{0}) \text{ and } S5 \overset{def}{=} s4.\langle\cdot\rangle.\overline{s6}.\mathbf{0} + s5.\langle\cdot\rangle.\overline{s6}.\mathbf{0}\ .$$

The enhancements for data flow touch the agents $S1$, $S2$, $S3$, and $S4$:

$$S1 \overset{def}{=} \langle\cdot\rangle.\overline{s1}\langle acc, rej\rangle.\mathbf{0}\ ,\ S2 \overset{def}{=} s1(acc, rej).\langle\cdot\rangle.(\overline{s2}\langle acc\rangle.\mathbf{0} + \overline{s3}\langle rej\rangle.\mathbf{0})\ ,$$

$$S3 \overset{def}{=} s2(acc).\langle\cdot\rangle.\overline{s4}.\mathbf{0} \text{ and } S4 \overset{def}{=} s3(rej).\langle\cdot\rangle.\overline{s5}.\mathbf{0}\ .$$

Example 7.10 (Agent Formalization of the Second Bank) The process graph from example 7.6 (Process Graph of the Second Bank) and the data flow graph from example 7.8 (Data Flow Graph of the Second Bank) are mapped to π-calculus agents according to algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents). We start by formalizing the process graph, where we omit recursive definitions since the process graph is acyclic:

$$T \overset{def}{=} (\nu t1, \ldots, t7\,) \prod_{i=1}^{7} Ti\ .$$

The nodes $T1$, $T2$, $T4$, $T5$, and $T7$ are placed inside sequence patterns:

$$T1 \overset{def}{=} \langle\cdot\rangle.\overline{t1}.\mathbf{0}\ ,\ T2 \overset{def}{=} t1.\langle\cdot\rangle.\overline{t2}.\mathbf{0}\ ,\ T4 \overset{def}{=} t3.\langle\cdot\rangle.\overline{t5}.\mathbf{0}\ ,\ T5 \overset{def}{=} t4.\langle\cdot\rangle.\overline{t6}.\mathbf{0}$$

$$\text{and } T7 \overset{def}{=} t7.\langle\cdot\rangle.\mathbf{0}\ .$$

Node $T3$ represents an exclusive choice according to pattern 5.4 (Exclusive Choice) and $T6$ represents a merge according to pattern 5.5 (Simple Merge):

$$T3 \overset{def}{=} t2.\langle\cdot\rangle.(\overline{t3}.\mathbf{0} + \overline{t4}.\mathbf{0}) \text{ and } T6 \overset{def}{=} t5.\langle\cdot\rangle.\overline{t7}.\mathbf{0} + t6.\langle\cdot\rangle.\overline{t7}.\mathbf{0}\ .$$

The enhancements for data flow touch the agents $T1$, $T2$, $T3$, and $T4$, and $T5$:

$$T1 \overset{def}{=} \langle\cdot\rangle.\overline{t1}\langle acc, rej, req\rangle.\mathbf{0}\ ,\ T2 \overset{def}{=} t1(acc, rej, req).\langle\cdot\rangle.\overline{t2}\langle acc, rej\rangle\ ,$$

$$T3 \overset{def}{=} t2(acc, rej).\langle\cdot\rangle.(\overline{t3}\langle acc\rangle.\mathbf{0} + \overline{t4}\langle rej\rangle.\mathbf{0})\ ,\ T4 \overset{def}{=} t3(acc).\langle\cdot\rangle.\overline{t5}.\mathbf{0} \text{ and}$$

$$T5 \overset{def}{=} t4(rej).\langle\cdot\rangle.\overline{t6}.\mathbf{0}\ .$$

7.1.3 The Broker

We start again by showing how the broker's business process is mapped to a process graph.

Example 7.11 (Process Graph of the Loan Broker) The process graph $P_{LB} = (N, E, T, A)$ of the loan broker from figure 7.1 is given by:

1. $N = \{B1, B2, B3\}$
2. $E = \{(B1, B2), (B2, B3)\}$
3. $T = \{(B1, MessageStartEvent), (B2, Task), (S3, MessageEndEvent)\}$
4. $A = \emptyset$

Example 7.12 (Data Flow Graph of the Loan Broker) The data flow graph $D_{LB} = (N, E, M)$ of the loan broker from figure 7.1 is given by:

1. $N = \{B1, B2, B3\}$
2. $E = \{(B1, B3)\}$
3. $M = \{((B1, B3), "ch")\}$

We omit the proof of the consistency of the data flow and process graph for the loan broker and continue with the agent formalization:

Example 7.13 (Agent Formalization of the Loan Broker) The process graph from example 7.11 (Process Graph of the Loan Broker) and the data flow graph from example 7.12 (Data Flow Graph of the Loan Broker) are mapped to π-calculus agents according to algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents). We start by formalizing the process graph, where we omit recursive definitions since the process graph is acyclic:

$$B \overset{def}{=} (\nu b1, b2)\prod_{i=1}^{3} Bi\ .$$

All three nodes of the process graph are placed inside sequence patterns:

$$B1 \stackrel{def}{=} \langle\cdot\rangle.\overline{b1}.\mathbf{0}\ ,\ B2 \stackrel{def}{=} b1.\langle\cdot\rangle.\overline{b2}.\mathbf{0} \text{ and } B3 \stackrel{def}{=} b2.\langle\cdot\rangle.\mathbf{0}\ .$$

The enhancements for data flow are given by:

$$B1 \stackrel{def}{=} \langle\cdot\rangle.\overline{b1}\langle ch\rangle.\mathbf{0}\ ,\ B2 \stackrel{def}{=} b1(ch).\langle\cdot\rangle.\overline{b2}\langle ch\rangle.\mathbf{0} \text{ and } B3 \stackrel{def}{=} b2(ch).\langle\cdot\rangle.\mathbf{0}\ .$$

7.1.4 The Loan Broker Interaction

After having defined the process graphs and corresponding agents of the different participants of the example, we can specify the complete interaction. We start by deriving the interaction graph:

Example 7.14 (Interaction Graph of the Loan Broker Interaction) The interaction graph $IG = (PS, C, L)$ of the example shown in figure 7.1 is given according to example 6.1 (Partly Mapping of a BPD to an Interaction Graph) by:

- $PS = \{P_C, P_B, P_{LB}\}$
- $C = \{(C3, B1), (B3, C5), (C5, S1), (S3, C7), (S4, C8)\}$
- $L = \{((C3, B1), broker(ch)), ((B3, C5), ch(bank)), ((C5, S1), bank(req, acc, rej)),$ $((S3, C7), acc), ((S4, C8), rej)\}$

We furthermore need to distinguish a subset of the interaction graph given by the service graph of the customer:

Example 7.15 (Service Graph of the Customer) The service graph $SG = (PS, C, L)$ of the customer from figure 7.1 is given by:

- $PS = P_C$
- $C = \{(C3, \perp), (\perp, C5), (\perp, S1), (S3, \perp), (S4, \perp)\}$
- $L = \{((C3, \perp), broker(ch)), ((B3, \perp), ch(bank)), ((\perp, S1), bank(req, acc, rej)),$ $((S3, \perp), acc), ((S4, \perp), rej)\}$

The interaction graph is used to modify the agent representations of the customer, the first bank, and the loan broker according to their interaction schemas. The service graph is used for reasoning later on.

Example 7.16 (Agent Formalization of the Loan Broker Interaction) The interaction graph from example 7.14 (Interaction Graph of the Loan Broker Interaction) is mapped to π-calculus agents according to algorithm 6.1 (Mapping Interaction Graphs to Agents). For the first step (mapping the process graphs contained in PS), we revert to the examples 7.4 (Agents of the Customer), 7.9 (Agents of the First Bank), and 7.13 (Agents of the Loan Broker).

In the second step, we iterate over all nodes of all process graphs from IG that are the target of an interaction edge. The π-calculus representation of these nodes is modified as follows:

$$B1 \overset{def}{=} broker(ch).\langle\cdot\rangle.\overline{b1}\langle ch\rangle.\mathbf{0}\ ,\ \ C5 \overset{def}{=} c4(ch).ch(bank).\langle\cdot\rangle.\overline{c6}\langle acc, rej\rangle.\mathbf{0}\ \text{ and}$$

$$S1 \overset{def}{=} bank(req, acc, rej).\langle\cdot\rangle.\overline{s1}\langle acc, rej\rangle.\mathbf{0}\ .$$

Note that the deferred choice construct of the customer needs a specific processing, since the incoming interaction flows are technically already required at $C6$ instead of $C7$ and $C8$ (see pattern 5.17). Since no objects are contained in the interaction flow edges $(S3, C7)$ and $(S4, C8)$, the mapping of the process graph to agents is already sufficient. The second sub-step (passing received objects to all further nodes) has already been done with the help of the data flow graphs.

In the third step, we iterate over all nodes of all process graphs from IG that are the source of an interaction edge. The π-calculus representation of these nodes is modified as follows:

$$C3 \overset{def}{=} \nu ch\ c2.\langle\cdot\rangle.\overline{broker}\langle ch\rangle.\overline{c4}\langle ch\rangle.\mathbf{0}\ ,$$

$$C5 \overset{def}{=} \nu req\ \nu acc\ \nu rej\ c4(ch).ch(bank).\langle\cdot\rangle.\overline{bank}\langle req, acc, rej\rangle.\overline{c6}\langle acc, rej\rangle.\mathbf{0}\ ,$$

$$S3 \overset{def}{=} s2(acc).\langle\cdot\rangle.\overline{acc}.\overline{s4}.\mathbf{0}\ \text{ and }\ S4 \overset{def}{=} s3(rej).\langle\cdot\rangle.\overline{rej}.\overline{s5}.\mathbf{0}\ .$$

The global agent I that represents the interaction graph IG of the example is given by:

$$I \overset{def}{=} \nu broker\ (C \mid S \mid B)\ .$$

We restricted the name *broker* inside the interaction, because it is the only channel already known at design time. The binding between the banks and the loan broker is discussed later on.

Up to now, we have only considered the concepts introduced in chapter 6 (Interactions) regarding the interaction between customer, first bank, and loan broker. Regarding practical feasibility, one concept left out so far is the replication of process graphs acting as services. The provided formalizations only allow one time evolution and cannot process requests at the same time. This limitation can be overcome by modifying the agent representation of a process graph with data flow and interaction edges to a service agent:

Algorithm 7.2 (Deriving Service Agents) A π-calculus mapping D of a process graph and data flow graph according to algorithm 7.1 (Mapping Process Graphs with Data Flow to Agents), contained inside a mapping of an interaction graph to agents according to algorithm 6.1 (Mapping Interaction Graphs to Agents) is enhanced to provide (1) multiple executions, and (2) parallel processing of different requests as follows:

1. The input prefix of the agent mapping representing the first node of the process graph is moved before the global agent representing the complete process graph.

2. Sequentially after the moved prefix follows the original definition of the global agent as well as in parallel a statement of the global agent identifier.

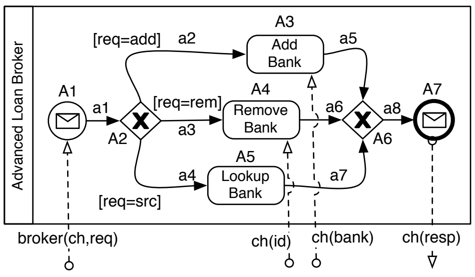


Figure 7.3: An advanced loan broker.

The algorithm is only applicable to structural sound process graphs contained inside an interaction graph, where there exists an incoming interaction edge for the first node of the process graph. Formally, $\exists n \in N$ with N being the set of nodes of all process graphs : $type(n) = MessageStartEvent \wedge \exists e \in E$ of IG : $target(e) = n$. □

According to this algorithm, the agent formalization of the loan broker can be enhanced to support multiple executions and parallel processing of requests:

$$B \overset{def}{=} broker(ch).(((\nu b1, b2)\prod_{i=1}^{3} Bi) \mid B) \text{ with } B1 \overset{def}{=} \langle\cdot\rangle.\overline{b1}\langle ch\rangle.\mathbf{0}\ .$$

The modification of the first bank is given by:

$$S \overset{def}{=} bank(req, acc, rej).(((\nu s1, \ldots, s6)\prod_{i=1}^{6} Si) \mid S) \text{ with } S1 \overset{def}{=} \langle\cdot\rangle.\overline{s1}\langle acc, rej\rangle.\mathbf{0}\ .$$

The second ban is enhanced accordingly by changing agent T (omitted). The formalization of the loan broker interaction is concluded by an extension of the loan broker with capabilities for dynamic registration and de-registration of different banks.

Example 7.17 (Loan Broker Extension—First Variant) The loan broker is extended with dynamic registrations capabilities as given by figure 7.3. The figure shows that the *advanced loan broker* is able to execute three different activities depending on the *req* value. All activities work on a *banklist* data structure that is globally available through all instances of the advanced loan broker. The *Add Bank* activity allows the addition of news banks during runtime, whereas the *Remove Bank* activity allows the removal of already registered banks. Both activities modify the *banklist* data structure, whereas the *Lookup Bank* activity only iterates over the structure. Since the advanced loan broker touches the functional perspective it cannot be mapped by the algorithms provided. Instead we provide a manual construction of the matching agents to show how (1) a broker can be represented formally, and (2) give an example of how the functional perspective can be represented formally using the π-calculus.

We start by denoting the service agent of the advanced loan broker, where we add a globally available list according to pattern 4.6 (Business Process Management System Data). Instead of a BPMS we use the service agent as environment.

$$AB(add, rem, src, ok) \overset{def}{=} list(l_{add}, l_{rem}, l_{it}).AB_1$$

and

$$AB_1 \overset{def}{=} broker(ch, req).(((\nu a1, \ldots, a8)\prod_{i=1}^{7} Ai) \mid AB_1)\ .$$

The parameters app, rem, and src of AB provide constants used for routing the control flow inside the process based on req. Ok is used to commit the removal of a bank. The agent $A1$ starts the processing of a new request:

$$A1 \overset{def}{=} \langle\cdot\rangle.\overline{a1}\langle ch, req\rangle.\mathbf{0}\ ,$$

and $A2$ routes it corresponding to the value of req:

$$\begin{aligned} A2 \overset{def}{=} & a1(ch, req).\langle\cdot\rangle. \\ & ([req = add]\overline{a2}\langle ch\rangle.\mathbf{0} + [req = rem]\overline{a3}\langle ch\rangle.\mathbf{0} + [req = src]\overline{a4}\langle ch\rangle.\mathbf{0})\ . \end{aligned}$$

The agent $A3$ is capable of dynamically register new banks, where the id inside the list is forwarded as a reference used for removing the registered bank later on:

$$A3 \overset{def}{=} \nu r\ a2(ch).ch(bank).\overline{l_{add}}\langle bank, r\rangle.r(id).\langle\cdot\rangle.\overline{a5}\langle ch, id\rangle.\mathbf{0}\ .$$

Note that the formalization as given allows the same bank to register multiple times. A bank is removed from the list in agent $A4$ by using the id returned after registration:

$$A4 \overset{def}{=} a3(ch).ch(id).\overline{l_{rem}}\langle id\rangle.\langle\cdot\rangle.\overline{a6}\langle ch, ok\rangle.\mathbf{0}$$

The search operation is contained inside agent $A5$, that non-deterministically returns a bank from the list. If currently no bank is available, the operation is blocked until a bank registers:

$$A5 \overset{def}{=} a4(ch).l_{it}(i, e).i(id, value).A5_1(value)$$

with

$$A5_1(value) \overset{def}{=} i(id, nextvalue).(A5_1(nextvalue) + A5_1(value)) + e.\langle\cdot\rangle.\overline{a7}\langle ch, value\rangle.\mathbf{0}\ .$$

The sequence flows are joined in agent $A6$:

$$\begin{aligned} A6 \overset{def}{=} & a5(resp, ch).\langle\cdot\rangle.\overline{a8}\langle resp, ch\rangle.\mathbf{0} + a6(resp, ch).\langle\cdot\rangle.\overline{a8}\langle resp, ch\rangle.\mathbf{0} + \\ & a7(resp, ch).\langle\cdot\rangle.\overline{a8}\langle resp, ch\rangle.\mathbf{0}\ . \end{aligned}$$

Finally, agent $A7$ returns the result:

$$A7 \overset{def}{=} a8(resp, ch).\langle\cdot\rangle.\overline{ch}\langle resp\rangle.\mathbf{0}\ .$$

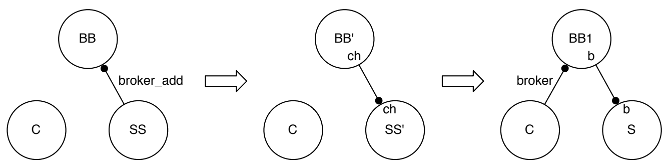


Figure 7.4: Initialization (first bank registers at the loan broker).

The formalization of the advanced loan broker can be extended further. However, we omit the discussion since the main concepts should have become clear by now. Instead, we provide a second variant of the loan broker extension that is not mapped from a process graph but instead direct encoded in π-calculus expressions. Due to the simplification, we are able to drop lists and provide a representation applicable for reasoning.

Example 7.18 (Loan Broker Extension—Second Variant) The second variant of the *Loan Broker* supports dynamic registration capabilities via a name $broker\_add$. At any time, a non-deterministic selection of a registered bank can be received from the loan broker via a name $broker$. Banks can remove their registration from the loan broker by interacting via a restricted name they receive after the interaction via $broker\_add$. The second variant of the loan broker is given by:

$$BB \overset{def}{=} broker\_add(name, ch).((\nu rem\ \overline{ch}\langle rem\rangle.BB1) \mid BB)$$

with

$$BB1 \overset{def}{=} broker(ch).(\overline{ch}\langle name\rangle.\mathbf{0} \mid BB1) + rem.\mathbf{0}\ .$$

In contrast to the derived agents of the first example, the second solution has a lower computational effort.

Before continuing with simulation in the next chapter, we provide the first and the second bank with a capability to register themselves at the second variant of the loan broker:

$$SS \overset{def}{=} \nu b\, \nu ch\ \overline{broker\_add}\langle b, ch\rangle.ch(rem).S\ .$$

$$TT \overset{def}{=} \nu b\, \nu ch\ \overline{broker\_add}\langle b, ch\rangle.ch(rem).T\ .$$

7.2 Simulation

In this subsection, we show excerpts of how the formalized system of the loan broker interaction can be simulated in a graphical manner using flow graphs. Therefore we assume all functional abstractions of the agent definitions to be filled with τ to abstract from the functional perspective of the different nodes that they represent. The initial state of the system is given by

$$I \overset{def}{=} \nu broker\ \nu broker\_add\ (BB \mid SS \mid C)$$

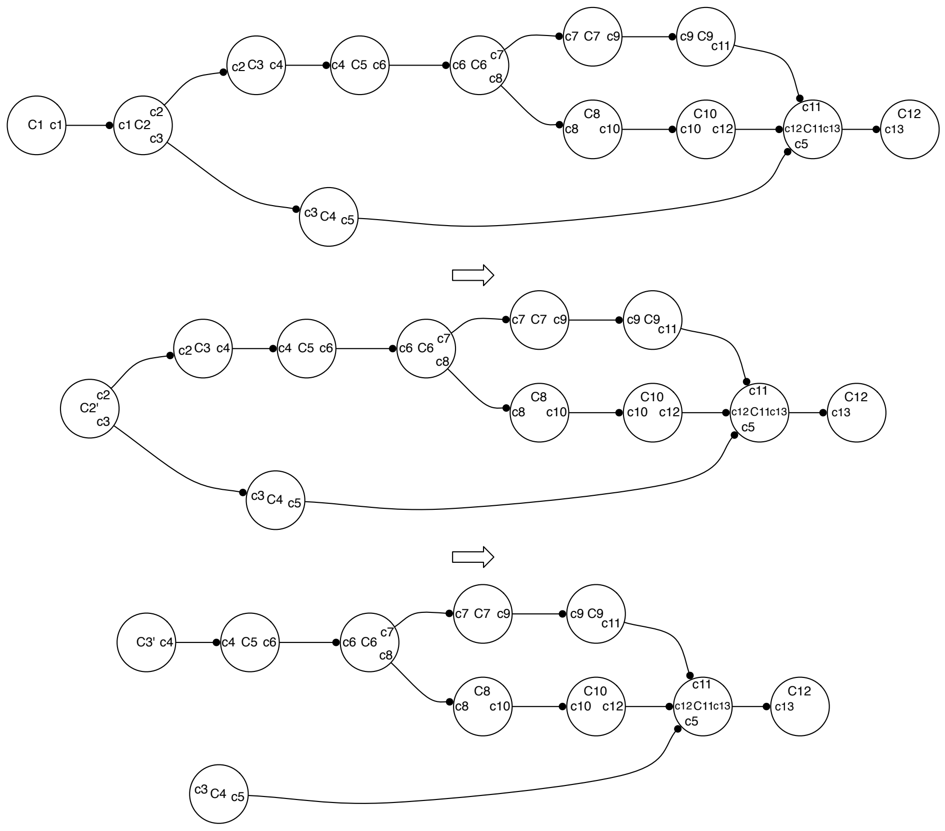


Figure 7.5: Evolution of the customer.

and depicted at the left hand side of figure 7.4. The system evolves as represented in the center of the figure by an interaction between SS and BB via $broker\_add$. In this step, the first bank registers itself at the loan broker:

$$I \xrightarrow{\tau} I' = \nu broker\ \nu broker\_add\ (BB' \mid SS' \mid C)\ .$$

The components evolve to

$$BB' = (\nu rem\ \overline{ch}\langle rem\rangle.BB1) \mid BB \text{ and } SS' \overset{def}{=} \nu bank\ \nu ch\ ch(rem).S\ ,$$

where we omit BB in the flow graph. Afterwards, an interaction between BB' and SS' takes place, where a restricted name rem is transmitted from BB' to SS' via ch. The name rem can be used by the first bank to remove its registration from the loan broker. However, this is not required for this simulation. The next state of the system is shown at the right hand side of figure 7.4. Since all agents contained have been given beforehand, we omit their duplicate definitions.

