A Visual Environment for the Simulation of Business Processes based on the Pi-Calculus

— Masterarbeit —

vorgelegt an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam,
Hasso Plattner Institut

von
Anja Bog

Potsdam
19. November 2006

Acknowledgements

I would like to express my sincere gratitude to my advisor, Frank Puhlmann, for providing me with continuous, very helpful comments and corrections of the theoretical part of this work, as well as improvements and suggestions during the development of the prototypical implementation.

I would like to thank Professor Mathias Weske for offering constructive feedback and support for my work. Finally, I would also like to thank the remaining members of the Business Process Technology research group of the Hasso Plattner Institute (HPI), to whom I have presented the concepts of the system and demonstrated the prototype, for their valuable remarks.

Potsdam, 19th November 2006 A.B.

Zusammenfassung

Der π-Kalkül stellt eine ernstzunehmende Alternative zu bestehenden Ansätzen zur formalen Repräsentation und Modellierung von Geschäftsprozessen dar. Im Gegensatz zu traditionellen formalen Modellierungstechniken, wie Petri Netzen und den auf Petri Netzen basierenden Weiterentwicklungen zur Unterstützung von Prozessmustern (z.B. Workflow Netze, YAWL), bietet der π-Kalkül zusätzlich die Möglichkeit zur Modellierung von Systemen mit dynamischen Strukturen. In solchen Systemen können sich sowohl die teilnehmenden Akteure, als auch die Kommunikationsverbindungen zwischen den Akteuren ändern. Im Hinblick auf die zunehmende Bedeutung von dienstbasierten Softwarearchitekturen, d.h. Systemen mit dynamischen Strukturen, steigt auch das Interesse an Modellierungstechniken für Prozesse, welche eben diese Eigenschaft darstellen können, und rückt den π-Kalkül weiter in den Vordergrund.

Diese Masterarbeit beschäftigt sich mit der Entwicklung einer Softwareanwendung zur Ausführung und Beobachtung von Geschäftsprozessen basierend auf dem π-Kalkül. Aufbauend auf Erläuterungen zu den Grundlagen von Geschäftsprozessen, deren Modellierung und Geschäftsprozessmanagement, sowie dem π-Kalkül werden die Konzepte zur Entwicklung einer Simulationsumgebung für π-Kalkül Prozesse beschrieben. Diese Konzepte umfassen sowohl die Gedanken zum Entwurf eines Abwicklers zur Ausführung von π-Kalkül Prozessen, als auch zu deren graphischer Darstellung. Der Abwickler benötigt eine interne Datenstruktur zur Speicherung und Verarbeitung der Prozesse. Diese Datenstruktur, sowie entsprechende Regeln für die Abarbeitung der Datenstruktur werden definiert.

Neben der Abwicklung von π-Kalkül Prozessen sollen dem Benutzer der Simulationsumgebung interaktive Möglichkeiten zur Einflussnahme auf die Weiterentwicklung des beobachteten Systems gegeben werden. Interaktionen manifestieren sich beispielsweise in der Auswahl des nächsten durchzuführenden Kommunikationsschrittes oder dem Hinzufügen von neuen Akteuren, die sofort mit bereits bestehenden Akteuren kommunizieren können. Eine intuitive Benutzungsweise, welche es dem Benutzer ermöglicht direkt mit Hilfe der graphischen Repräsentation zu interagieren, wird dabei angestrebt. Bestehende Ansätze zur graphischen Darstellung von π-Kalkül Prozessen werden analysiert, bzw. entsprechend erweitert, um eine adäquate graphische Darstellung für die Simulationsumgebung zu erhalten und genügend Informationen zur Interaktion mit dem Benutzer bereitzustellen.

Der Rahmen dieser Arbeit umfasst außerdem eine prototypische Implementierung der erarbeiteten Konzepte. Die Anforderungen an den Prototypen der Simulationsumgebung und die daraus entstehende Architektur, sowie deren Umsetzung werden beschrieben. Das Aussehen und die Funktionsweise des Prototyps werden anschließend anhand eines Beispiels präsentiert.

Abstract

The π-calculus poses as a serious alternative to existing approaches for formal representation and modeling of business processes. In contrast to traditional formal modeling techniques, like Petri nets and enhancements of Petri nets for process pattern support (e.g. Workflow nets, YAWL), the π-calculus additionally offers the possibility to model systems with dynamic structures. In such systems, the actors as well as the communication links between these actors are subject to changes, e.g. omission of existing actors or addition of new actors and communication links, respectively. With regard to the increasing relevance of service oriented architectures, i.e. systems with dynamic structures, the interest in modeling techniques for processes being able to describe these properties rises and lets the π-calculus come to the fore.

This thesis deals with the development of a software application for the execution and observation of business processes based on the π-calculus. Starting from explanations of the basic principles of business processes, business process modeling, business process management and the π-calculus, the concepts for the development of a simulation environment for π-calculus processes are described. These concepts comprise both, ideas for the design of an execution engine for π-calculus processes as well as their graphical representation. The execution engine requires an internal data structure for storing and handling the processes. This data structure and appropriate rules for processing the data structure are defined.

Besides the execution of π-calculus processes, the user of the simulation environment is offered interactive means for exerting influence on the further evolution behavior of the observed system. Interactions become manifest, for example, in choosing the next communication step or adding new actors to the system that are immediately able to communicate with already existing actors. An intuitional use, which allows a user to directly manipulate the graphical representation, is striven for. Existing approaches for graphically representing π-calculus processes are analyzed. One of them is extended to obtain an adequate graphical representation for the simulation environment and to provide sufficient information for user interaction.

Moreover, the scope of this work contains a prototypical implementation of the elaborated concepts. The requirements for the prototype of the simulation environment and the resultant architecture as well as its implementation are described. The appearance and functionality of the prototype are presented by means of an example, subsequently.

Contents

List of Figures

Chapter 1

Introduction

Business processes are a collection of consecutive, alternative and parallel units of work with the objective to create value to customers or the organization they are part of. In order to accomplish their goals, business processes communicate with other processes that may belong to the same organization or to another one. Therefore, business processes are part of communicating systems. Nowadays, with regard to the development of service oriented architectures (SOA) [31] and the demand for the business processes of an enterprise to adapt to changes continuously, the communication links between business processes are by no means permanent. They are subject to perpetual change, for example, as the effect of utilizing external services offered by a service registry to extend the functionality of the own system. In this case the communication links between the enterprises business process using the service and the other enterprise offering the service are established dynamically. On this account, especially for showing the mobile aspect of business processes, the π-calculus is proposed as a representation for business processes.

The π-calculus is a formal algebra for concurrent, communicating and mobile systems. It provides a framework for the representation and analysis of these systems. Concurrent processes within such a system communicate with each other via channels sending and receiving messages. The contents of messages are names that again represent channels and can therefore be used for further communication. Thus, new links between processes within a π-calculus system can be created during execution, which constantly changes the circuits in the system and thereby make it mobile.

1.1 Motivation

As already mentioned above, by using the π-calculus as a formal representation of business processes, the dynamic linking behavior of processes within service oriented architectures can be modeled. Other formal approaches for modeling business processes exist, e.g. Petri nets [44] and Workflow nets [9]. Nevertheless, the problem of these approaches is that they represent systems with static structures, which means that all links for communications and all actors within the system have to be known beforehand at design time. Already a simple example, found in everyday life, illustrates the problem.

Lets assume Steve wants to communicate with Mary using a telephone, but does not know Mary's telephone number. Without her number, no communication can be established. To resolve this problem Steve calls the directory assistance, whose telephone number is publicly known, and asks for Mary's number. The directory assistance can provide him with her telephone number, since Mary has signed up in the telephone book. The state of the system is depicted in Figure 1.1(a). After receiving the number, Steve is able to call Mary. Using the received number he establishes a new communication link between himself and Mary, which did not exist before. The new state of the linking structure is shown in Figure 1.1(b).