The first evolutions of the customer agents are shown in figure 7.5. We assume to make a purchase above the threshold limit, such that agent $C2$ decides to emit the name $c2$ used to

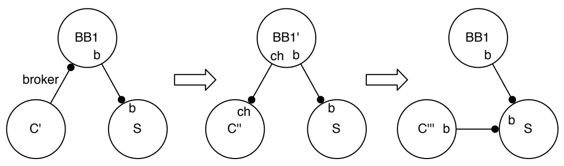


Figure 7.6: Discovery and dynamic binding in the example.

trigger agent $C3$. This agent corresponds to the *Find Bank* activity of the customer. In this activity, the loan broker is asked for an available bank. The agent $C3'$ is given by:

$$C3' = \nu ch\ \overline{broker}\langle ch\rangle.\overline{c4}\langle ch\rangle.\mathbf{0}\ .$$

Right now, an interaction between the customer, given by C' and the loan broker given by $BB1$ is possible. C' is given by the evolved state of C:

$$C' = (\nu c3, \ldots, c13\ )((\prod_{i=4}^{12} Ci) \mid C3')\ .$$

In this interaction, the customer acquires the link to a bank that is currently registered at the loan broker. This evolution is depicted at the left hand side of figure 7.6. The evolved agents $BB1'$ and C'' are given by

$$BB1' = \overline{ch}\langle name\rangle.\mathbf{0} \mid BB1 \ \text{ and } \ C'' = (\nu c3, c5, \ldots, c13\ )((\prod_{i=6}^{12} Ci) \mid C4 \mid C5')\ ,$$

with

$$C5' = \nu req, acc, rej\ \ ch(b).\tau.\bar{b}\langle req, acc, rej\rangle.\overline{c6}\langle acc, rej\rangle.\mathbf{0}\ .$$

We α-converted the name *bank* from the original definition of $C5$ to b for a shorter representation in the figure. In the evolution shown at the right hand side of figure 7.6, the dynamic binding between the customer and the bank received from the loan broker takes places via ch. Since $BB1'$ evolves with $\xrightarrow{\overline{ch}\langle name\rangle}$ to $BB1$ again, only the evolved structure of C''' has to be considered:

$$C'' \xrightarrow{ch(b)} C''' = (\nu c3, c5, \ldots, c13\ )((\prod_{i=6}^{12} Ci) \mid C4 \mid C5'')\ ,$$

with

$$C5'' = \nu req\ \nu acc\ \nu rej\ \ \tau.\bar{b}\langle req, acc, rej\rangle.\overline{c6}\langle acc, rej\rangle.\mathbf{0}\ .$$

The evolution of the interactions between the customer and the first bank is shown in figure 7.7. We added rectangles to the flow graphs for denoting the participants where the agent definitions belong to. In the upper part of the figure, an interaction between the customer and the first bank is possible. In this interaction three restricted names *req*, *acc*, and *rej* are transmitted via b. The system evolves in several steps to the flow graph shown in the lower part of the figure. As

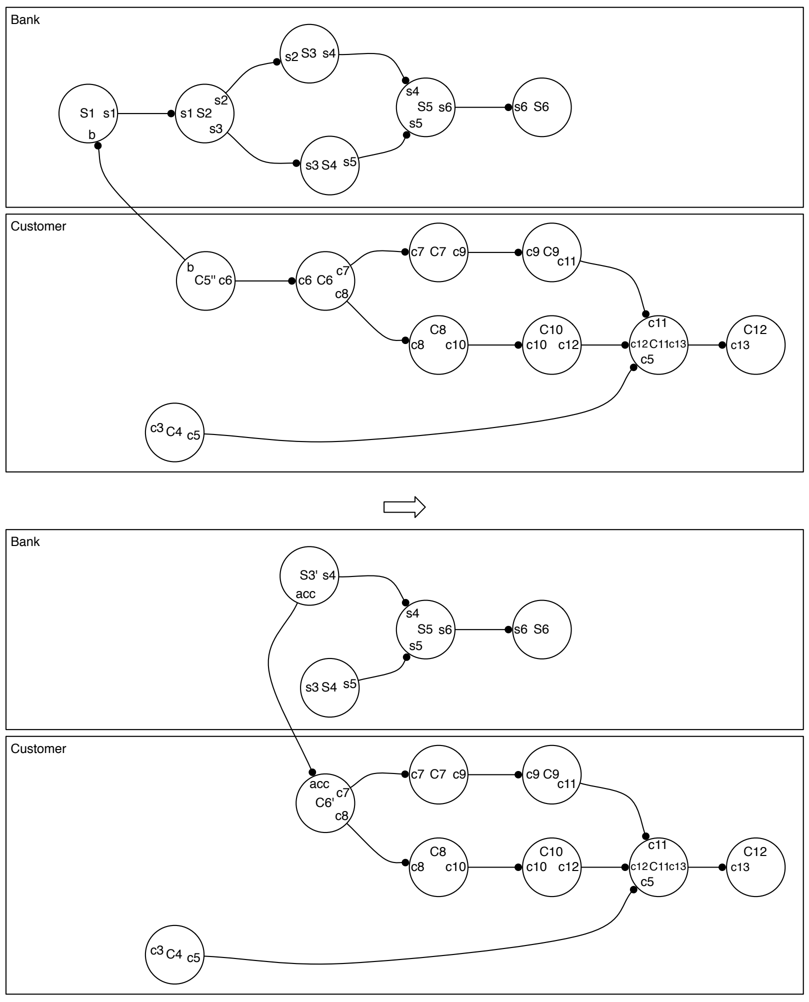


Figure 7.7: Interaction between customer and first bank.

can be conducted, the exclusive choice in agent $S2$ of the first bank has been resolved in favor of accepting the loan request. Hence, agent $S3'$ of the first bank can interact via acc with agent $C6'$ of the customer:

$$S3' = \overline{acc}.\overline{s4}.\mathbf{0} \quad \text{and} \quad C6' = acc.\overline{c7}\langle acc\rangle.\mathbf{0} + rej.\overline{c8}\langle rej\rangle.\mathbf{0}\ .$$

The remaining components belonging to S and C of the system I are depicted in figure 7.8. As can bee seen, while no further actions are possible, agents are remaining. These agents can never be evolved, since the names used as their preconditions are restricted inside either the remainder of S or C. However, inside I another recursive copy of SS can register itself at BB as a new representation of the first bank. Since no more agents representing a customer are available, this case is out of scope for the simulation.

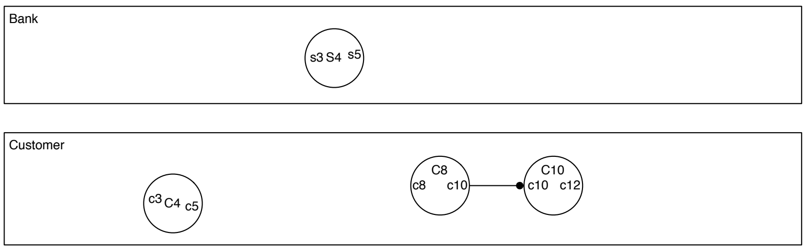


Figure 7.8: Final System of the first bank and the customer.

7.3 Reasoning

In this section, properties of the formal representation of the loan broker interaction are investigated. In particular, we investigate lazy soundness for the process graph of the customer, extend it to interaction soundness with a given set of banks, and finally evaluate if the two different banks given are interaction equivalent.

7.3.1 Lazy Soundness of the Customer

The first property that we investigate is lazy soundness (definition 5.10) for the customer. Lazy Soundness states that a structural sound process graph is deadlock and livelock free as long as the final node has not been executed. Once the final node has been executed for the first time, other nodes might still be executed, however the final node is not executed again. Since lazy soundness is given for structural sound process graphs, we first have to show the structural soundness of the customer's process graph.

Proof 7.3 (Structural Soundness of the Customer's Process Graph) Direct proof. According to definition 5.9 (Structural Sound) we have to show three properties for the process graph $P_C = (N, E, T, A)$ from example 7.1 (Process Graph of the Customer):

- $C1$ is the only initial node, since $pre(C1) = \emptyset \wedge \forall n \in N \backslash \{C1\} : pre(n) \neq \emptyset$ holds.
- $C12$ is the only final node, since $post(C12) = \emptyset \wedge \forall n \in N \backslash \{C12\} : post(n) \neq \emptyset$ holds.
- Every node is on a path from $C1$ to $C12$. We give three paths from P_C that cover all nodes betwen $C1$ and $C12$:

  1. $\langle C1, C2, C3, C5, C6, C7, C9, C11, C12 \rangle$,
  2. $\langle C1, C2, C3, C5, C6, C8, C10, C11, C12 \rangle$, and
  3. $\langle C1, C2, C4, C11, C12 \rangle$.

Since all three properties are fulfilled, the process graph of the customer is structural sound. □

To prove the structural sound process graph of the customer to be lazy sound, we first have to annotate the π-calculus mapping C and thereafter show that it is weak ground bisimilar to

S_{LAZY}. The annotation of C is done according to algorithm 5.3 (Lazy Soundness Annotated π-calculus Mapping). The functional abstractions of the agent definitions $C2, \dots, C11$ are filled with τ, while $C1$ and $C12$ are given by:

$$C1 \overset{def}{=} i.\tau.\overline{c1}.\mathbf{0} \text{ and } C12 \overset{def}{=} c13.\tau.\overline{o}.\mathbf{0}\ .$$

Furthermore, we need to abstract from data-based exclusive choices (see pattern 5.4) and environmental triggered deferred choices (see pattern 5.17) to simplify the reasoning. Using these abstractions, the formalization of the process graph is sufficient to prove lazy soundness. We do not require the formalization of the data flow graph. The abstractions require a modification of the agents $C2$ and $C6$ defined inside C:

$$C2 \overset{def}{=} c1.\tau.(\overline{c2}.\mathbf{0} + \overline{c3}.\mathbf{0}) \text{ and } C6 \overset{def}{=} c6.\tau.(\overline{c7}.\mathbf{0} + \overline{c8}.\mathbf{0})\ . \tag{7.2}$$

Proof 7.4 (Lazy Soundness of the Customer's Process Graph) Using weak ground bisimulation equivalence. According to definition 5.10 (Lazy Sound Process Graph), the process graph P_C of the customer is lazy sound if for the lazy soundness annotated π-calculus mapping C of P_C it holds that $C \approx S_{LAZY}$. Since $C \approx S_{LAZY}$ holds, the process graph of the customer is lazy sound. □

A tool supported proof of $C \approx S_{LAZY}$ is given in appendix A.3.1.

7.3.2 Interaction Soundness of the Customer

After having shown lazy soundness for the process graph of the customer, we now extend the reasoning to include the interactions of the customer. By proving interaction soundness, we can show that all given services that can be used by the customer will not lead to deadlock situations. Due to performance reasons, we use the second variant of the loan broker as given by example 7.18 (Loan Broker Extension - Second Variant). Furthermore, we need to include the environmental triggered deferred choice found in agent $C6$ that has been removed for the lazy soundness proof as well as a mapping of the customer's data flow graph. Due to performance reasons, we still abstract from the data-based exclusive choice found in agent $C2$.

In a first investigation, we place banks of the first variant as given by example 7.5 (Process Graph of the Bank) together with the loan broker and the customer. The initial state of the system corresponds with I from section 7.2 (Simulation), with the exception of annotating $C1$ and $C12$ according to interaction soundness (algorithm 6.2) as shown below:

$$I1 \overset{def}{=} \nu broker\ \nu broker\_add\ (BB \mid SS \mid C)$$

with

$$C1 \overset{def}{=} i.\tau.\overline{c1}.\mathbf{0} \text{ and } C12 \overset{def}{=} c13.\tau.\overline{o}.\mathbf{0}\ .$$

Further modifications have to be made according the removal of the data-based choice in agent $C2$. The complete set of agent definitions is shown in appendix A.3.2. $I1$ is composed out of two different parts, a service graph mapped to agents and an environment agent. The former is given by the component C of $I1$, while the latter is given by the components BB and SS of $I1$. Both can be unified as shown ($(BB \mid SS) \uplus C$).

Proof 7.5 (Unification of the Customer's π-calculus Mapping with an Environment Agent) Direct proof. According to definition 6.6 (Environment Agent), the π-calculus mapping of the customer given by C has to have at least one common free name with the environment agent given by $BB \mid SS$. Since $broker\_add \in fn(C) \cap fn(BB \mid SS)$, the unification is possible. $\square$.

After having shown that the customer has the possibility to interact with the environment given by the loan broker and the first bank with at least one static interaction edge, we can prove the interaction soundness of the customer:

Proof 7.6 (Interaction Soundness of the Customer with an Environment containing the First Bank) Using weak ground bisimulation equivalence. According to definition 6.7 (Interaction Sound Service Graph), the service graph of the customer represented inside $I1$ is interaction sound if $I1 \approx S_{LAZY}$. Since $I1 \approx S_{LAZY}$ holds, the service graph of the customer is interaction sound regarding the environment contained inside $I1$. $\square$

The second investigation shows if the service graph of the customer is still interaction sound even if an instance of the second bank is contained in the environment. The interaction soundness annotated system $I2$ is given y:

$$I2 \stackrel{def}{=} \nu broker\ \nu broker\_add\ (BB \mid SS \mid TT \mid C)$$

with

$$C1 \stackrel{def}{=} i.\tau.\overline{c1}.\mathbf{0} \text{ and } C12 \stackrel{def}{=} c13.\tau.\overline{o}.\mathbf{0}\ .$$

The complete set of agent definitions is shown in appendix A.3.2. In contrast to the last proof, the π-calculus mapping of the service graph of the customer unified with an environment agent E containing the first and the second bank is not interaction sound (we omit the proof for the unification):

Proof 7.7 (Disrupted Interaction Soundness of the Customer with an Environment containing the First and the Second Bank) Using weak ground bisimulation equivalence. According to definition 6.7 (Interaction Sound Service Graph), the service graph of the customer represented inside $I2$ is interaction sound if $I2 \approx S_{LAZY}$. Since $I2 \not\approx S_{LAZY}$, the service graph of the customer is not interaction sound regarding the environment contained inside $I2$. $\square$

Tool supported proofs of $I1 \approx S_{LAZY}$ and $I2 \not\approx S_{LAZY}$ are given in appendix A.3.2.

7.3.3 Interaction Equivalence of the Banks

As a final investigation, we analyze if the first bank and the second bank are interaction equivalent. If both are interaction equivalent, they can be exchanged by each other, so that any service using them is not aware of any differences. The agent formalization of the first bank is given by S, whereas the agent formalization of the second bank is given by T, according to examples 7.9

and 7.10.

Proof 7.8 (Interaction Equivalence of the First and the Second Bank) Using weak open d-bisimulation. According to definition 6.8 (Interaction Equivalence), the environment agents S and T representing the first bank and the second bank are interaction equivalent if $S \approx_O^D T$. Since $S \not\approx_O^D T$, the agents are not interaction equivalent and hence the banks they represent have a different interaction behavior. □

A tool supported proof is given in appendix A.3.3.

7.3.4 Conclusion

The different kinds of reasoning applied to the loan broker interaction showed the expected results. However, even in this comparatively small example several errors have been made during the preparation. Fortunately, most of them have been detected with the help of existing tools as described in chapter A. Hence, even if we think the formalization is correct, automated reasoning on different properties helps in making sure that the process and interaction models are indeed formally correct. For larger processes and interactions, this is even more important.

Regarding the example investigated, it has been shown that the process graph of the customer is deadlock and livelock free (as long as the final node has not been reached). Since no cycles or critical patterns (Discriminator, N-out-of-M-Join, Multiple Instances without Synchronization) are contained, this even holds without the condition stated in brackets. Indeed, the process graph of the customer fulfills weak and relaxed soundness (proofs omitted). By having formally proved these properties, we can be sure that each instance of the customer's process graph will terminate.

Since the customer is not isolated, we also investigated its interaction soundness regarding two different environments. While the customer's service graph can be executed without deadlocks in the first environment where only the first bank is contained, we should avoid enabling it without modifications in the second environment. If the loan broker returns a link to the second bank, the process of the customer deadlocks, since the interaction behavior does not match.

As can already be conducted from the investigation on interaction soundness, the different banks cannot replace each other. This holds for a certain customer, as has been shown using interaction soundness, as well as in the general case, as has been shown using interaction equivalence. The banks are not even able to simulate their interactions in one direction (proof omitted). If we want to integrate a bank that requires a security and is at least interaction simulation compliant to the first bank, we have to find another solution.

Chapter 8

Discussion

This chapter discusses the results that have been presented in the previous part. It starts by re-examining the trends sketched as the shifting focus, continues with gathering restrictions on using the π-calculus for BPM, and concludes with a classification of related work.

8.1 Revisiting the Shifting Focus

In this section we explain how the shifting focus from WfM to BPM is supported by the results of this thesis. The discussion is split into three parts, reflecting the shifting requirements as introduced in section 1.1.

8.1.1 Dynamic Binding

As has been shown in chapter 6 (Interactions), the π-calculus is able to directly express the concept of dynamic binding by its link passing mobility capability. Hence, it fulfills the first requirement. The investigations led to interaction and service graphs (see definitions 6.1 and 6.3). The added value in contrast to existing approaches is given by the interaction flow labels (see definition 6.2). Due to the labeling of interaction edges, derived from π-calculus names, the dynamic passing of interaction channels can be described. Consider for instance a service graph $SG_{Req} = (PS, C, L)$ according to figure 8.1 with the following components:

1. $PS = \{N1, \dots, N5\}$,
2. $C = \{(N2, \perp), (\perp, N3), (N4, \perp)\}$, and
3. $L = \{((N2, \perp), \mathit{find}(ch)), ((\perp, N3), ch(resp)), ((N4, \perp), resp(req))\}$.

Node $N2$ is the source of an interaction edge with the label $\mathit{find}(ch)$. The data of the label is given by ch, which is used in node $N3$ as a response channel. The name ch has been used to correlate the request sent in $N2$ with the response received in $N3$. According to figure 3.9 (The service-oriented architecture), these two interactions correspond to the find arrow between a service requestor and a service provider. The dynamic binding is established by the interaction edge originating from node $N4$. Its label contains the name $resp$ received in node $N3$ that is

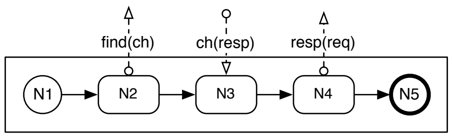


Figure 8.1: A business process with dynamic binding.

used to establish a connection to a service provider. The mapping of interaction and service graphs to π-calculus agents is further on described in algorithm 6.1.