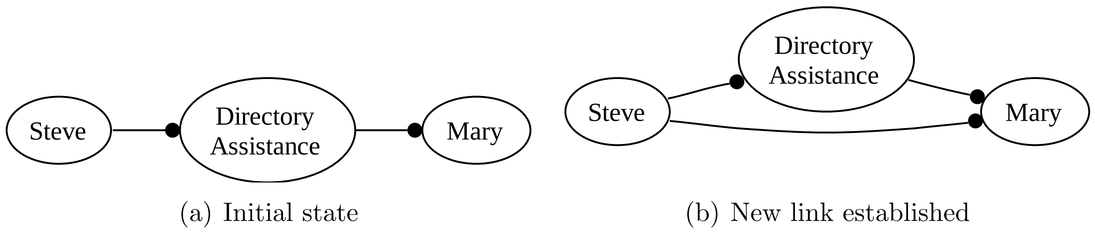


Figure 1.1: Change of linking structure during evolution

Such evolution can not be modeled by Petri or Workflow nets. They represent static structures and do not provide means to model the addition, omission, or change of the source or target of communication links between actors during the evolution of the system. According to service oriented architectures even actors themselves do not have to be part of a system permanently, but can be added to the system or leave it at any time. As an example, a new company registers an own service to a public service registry and thereby makes it available to other companies. Thus, an additional actor - the new company providing the service - has been added to the system.

However, a problem is the lacking tool support for π-calculus simulation. Therefore, this thesis presents essential concepts needed for the implementation of an environment for the interactive simulation of orchestrations and choreographies of business processes based on the π-calculus. Orchestrations and choreographies have been defined by the W3C Working Group in [27] as follows:

> "An orchestration defines the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function.

> I.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal."
>
> "A choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state."

A definition of Web services can also be found in [27]. Orchestrations have a single point of control, in contrast to choreographies not having any point of control. Thus orchestrations refer to the operating sequence and conditions of work units within a single business process. Choreographies describe the interactions between different business processes probably belonging to different companies.

A prototypical implementation developed as part of this work serves as proof of the elaborated concepts. In contrast to existing approaches, the π-calculus is utilized to support link passing mobility required in service oriented architectures. The prototypical implementation "PiVizTool" provides an interactive graphical representation of complex orchestrations and choreographies with changing structures. It enables the exertion of direct influence on the steps of execution taken by the simulated business processes.

1.2 Outline & Contribution

This thesis is structured as follows. The second chapter lays out the basics needed for constructing a simulation environment for business processes based on the π-calculus. Therefore, business processes, business process management and modeling are explained. The π-calculus including its syntax for defining processes as well as its rules for executing processes is introduced. Additionally, some further existing work and tools addressing the π-calculus, e.g. concerning its reasoning capabilities for comparing different systems and checking soundness criteria, or regarding visualization concepts of π-calculus systems are presented. In this chapter an example business process that helps explaining the concepts needed for a visual simulation environment for π-calculus systems is introduced. Parts of this example or simplified versions are used throughout this work.

The third chapter starts with an analysis of the functional requirements for a simulation environment referring to different points of view, as for example the user interaction view or the simulation view. Tools for aiding the implementation of the prototype are presented and concluding this chapter a possible architecture for the prototype is proposed and explained.

In the 4th chapter the underlying concepts of the simulation environment are described. The first part focuses on ideas with respect to the simulation engine realizing the execution of π-calculus systems. Fundamental issues like an appropriate data structure for storing the π-calculus process definitions that can further on be used for their execution and rules producing the π-calculus evolution behavior fitted to the proposed data structure are examined. The second part deals with the visualization concepts comprising an extension to an existing visualization approach of π-calculus systems, which is needed for the interaction with users. Implementation details are given to both parts including an overview of the static structure of the implementation.

The 5th chapter presents information about the realization of the user interface and provides screenshots for a deeper insight. Additionally, the example as it was introduced in the second chapter is taken up in its entirety and the evolution of the according π-calculus system is shown by means of a selection of intermediate steps that are graphically represented. This example finally gives an overview of the elaborated concepts working together to obtain an environment for the simulation and visualization of π-calculus systems.

Finally a conclusion summarizes the realized results of this work and gives an overview of some ideas for potential further research topics in this field.

Chapter 2

Preliminaries

This chapter introduces the basic vocabulary and theoretical concepts needed for developing the desired simulation environment for business processes based on the π-calculus. The chapter is divided into two parts. The first part takes care of terms related to business processes, like business process management and business process modeling. The aspect of modeling business processes will be accompanied by the introduction of an example process that will be picked up throughout this work in differing variants. This part will furthermore reflect on the issues that bring on the π-calculus as a foundation for business processes and presents a possibility to create π-calculus systems out of business process models.

The second part establishes a basic knowledge of the π-calculus including its syntax, the interpretation of its constructs, as well as their behavior during the evolution of a π-calculus system and a theory called the reduction semantics specifying how π-calculus systems may evolve. Beyond that an overview of related work, e.g. tools for reasoning with π-calculus systems and research towards visual representations of π-calculus systems will be given. Finally, the idea of creating π-calculus systems out of business process models, mentioned above, will be delved into.

2.1 Business Processes

Work within an enterprise is organized in work items or tasks, which are fulfilled by different employees, machines, or external services. Generally, these work items are accomplished in correlation to other work items being worked on before, after, or in parallel, maybe depending on them. Thereby, a structure is established relating the tasks in causal connections, e.g. the effects or products of one task are the inputs of another and hence the second task should be conducted after the first. These structures are not subject to constant changes. For example, the tasks within a production chain remain relatively stable over a certain amount of time. In [52](p. 144), those structures are called processes, being defined "[. . .] as anything that displays consistency of structure over time."

Different types of processes within an organization can be distinguished. These are management, operational and supporting processes. Management processes are those processes running the production and that obey the organization's requirements, e.g. its strategies, values and culture. Operational processes are those generating an organization's products or contributing to its service provisioning, directly creating value for its customers. Supplying processes are all other processes that do not directly create value for the customers, but are aiding to a smooth functioning of the operational and management processes. Based on this, business processes are mostly the operational processes of an enterprise.

Business Processes according to [50] are a collection of activities, that create value to the organization or to people as the organization's stakeholders or customers. Value is created by transforming raw materials, manufacturing goods, offering services, knowledge, or human effort. In general, business processes can be thought of as a structured chain of activities, transforming inputs into commercially useful outputs. Business processes are not a new concept, but are implicitly present since the beginning of business and commerce [56]:

> "They are the work, and how the work gets done; they exist quite independently of any technology."

2.1.1 Business Process Management

With the existence of business processes comes the need to manage these processes. Managing in this case refers to the activities undertaken by organizations to design, analyze, optimize, and adapt their processes. These activities are summarized under the term Business Process Management (BPM). See [15, 25, 29] for more detailed knowledge on BPM. Business process management comprises different phases in its lifecycle [6] to fulfill its tasks. These phases are depicted in Figure 2.1.

The four phases of the BPM lifecycle are process design and analysis, system configuration, process enactment, and diagnosis. In the design and analysis phase the business processes are modeled. Because quality of the business processes has a large influence on the performance of the enterprise utilizing them [32], a thorough design is invaluable. Design of business processes is supported by various modeling notations making a contribution to easier understanding and giving a better overview during construction. Several possibilities to model business processes will be discussed in section 2.1.2.

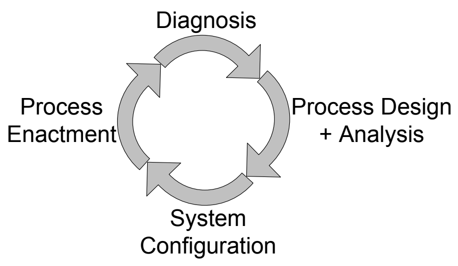


Figure 2.1: BPM lifecycle [6]

Different types of analysis exist for business process analysis. These are verification, validation, and simulation. Verification is the act of proving or disproving the correctness of a modeled business process with respect to a formal specification. For example the checking of a business process regarding structural soundness criteria belongs to this kind of analysis. Validation comprises the process of checking if a designed business process has the intended outcomes, i.e. if it actually does what it is intended to do according to its specification. Simulation aims at observing the behavior of a modeled business process, e.g. the sequence of processing certain work units according to special conditions, without actually implementing it in the real environment. Thus, behavioral problems within a process, or for example bottlenecks concerning resource allocation, might be detected and compensated for at design time before the process is put to work and actually starts to consume costly resources. Due to developing an environment for virtually enacting business processes and monitoring their behavior, this work is focused on the area of simulation.

After the design and analysis phase, the designs have to be implemented, which is done by configuring a business process management system (BPMS). BPMS, as stated in [17](p. 153), can be considered

> "[...] as the next evolution of WfMS. BPMS adds robust application integration, application development, process analysis, and richer process simulation and modeling capabilities that traditional workflow systems are lacking."