Since restrictions on interaction and service graphs require that each node of the contained process graphs should have at most one interaction edge (with the exception of service nodes), complex interactions can only be modeled by providing a complex process graph structure. These restrictions are necessary, because otherwise a redundant definition of process behavior inside nodes had to be given. We introduced interpretations of the service interaction patterns in an extended BPMN notation to represent complex interactions. Due to the informal description of the service interaction patterns, only a reduced subset based on implicit assumptions has been investigated. Beside the practical value of having patterns for graphically modeling service interactions, the descriptions given cannot be exhaustive—due to the informal description of the patterns—and thus might not be applicable in all cases.

Furthermore, we introduced reasoning on systems of services connected using dynamic binding. The property developed is *interaction soundness* (definition 6.7), which defines when a service graph interacting with a given set of services inside an environment is deadlock free under consideration of all possible bindings. Formally, link passing mobility is only required inside the investigated system made up of the π-calculus mapping of a service graph and an environment agent. The formal representation has the advantage of being compact, since only required components, but not all possible bindings have to be enumerated.

8.1.2 Composition and Visibility

Composition and visibility of components are given by the π-calculus concepts of parallel composition and restricted names, thus fulfilling the second requirement. At any time during the evolution of a system, additional components can be added. If the added components have knowledge about free names of the initial system, they may interact. An initial system can be given by an agent representing a service broker according to example 7.18. The broker offers two free names, $broker\_add$ and $broker$. Using these names, additional components given by π-calculus agents can register themselves as services or request names of registered services. As suggested, π-calculus components can represent services. According to algorithm 6.1 (Mapping Interaction Graphs to Agents), all internal dependencies of a service are handled by restricted names. Due to this, they cannot be disturbed by external events beside the ones especially denoted. By using restricted names, the visibility of the services can be defined. Both concepts—composition and visibility—depend on link passing mobility to share knowledge about interaction possibilities via scope extrusion. Beside the application of concepts provided directly by the π-calculus, we developed algorithms for describing how the business processes inside services can be modeled, encapsulated, and verified.

A key concept required for business processes is the representation of data. Data is used for internal calculations and decision-making, it describes cases that run through a BPMS, and it also describes environmental values. Since the π-calculus can encode the λ-calculus, all kinds of data can be represented. Furthermore, using restricted names, the visibility of data is supported. In chapter 4 we investigated how basic structures like memory cells, stacks, and queues can be formally represented. The investigations led to the definition of natural numbers as a lightweight extension to the π-calculus. We also showed examples of how the data patterns can be formally represented.

Another key concept is the representation of control flow dependencies between activities. We introduced process graphs (see definition 5.1) that provide a formal model for the structure of business processes. A formal semantics is given by applying algorithm 5.1 (Mapping Process Graphs to Agents). The nodes of a process graph are mapped according to a formalization of common process patterns given in section 5.2. Due to the construction of the mapping algorithm and the pattern formalizations, each formalized process graph is encapsulated inside a common agent denoted as N. Furthermore, N contains no free names and is thus completely encapsulated from the outside. The business process is enacted by evolving N.

To verify the internal business processes of components representing services, soundness properties have been investigated. We developed bisimulation-based verification techniques for process graphs mapped to agents according to weak soundness (definition 3.31) and relaxed soundness (definition 3.39). If both properties hold for a given process graph, soundness (definition 3.30) is given. However, since the former kinds of soundness are too strong regarding business processes containing patterns that can leave running (lazy) activities behind, we developed lazy soundness (definition 5.10). Lazy soundness proves a process graph to be free of deadlocks and livelocks as long as the final node has not been reached. Thereafter, activities might remain active but are not permitted to trigger the final node again. Besides supporting lazy activities, the main advantage of the π-calculus characterization of lazy soundness using bisimulation is given by its simplicity. Due to this, the efforts for reasoning are lower than for weak and relaxed soundness, as will be discussed later on.

8.1.3 Change

The requirement of supporting change is fulfilled by different concepts investigated in chapter 6 (Interactions). They are once again based on link passing mobility as well as the prototypical nature of the π-calculus. First of all, we provided the definition of an environment made up of agents (definition 6.6) that can be unified with the π-calculus mapping of a service graph. Due to this, business processes encapsulated as services can be *plugged* into different environments. The minimum requirement on a service graph unified with an environment is at least one static interaction edge between both. This static interaction edge provides an initial communication channel, where all other interaction edges can be retrieved from using link passing mobility. The compatibility of a service graph with an environment made up of different services can be verified using interaction soundness. Two other properties regard the replaceability of (parts of) environments. Using *interaction equivalence* and *interaction simulation* (definition 6.8 and 6.9), we can formally show that two environments have the same observable behavior. In contrast to existing approaches, we consider dynamic binding using link passing mobility.

The prototypical nature of the π-calculus has already been discussed in section 5.1.3. In WfMS, a distinction is made between processes (schemas) and process instances. In a BPMS implementing a SOA, this distinction is blurred. Most notable, this is due to the distributed nature of service-oriented environments. The business processes inside services can still be divided into processes and process instances. However, the deployment of new services into already running systems requires special care. An illustrating example is given by the deployment of a new service to the Internet. Does this deployment require the re-deployment of the Internet? Since from a practical viewpoint it does not, a change to a running system (the Internet) is made. By using the π-calculus, exactly this prototypical approach can be analyzed from a theoretical point of view.

8.2 Formal Foundations

This section critically discusses the π-calculus as a formal foundation for business process management by exposing limitations and drawbacks of the investigations regarding the formal theory.

8.2.1 Minimum Bisimulation Equivalence Requirements

We start with discussing the minimum bisimulation requirements for the different kinds of soundness. Lazy soundness requires at least a weak ground bisimulation equivalence. Even while the patterns 5.15 (Multiple Instances with a priori Runtime Knowledge) and 5.16 (Multiple Instances without a priori Runtime Knowledge) use link passing mobility, the corresponding interactions occur inside the system. According to transition rule COMM, these interactions result in τ-transitions. Since S_{LAZY} contains no objects in its prefixes, a ground bisimulation is sufficient. Weak soundness and relaxed soundness contain the activity (loop) observation agents (definition 5.11 and 5.13), which use link passing mobility for acknowledgment. Again, a weak ground bisimulation equivalence is sufficient, where the same arguments apply as for lazy soundness. Even interaction soundness can be proved using weak ground bisimulation equivalence, since the link passing mobility, required for dynamic binding, is kept inside the observed system. In contrast, interaction equivalence requires a weak open d-bisimulation equivalence, since it shows the conformance of two environment agents in arbitrary contexts. Hence, a congruence on agent terms supporting link passing mobility is required.

Instead of using a ground bisimulation, we can also apply open bisimulation for comparing S_{LAZY}, S_{WEAK}, and $S_{RELAXED}$. Open bisimulation is sufficient because no internal interaction can be provoked inside S_{LAZY}, S_{WEAK}, and $S_{RELAXED}$ by any substitution. The interesting case is given by equating i and o for the π-calculus mapping of an arbitrary process graph. Hence, an interaction according to COMM between the agent representing the initial and the agent representing the final node could be possible. However, this can never occur since all described algorithms derive the agent terms from a structural sound process graph. According to structural soundness, the final node has a precondition given by a restricted name before $\overline{o}$,

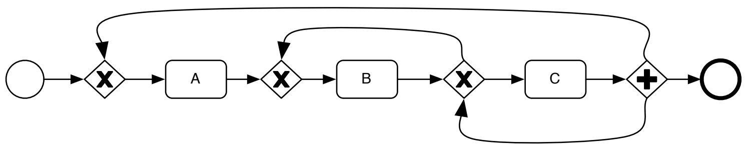


Figure 8.2: A defective business process with loops.

that can only be fulfilled after the preceding node has emitted the postcondition given by the restricted name. If this happens, the free name i has already emitted by the agent representing the initial node.

8.2.2 Efforts for Bisimulation

An interesting constraint of using bisimulation equivalence for proving formal properties of business process and interactions is the practical applicability. From a theoretical point of view, even ground bisimulation between two π-calculus terms is undecidable due to the halting problem. The halting problem can be applied, since the π-calculus is Turing complete, as for instance shown by a mapping to the λ-calculus in [118]. A practical example is already given by pattern 5.16 (Multiple Instances without a priori Runtime Knowledge). In theory this pattern allows an infinite number of instances to be created. Thus, we can reason an infinite amount of time on simulating its transitions. Furthermore, applying loop detection algorithms would hardly succeed, since restricted names and link passing mobility are used for synchronizing the created instances. As can be concluded, existing π-calculus reasoners fail on deciding any kind of soundness for business processes containing this pattern. They also fail on deciding soundness for business processes with interleaved loop structures that reach the final node more than once in reasonable time (see for instance figure 8.2). This might be due to their restriction on depth-first search. The evaluated tools are introduced in appendix A.

A more elaborate evaluation of the efforts is given by a measurement of the execution times for deciding different kinds of soundness. Therefore we applied lazy, weak, and relaxed soundness to different examples given in the appendix of this thesis. Table 8.1 shows the results taken from the *user* output of the Unix tool `time`. An optimization has been made on example A.3. Since a process containing the synchronizing merge pattern can never be lazy or weak sound, these measurements have been skipped. Furthermore, weak soundness analysis for example A.1 has been aborted, since it did not finish within a reasonable timeframe. Lazy and weak soundness have been analyzed with two different tools, *Another Bisimulation Checker* (ABC) and the *Mobility Workbench* (MWB); see appendix A for an introduction. Relaxed soundness has only been measured with ABC since it requires simulation. Currently, only ABC is able to prove simulation using an undocumented command. Regarding example A.6 (Lazy Soundness of the Customer's Process Graph), we replaced the task labeling with letters for a shorter representation. The task *Buy Direct* is denoted as A, *Find Bank* as B, *Request Loan* as C, *Buy* as D, and *Reject Purchase* as E.

Since all measured examples are lazy sound, the respective times in the corresponding rows investigate the full state space. Unsound processes might require a smaller subset that contains a

Soundness		Example A.1 (8 nodes)	Example A.2 (10 nodes)	Example A.3 (12 nodes)	Example A.6 (12 nodes)
Lazy	(ABC)	$6.066s$	$0.598s$	–	$1.077s$
	(MWB)	$7.973s$	$0.360s$	–	$0.397s$
Weak	(ABC)	$> 20m$ (aborted)	$13.327s$	–	$8.971s$
	(MWB)	$> 20m$ (aborted)	$18.364s$	–	$6.346s$
Relaxed	(ABC)	$A : 16.547s$ $B : 16.352s$ $C : 16.415s$ $D : 10.479s$ $\sum = 59.793s$	$A : 1.588s$ $B : 2.060s$ $C : 0.976s$ $D : 1.603s$ $E : 1.568s$ $\sum = 7.795s$	$A :$ 3:55.161s $B :$ 3:11.245s $C :$ 6:20.794s $D :$ 6:03.905s $\sum \approx 19m$	$A : 2.260s$ $B : 3.025s$ $C : 2.871s$ $D : 2.232s$ $E : 2.234s$ $\sum = 12.622s$

Setup: 1.8GHz iMac G5 with 2GB DDR SDRAM, MacOS 10.4.8, ABC.opt v1.0.7, and MWB v4.136.

Table 8.1: Measured efforts for bisimulation based soundness proofs.

counterexample, or have an infinite state space. Nevertheless, due to defective process structures, the state space can also easily explode as for instance in figure 8.2. The measured times for deciding lazy soundness can be seen as a baseline for comparing weak and relaxed soundness. The computation of weak soundness for example A.2 is nearly 22–51 times slower than deciding lazy soundness. Again, the full state space is investigated since the example is weak sound. Deciding weak soundness for example A.6, which is also weak sound, is 8–16 times slower. As can be seen, there is no preference for a certain tool. The measurement of relaxed sound business processes has been reduced to analyzing nodes of the type *Task*. The differences of the examples range between twice as fast as weak soundness (example A.2) up to 31 times as slow (example A.6). Noteworthy, example A.3, which includes a synchronizing merge pattern, has a high effort. The measurements only give examples of possible efforts. A further investigation would require the analysis of a large set of common reference business process models.

To conclude the measurements, we also provide results for interaction soundness, equivalence, and simulation given in table 8.2. Exemplary, we discuss example A.7 in detail. The example contains a service broker, which is able to dynamically register services at runtime, return a registered service in a non-deterministic manner, and also allows the removal of registered services. In the example given, the services can register at any time during the execution of the system. However, the service broker blocks a find request until a service has been registered. This way, a service registration is enforced. The results for interaction soundness differentiate between the tools as shown in the table. If we define the agent $I2$ to contain three services of the same type, such as

$$I2 \overset{def}{=} \nu broker\ \nu broker\_add\ (BB \mid SS \mid SS \mid SS \mid C)\ ,$$

reasoning on interaction soundness was not possible in reasonable time ($< 30m$) on the given

	Interaction Soundness				*Interaction Equivalence*		*Interaction Simulation*
	Ex. A.4a	Ex. A.4c	Ex. A.7	Ex. A.8	Ex. A.5a	Ex. A.9	Ex. A.5b
(ABC)	3:39.342s	18.812s	17.289s	1:4.376s	6.599s	$> 3m$	2.130s
(MWB)	19.157s	2.117s	3.205s	26.906s	4.647s	1.446s	–

Setup: 1.8GHz iMac G5 with 2GB DDR SDRAM, MacOS 10.4.8, ABC.opt v1.0.7, and MWB v4.136.

Table 8.2: Measured efforts for (bi)-simulation based interaction proofs.

setup. By enforcing all services to register before the agent C could start, as given by

$$I2 \stackrel{def}{=} \nu broker\ \nu broker\_add\ \nu b1\ \nu b2\ \nu b3\ (BB \mid S(b1) \mid S(b2) \mid S(b3) \mid CC)\ ,$$

with

$$CC \stackrel{def}{=} \nu r\ \overline{broker\_add}\langle b1, r\rangle.r(c).\overline{broker\_add}\langle b2, r\rangle.r(c).\overline{broker\_add}\langle b3, r\rangle.r(c).C\ ,$$

interaction soundness was decided in *59.458s* using MWB. Thus, by limiting the state space (early registration of services instead of any time registration), reasoning is more applicable.

As has already been shown by the small examples measured, reasoning on soundness and equivalence on processes and interactions represented by π-calculus agents is costly, often inefficient, and sometimes impossible. Solutions include further abstractions, inclusion of additional knowledge, or domain specific reasoners. Regarding abstractions, we already abstracted from data flow for reasoning. While reasoning on formalized business processes that contain data-based exclusive choices is theoretically possible, the effort is way too high. Other abstractions reduce for instance the complexity of a process, e.g. by separating it into different complex activities that can be analyzed independently, map complex patterns to simple ones, e.g. replace multiple instance patterns by simple activities, or modify the process and interaction structure, as for instance given by enforcing an early registration of services. Regarding additional knowledge and domain specific reasoners, an optimized reasoner could for instance take the process or interaction graph into consideration, as well as apply heuristics and breadth-first search. However, the last techniques can only find counterexamples more efficiently or give a feasibility for bisimulation equivalence. Regarding the investigated soundness properties, lazy soundness is most likely to be practically applicable due to its simple representation.

8.2.3 Expressiveness of Bisimulations for Soundness

Another interesting topic is the expressiveness of bisimulation for soundness. While a proven equivalence clearly states that the process graph fulfills a certain property, a mismatch of the agent terms is currently only of restricted use. Since bisimulation is a binary criteria, its answer is either yes or no. While analyzing business processes, however, we are interested in the place where the error occurs. Existing tools provide a trace of actions that led to a contradiction of the bisimulation. While these do not provide a solution, they nevertheless give hints where the problem occurs. Additionally, knowledge of the visited states is required, since non-determinism can occur, as the simple sequence $\varphi = \langle a, b \rangle$ shows:

$$T \overset{def}{=} a.(b.T' + b.T'') \xrightarrow{\varphi} T' \text{ or } \xrightarrow{\varphi} T'' \ .$$

Regarding formalized processes and interaction, this knowledge is also of restricted use, since interactions inside the system are only denoted as τ.

An exemplary debugging session with an available tool is shown in appendix A.3.4. The report of a broken bisimulation equivalence contains several issues. First, restricted names are α-converted. While this is technically correct, it complicates the correlation between the edges of a process graph and the π-calculus names. Second, all agent identifiers beside recursive enumerations are lost. Furthermore, no meta information can be attached to an agent, such as providing a link back to a node of a process graph. Regarding the example shown, it can be deduced from the last transition $\xrightarrow{\overline{o}}$ (Which S_{LAZY} can follow, while N cannot), that a deadlock somewhere in N occurs. Since further possible transitions inside N might have been executed, the provided trace is just one possibility and not the shortest trace. Indeed, we modified agent 1138 (representing an AND gateway) to represent an XOR gateway before executing the bisimulation test. Thus, the problem occurs in the beginning of the process, while the complete trace consists of nine actions (with seven unobservable ones). Hence, the problem is difficult to detect. However, all discussed problems can be solved by a domain specific reasoner that also allows round trip engineering and simulation of process and interaction graphs.

8.2.4 Drawbacks of (Bi)-Simulation for Service Equivalence

Beside the high effort and (reduced) expressiveness of bisimulation for soundness, bisimulation as well as simulation have also drawbacks as a conformance notion. The definitions of interaction equivalence (definition 6.8) and interaction simulation (definition 6.9) are either too strong or too weak regarding the conformance of different environments. To focus on the problem, we use different environment agents that represent placeholders for π-calculus mappings of service graphs. Consider two environment agents P and P' given by:

$$P \overset{def}{=} a.(\tau.\overline{b}.\mathbf{0} + \tau.\overline{c}.\mathbf{0}) \text{ and } P' \overset{def}{=} a.\overline{b}.\mathbf{0} \ .$$

P represents a specification, whereas P' represents an implementation. From the viewpoint of interaction equivalence, $P \not\approx^D_O P'$, since P has the choice to emit via c that P' is unable to mimic. From the implementation's viewpoint, P' is conforming to P, since the decision of which part of the summation is chosen should be internal to the implementation. This behavior can be shown using interaction simulation, where $P' \precsim^D_{\approx O} P$ holds. Thus, bisimulation is too strong regarding the example. A more elaborate example is given in appendix A.2.2.

While simulation solves the problem of the first example, it also relates environments that cause problem regarding interaction soundness. Consider for instance two environments P and Q given by:

$$P \overset{def}{=} a.(\overline{b}.\mathbf{0} + \overline{c}.\mathbf{0}) \text{ and } Q \overset{def}{=} \overline{a}.(b.\mathbf{0} + c.\mathbf{0}) \ .$$

By annotating another P according to interaction soundness, it can be shown that $P \uplus Q$ is interaction sound:

$$\nu a\, \nu b\, \nu c\, (P_i \mid Q) \approx S_{LAZY} \text{ with } P_i \overset{def}{=} i.a.(\overline{b}.\overline{o}.\mathbf{0} + \overline{c}.\overline{o}.\mathbf{0})\ .$$

Since we consider interaction simulation as a conformance relation, we can give P' and Q' as implementations of P and Q by:

$$P' \overset{def}{=} a.\overline{c}.\mathbf{0} \precsim^{D}_{O} P \text{ and } Q' \overset{def}{=} \overline{a}.b.\mathbf{0} \text{ with } Q' \precsim^{D}_{O} Q\ .$$

However, the expected property of interaction soundness is lost for P' unified with Q' :

$$\nu a\, \nu b\, \nu c\, (P'_i \mid Q') \not\approx^{D}_{O} S_{LAZY} \text{ with } P'_i \overset{def}{=} i.a.\overline{c}.\overline{o}.\mathbf{0}\ .$$

In a nutshell, while the specifications are interaction sound—and each implementation conforms to a specification according to interaction simulation—the implementations themselves might contain deadlock behavior not found in the specification. Thus, bisimulation as well as simulation have only limited applicability in real application domains. A possible solution discarding dynamic binding has been presented by Baldoni et al. [22].