WfMS stands for Workflow Management Systems and they "allow organizations to define and control the various activities associated with a business process" [18]. [17](p. 153) distinguishes workflows from business processes by defining workflow as "[...] automation of a business process, that involves multiple participants, and results in documents and tasks to be passed from one participant to another" and specifying a business process as "[...] a standardized and coordinated flow of activities, performed by humans or machines, which can cross functional or departmental boundaries to create value to customers." As can be seen, business processes are centered around the customer including process communications across the boundaries of enterprises, instead of workflows that may reflect any process cycle with information or material flow within one enterprise. More information on workflow management can be found in [8, 19, 58].

In the next phase, process enactment, the business processes are executed on the configured system. During execution data is collected, which will be used in the diagnosis phase to analyze the executed processes and identify problems and bottlenecks that can be eliminated by a redesign of the corresponding process in the succeeding design phase. The BPM lifecycle ensures that processes can continually be changed, optimized, and adapted to new requirements, which is one important aspect for keeping enterprises competitive in an ever changing environment.

In the following section the design phase including the modeling of business processes will be further examined.

2.1.2 Business Process Modeling

The objective of business process modeling is to design new and redesign existing business processes during build time. In [7] the authors argument that "Business process management systems allow organizations to change their processes by merely changing their models." Since most modeling notations offer graphical elements to model processes, understanding of processes can be achieved easily, which is vitally important for applying useful changes to the processes. The modeling and remodeling of processes is done in the design phase. Remodeling bases on data collected during the previous execution phases that are categorized and evaluated during the diagnosis phase.

A traditional approach to model business processes utilizes Petri nets invented by Carl Adam Petri [44]. Notations based on Petri nets, like Workflow nets [9] and Yet Another Workflow Language(YAWL) [11], add support for process patterns [10]. The advantages of these approaches is that they are based on a formal specification. Therefore, the models have a defined interpretation that does not depend on the modeler or certain context information. Especially Workflow nets and YAWL provide a broad support for process patterns. However, a drawback of Workflow nets and YAWL is that they do not support the modeling of choreographies, since communication within a process of a company can not be differentiated from communication between processes of different companies. Although choreographies can be modeled with Petri nets, the problem of this approach is the lacking support for process patterns.

Other modeling approaches very common are the Unified Modeling Language (UML) [13] providing state charts and activity diagrams for process modeling and the Business

Process Modeling Notation (BPMN) [16]. Both notations have a semi-formal specification leaving room for interpretation and resulting in further effort producing additional specifications needed for the execution of the models. The BPMN specification, for example, provides mappings between the graphics of the notation to the constructs of an underlying execution language, e.g. Business Process Execution Language for Web Services (BPEL4WS) [12]. Furthermore, UML and BPMN allow for the modeling of orchestrations as well as choreographies and provide a fair support for modeling the process patterns.

In contrast to UML, BPMN offers a large variety of different elements for modeling processes, e.g. lots of special gateways for controlling the sequence flow, or several different event types. Therefore, the main concepts of modeling business processes will be explained by means of BPMN in the following. These concepts especially show what components business processes are made of and how they can be connected to each other, e.g. causal dependencies, and be influenced by each other during the process flow.