8.2.5 Drawbacks of the Pi-Calculus Semantics

The π-calculus semantics as given in chapter 2 has two drawbacks regarding the application of the π-calculus into the domain of business process management. The first drawback is the unenforceability of a transition, meaning that a transition *can* occur. A transition can only be enforced in a distributed system described in the π-calculus by synchronizing all concurrent components. Consider for instance

$$S \overset{def}{=} \nu a\, (\overline{a}.P' \mid a.Q') \mid R\ ,$$

where the transition $\xrightarrow{\tau}$ between the first two components cannot be enforced at a given point in time (however, by assuming fairness it will occur at some point in time). Instead, arbitrary transitions that might be contained inside R can be executed first. The only possible solution is given by a global synchronization, as for instance by a modification

$$S \overset{def}{=} \nu sync\, (\nu a\, (\overline{a}.\overline{sync}.P' \mid a.Q') \mid sync.R)\ .$$

However, global synchronizations contradict the concurrent execution of different activities or services in the BPM domain. We provide two example where the enforcement of transitions at a given point in time is required.

Consider for instance pattern 5.17 (Deferred Choice), which per definition should make the decision inside a node occurring after the pattern. Therefore the succeeding node should cancel concurrent nodes:

$$A \overset{def}{=} \langle\cdot\rangle.(\overline{b}.\mathbf{0} \mid \overline{c}.\mathbf{0})\ ,\ B \overset{def}{=} b.(b_{env}.\overline{kill}.\langle\cdot\rangle.B' + kill.\mathbf{0}) \text{ and } C \overset{def}{=} c.(c_{env}.\overline{kill}.\langle\cdot\rangle.C' + kill.\mathbf{0})$$

inside a system

$$A \mid \nu kill\ (B \mid C)\ .$$

After a transition $\xrightarrow{b_{env}}$ occurred, the node represented by agent C should be "killed", i.e. become inaction. This behavior *can* occur if an interaction between the components B and C via $kill$ occurs. However, sometimes after transition $\xrightarrow{b_{env}}$, but before the interaction between B and C, another transition $\xrightarrow{c_{env}}$ might occur, leading to a deadlock. This is the reason why pattern 5.17 (Deferred Choice) could not be directly implemented and hence requires a more complex processing. A solution to the first problem might be to permit the environment to either signal exclusively b_{env} or c_{env}. However, this is difficult with a deferred choice inside a loop (when should the environment be permitted to emit a name again?) as well as with timers. A timer is given by an agent running concurrently with the π-calculus mapping of a process graph. A timer agent abstracts from concrete time but rather states that after activation it is able to signal a timeout or receive a cancel signal:

$$TIMER \overset{def}{=} set\_timer(timeout, cancel).((\tau.\overline{timeout}.\mathbf{0} + cancel.\mathbf{0}) \mid TIMER)\ .$$

The actual timeout is represented by the execution of τ. Again, since τ cannot be enforced at any point in time, the timer only *can* work as expected. Possible extensions for time and transactions have been provided for instance by Laneve and Zavattaro in [82]. Since the extensions are not lightweight (i.e. can be mapped to the syntax and semantics introduced in chapter 2), we do not investigate them further.

The second drawback, regarding simulation and execution of process and interaction graphs mapped to π-calculus agents, is missing garbage collection. Without garbage collection, remaining—but no longer required—agents can flood the memory. An example is given by a process graph containing pattern 5.4 (Exclusive Choice). A node of the type exclusive choice is always followed by at least two other nodes of which only one will be executed. The agent that represents the unchosen node will never be activated if no loop is contained. Another example is given by the agent $STACK$ (definition 4.3). A stack uses *fresh* $TRIPLE$ agents each time a name is *pushed* on the stack. After each *pop* operation, the agent structure of the corresponding tuple remains. A structural congruence rule for garbage collection of agents prefixed by restricted inputs as given by

$$(\text{SC-INPUT-GARBAGE})\quad \nu z\ z.P \equiv \mathbf{0}$$

is in most cases sufficient. Consider for instance two agents representing a sequence of activities that have been neglected by a preceding exclusive choice:

$$A \overset{def}{=} a.\tau.\bar{b}.\mathbf{0} \text{ and } B \overset{def}{=} b.\tau.\mathbf{0} \text{ inside } \nu a\ \nu b\ (A \mid B)\ .$$

Due to SC-RES-COMP, the term is structural congruent to

$$\nu b\ ((\nu a\ A) \mid B)\ .$$

Using SC-INPUT-GARBAGE and SC-COMP-INACT, component A is dropped and the remaining term is given by

$$\nu b\ B\ .$$

Again, SC-INPUT-GARBAGE can be applied resulting in inaction. As a concluding remark, the drawbacks of the introduced semantics of the π-calculus can be overcome by using existing extensions.

8.3 Related Work

This section discusses the contributions of this thesis in contrast to related work.

8.3.1 Data, Process, and Interaction Patterns

In sections 4.3 (Data Patterns), 5.2 (Process Patterns), and 6.2 (Interaction Patterns), we provided formal representations of data, process, as well as interaction patterns. While the former and the latter are shown by exemplary applications, the process patterns are described in a generic manner. By using the process pattern formalizations, algorithm 5.1 (Mapping Process Graphs to Agents) provides a mapping from process graphs to π-calculus agents. Due to the nature of the patterns—that are only given in natural language—some implicit assumptions have been made explicit in the formalizations. Consider for instance pattern 5.9 (Discriminator). According to the corresponding documentation [12], a discriminator activates subsequent activities if one of the incoming branches is completed. Thereafter it waits for all remaining branches and resets itself. While the pattern formalization captures exactly the textual description, practical application has led to different variants of the discriminator. One variant of the discriminator is given in the YAWL language [11], where all remaining activities beforehand a discriminator are canceled if one incoming branch is activated. Another example is given by the patterns 5.5 (Exclusive Choice) and 5.8 (Multiple Merge). If the former pattern is applied inside a loop, its formal definition and semantics matches the latter one. The same assumptions have been made in YAWL.

Regarding the formal representation of the process patterns, related work is available. First of all, YAWL can be cited once again. YAWL has been designed with direct support for the workflow patterns. Since the semantics of YAWL is given by a transition system (see figure 3.7), it provides a means of formalizing the given process patterns. In contrast to the approach proposed in this thesis, the YAWL semantics does not formally support data and interactions. Furthermore, the transition system of YAWL is proprietary and fixed to the workflow patterns. While it provides a direct support for the patterns, it might be difficult to extend. In contrast, the pattern formalizations given in this thesis are based on a common process calculus that also proved able to support data and interactions.

Cook et al. have proposed other formalizations of process pattern in [47] using Orc, Stefansen in [120] using CCS, Wong and Gibbons in [135] using CSP, as well as Dong and Shen-Sheng in [53] using π-calculus. All approaches claim to support all workflow patterns. This has not been confirmed for [120] and [135], since the related articles are unpublished. Beside the work of Dong and Shen-Sheng, all formalizations use theories without link passing mobility, which complicates the representation of dynamic binding. Since dynamic binding is crucial to support the requirements from section 1.1, the proposed approaches are limited regarding extensions to interacting business processes.

8.3.2 Extended BPMN

In section 3.3 (Graphical Notations) and 6.1.2 (Interaction Graphs) we extended the BPMN to provide a more direct representation of process patterns and means for representing dynamic binding in public and global business process diagrams. Wohed et al. concluded in their investigation on the suitability of BPMN regarding common process patterns, that not all patterns are supported [134]. Since advanced patterns furthermore require utilization of BPMN attributes which are not graphically represented, we provided proprietary extensions to denote multiple instance and discriminator patterns. The author knows no other approach of representing dynamic binding in BPMN.

8.3.3 Abstract Views of Processes and Interactions with Dynamic Binding

In definitions 5.1 (Process Graph), 6.1 (Interaction Graphs), and 6.3 (Service Graphs) we provided abstract views of processes and interactions. The abstract views represent a layer between graphical notations and formal representations. The given graphs capture the structures of processes and interactions, while graphical notations give visualizations and the formalization a formal semantics. Due to their complex nature, we provided only sketches for mapping business process diagrams to process and interaction graphs, as given by examples 5.1 (Partly Mapping of a BPD to a Process Graph) and 6.1 (Partly Mapping of a BPD to an Interaction Graph). Mappings to π-calculus are given by algorithms 5.1 (Mapping Process Graphs to Agents) and 6.1 (Mapping Interaction Graphs to Agents). Process and interaction graphs are used to define different soundness properties at a level above concrete formalization in an informal manner. The approach has the advantage of being generic, since different formalizations can be applied to prove soundness as will be shown in the next section. Furthermore, other graphical notations can be mapped to process and interaction graphs with low effort. Due to the pattern-based mapping from process and interaction graphs to π-calculus, the definition of a formal semantics is straightforward.

Regarding related work, a number of direct mappings from graphical notations to formal representations exist. A strong focus is set on Petri nets that provide a rich formal foundation as shown in chapter 3. Contributions regarding major notations are given for instance by van der Aalst for Event-driven process chains [4] and Stoerrle for UML2 activity diagrams [121]. Regarding BPMN, a mapping via BPEL—as given in the specification—to Petri nets is proposed for instance by Hinz et al. in [70]. Beside direct mappings, other approaches use existing notations as intermediate layers. For instance, Brogi and Popescu map BPEL to YAWL [39], which in turn is mapped to Petri nets for analysis. However, all these approaches are fixed to specific notations and do not support dynamic binding.

8.3.4 Lazy Soundness

In chapter 5 we introduced lazy soundness (definition 5.10). Lazy soundness advances the state-of-the-art in two directions. First of all, it provides a property for proving business processes containing lazy activities to be free of deadlocks and livelocks. Second, it provides bisimulation-based reasoning that is extended to weak and relaxed soundness. Regarding the first issue, also

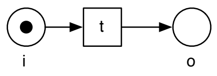


Figure 8.3: A lazy sound Petri net.

weak termination (definition 3.46) is applicable. In contrast to lazy soundness, it enforces the explicit enumeration of all final markings resulting in a complex representation. Regarding the second issue, the author knows no other approach. However, projection inheritance by Basten (definition 3.38) can be applied for reasoning on lazy soundness. A net corresponding to S_{LAZY} is given in figure 8.3. By applying an abstraction (definition 3.37) to all transitions of a more complex net, weak branching bisimulation between the complex net and the one shown in figure 8.3 can be decided. Using branching bisimulation, tokens can remain in the complex net representing a business process. The bisimulations for weak and relaxed soundness require a complex preparation of the investigated Petri net, since the activity observation agent (definition 5.11) and the activity loop observation agent (definition 5.13) have to be included accordingly.

8.3.5 Interaction Soundness and Equivalence

Interaction soundness (definition 6.7) and equivalence (definition 6.8) introduce compatibility and conformance notions for service graphs and environment agents. Composed systems (definition 3.41) and environments (definition 3.42) for workflow modules by Martens match definition 6.5 (Environment) and definition 6.6 (Environment Agent) for service graphs and agents. Interaction soundness resembles usability (definition 3.43) for workflow modules. In addition to usability, interaction soundness supports dynamic binding and thus does not require knowledge of all interaction edges at design time. Benatallah et al. introduced a notion of partial compatibility in [29]. This notion has the advantage of considering only subsets of behavior, but does also not support dynamic binding. Equivalence of workflow modules (definition 3.44) is related to interaction equivalence. The latter has the advantage of supporting dynamic binding, which is not possible in the former.

The author knows no other approach that supports dynamic binding via link passing mobility for compatibility or conformance. The only slightly related approach is given by Canal et al. in [43], where the authors discuss the representation of compatibility in software architectures with runtime binding via a π-calculus representation. The lack of related work is especially interesting, since the recent standard BPEL4WS supports link passing mobility via the `assign fromPartnerLink` statement. Indeed, any asynchronous callback behavior such as given by pattern 6.3 (Send/Receive) requires link passing. Formal semantics for BPEL4WS, such as given in [70, 39, 62], explicitly state that they abstract from link passing mobility. In contrast to lazy soundness, interaction soundness and equivalence cannot be proved using Petri nets as given by definition 3.26, since Petri nets do no support dynamic binding due to their static structure.

8.3.6 Related Formalizations

Regarding support for dynamic binding, other theories can be used as well. First of all, *actors* as introduced by Agha in [17] has to be mentioned. Actors introduce several concepts closely related to the topics of this thesis. They describe a dynamic topology, where channels are determined dynamically. Furthermore, they derive the need for dynamic resource allocation in open systems. Channel passing as well as dynamic resource allocation then build the foundation for reconfigurable and extensible systems. Both concepts are closely related to the π-calculus concepts of link passing mobility and the restriction operator. Regarding extensions to Petri nets, several approaches that might be able to represent dynamic binding are available. However, no publication on how dynamic binding in a SOA can be represented using extended Petri nets is known to the author. A promising approach might be using object Petri nets by Lakos [81] that formally describe *umbrellas* and *subscribers* as special kinds of nets that can interact dynamically. Object Petri nets are based on colored Petri nets. While colored Petri nets are Turing-complete (and thus should be able to somehow represent dynamic binding), they easily allow the representation of correlations using colored tokens. Common patterns have been collected by Mulyar and van der Aalst in [102]. As stated, also this pattern catalogue lacks the representation of dynamic binding. A last approach that should be mentioned are nets in nets, that has been applied to workflow in [13].

8.3.7 Work in Progress

This subsection introduces miscellaneous related work that is currently in progress but has already generated publications. All approaches include investigations on how to formally represent processes and interactions based on different theoretical foundations. Due to their preliminary nature, a detailed discussion is omitted.

SOCK. A holistic approach considering foundational theories, techniques, and methods inside a software engineering approach is investigated inside the *SENSORIA* project funded as an IST project in the 6th framework program of the European union.[1] The project is due on August 2009, thus only preliminary results are available. They include a service-oriented computing kernel, abbreviated as *SOCK* [68]. SOCK is composed out of different calculi focusing on service behavior, service declaration, service engine, and service systems. In contrast to this thesis, which investigates the π-calculus for description on and reasoning about process and interaction behavior, the scope investigated is much larger. An example is given by another publication of the SENSORIA project, the service centered calculus (SCC) [34]. In advantage to the concept of link passing mobility found in the π-calculus, SCC supports explicit modeling of sessions, which can be named and scoped, as well as interruption and cancelation mechanisms. Eventually, SOCK will provide a uniform foundation for service-oriented architectures. It is expected that a more consolidated representation of interactions than in the π-calculus will be possible.

[1] URL: `http://sensoria.fast.de/`

CPN Pattern. A draft technical report on a revised view of the control flow patterns is available in [112]. In contrast to the original publication [12], a more fine grained distinction between the patterns is made, leading to a total of 43 patterns. Furthermore, many implicit assumptions have been made explicit by providing each pattern with a colored Petri net semantics. However, the preliminary report does not explain how dynamic binding can be represented in colored Petri nets, thus leaving the extension from processes to interactions open. Furthermore, it has to be seen how the algorithms developed for Petri nets, i.e. soundness, can be adapted to colored nets. Due to the fact that colored Petri nets as well as π-calculus are Turing complete—and thus are computational equivalent—both are applicable to the BPM domain. Hence, the discussion narrows down to the suitability of either one or the other formalism for representing processes and interactions. We presented results on how the π-calculus can be applied. The reader might individually find his or her view on the suitability. A comparison is out of scope of this thesis.

Chapter 9

Conclusion

This chapter concludes the thesis by summarizing the results and showing paths for future research. Additionally, an outlook on a broader area of computer science will be given as a concluding remark.

9.1 Summary

This thesis investigated the application of a theory for mobile systems, the π-calculus, as a formal foundation for business process management. The investigated areas included the formal representation and verification of data, processes, and interactions. In contrast to existing formal foundations for workflow management, the π-calculus inherently supports dynamic binding via link passing mobility. While link passing mobility is not required for the representation of static processes, it builds the core of dynamic interactions. By supporting link passing mobility, the π-calculus allows the direct extension from processes to interactions. Since interactions occur between a set of processes, composition and visibility requirements arise. Once again, the π-calculus inherently fulfills these by its composition and restriction operators. Furthermore, the π-calculus supports change due to link passing mobility as well as the prototypical nature of the calculus. The prototypical representation of mobile systems of processes directly resembles the structure of the Internet. New processes representing services can be deployed, changed, or removed without affecting the Internet as a whole. There is no need to *re-deploy* the whole system such as static system theory would enforce.

During the investigations, several results have been achieved that are not yet to be found in related work. We were among the first that provided formal representations of the different patterns given in the chapters of Part II. Due to the informal nature of the pattern descriptions, which are given in natural language only, several implicit assumptions had to be made explicit. In particular, the scope of the data and interaction patterns is too broad for a complete formalization, hence we focused on key issues. In contrast, the process patterns, that build the heart of BPM, have been captured completely in a generalized manner. Using these formalizations, algorithms for mapping graphical notations to π-calculus have been introduced. Furthermore, an extension for the representation of dynamic binding in graphical notations has been proposed.

Beside providing an unambiguous description of processes and interactions, the formal representations opened the door for verification. We introduced lazy soundness as a new kind of soundness that is able to deal with lazy activities, i.e. activities that might remain active while the business process itself already reached its final node. These *clean-up* activities can be found for instance before an n-out-of-m-join, a pattern frequently used in interactions. Reasoning on lazy soundness has been grounded in bisimulation equivalence. We furthermore provided reasoning based on bisimulation for weak and relaxed soundness. Lazy soundness has then been extended to interaction soundness, a compatibility notion between a set of services and a business process. In addition to lazy soundness, interaction soundness considers internal *and* external dependencies between activities, ensuring deadlock freedom. In contrast to existing work, dynamic binding of, and between, services is supported. The investigations have been concluded with a conformance notion between environments made up of different services. The conformance is called interaction equivalence and is grounded in weak open bisimulation. This kind of bisimulation takes care of link passing mobility regarding arbitrary contexts and hence supports dynamic binding.

The investigated concepts have been brought together in a larger example. It has been shown how data and processes can be integrated as well as how processes can interact together. All kinds of soundness have been practically applied using a tool chain developed during this thesis. Process and interaction models depicted in an enhanced graphical variant of the BPMN have been mapped to process and interaction graphs, which in turn have been converted to π-calculus agents. However, while reasoning on lazy soundness, interaction soundness, and interaction equivalence is practically possible, the effort is high. This is on the one hand due to the lack of efficient tools and on the other hand based on the size of the state space that has to be investigated. In a sentence, the π-calculus has its strengths in representing interacting processes with dynamic binding, whereas verification based on bisimulation equivalence requires high efforts.

9.2 Future Work

Future research on mobile systems for business process management can be split into several directions according to the BPM lifecycle. Regarding the *Design and Analysis* activity, that has partly been investigated in this thesis, several steps are conceivable. First of all, the practical effort for verification has to be dropped by developing bisimulation checkers that are optimized for the BPM domain. These checkers should use additional knowledge as given by process and interaction graphs. Second, the drawbacks of bisimulation and simulation for interaction equivalence can be overcome by constructing an asymmetric bisimulation, which differentiates input and output prefixes under a summation. Third, type systems can be defined for further reasoning on static properties. Regarding the *Configuration* and *Enactment* activity, a BPM engine based on some kind of π-calculus *bytecode* can be developed. For performance issues, the engine has to natively support different data types by intercepting access to π-calculus names typed as data. Since the π-calculus is Turing complete, all kinds of existing standards, either graphical (e.g. BPMN) or XML–structured (e.g. BPEL4WS), should be executable by the engine after a mapping has been defined. The architectural style should be based on REST [59], since the prototypical representation of processes and interactions in the π-calculus closely resembles the architecture of the Internet. The *Evaluation* activity of the lifecycle can be supported by developing round-trip environments. These should allow a graphical design of processes and interactions as well as supporting verification, simulation, deployment, execution, and mining of runtime data.

9.3 Concluding Remarks

Business process management and service-oriented architectures can be seen as driving technologies for programming-in-the-large, which means assembling instead of programming. Today, this comprises nearly every it-related business. However, assembling is very different to programming and hence requires different methodologies and theories. We like to quote a classic from DeRemer and Kron, already published in 1975:

> ”We argue that structuring a large collection of modules to form a ”system” is an essentially distinct and different intellectual activity from that of constructing the individual modules.” [52]

They continue by motivating the need for a programming-in-the-large notation:

> ”That is, we distinguish between block structure and module interconnectivity. Block structure works well on a small scale, but humans simply cannot keep track of nesting levels after a few pages. Furthermore, and perhaps most important, module interconnectivity must in many cases take the shape of a *graph* or *partial ordering*. The more limited tree structure of nested blocks forces us to place some low-level modules at high places, extending their scope of definition to inappropriate places. It follows, then, that we need a separate language, or at least separate language constructs, for describing module interconnectivity, rather than complicating existing constructs that are well suited for modeling in the small.” [52]

Programming-in-the-large has been brought forward since then, where service-oriented architecture form one branch. Indeed, by replacing the term *module* by *service*, one can easily motivate the need for service orchestration languages from the quotes above. However, what has not been seen at this time is the distributed, concurrent execution of ”modules” in open environments with constant change. The π-calculus is a theory for describing systems in the large that support distribution, concurrency, and link passing mobility. This thesis showed how the π-calculus might be applied to the domain of business process management, acting as a programming-in-the-large language.

The End.

Part IV

Appendix

Appendix A

Examples

This chapter contains examples that support the practical feasibility of the investigations. We used two existing π-calculus bisimulation checkers, the Advanced Bisimulation Checker (ABC) [37] and the Mobility Workbench (MWB) [125]. Most of the examples have been converted from BPMN to π-calculus agents with the help of a tool chain that has been developed during the work on this thesis.[1] To make the presentation as authentic as possible, we provide the generated XML and π-calculus agents in the original format without any other modifications than line wraps.

The tool chain is based on the concepts and algorithms presented in chapter 5 (Processes) and 6 (Interactions). The theoretical foundations are based on chapter 2 (The Pi-Calculus). Figure A.1 depicts the tool dependencies and document flows in the tool chain. Tools or scripts are shown as rectangles, whereas documents are denoted as notes. The components developed during the thesis are highlighted. First of all, we use a *graphical editor* for designing business process diagrams. The editor is equipped with a set of *BPMN stencils* annotated with additional information. Based on this information, an *XML exporter* script is able to generate an XML description of the business process diagram by interacting with the editor. The *XML* representation of the business process can be checked for structural soundness by a *structural soundness checker* script. Furthermore, it can be used as input for a *pi-calculus converter* script that maps the XML file to a proprietary ASCII notation representing π-calculus agents. The implementation is based on example 5.1 (Partly Mapping of a BPD to a Process Graph), algorithm 5.1 (Mapping Process Graphs to Agents), example 6.1 (Partly Mapping of a BPD to an Interaction Graph), and algorithm 6.1 (Mapping Interaction Graphs to Agents). The generated file containing the π-calculus agents can then directly be used as an input for existing π-calculus bisimulation checkers or the *PiVizTool* [31] for simulation. The PiVizTool is a graphical environment for simulating interacting business processes represented in the π-calculus according to chapter 6 (Interactions) and chapter 7 (Unification). It has been developed as part of a Master thesis supervised by the author of this work.

Technically, the feasibility study has been developed on *Mac OS X*. *OmniGraffle Professional* is used as a graphical editor.[2] It is fully programmable in *AppleScript*, which was used

[1] `http://bpt.hpi.uni-potsdam.de/twiki/bin/view/Piworkflow/Reasoner`
[2] `http://www.omnigroup.com/applications/omnigraffle`

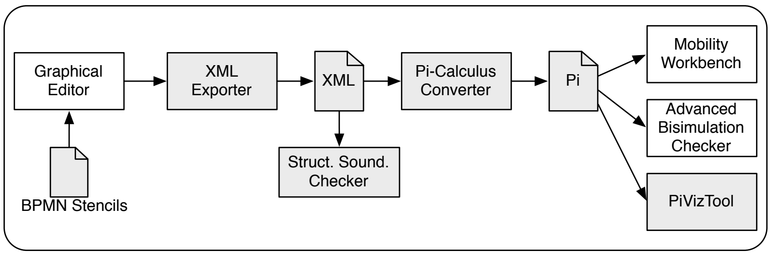


Figure A.1: Architecture of the tool chain.

for implementing the XML exporter. The π-calculus converter and the structural soundness checker have been implemented as os-independend Ruby scripts. The π-calculus tools compatible with the tool chain are the bisimulation checkers MWB and ABC for verification as well as the PiVizTool. MWB and ABC are written in the functional programming languages SML and OCaml that are available for a wide variety of platforms. The PiVizTool is written in Java.

A.1 Processes

This section contains examples for chapter 5 (Processes).

A.1.1 Lazy Soundness

We illustrate lazy soundness by example in the corresponding input style for MWB/ABC.

Example A.1 (Lazy Soundness Tool Example) The XML representation of the process graph according to the business process diagram shown in figure A.2 is given by:

```
<model>
   <process id="1" type="BPMN">
      <node id="1155" type="MI without Sync" name="D" count="3"/>
      <node id="1146" type="End Event"/>
      <node id="1145" type="Task" name="B"/>
      <node id="1144" type="Task" name="C"/>
      <node id="1143" type="Task" name="A"/>
      <node id="1138" type="AND Gateway"/>
      <node id="1137" type="N-out-of-M-Join" continue="2"/>
      <node id="1136" type="Start Event"/>
      <flow id="1163" type="Sequence Flow" from="1155" to="1146"/>
      <flow id="1154" type="Sequence Flow" from="1137" to="1155"/>
      <flow id="1153" type="Sequence Flow" from="1144" to="1137"/>
      <flow id="1152" type="Sequence Flow" from="1145" to="1137"/>
      <flow id="1151" type="Sequence Flow" from="1143" to="1137"/>
      <flow id="1150" type="Sequence Flow" from="1138" to="1144"/>
      <flow id="1149" type="Sequence Flow" from="1138" to="1145"/>
      <flow id="1148" type="Sequence Flow" from="1138" to="1143"/>
      <flow id="1147" type="Sequence Flow" from="1136" to="1138"/>
   </process>
</model>
```

The lazy soundness annotated π-calculus mapping of the process graph is given by:

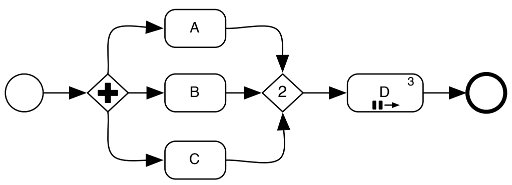


Figure A.2: Business process example 1.

```
agent N1155(e1154,e1163)=e1154.(t.0 | t.0 | t.0 | 'e1163.0 | N1155(e1154,e1163))
agent N1138(e1147,e1150,e1149,e1148)=e1147.t.('e1150.0 | 'e1149.0 | 'e1148.0 |
      N1138(e1147,e1150,e1149,e1148))
agent N1137(e1153,e1152,e1151,e1154)=(^h,run)(N1137_1(e1153,e1152,e1151,e1154,h,run) |
      N1137_2(e1153,e1152,e1151,e1154,h,run))
agent N1137_1(e1153,e1152,e1151,e1154,h,run)=e1153.'h.0 | e1152.'h.0 | e1151.'h.0
agent N1137_2(e1153,e1152,e1151,e1154,h,run)=h.h.'run.h.
      N1137(e1153,e1152,e1151,e1154) | run.t.'e1154.0
agent N1136(e1147,i)=i.t.'e1147.0
agent N1146(e1163,o)=e1163.t.'o.N1146(e1163,o)
agent N1145(e1149,e1152)=e1149.t.('e1152.0 | N1145(e1149,e1152))
agent N1144(e1150,e1153)=e1150.t.('e1153.0 | N1144(e1150,e1153))
agent N1143(e1148,e1151)=e1148.t.('e1151.0 | N1143(e1148,e1151))
agent N(i,o)=(^e1163,e1154,e1153,e1152,e1151,e1150,e1149,e1148,e1147)(
      N1155(e1154,e1163) | N1138(e1147,e1150,e1149,e1148) |
      N1137(e1153,e1152,e1151,e1154) | N1136(e1147,i) |
      N1146(e1163,o) | N1145(e1149,e1152) | N1144(e1150,e1153) |
      N1143(e1148,e1151))
agent S_LAZY(i,o)=i.t.'o.0
```

We can ask MWB for deciding weak open d-bisimulation equivalence on N and S_{LAZY}, thus deciding lazy soundness for the process graph from example 5.2 (Simple Business Process):

```
MWB>weq N(i,o) S_LAZY(i,o)
The two agents are equal.
Bisimulation relation size = 317.
```

Since $N \approx_O^D S_{LAZY}$ holds, the corresponding process graph is lazy sound. By modifying the AND Gateway of the example given in figure A.2 to an XOR Gateway in the corresponding lazy soundness annotated π-calculus mapping, we can prove the corresponding process graph to be not lazy sound:

```
MWB>agent N1138(e1147,e1150,e1149,e1148)=e1147.t.(('e1150.0 + 'e1149.0 + 'e1148.0) |
N1138(e1147,e1150,e1149,e1148))
MWB>weq N(i,o) S_LAZY(i,o)
The two agents are NOT equal.
```

Obviously, the modified process graph is not lazy sound as it contains a deadlock.

A.1.2 Weak Soundness

Weak soundness is illustrated in the corresponding input style for MWB/ABC.

Example A.2 (Weak Soundness Tool Example) The XML representation of the process graph according to the business process shown in figure A.3 is given by:

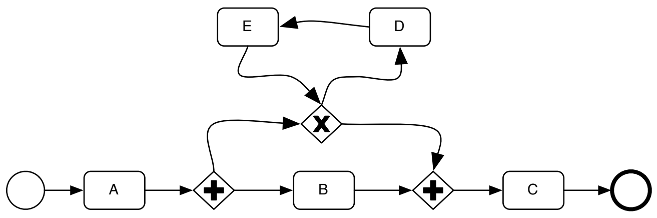


Figure A.3: Business process example 2.

```
<model>
   <process id="1" type="BPMN">
      <node id="924" type="AND Gateway"/>
      <node id="782" type="Task" name="E"/>
      <node id="781" type="Task" name="D"/>
      <node id="773" type="XOR Gateway"/>
      <node id="772" type="End Event"/>
      <node id="1189" type="Task" name="C"/>
      <node id="1188" type="Task" name="B"/>
      <node id="1183" type="AND Gateway"/>
      <node id="1182" type="Task" name="A"/>
      <node id="1181" type="Start Event"/>
      <flow id="1194" type="Sequence Flow" from="782" to="773"/>
      <flow id="1193" type="Sequence Flow" from="781" to="782"/>
      <flow id="1192" type="Sequence Flow" from="773" to="781"/>
      <flow id="1191" type="Sequence Flow" from="773" to="924"/>
      <flow id="1190" type="Sequence Flow" from="1183" to="773"/>
      <flow id="788" type="Sequence Flow" from="1189" to="772"/>
      <flow id="787" type="Sequence Flow" from="924" to="1189"/>
      <flow id="786" type="Sequence Flow" from="1188" to="924"/>
      <flow id="785" type="Sequence Flow" from="1183" to="1188"/>
      <flow id="784" type="Sequence Flow" from="1182" to="1183"/>
      <flow id="783" type="Sequence Flow" from="1181" to="1182"/>
   </process>
</model>
```

The weak soundness annotated π-calculus mapping is given by:

```
agent N924(e1191,e786,e787,x)=e1191.e786.(^ack)'x<ack>.ack.('e787.0 |
      N924(e1191,e786,e787,x))
agent N1189(e787,e788,x)=e787.(^ack)'x<ack>.ack.('e788.0 | N1189(e787,e788,x))
agent N1188(e785,e786,x)=e785.(^ack)'x<ack>.ack.('e786.0 | N1188(e785,e786,x))
agent N1183(e784,e1190,e785,x)=e784.(^ack)'x<ack>.ack.(N1183(e784,e1190,e785,x) |
      'e1190.0 | 'e785.0)
agent N1182(e783,e784,x)=e783.(^ack)'x<ack>.ack.('e784.0 | N1182(e783,e784,x))
agent N1181(e783,i,x)=i.(^ack)'x<ack>.ack.'e783.0
agent N782(e1193,e1194,x)=e1193.(^ack)'x<ack>.ack.('e1194.0 | N782(e1193,e1194,x))
agent N781(e1192,e1193,x)=e1192.(^ack)'x<ack>.ack.('e1193.0 | N781(e1192,e1193,x))
agent N773(e1194,e1190,e1192,e1191,x)=(e1194.N773_1(e1194,e1190,e1192,e1191,x) +
      e1190.N773_1(e1194,e1190,e1192,e1191,x))
agent N773_1(e1194,e1190,e1192,e1191,x)=(^ack)'x<ack>.ack.(('e1192.0 + 'e1191.0) |
      N773(e1194,e1190,e1192,e1191,x))
agent N772(e788,o,x)=e788.(^ack)'x<ack>.ack.'o.N772(e788,o,x)
agent N(i,o,s)=(^e1194,e1193,e1192,e1191,e1190,e788,e787,e786,e785,e784,e783,x)
      (N924(e1191,e786,e787,x) | N1189(e787,e788,x) | N1188(e785,e786,x) |
      N1183(e784,e1190,e785,x) | N1182(e783,e784,x) | N1181(e783,i,x) |
      N782(e1193,e1194,x) | N781(e1192,e1193,x) | N773(e1194,e1190,e1192,e1191,x) |
      N772(e788,o,x) | X(x,s))
agent X(x,s)=x(ack).(t.'ack.0 | X(x,s)) + x(ack).('s.'ack.0 | X_1(x))
agent X_1(x)=x(ack).(t.'ack.0 | X_1(x))
agent S_WEAK(i,o,s)=i.(t.'o.0 + t.'s.'o.0)
```

We can ask MWB for deciding weak open d-bisimulation equivalence on N and S_{WEAK}, thus deciding weak soundness for the process graph:

```
MWB>weqd (i,o,s) N(i,o,s) S_WEAK(i,o,s)
The two agents are equal.
Bisimulation relation size = 258.
```

Since $N \approx_O^D S_{WEAK}$ holds, the corresponding process graph is weak sound. A counterexample can be given by checking weak soundness for the process graph from example A.1. The corresponding weak soundness annotated π-calculus mapping is given by:

```
agent N1155(e1154,e1163,x)=e1154.(t.0 | t.0 | t.0 | 'e1163.0 | N1155(e1154,e1163,x))
agent N1138(e1147,e1150,e1149,e1148,x)=e1147.(^ack)'x<ack>.ack.('e1150.0 | 'e1149.0 |
      'e1148.0 | N1138(e1147,e1150,e1149,e1148,x))
agent N1137(e1153,e1152,e1151,e1154,x)=(^h,run)(
       N1137_1(e1153,e1152,e1151,e1154,x,h,run) |
       N1137_2(e1153,e1152,e1151,e1154,x,h,run))
agent N1137_1(e1153,e1152,e1151,e1154,x,h,run)=e1153.'h.0 | e1152.'h.0 | e1151.'h.0
agent N1137_2(e1153,e1152,e1151,e1154,x,h,run)=
       h.h.'run.h.N1137(e1153,e1152,e1151,e1154,x) |
      run.(^ack)'x<ack>.ack.'e1154.0
agent N1136(e1147,i,x)=i.(^ack)'x<ack>.ack.'e1147.0
agent N1146(e1163,o,x)=e1163.(^ack)'x<ack>.ack.'o.N1146(e1163,o,x)
agent N1145(e1149,e1152,x)=e1149.(^ack)'x<ack>.ack.('e1152.0 | N1145(e1149,e1152,x))
agent N1144(e1150,e1153,x)=e1150.(^ack)'x<ack>.ack.('e1153.0 | N1144(e1150,e1153,x))
agent N1143(e1148,e1151,x)=e1148.(^ack)'x<ack>.ack.('e1151.0 | N1143(e1148,e1151,x))
agent N(i,o,s)=(^e1163,e1154,e1153,e1152,e1151,e1150,e1149,e1148,e1147,x)
      (N1155(e1154,e1163,x) | N1138(e1147,e1150,e1149,e1148,x) |
       N1137(e1153,e1152,e1151,e1154,x) | N1136(e1147,i,x) | N1146(e1163,o,x) |
       N1145(e1149,e1152,x) | N1144(e1150,e1153,x) | N1143(e1148,e1151,x))
agent X(x,s)=x(ack).(t.'ack.0 | X(x,s)) + x(ack).('s.'ack.0 | X_1(x))
agent X_1(x)=x(ack).(t.'ack.0 | X_1(x))
agent S_WEAK(i,o,s)=i.(t.'o.0 + t.'s.'o.0)
```

We can ask MWB again for deciding weak open d-bisimulation equivalence:

```
MWB>weqd (i,o,s) N(i,o,s) S_WEAK(i,o,s)
The two agents are NOT equal.
```

This time, the process graph is not weak sound, since it contains lazy activities which can be active after the final node has been reached.

A.1.3 Relaxed Soundness

Relaxed soundness is illustrated by example in the corresponding input style for ABC.

Example A.3 (Relaxed Soundness Tool Example) The XML representation of the process graph according to the business process shown in figure A.4 is given by:

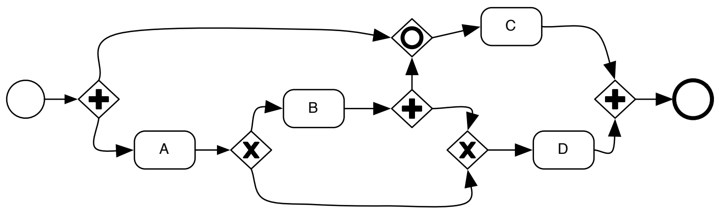


Figure A.4: Business process example 3.

```
<model>
   <process id="1" type="BPMN">
      <node id="538" type="End Event"/>
      <node id="1346" type="Task" name="D"/>
      <node id="1339" type="AND Gateway"/>
      <node id="1335" type="Task" name="C"/>
      <node id="714" type="OR Gateway"/>
      <node id="1324" type="XOR Gateway"/>
      <node id="1318" type="AND Gateway"/>
      <node id="1316" type="Task" name="B"/>
      <node id="1306" type="XOR Gateway"/>
      <node id="1305" type="Task" name="A"/>
      <node id="1300" type="AND Gateway"/>
      <node id="1299" type="Start Event"/>
      <flow id="1350" type="Sequence Flow" from="1339" to="538"/>
      <flow id="1349" type="Sequence Flow" from="1346" to="1339"/>
      <flow id="1347" type="Sequence Flow" from="1324" to="1346"/>
      <flow id="1344" type="Sequence Flow" from="1335" to="1339"/>
      <flow id="1338" type="Sequence Flow" from="1318" to="714"/>
      <flow id="1337" type="Sequence Flow" from="714" to="1335"/>
      <flow id="1336" type="Sequence Flow" from="1300" to="714"/>
      <flow id="1333" type="Sequence Flow" from="1306" to="1324"/>
      <flow id="1332" type="Sequence Flow" from="1318" to="1324"/>
      <flow id="1323" type="Sequence Flow" from="1316" to="1318"/>
      <flow id="1317" type="Sequence Flow" from="1306" to="1316"/>
      <flow id="1315" type="Sequence Flow" from="1300" to="1305"/>
      <flow id="1314" type="Sequence Flow" from="1305" to="1306"/>
      <flow id="671" type="Sequence Flow" from="1299" to="1300"/>
   </process>
</model>
```

The relaxed soundness annotated π-calculus mapping is given as follows. Since relaxed soundness requires $n - 2$ different agent representations of a process graph with $n > 2$ nodes, we only give an example regarding the activities *A* and *B* to showcase the reasoning. The relaxed soundness annotated π-calculus mapping for investigating activity *A* is given by:

```
agent N538(e1350,o)=e1350.t.'o.N538(e1350,o)
agent N1324(e1333,e1332,e1347)=(e1333.N1324_1(e1333,e1332,e1347) +
      e1332.N1324_1(e1333,e1332,e1347))
agent N1324_1(e1333,e1332,e1347)=t.('e1347.0 | N1324(e1333,e1332,e1347))
agent N1318(e1323,e1338,e1332)=e1323.t.('e1338.0 | 'e1332.0 | N1318(e1323,e1338,e1332))
agent N1316(e1317,e1323)=e1317.t.('e1323.0 | N1316(e1317,e1323))
agent N1306(e1314,e1333,e1317)=e1314.t.(('e1333.0 + 'e1317.0) |
      N1306(e1314,e1333,e1317))
agent N1305(e1315,e1314,x)=e1315.(^ack)'x<ack>.ack.('e1314.0 | N1305(e1315,e1314,x))
agent N1300(e671,e1336,e1315)=e671.t.('e1336.0 | 'e1315.0 | N1300(e671,e1336,e1315))
agent N1299(e671,i)=i.t.'e671.0
agent N1346(e1347,e1349)=e1347.t.('e1349.0 | N1346(e1347,e1349))
agent N1339(e1349,e1344,e1350)=e1349.e1344.t.('e1350.0 | N1339(e1349,e1344,e1350))
agent N1335(e1337,e1344)=e1337.t.('e1344.0 | N1335(e1337,e1344))
agent N714(e1338,e1336,e1337)=(^c,w,d)( (e1338.('d.0 + w.c.0) + c.0) |
      (e1336.('d.0 + w.c.0) + c.0) | 'w.0 | d.'c.t.('e1337.0 |
      N714(e1338,e1336,e1337)))
agent N(i,o,s)=(^e1350,e1349,e1347,e1344,e1338,e1337,e1336,e1333,e1332,e1323,e1317,
      e1315,e1314,e671,x)(N538(e1350,o) | N1324(e1333,e1332,e1347) |
      N1318(e1323,e1338,e1332) | N1316(e1317,e1323) | N1306(e1314,e1333,e1317) |
      N1305(e1315,e1314,x) | N1300(e671,e1336,e1315) | N1299(e671,i) |
      N1346(e1347,e1349) | N1339(e1349,e1344,e1350) | N1335(e1337,e1344) |
      N714(e1338,e1336,e1337) | X(x,s))
agent X(x,s)=x(ack).(t.'ack.0 | X(x,s)) + x(ack).('s.'ack.0 | X_1(x))
agent X_1(x)=x(ack).(t.'ack.0 | X_1(x))
agent S_RELAXED(i,o,s)=i.'s.'o.0
```

We can ask ABC for deciding weak open simulation equivalence:

```
abc > wlt S_RELAXED(i,o,s) N(i,o,s)
The two agents are weakly related (4).
Do you want to see the core of the simulation (yes/no) ? no
```

Since $S_{RELAXED} \precsim^{D}_{\approx O} N$ holds, activity A participates in the process in at least one valid execution sequence. The relaxed soundness annotated π-calculus mapping for investigating activity *B* is given by:

```
agent N538(e1350,o)=e1350.t.'o.N538(e1350,o)
agent N1324(e1333,e1332,e1347)=(e1333.N1324_1(e1333,e1332,e1347) +
      e1332.N1324_1(e1333,e1332,e1347))
agent N1324_1(e1333,e1332,e1347)=t.('e1347.0 | N1324(e1333,e1332,e1347))
agent N1318(e1323,e1338,e1332)=e1323.t.('e1338.0 | 'e1332.0 | N1318(e1323,e1338,e1332))
agent N1316(e1317,e1323,x)=e1317.(^ack)'x<ack>.ack.('e1323.0 | N1316(e1317,e1323,x))
agent N1306(e1314,e1333,e1317)=e1314.t.(('e1333.0 + 'e1317.0) |
      N1306(e1314,e1333,e1317))
agent N1305(e1315,e1314)=e1315.t.('e1314.0 | N1305(e1315,e1314))
agent N1300(e671,e1336,e1315)=e671.t.('e1336.0 | 'e1315.0 | N1300(e671,e1336,e1315))
agent N1299(e671,i)=i.t.'e671.0
agent N1346(e1347,e1349)=e1347.t.('e1349.0 | N1346(e1347,e1349))
agent N1339(e1349,e1344,e1350)=e1349.e1344.t.('e1350.0 | N1339(e1349,e1344,e1350))
agent N1335(e1337,e1344)=e1337.t.('e1344.0 | N1335(e1337,e1344))
agent N714(e1338,e1336,e1337)=(^c,w,d)( (e1338.('d.0 + w.c.0) + c.0) |
      (e1336.('d.0 + w.c.0) + c.0) | 'w.0 |
      d.'c.t.('e1337.0 | N714(e1338,e1336,e1337)))
agent N(i,o,s)=(^e1350,e1349,e1347,e1344,e1338,e1337,e1336,e1333,e1332,e1323,e1317,
      e1315,e1314,e671,x)(N538(e1350,o) | N1324(e1333,e1332,e1347) |
      N1318(e1323,e1338,e1332) | N1316(e1317,e1323,x) | N1306(e1314,e1333,e1317) |
      N1305(e1315,e1314) | N1300(e671,e1336,e1315) | N1299(e671,i) |
      N1346(e1347,e1349) | N1339(e1349,e1344,e1350) | N1335(e1337,e1344) |
      N714(e1338,e1336,e1337) | X(x,s))
agent X(x,s)=x(ack).(t.'ack.0 | X(x,s)) + x(ack).('s.'ack.0 | X_1(x))
agent X_1(x)=x(ack).(t.'ack.0 | X_1(x))
agent S_RELAXED(i,o,s)=i.'s.'o.0
```

We can ask ABC for deciding weak open simulation equivalence:

```
abc > wlt S_RELAXED(i,o,s) N(i,o,s)
The two agents are weakly related (4).
Do you want to see the core of the simulation (yes/no) ? no
```

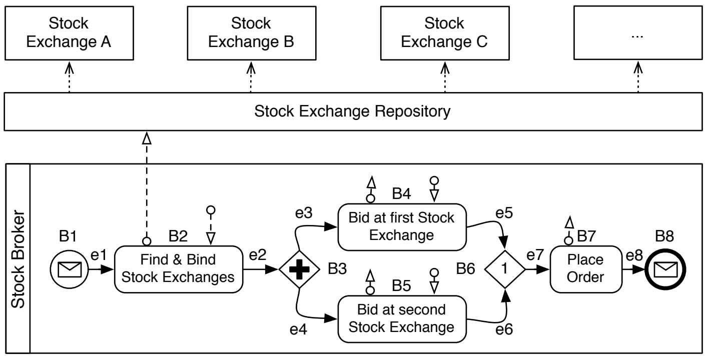

Stock Exchange A
Stock Exchange B
Stock Exchange C
...
Stock Exchange Repository
Stock Broker
B1
e1
B2
Find & Bind Stock Exchanges
e2
B3
e3
e4
B4
Bid at first Stock Exchange
B5
Bid at second Stock Exchange
e5
e6
B6
1
e7
B7
Place Order
e8
B8

Figure A.5: Stock broker interaction.

Since again $S_{RELAXED} \precapprox^D_O N$ holds, activity B also participates in the process. We omit the proofs for further activities.

A.2 Interactions

A.2.1 Interaction Soundness

We illustrate interaction soundness by example in the corresponding input style for MWB/ABC.

Example A.4 (Interaction Soundness Tool Example) The π-calculus representation of the interaction shown in figure A.5 is given by:

```
agent SE_A(ch) = (^o)ch(b).t.'b<o>.o.SE_A(ch)
agent SE_B(ch) = (^o)ch(b).t.'b<o>.o.SE_B(ch)
agent SE_C(ch) = (^o)ch(b).t.'b<o>.o.SE_C(ch)

agent R(r,s1,s2,s3)=r(ch).'ch<s1>.r(ch).'ch<s2>.R(r,s1,s2,s3) +
      r(ch).'ch<s2>.r(ch).'ch<s3>.R(r,s1,s2,s3) +
      r(ch).'ch<s1>.r(ch).'ch<s3>.R(r,s1,s2,s3)

agent B(i,o,r)=(^e1,e2,e3,e4,e5,e6,e7,e8)( B1(e1,i) | B2(e1,e2,r) | B3(e2,e3,e4) |
      B4(e3,e5) | B5(e4,e6) | B6(e5,e6,e7) | B7(e7,e8) |B8(e8,o))
agent B1(e1,i)=i.t.'e1.0
agent B2(e1,e2,r)=(^ch)e1.'r<ch>.ch(s1).'r<ch>.ch(s2).t.('e2<s1,s2>.0 | B2(e1,e2,r))
agent B3(e2,e3,e4)=e2(s1,s2).t.('e3<s1>.0 | 'e4<s2>.0 | B3(e2,e3,e4))
agent B4(e3,e5)=(^b)e3(s).'s<b>.b(o).t.('e5<o>.0 | B4(e3,e5))
agent B5(e4,e6)=(^b)e4(s).'s<b>.b(o).t.('e6<o>.0 | B5(e4,e6))
agent B6(e5,e6,e7)=(^h,run)(B6_1(e5,e6,e7,h,run) | B6_2(e5,e6,e7,h,run))
agent B6_1(e5,e6,e7,h,run)=e5(o).'h<o>.0 | e6(o).'h<o>.0
agent B6_2(e5,e6,e7,h,run)=h(o).'run<o>.h(o).B6(e5,e6,e7) | run(o).t.'e7<o>.0
agent B7(e7,e8)=e7(o).'o.t.('e8.0 | B7(e7,e8))
agent B8(e8,o)=e8.t.'o.B8(e8,o)

agent SYS(i,o) = (^s1,s2,s3,r)( SE_A(s1) | SE_B(s2) | SE_C(s3) | R(r,s1,s2,s3) |
      B(i,o,r))

agent S_LAZY(i,o)=i.t.'o.0
```

The first three lines of the example denote simple kinds of services that are used for reasoning. They create an order token o and wait for a connection via $ch(b)$, where b is a response channel used to signal back the o token. In between, however, complex computation takes place that is abstracted from by τ. Note that the different services do not differ in their interaction behavior, thus we may use each of them inside the stock broker.

The agent R denotes a simple kind of a repository that returns two arbitrary services. We omitted a complex structure based on lists that would allow arbitrary services to register and to de-register. The stock broker's process is represented in the third block. B is a π-calculus agent containing all activities of the stock broker, that in turn are represented according to figure A.5 by $B1 \dots B8$. Note the agents $B2$, where the stock exchanges are found at the repository (`'r<ch>.ch(s1).'r<ch>.ch(s2)`), and $B4$ and $B5$, where the stock exchanges are dynamically bound and invoked (`'s<b>.b(o)`). Furthermore, the successful bidding activity forwards the order token to $B7$, where the order is finally placed. To allow agents to be observed according to lazy soundness, $B1$ and $B8$ are enhanced with `i` and `'o` accordingly. The agent $SYS(i, o)$ places all participants into a system leaving only i and o as free names. SYS can then be compared to S_LAZY required for deciding lazy soundness.

Part A. A tool session using MWB to prove interaction soundness is shown below:

```
MWB>weq SYS(i,o) S_LAZY(i,o)
The two agents are equal.
```

The agent of the stock broker inside the environment represented by SYS is weak open d-bisimulation equivalent to S_LAZY, hence the service graph contained is interaction sound. Since the service graph includes the interactions with the repository and stock exchanges, all possible behaviors of the services are acceptable and will not lead to a deadlock.

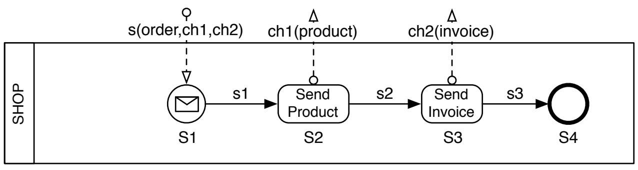
Part B. But what happens if one of the possible interaction partners, e.g. one of the services, shows a different interaction behavior? This can be investigated, for instance, by changing the definition of $SE\_A(ch)$ to wait for a confirmation of the bidding via b before proceeding.

```
MWB>agent SE_A(ch) = (^o)ch(b).b(confirm).t.'b<o>.o.SE_A(ch)
MWB>weq SYS(i,o) S_LAZY(i,o)
The two agents are equal.
```

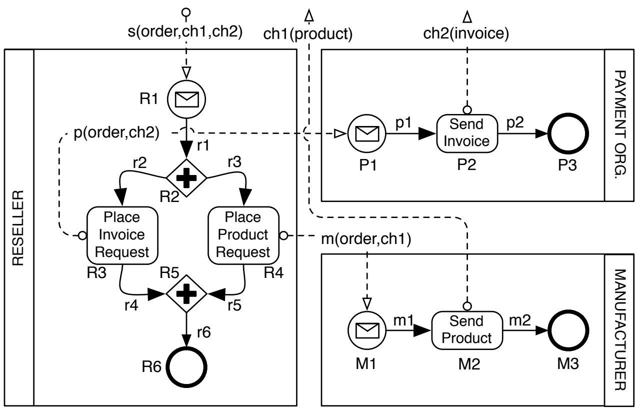
Once again, the service graph SG related to SYS is interaction sound. This is even true if the "defective" service represented by agent $SE\_A(ch)$ is dynamically bound. In this case, always the second service will be used (due to the discriminator). Hence, the service graph will not deadlock even if a non-matching, defective service is contained in the environment.

Part C. However, if we introduce a second defective service by changing $SE\_B(ch)$, the possibility of selecting and binding to two defective services exists, thus leading to a serious problem:

```
MWB>agent SE_B(ch) = (^o)ch(b).b(confirm).t.'b<o>.o.SE_A(ch)
MWB>weq SYS(i,o) S_LAZY(i,o)
The two agents are NOT equal.
```



(a) Environment 1.



(b) Environment 2.

Figure A.6: Two different environments.

The service graph SG contained in the modified system is not interaction sound anymore, since there exist possible combinations of services in it that will lead to deadlock situations.

A.2.2 Interaction Equivalence

We illustrate interaction simulation and interaction equivalence by example in the corresponding input style for ABC.

Example A.5 (Interaction Equivalence Tool Example) The environment agents of the environments shown in figure A.6 are given by:

```
agent S(x) = (^s1,s2,s3)( S1(x,s1) | S2(s1,s2) | S3(s2,s3) | S4(s3) )
agent S1(x,s1) = x(ch1).ch1(ch2).ch1(order).t.'s1<ch1,ch2>.0
agent S2(s1,s2) = (^invoice)(s1(ch1,ch2).t.'ch1<invoice>.'s2<ch2>.0)
agent S3(s2,s3) = (^product)(s2(ch2).t.'ch2<product>.'s3.0)
agent S4(s3) = s3.t.0

agent R(x,p,m) = (^r1,r2,r3,r4,r5,r6)( R1(x,r1) | R2(r1,r2,r3) | R3(r2,p,r4) |
      R4(r3,m,r5) | R5(r4,r5,r6) | R6(r6) )
agent R1(x,r1) = x(ch1).ch1(ch2).ch1(order).t.'r1<ch1,ch2,order>.0
agent R2(r1,r2,r3) = r1(ch1,ch2,order).t.('r2<ch1,order>.0 | 'r3<ch2,order>.0)
agent R3(r2,p,r4) = r2(ch1,order).t.'p<ch1,order>.'r4.0
agent R4(r3,m,r5) = r3(ch2,order).t.'m<ch2,order>.'r5.0
agent R5(r4,r5,r6) = r4.r5.t.'r6.0
agent R6(r6) = r6.t.0
agent M(m) = (^m1,m2,product)( M1(m,m1) | M2(m1, product, m2) | M3(m2))
agent M1(m,m1) = m(ch,order).t.'m1<ch,order>.0
agent M2(m1, product, m2) = m1(ch,order).t.'ch<product>.'m2.0
agent M3(m2) = m2.t.0

agent P(p) = (^p1,p2,invoice)(P1(p,p1) | P2(p1,invoice,p2) | P3(p2))
agent P1(p,p1) = p(ch,order).t.'p1<ch,order>.0
agent P2(p1,invoice,p2) = p1(ch,order).t.'ch<invoice>.'p2.0
agent P3(p2) = p2.t.0
```

Part A. We can ask ABC for deciding interaction equivalence between the shop, represented by S and the reseller-construct, represented by R, P, and M:

```
abc > weq S(x) (^p,m)(R(x,p,m) | P(p) | M(m))
The two agents are not weakly related (8).
Do you want to see some traces (yes/no) ? no
```

Interestingly, the shop cannot be simply replaced by the reseller-construct, since they are not interaction equivalent. Further analysis showed indeed a different behavior. The shop always sends the invoice first followed by the product, whereas the reseller has non-deterministic behavior.

Part B. What can be proven, however, is an interaction simulation. Since the shop implements a part of the reseller behavior, the latter should be able to simulate the interactions of the former:

```
abc > wlt S(x) (^p,m)(R(x,p,m) | P(p) | M(m))
The two agents are weakly related (18).
Do you want to see the core of the simulation (yes/no) ? no
```

The reseller extends the possible behavior of the shop, so that whenever an interaction behavior as given by the shop is required, also the reseller can be used.

Part C. The opposite direction should not hold:

```
abc > wlt (^p,m)(R(x,p,m) | P(p) | M(m)) S(x)
The two agents are not weakly related (24).
Do you want to see some traces (yes/no) ? no
```

As expected, ABC proves the anticipations.

A.3 Unification

The figures A.7 and A.8 give an illustration of the examples described in chapter 7 (Unification). In this section we show the tool supported proofs for lazy and interaction soundness of the customer and interaction equivalence of the banks.

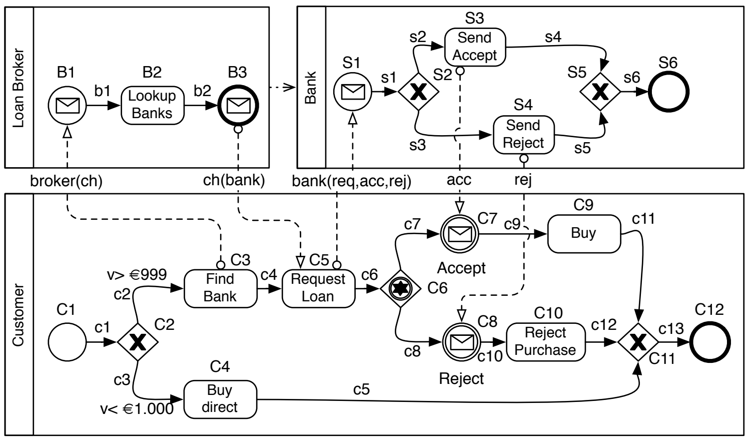


Figure A.7: Loan broker interaction.

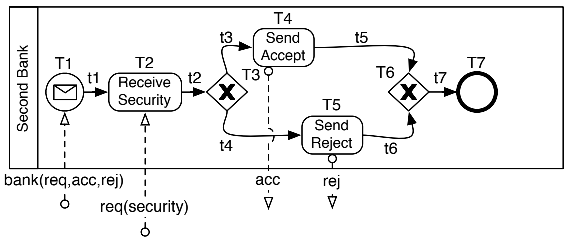


Figure A.8: Another bank for the loan broker interaction.

A.3.1 Lazy Soundness of the Customer

Example A.6 (Tool Supported Investigation of Proof 7.4 (Lazy Soundness of the Customer's Process Graph)) The lazy soundness annotated π-calculus mapping of the customer's process graph is given by:

```
agent C(i,o) = (^c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13)( C1(c1,i) | C2(c1,c2,c3) |
      C3(c2,c4) | C4(c3,c5) | C5(c4,c6) | C6(c6,c7,c8) | C7(c7,c9) | C8(c8,c10) |
      C9(c9,c11) | C10(c10,c12) | C11(c5,c11,c12,c13) | C12(c13,o) )
agent C1(c1,i) = i.t.'c1.0
agent C2(c1,c2,c3) = c1.t.('c2.0 + 'c3.0)
agent C3(c2,c4) = c2.t.'c4.0
agent C4(c3,c5) = c3.t.'c5.0
agent C5(c4,c6) = c4.t.'c6.0
agent C6(c6,c7,c8) = c6.t.('c7.0 + 'c8.0)
agent C7(c7,c9) = c7.t.'c9.0
agent C8(c8,c10) = c8.t.'c10.0
agent C9(c9,c11) = c9.t.'c11.0
agent C10(c10,c12) = c10.t.'c12.0
agent C11(c5,c11,c12,c13) = c5.t.'c13.0 + c11.t.'c13.0 + c12.t.'c13.0
agent C12(c13,o) = c13.t.'o.0

agent S_LAZY(i,o) = i.t.'o.0
```

Note that the abstraction from the deferred choice in agent *C6*. We can ask MWB for deciding weak open d-bisimulation on C and S_{LAZY}, thus deciding lazy soundness for the process graph of example 7.1 (Process Graph of the Customer):

```
MWB>weq S_LAZY(i,o) C(i,o)
The two agents are equal.
Bisimulation relation size = 32.
```

Since $C \approx_O^D S_{LAZY}$, the process graph of the customer is lazy sound.

A.3.2 Interaction Soundness of the Customer

Example A.7 (Tool Supported Investigation of Proof 7.6 (Interaction Soundness of the Customer with an Environment containing the First Bank)) The interaction soundness annotated system *I1* consisting of the customer, the loan broker, and the bank is given by:

```
agent BB(broker,broker_add) = broker_add(name,ch).( ((^rem)
      'ch<rem>.BB1(broker,name,rem)) | BB(broker,broker_add))
agent BB1(broker,name,rem) = broker(ch).('ch<name>.0 | BB1(broker,name,rem)) + rem.0

agent S_S(broker_add) = (^b,ch) 'broker_add<b,ch>.ch(rem).S(b)
agent S(b) = (^s1,s2,s3,s4,s5,s6) b(req,acc,rej).( S1(s1,req,acc,rej) | S2(s1,s2,s3) |
      S3(s2,s4) | S4(s3,s5) | S5(s4,s5,s6) | S6(s6) | S(b))
agent S1(s1,req,acc,rej) = t.'s1<acc,rej>.0
agent S2(s1,s2,s3) = s1(acc,rej).t.('s2<acc>.0 + 's3<rej>.0)
agent S3(s2,s4) = s2(acc).t.'acc.'s4.0
agent S4(s3,s5) = s3(rej).t.'rej.'s5.0
agent S5(s4,s5,s6) = s4.t.'s6.0 + s5.t.'s6.0
agent S6(s6) = s6.t.0

agent C(broker,i,o) = (^c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13)( C1(c1,i) |
      C2(c1,c2,c3) | C3(c2,c4,broker) | C4(c3,c5) | C5(c4,c6) | C6(c6,c7,c8) |
      C7(c7,c9) | C8(c8,c10) | C9(c9,c11) | C10(c10,c12) | C11(c5,c11,c12,c13) |
      C12(c13,o) )
agent C1(c1,i) = i.t.'c1.0
agent C2(c1,c2,c3) = c1.t.('c2.0 + 'c3.0)
agent C3(c2,c4,broker) = (^ch)c2.t.'broker<ch>.'c4<ch>.0
agent C4(c3,c5) = c3.t.'c5.0
agent C5(c4,c6) = (^req,acc,rej)c4(ch).ch(bank).t.'bank<req,acc,rej>.'c6<acc,rej>.0
agent C6(c6,c7,c8) = c6(acc,rej).t.(acc.'c7.0 + rej.'c8.0)
agent C7(c7,c9) = c7.t.'c9.0
agent C8(c8,c10) = c8.t.'c10.0
agent C9(c9,c11) = c9.t.'c11.0
agent C10(c10,c12) = c10.t.'c12.0
agent C11(c5,c11,c12,c13) = c5.t.'c13.0 + c11.t.'c13.0 + c12.t.'c13.0
agent C12(c13,o) = c13.t.'o.0

agent I1(i,o) = (^broker,broker_add)( BB(broker,broker_add) | S_S(broker_add) |
      C(broker,i,o))

agent S_LAZY(i,o) = i.t.'o.0
```

For technical reasons regarding MWB, we had to denote the agent *SS* as *S_S*. Interaction soundness for the service graph of the customer is decided by evaluating $I1 \approx_O^D S_{LAZY}$:

```
MWB>weq S_LAZY(i,o) I1(i,o)
The two agents are equal.
Bisimulation relation size = 151.
```

Since $I1 \approx_O^D S_{LAZY}$ holds, the service graph of the customer is interaction sound inside an environment consisting of the loan broker and the first bank.

Example A.8 (Tool Supported Investigation of Proof 7.7 (Disrupted Interaction Soundness of the Customer with an Environment containing the First and the Second Bank)) The interaction soundness annotated system $I2$ consisting of the customer, the loan broker, the first bank, and the second bank is given by:

```
agent BB(broker,broker_add) = broker_add(name,ch).( ((^rem)
      'ch<rem>.BB1(broker,name,rem)) | BB(broker,broker_add))
agent BB1(broker,name,rem) = broker(ch).('ch<name>.0 | BB1(broker,name,rem)) + rem.0

agent S_S(broker_add) = (^b,ch) 'broker_add<b,ch>.ch(rem).S(b)
agent S(b) = (^s1,s2,s3,s4,s5,s6) b(req,acc,rej).( S1(s1,req,acc,rej) | S2(s1,s2,s3) |
      S3(s2,s4) | S4(s3,s5) | S5(s4,s5,s6) | S6(s6) | S(b))
agent S1(s1,req,acc,rej) = t.'s1<acc,rej>.0
agent S2(s1,s2,s3) = s1(acc,rej).t.('s2<acc>.0 + 's3<rej>.0)
agent S3(s2,s4) = s2(acc).t.'acc.'s4.0
agent S4(s3,s5) = s3(rej).t.'rej.'s5.0
agent S5(s4,s5,s6) = s4.t.'s6.0 + s5.t.'s6.0
agent S6(s6) = s6.t.0

agent T_T(broker_add) = (^b,ch) 'broker_add<b,ch>.ch(rem).T(b)
agent T(b) = (^t1,t2,t3,t4,t5,t6,t7) b(req,acc,rej).( T1(t1,req,acc,rej) | T2(t1,t2) |
      T3(t2,t3,t4) | T4(t3,t5) | T5(t4,t6) | T6(t5,t6,t7) | T7(t7) | T(b))
agent T1(t1,req,acc,rej) = t.'t1<acc,rej,req>.0
agent T2(t1,t2) = t1(acc,rej,req).req(security).t.'t2<acc,rej>.0
agent T3(t2,t3,t4) = t2(acc,rej).t.('t3<acc>.0 + 't4<rej>.0)
agent T4(t3,t5) = t3(acc).t.'acc.'t5.0
agent T5(t4,t6) = t4(rej).t.'rej.'t6.0
agent T6(t5,t6,t7) = t5.t.'t7.0 + t6.t.'t7.0
agent T7(t7) = t7.t.0

agent C(broker,i,o) = (^c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13)( C1(c1,i) |
      C2(c1,c2,c3) | C3(c2,c4,broker) | C4(c3,c5) | C5(c4,c6) | C6(c6,c7,c8) |
      C7(c7,c9) | C8(c8,c10) | C9(c9,c11) | C10(c10,c12) | C11(c5,c11,c12,c13) |
      C12(c13,o) )
agent C1(c1,i) = i.t.'c1.0
agent C2(c1,c2,c3) = c1.t.('c2.0 + 'c3.0)
agent C3(c2,c4,broker) = (^ch)c2.t.'broker<ch>.'c4<ch>.0
agent C4(c3,c5) = c3.t.'c5.0
agent C5(c4,c6) = (^req,acc,rej)c4(ch).ch(bank).t.'bank<req,acc,rej>.'c6<acc,rej>.0
agent C6(c6,c7,c8) = c6(acc,rej).t.(acc.'c7.0 + rej.'c8.0)
agent C7(c7,c9) = c7.t.'c9.0
agent C8(c8,c10) = c8.t.'c10.0
agent C9(c9,c11) = c9.t.'c11.0
agent C10(c10,c12) = c10.t.'c12.0
agent C11(c5,c11,c12,c13) = c5.t.'c13.0 + c11.t.'c13.0 + c12.t.'c13.0
agent C12(c13,o) = c13.t.'o.0

agent I2(i,o) = (^broker,broker_add)( BB(broker,broker_add) | S_S(broker_add) |
      T_T(broker_add) | C(broker,i,o))

agent S_LAZY(i,o) = i.t.'o.0
```

Interaction soundness for the service graph of the customer is again decided by evaluating $I1 \approx_O^D S_{LAZY}$:

```
MWB>weq S_LAZY(i,o) I2(i,o)
The two agents are NOT equal.
```

Since $I2 \approx_O^D S_{LAZY}$ does not hold, the service graph of the customer is not interaction sound inside an environment consisting of the loan broker, the first bank, and the second bank.

A.3.3 Interaction Equivalence of the Banks

Example A.9 (Tool Supported Investigation of Proof 7.8 (Interaction Equivalence of the First and the Second Bank)) For proving interaction equivalence of the first bank and the second bank, we can re-use the agents defined in example A.8. Interaction equivalence is shown by deciding if S (the agent representation of the first bank) is weak open d-bisimulation equivalent to T (the agent representation of the second bank:

```
MWB>weq S(b) T(b)
The two agents are NOT equal.
```

Since $S \not\approx_O^D T$, the banks are not interaction equivalent.

A.3.4 Debugging Session

A modified version of example A.1 gives an impression of a ”debugging” session using ABC:

```
abc > weq N(i,o) S_LAZY(i,o)
The two agents are not weakly related (9).
Do you want to see some traces (yes/no) ? yes
traces of

N i o
S_LAZY i o

-i->
=i=>

(^x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)(x0.t.'o.#1 | x1.#2 | x2.'x9.0 | x3.'x9.0 |
x4.'x9.0 | x5.t.#3 | x6.t.#4 | x7.t.#5 | x8.t.#6 | x9.x9.#8 | x10.t.'x1.0 | t.'x8.0)
'o.0

-t->
=t=>

(^x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)('x8.0 | x0.t.'o.#1 | x1.#2 | x2.'x9.0 |
x3.'x9.0 | x4.'x9.0 | x5.t.#3 | x6.t.#4 | x7.t.#5 | x8.t.#6 | x9.x9.#8 | x10.t.'x1.0)
'o.0

-t->
=t=>

(^x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)(x0.t.'o.#1 | x1.#2 | x2.'x9.0 | x3.'x9.0 |
x4.'x9.0 | x5.t.#3 | x6.t.#4 | x7.t.#5 | x9.x9.#8 | x10.t.'x1.0 | t.#6)
'o.0

[...]

(^x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)(x0.t.'o.#1 | x1.#2 | x3.'x9.0 | x4.'x9.0 |
x5.t.#3 | x6.t.#4 | x7.t.#5 | x8.t.#6 | x9.#8 | x10.t.'x1.0)
'o.0
```

```
='o=>
-'o->

*
0

#1 ::= N1146 x0 o
#2 ::= (N1155 x1 x0 | 'x0.0 | t.0 | t.0 | t.0)
#3 ::= (N1144 x5 x2 | 'x2.0)
#4 ::= (N1145 x6 x3 | 'x3.0)
#5 ::= (N1143 x7 x4 | 'x4.0)
#6 ::= (N1138 x8 x5 x6 x7 | ('x5.0 + 'x6.0 + 'x7.0))
#7 ::= N1137 x2 x3 x4 x1
#8 ::= 'x10.x9.#7
```

Appendix B

Bibliography

[1] AALST, W.: *Verification of Workflow Nets.* In AZÉMA, P.; BALBO, G. (Eds.): *Application and Theory of Petri Nets 1997, volume 1248 of LNCS.* Springer Verlag, Berlin, 1997, pages 407–426

[2] AALST, W.: *The Application of Petri Nets to Workflow Management.* In *The Journal of Circuits, Systems and Computers* 8(1), 1998: pages 21–66

[3] AALST, W.: *Three Good Reasons for Using a Petri-net-based Workflow Management System.* In WAKAYAMA, T.; KANNAPAN, S.; KHOONG, C.; NAVATHE, S.; YATES, J. (Eds.): *Information and Process Integration in Enterprises: Rethinking Documents, volume 428 of The Kluwer International Series in Engineering and Computer Science.* Kluwer Academic Publishers, Boston, Massachusetts, 1998, pages 161–182

[4] AALST, W.: *Formalization and Verification of Event-driven Process Chains.* In *Information and Software Technology* 41(10), 1999: pages 639–650

[5] AALST, W.: *Inheritance of Workflow Processes: Four Problems - One Solution?.* In CUMMINS, F. (Eds.): *Proceedings of the Second OOPSLA Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems.* Denver, Colorado, 1999, pages 1–22

[6] AALST, W.; BASTEN, T.: *Inheritance of Workflows: An approach to tackling problems related to change.* Computing science reports 99/06, Eindhoven University of Technology, Eindhoven, 1999

[7] AALST, W.; DESSEL, J.; KINDLER, E.: *On the Semantics of EPCs: A Vicious Circle.* In NÜTTGENS, M.; RUM, F. (Eds.): *EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten.* Trier, 2002, pages 71–79

[8] AALST, W.; DUMAS, M.; HOFSTEDE, A.: *Pattern Based Analysis of BPEL4WS.* Technical report FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002

[9] AALST, W.; HEE, K.: *Workflow Management.* MIT Press, 2002

[10] AALST, W.; HEE, K.; HOUBEN, G.: *Modeling and Analysing Workflow using a Petri-net based Approach*. In DE MICHELIS, G.; ELLIS, C.; MEMMI, G. (Eds.): *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*. 1994, pages 31–50

[11] AALST, W.; HOFSTEDE, A.: *YAWL: Yet Another Workflow Language (Revised version)*. Technical report FIT-TR-2003-04, Queensland University of Technology, Brisbane, 2003

[12] AALST, W.; HOFSTEDE, A.; KIEPUSZEWSKI, B.; BARROS, A.: *Workflow Patterns*. In *Distributed and Parallel Databases* 14(1), 2003: pages 5–51

[13] AALST, W.; MOLDT, D.; VALK, R.; WIENBERG, F.: *Enacting Interorganizational Workflow Using Nets in Nets*. In BECKER, J.; MUEHLEN, M.; ROSEMANN, M. (Eds.): *Workflow Management '99*. University of Munster, 1999, pages 117–136

[14] AALST, W.; TER HOFSTEDE, A.; WESKE, M.: *Business Process Management: A Survey*. In AALST, W.; HOFSTEDE, A.; WESKE, M. (Eds.): *Business Process Management, volume 2678 of LNCS*. Springer Verlag, Berlin, 2003, pages 1–12

[15] AALST, W.; WESKE, M.: *The P2P Approach to Interorganizational Workflow*. In DITTRICH, K.; GEPPERT, A.; NORRIE, M. (Eds.): *Advanced Information Systems Engineering: 13th International Conference, CAiSE 2001, volume 2068 of LNCS*. Springer Verlag, Berlin, 2001, pages 140–156

[16] ABADI, M.; GORDON, A. D.: *A Calculus for Cryptographic Protocols: The Spi Calculus*. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*. ACM Press, New York, NY, USA, 1997, pages 36–47

[17] AGHA, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986

[18] ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V.: *Web Services: Concepts, Architectures and Applications*. Springer Verlag, Berlin, 2004

[19] ARPINAR, İ.; HALICI, U.; ARPINAR, S.; DOĞAÇ, A.: *Formalization of Workflows and Correctness Issues in the Presence of Concurrency*. In *Distributed and Parallel Databases* 7(2), 1999: pages 199–248

[20] BAETEN, J.: *A Brief History of Process Algebra*. In *Theoretical Computer Science* 335(2-3), 2005: pages 131–146

[21] BAETEN, J.; WEIJLAND, W.: *Process Algebra*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1990

[22] BALDONE, M.; BAROGLIO, C.; MARTELLI, A.; PATTI, V.: *A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments*. In DAM, A.; LAMERSDORF, W. (Eds.): *Service-Oriented Computing – ICSOC 2006, volume 4294 of LNCS*. Springer Verlag, Berlin, 2006, pages 339–351

[23] BARENDREGT, H. P.: *The Lambda Calculus*. Elsevier, Amsterdam, 1985

[24] BARROS, A.; DUMAS, M.; HOFSTEDE, A.: *Service Interaction Patterns*. In AALST, W.; BENATALLAH, B.; CASATI, F. (Eds.): *Business Process Management, volume 3649 of LNCS*. Springer Verlag, Berlin, 2005, pages 302–318

[25] BARROS, A.; DUMAS, M.; HOFSTEDE, A.: *Service Interaction Patterns: Towards a Reference Framework for Service-oriented Business Process Interconnections*. Technical report, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 2005

[26] BARROS, A.; DUMAS, M.; OAKS, P.: *A Critical Overview of the Web Services Choreography Description Language (WS-CDL)*. In *BPTrends Newsletter* 3(3), March 2005

[27] BASTEN, T.: *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1998

[28] BEA Systems, IBM, Microsoft, SAP, Siebel Systems: *Business Process Execution Language for Web Services Version 1.1 (BPEL4WS)*, May 2003. `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`

[29] BENATALLAH, B.; CASATI, F.; TOUMANI, F.: *Analysis and Management of Web Service Protocols*. In ATZENI, P.; CHU, W.; LU, H.; ZHOU, S.; LING, T. (Eds.): *23rd International Conference on Conceptual Modeling (ER 2004), volume 3288 of LNCS*. Springer Verlag, Berlin, 2004, pages 524–541

[30] BLOOM, B.; ISTRAIL, S.; MEYER, A. R.: *Bisimulation can't be Traced*. In *Journal of the ACM* 42(1), 1995: pages 232–268

[31] BOG, A.: *A Visual Environment for the Simulation of Business Processes based on the Pi-Calculus*. Master thesis, Hasso-Plattner-Institute, Potsdam, Germany, 2006

[32] BORDEAUX, L.; SALAÜN, G.: *Using Process Algebra for Web Services: Early Results and Perspectives*. In SHAN, M.; DAYAL, U.; HSU, M. (Eds.): *Technologies for E-Services, volume 3324 of LNCS*. Springer Verlag, Berlin, 2005, pages 54–68

[33] BORDEAUX, L.; SALAÜN, G.; BERARDI, D.; MECELLA, M.: *When are Two Web Services Compatible?*. In SHAN, M.; DAYAL, U.; HSU, M. (Eds.): *Technologies for E-Services, volume 3324 of LNCS*. Springer Verlag, Berlin, 2005, pages 15–28

[34] BOREALE, M.; BRUNI, R.; L. CAIRE AND, R. D.; LANESE, I.; LORETI, M.; MARTINS, F.; MONTANARI, U.; RAVARA, A.; SANGIORGI, D.; VASCONCELOS, V.; ZAVATTARO, G.: *SCC: A Service Centered Calculus*. In BRAVETTI, M.; NÚÑEZ, M.; ZAVATTARO, G. (Eds.): *Web Services and Formal Methods, volume 4184 of LNCS*. Springer Verlag, Berlin, 2006, pages 38–59

[35] BPMI.ORG: *Business Process Modeling Language*, 2002

[36] BPMI.ORG: *Business Process Modeling Notation*, 1. edition, May 2004.
`http://www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf`

[37] BRIAIS, S.: *ABC Bisimulation Checker*, 2003.
`http://lamp.epfl.ch/~sbriais/abc/abc.html`

[38] BROGI, A.; CANAL, C.; E.PIMENTEL; VALLECILLO, A.: *Formalizing Web Service Choreographies.* In *Proceedings of First International Workshop on Web Services and Formal Methods*. Electronic Notes in Theoretical Computer Science, Elsevier, 2004

[39] BROGI, A.; POPESCU, R.: *From BPEL Processes to YAWL Workflows.* In BRAVETTI, M.; NÚÑEZ, M.; ZAVATTARO, G. (Eds.): *Web Services and Formal Methods, volume 4184 of LNCS*. Springer Verlag, Berlin, 2006, pages 107–122

[40] BROOKES, S.; HOARE, C.; ROSCOE, A.: *A Theory of Communicating Sequential Processes.* In *Journal of the ACM* 31(3), 1984: pages 560–599

[41] BURBECK, S.: *The Tao of E-Business Services*, 2000.
`http://www-128.ibm.com/developerworks/library/ws-tao/`

[42] BUSI, N.; GORRIERI, R.; GUIDI, C.; LUCCHI, R.; ZAVATTARO, G.: *Choreography and Orchestration: A Synergic Approach to System Design.* In BENATALLAH, B.; CASATI, F.; TRAVERSO, P. (Eds.): *Service-Oriented Computing – ICSOC 2005, volume 3826 of LNCS*. Springer Verlag, Berlin, 2005, pages 228–240

[43] CANAL, C.; PIMENTEL, E.; TROYA, J. M.: *Compatibility and inheritance in software architectures.* In *Science of Computer Programming* 41(2), 2001: pages 105–138

[44] CARDELLI, L.; GORDON, A.: *Mobile Ambients.* In NIVAT, M. (Eds.): *Foundations of Software Science and Computation Structures, volume 1378 of LNCS*. Springer Verlag, Berlin, 1998, pages 140–155

[45] CERF, V.: *RFC 20, ASCII format for Network Interchange*, 1969.
`http://www.ietf.org/rfc/rfc20.txt`

[46] CHRISTENSEN, E.; CURBERA, F.; MEREDITH, G.; SANJIVA, W.: *Web Service Description Language (WSDL) 1.1.* IBM, Microsoft, March 2001. W3C Note,
`http://www.w3.org/TR/wsdl`

[47] COOK, W. R.; PATWARDHAN, S.; MISRA, J.: *Workflow Patterns in Orc.* In CIANCARINI, P.; WIKLICKY, H. (Eds.): *Coordination Models and Languages, volume 4038 of LNCS*. Springer Verlag, 2006, pages 82–96

[48] CURTIS, B.; KELLNER, M. I.; OVER, J.: *Process Modeling.* In *Communications of the ACM* 35(9), 1992: pages 75–90

[49] DAVULCU, H.; KIFER, M.; RAMAKRISHNAN, C.; RAMAKRISHNAN, I.: *Logic Based Modeling and Analysis of Workflows*. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM Press, 1998, pages 25–33

[50] DAYAL, U.; HSU, M.; LADIN, R.: *Organizing Long-Running Activities with Triggers and Transactions*. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. ACM Press, New York, 1990, pages 204–214

[51] DEHNERT, J.; RITTGEN, P.: *Relaxed Soundness of Business Processes*. In DITTRICH, K.; GEPPERT, A.; NORRIE, M. (Eds.): *anced Information Systems Engineering: 13th International Conference (CAiSE 2001), volume 2068 of LNCS*. Springer Verlag, Berlin, 2001, pages 157–170

[52] DEREMER, F.; KRON, H.: *Programming-in-the-Large versus Programming-in-the-Small*. In *IEEE Transactions on Software Engineering* SE-2(2), 1976

[53] DONG, Y.; SHEN-SHENG, Z.: *Approach for workflow modeling using π-calculus*. In *Journal of Zhejiang University Science* 4(6), 2003: pages 643–650

[54] EHRIG, H.; MAHR, B.; CORNELIUS, F.; GROSSE-RHODE, M.; ZEITZ, P.: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer Verlag, Berlin, 2. edition, 2001

[55] EMMERICH, W.; GRUHN, V.: *FUNSOFT nets: a Petri-net based software process modeling language*. In *IWSSD '91: Proceedings of the 6th international workshop on Software specification and design*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pages 175–184

[56] ENGBERG, U.; NIELSEN, M.: *A Calculus of Communication Systems with Label Passing*. Technical report DAIMI PB-208, University of Aarhus, 1986

[57] FARAHBOD, R.; GLÄSSER, U.; VAJIHOLLAHI, M.: *Specification and Validation of the Business Process Execution Language for Web Services*. In ZIMMERMANN, W.; THALHEIM, B. (Eds.): *Abstract State Machines 2004. Advances in Theory and Practice: 11th International Workshop (ASM 2004), volume 3052 of LNCS*. Springer Verlag, Berlin, 2004, pages 78–94

[58] FERRARA, A.: *Web Services: A Process Algebra Approach*. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. ACM Press, New York, NY, USA, 2004, pages 242–251

[59] FIELDING, R.: *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, USA, 2000.
`http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

[60] FISTEUS, J.; FERNÁNDEZ, L.; KLOOS, C.: *Formal Verification of BPEL4WS Business Collaborations*. In BAUKNECHT, K.; BICHLER, M.; PRÖLL, B. (Eds.): *E-Commerce and Web Technologies: 5th International Conference (EC-Web 2004), volume 3182 of LNCS*. Springer Verlag, Berlin, 2004, pages 76–85

[61] FOURNET, C.; GONTHIER, G.: *The reflexive CHAM and the join-calculus*. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 1996, pages 372–385

[62] FU, X.; BULTAN, T.; SU, J.: *Analysis of interacting BPEL web services*. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*. ACM Press, New York, NY, USA, 2004, pages 621–630

[63] GEORGAKOPOULOS, D.; HORNICK, M.; SHETH, A.: *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. In *Distributed and Parallel Databases* 3(2), 1995: pages 119–153

[64] GLABBEEK, R.; WEIJLAND, W.: *Branching Time and Abstraction in Bisimulation Semantics*. In *Journal of the ACM* 43(3), 1996: pages 555–600

[65] GORRIERI, R.; GUIDI, C.; LUCCHI, R.: *Reasoning About Interaction Patterns in Choreography*. In BRAVETTI, M.; KLOUL, L.; ZAVATTARO, G. (Eds.): *Formal Techniques for Computer Systems and Business Processes, volume 3670 of LNCS*. Springer Verlag, Berlin, 2005, pages 333–348

[66] GOTTSCHALK, K.: *Web Services Architecture Overview*, 2000. `http://www-128.ibm.com/developerworks/webservices/library/w-ovr/`

[67] GUIDI, C.; LUCCHI, R.: *Mobility Mechanisms in Service Oriented Computing*. In GORRIERI, R.; WEHRHEIM, H. (Eds.): *Formal Methods for Open Object-Based Distributed Systems, volume 4037 of LNCS*. Springer Verlag, Berlin, 2006, pages 233–250

[68] GUIDI, C.; LUCCHI, R.; GORRIERI, R.; BUSI, N.; ZAVATTARO, G.: *SOCK: A Calculus for Service Oriented Computing*. In DAM, A.; LAMERSDORF, W. (Eds.): *Service-Oriented Computing – ICSOC 2006, volume 4294 of LNCS*. Springer Verlag, Berlin, 2006, pages 327–338

[69] HADDAD, S.; POITRENAUD, D.: *Theoretical Aspects of Recursive Petri Nets*. In DONATELLI, S.; KLEIJN, J. (Eds.): *Applications and Theory of Petri Nets 1999, volume 1639 of LNCS*. Springer Verlag, Berlin, 1999, pages 228–247

[70] HINZ, S.; SCHMIDT, K.; STAHL, C.: *Transforming BPEL to Petri nets*. In AALST, W.; BENATALLAH, B.; CASATI, F. (Eds.): *Business Process Management, volume 3649 of LNCS*. Springer Verlag, Berlin, 2005, pages 220–235

[71] HOARE, C.: *Communicating Sequential Processes*. In *Communications of the ACM* 21(8), 1978: pages 666–677

[72] HOARE, C.: *Communicating Sequential Processes*. Prentice Hall, New York, 1985

[73] HOLLINGSWORTH, D.: *The Workflow Reference Model*. Technical report, Workflow Management Coalition, Hampshire, 1995.
`http://www.wfmc.org/standards/docs/tc003v11.pdf`

[74] HUHNS, M.; SINGH, M.: *Workflow Agents*. In *IEEE Internet Computing* July 1998: pages 94–96

[75] HÜNDLING, J.; WESKE, M.: *Web Services: Foundation and Composition*. In *EM - Electronic Markets Journal* 13(2), June 2003: pages 108–119.
`http://www.electronicmarkets.org/modules/pub/view.php/electronicmarkets-378`

[76] IBM: *Web Services Flow Language (WSFL 1.0)*, May 2001

[77] JENSEN, K.: *Coloured Petri Nets*. Springer Verlag, Berlin, 2. edition, 1997

[78] KELLER, G.; NÜTTGENS, M.; SCHEER, A.: *Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)"*. Technical report 89, Institut für Wirtschaftsinformatik, Saarbrücken, 1992

[79] KNOLMAYER, G.; ENDL, R.; PFAHRER, M.: *Modeling Processes and Workflows by Business Rules*. In AALST, W.; DESEL, J.; OBERWEIS, A. (Eds.): *Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of LNCS*. Springer Verlag, Berlin, 2000, pages 16–29

[80] KNUTH, D. E.: *The Art of Computer Programming*, Volume 1. Addison–Wesley, 3. edition, 1997

[81] LAKOS, C.: *From Coloured Petri nets to Object Petri nets*. In DE MICHELIS, G.; DIAZ, M. (Eds.): *Application and Theory of Petri Nets 1995, volume 935 of LNCS*. Springer Verlag, 1995, pages 278–297

[82] LANEVE, C.; ZAVATTARO, G.: *Foundations of Web Transactions*. In SASSONE, V. (Eds.): *Foundations of Software Science and Computational Structures, volume 3441 of LNCS*. Springer Verlag, Berlin, 2005, pages 282–298

[83] LEYMANN, F.; ROLLER, D.: *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, New Jersey, 2000

[84] MARTENS, A.: *On Compatibility of Web Services*. In *Petri Net Newsletter* 65, 2003: pages 12–20

[85] MARTENS, A.: *Analyzing Web Service based Business Processes*. In CERIOLI, M. (Eds.): *Fundamental Approaches to Software Engineering (FASE'05), volume 3442 of LNCS*. Springer Verlag, April 2005, pages 19–33

[86] MASSUTHE, P.; REISIG, W.; SCHMIDT, K.: *An Operating Guideline Approach to the SOA*. In *Annals of Mathematics, Computing & Teleinformatics* 1(3), 2005: pages 35–43

[87] MASSUTHE, P.; SCHMIDT, K.: *Operating Guidelines - an Automata-Theoretic Foundation for the Service-Oriented Architecture*. In *Proceedings of the 5th International Conference on Quality Software (QSIC'05)* 2005: pages 452–457

[88] MAZZARA, M.; LANESE, I.: *Towards a Unifying Theory for Web Service Composition*. In BRAVETTI, M.; NÚÑEZ, M.; ZAVATTARO, G. (Eds.): *Web Services and Formal Methods, volume 4184 of LNCS*. Springer Verlag, Berlin, 2006, pages 257–272

[89] Microsoft: *XLang Web Services for Business Process Design*, 2001

[90] MILNER, R.: *An Algebraic Definition of Simulation between Programs*. In *Proceedings of the 2nd International Joint Conference on Artifical Intelligence*. British Computer Society, 1971, pages 481–489

[91] MILNER, R.: *Flowgraphs and Flow Algebras*. In *Journal of the ACM* 26(4), 1979: pages 794–818

[92] MILNER, R.: *A Calculus of Communicating Systems*, Volume 94 of LNCS. Springer Verlag, 1980

[93] MILNER, R.: *Lectures on a Calculus for Communicating Systems*. In BROOKES, S.; ROSCOE, A.; WINSKEL, G. (Eds.): *Seminar on Concurrency: Carnegie-Mellon University Pittsburgh, volume 197 of LNCS*. Springer Verlag, Berlin, 1985, pages 197–220

[94] MILNER, R.: *Communication and Concurrency*. Prentice Hall, New York, 1989

[95] MILNER, R.: *Functions As Processes*. In PATERSON, M. (Eds.): *Automata, Languages, and Programming, volume 443 of LNCS*. Springer Verlag, 1990, pages 167–180

[96] MILNER, R.: *The polyadic π–Calculus: A tutorial*. In BAUER, F. L.; BRAUER, W.; SCHWICHTENBERG, H. (Eds.): *Logic and Algebra of Specification*. Springer Verlag, Berlin, 1993, pages 203–246

[97] MILNER, R.: *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, Cambridge, 1999

[98] MILNER, R.: *Bigraphical Reactive Systems*. In ACETO, L.; INGÓLFSDÓTTIR, A. (Eds.): *Foundations of Software Science and Computation Structures, volume 3921 of LNCS*. Springer Verlag, London, UK, 2001, pages 16–35

[99] MILNER, R.; PARROW, J.; WALKER, D.: *A Calculus of Mobile Processes, Part I/II*. In *Information and Computation* 100, September 1992: pages 1–77

[100] MOLDT, D.; VALK, R.: *Object Oriented Petri Nets in Business Process Modeling*. In AALST, W.; DESEL, J.; OBERWEIS, A. (Eds.): *Business Process Management, volume 1806 of LNCS*. Springer Verlag, Berlin, 2000, pages 254–273

[101] MOLDT, H., DANIEL UND RÖLKE: *Pattern Based Workflow Design Using Reference Nets*. In AALST, W.; HOFSTEDE, A.; WESKE, M. (Eds.): *Business Process Management, volume 2678 of LNCS*. Springer Verlag, Berlin, 2003, pages 246–260

[102] MULYAR, N.; AALST, W.: *Patterns in Colored Petri nets*. BETA Working Paper Series WP 139, Eindhoven University of Technology, Eindhoven, 2005

[103] NESTMANN, U.: *Welcome to the Jungle: A Subjective Guide to Mobile Process Calculi*. In BAIER, C.; HERMANNS, H. (Eds.): *CONCUR 2006 – Concurrency Theory, volume 4137 of LNCS*. Springer Verlag, Berlin, 2006, pages 52–63

[104] NEWCOMER, E.; LOMOV, G.: *Understanding SOA with Web Services*. Addison–Wesley, 2005

[105] OASIS: *UDDI Version 3.0.2*, October 2004

[106] OMG: *UML 2.0 Superstructure Final Adopted specification*, 2003

[107] PARK, D.: *Concurrency and Automata on Infinite Sequences*. In DEUSSEN, P. (Eds.): *Theoretical Computer Science: 5th GI-Conference , volume 104 of LNCS*. Springer Verlag, Berlin, 1981, pages 167–183

[108] PARROW, J.: *An Introduction to the π–Calculus*. In BERGSTRA, J.; PONSE, A.; SMOLKA, S. (Eds.): *Handbook of Process Algebra*. Elsevier, 2001, pages 479–543

[109] PARROW, J.; VICTOR, B.: *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 1998, pages 176–

[110] PETRI, C. A.: *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962

[111] ROSCOE, A.: *Theory and Practice of Concurrency*. Prentice Hall, 2005

[112] RUSELL, N.; HOFSTEDE, A.; AALST, W.; MULYAR, N.: *Workflow Control Flow Patterns: A Revised View*.
`http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-22.pdf`

[113] RUSSELL, N.; HOFSTEDE, A.; EDMOND, D.; AALST, W.: *Workflow Data Patterns*. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.
`http://is.tm.tue.nl/research/patterns/download/data_patterns%20BETA%20TR.pdf`

[114] RUSSELL, N.; HOFSTEDE, A.; EDMOND, D.; AALST, W.: *Workflow Resource Patterns*. BETA Working Paper Series WP 127, Eindhoven University of Technology, Eindhoven, 2004.

`http://is.tm.tue.nl/research/patterns/download/Resource%20Patterns%20BETA%20TR.pdf`

[115] SALAÜN, G.; BORDEAUX, L.; SCHAERF, M.: *Describing and Reasoning on Web Services using Process Algebra.* In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04).* IEEE Computer Society, Washington, DC, USA, 2004, pages 43–

[116] SANGIORGI, D.: *A Theory of Bisimulation for the Pi-Calculus.* In BEST, E. (Eds.): *CONCUR'93, volume 715 of LNCS.* Springer Verlag, Berlin, 1993, pages 127–142

[117] SANGIORGI, D.: *An Investigation into Functions as Processes.* In MAIN, M.; MELTON, A.; MISLOVE, M.; SCHMIDT, D. (Eds.): *Mathematical Foundations of Programming Semantics, volume 802 of LNCS.* Springer Verlag, 1994, pages 143–159

[118] SANGIORGI, D.; WALKER, D.: *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, Cambridge, paperback edition, 2003

[119] SCHLINGLOFF, B.; MARTENS, A.; SCHMIDT, K.: *Modeling and Model Checking Web Services.* In *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems* 126, 2005: pages 3–26

[120] STEFANSEN, C.: *Expressing Workflow Patterns in CCS*, 2005. PhD report,
`http://www.stefansen.dk/papers/workflowpatterns.pdf`

[121] STOERRLE, H.: *Semantics of Control-Flow in UML 2.0 Activities.* In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04).* IEEE Computer Society, Washington, DC, USA, 2004, pages 235–242

[122] TURING, A.: *On Computable Numbers, with an Application to the Entscheidungsproblem.* In *Proceedings of the London Mathematical Society* 2(42), 1936: pages 230–265

[123] TURNER, D. N.: *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, Edinburgh, 1995

[124] VALK, R.: *Self-Modifying Nets, a Natural Extension of Petri Nets.* In AUSIELLO, G.; BÖHM, C. (Eds.): *Automate, Languages, and Programming, volume 62 of LNCS.* Springer Verlag, Berlin, 1978, pages 464–476

[125] VICTOR, B.; MOLLER, F.; DAM, M.; ERIKSSON, L.-H.: *The Mobility Workbench*, 2005.
`http://www.it.uu.se/research/group/mobility/mwb`

[126] W3C: *Web Services Glossary*, 2004.
`http://www.w3.org/TR/ws-gloss/`

[127] W3C.org: *Web Service Choreography Interface (WSCI)*, 1. edition, August 2002.
`http://www.w3.org/TR/wsci/`

[128] W3C.org: *Web Service Choreography Description Language (WS-CDL)*, 1. edition, April 2004.
http://www.w3.org/TR/ws-cdl-10/

[129] WESKE, M.: *Deadlocks in Computersystemen*. Thomson Publishing, Bonn, 1995

[130] WESKE, M.: *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*. Habilitationsschrift, Fachbereich Mathematik und Informatik, Universität Münster, Münster, 2000

[131] WESKE, M.; VOSSEN, G.; PUHLMANN, F.: *Handbook on Architectures of Information Systems*, Springer Verlag, Berlin, Chapter Workflow Languages. 2. edition, 2005, pages 369–390

[132] WHITE, S. A.: *Introduction to BPMN*. Technical report, IBM, 2004.
http://bpmn.org/Documents/Introduction%20to%20BPMN.pdf

[133] WHITE, S. A.: *Process Modeling Notations and Workflow Patterns*. Technical report, IBM, 2004.
http://www.bpmn.org/Documents/Notations%20and%20Workflow%20Patterns.pdf

[134] WOHED, P.; AALST, W.; DUMAS, M.; HOFSTEDE, A.; RUSSELL, N.: *On the Suitability of BPMN for Business Process Modelling*. In DUSTDAR, S.; FIADEIRO, J.; SHETH, A. (Eds.): *Business Process Management, volume 4102 of LNCS*. Springer Verlag, Berlin, 2006, pages 161–176

[135] WONG, P. Y.; GIBBONS, J.: *A Process Algebraic Approach to Workflow Verification*, 2006. Unpublished report,
http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/pattern.pdf

[136] WOODLEY, T.; GAGNON, S.: *BPM and SOA: Synergies and Challenges*. In NGU, A.; KITSUREGAWA, M.; NEUHOLD, E.; CHUNG, J.; SHENG, Q. (Eds.): *Web Information Systems Engineering – WISE 2005: 6th International Conference on Web Information Systems Engineering, volume 3806 of LNCS*. Springer Verlag, Berlin, 2005, pages 679–688

[137] WYNN, M.; EDMOND, D.; AALST, W.; HOFSTEDE, A.: *Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets*. In CIARDO, G.; DARONDEAU, P. (Eds.): *Applications and Theory of Petri Nets 2005, volume 3536 of LNCS*. Springer Verlag, Berlin, 2005, pages 423–443