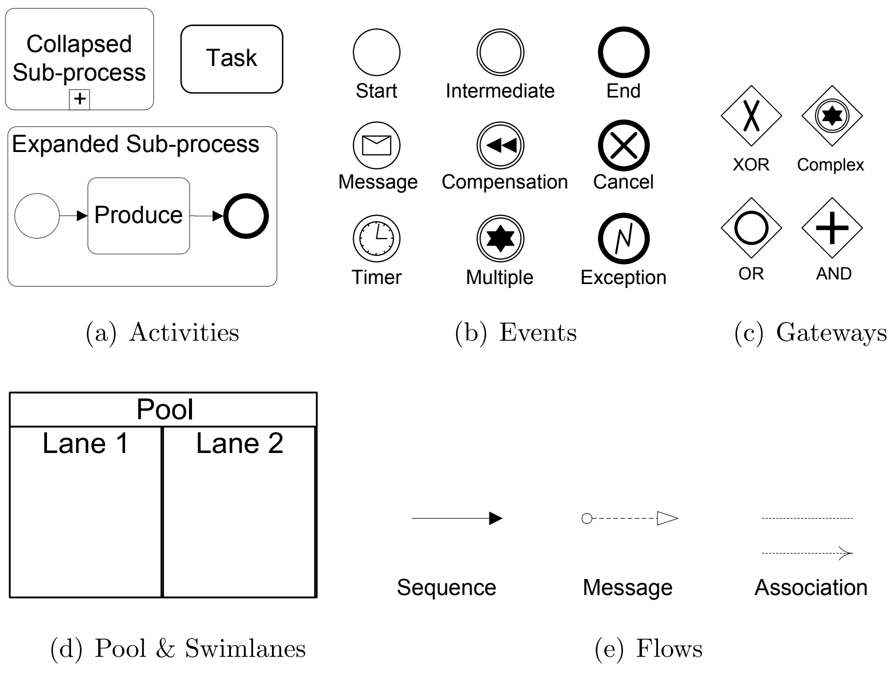
Core elements of business processes as defined in the BPMN specification [16] are flow objects, connecting objects, swimlanes, and artifacts. Examples for the elements of BPMN are depicted in Figure 2.2. A complete outline of the elements BPMN contains can be found in its specification.

Flow objects are activities, events, and gateways. They define the behavior of a business process. Activities are a companies work items or tasks that are performed by humans or machines. They are either atomic or compound. Compound activities are those that can be decomposed into other activities.

Events can happen during the course of a business process and have a trigger. Triggers for events can, for example, be incoming messages, depicted by a letter symbol, timers, shown by a clock symbol, defined rules, or multiple kinds of them, where the occurrence of one of them triggers the event. Events are distinguished into three categories depending on their position of occurrence within the process. If they occur at the beginning of a process, they are called start events and are marked by a single circle. Events occurring within are called intermediate events, denoted by a double circle and events occurring at the end have a bold circle surrounding them.

Gateways control the splitting and joining of sequence flows. Depending on the type of the gateway all paths (parallel/AND gateway), exactly one path (exclusive choice/XOR gateway), or multiple paths (inclusive/complex gateway) are taken starting from this gateway or are joined at this gateway.

Three types of connecting objects are defined, which are, as in the order depicted in Figure 2.2(e) from left to right, sequence flows, message flows, and associations. Se-



(a) Activities (b) Events (c) Gateways

(d) Pool & Swimlanes (e) Flows

Figure 2.2: BPMN core elements

quence flows specify the order of activities within a process and message flows show the information exchange between two participants. Participants being able to communicate via message flow are represented by pools in BPMN meaning that no activities within one pool may communicate with each other via message flow. This kind of communication is reserved for communication of activities of different pools, where sequence flows are not applicable. Associations attach information to flow objects in form of artifacts. As already stated, pools represent participants in a business process, e.g. two companies communicating with each other. They are a container for the activities taking place within each of the companies. Lanes are a further refinement of pools, for example, dividing the pool representing a company.

Artifacts provide additional information about a process and are differentiated into data objects, groups, and text annotations. Data objects offer information about data that are produced or needed by a certain activity. Groups, for example, associate activities regarding some characteristics they have in common, and text annotations provide easier readability of a business process diagram.

Using these elements, business processes can be modeled as for example the model in Figure 2.3 shows. Depicted are the processes of four independent communication partners in different pools: a customer, reseller, manufacturer and payment organization. The customer process is composed of the three activities “Order”, “Receive Invoice” and “Receive Product” with the last two activities being executable in parallel as indicated by the AND split gateway. The customer process ends after both activities have completed, which is denoted by the AND join gateway waiting for their completion. The reseller, manufacturer, and payment organization processes are all triggered by incoming messages. A customer, for example, may send an email to a reseller, with the contents specifying ordered items. After sending this order, the customer waits for the ordered items and the invoice. Upon receiving the order, the reseller proceeds to handling the request by running the activities of triggering a manufacturer to produce the desired items and prompting a payment organization to take care of the customer's payment concerning the desired items. The manufacturer produces the items and sends them back to the customer. The payment organization creates an invoice for the ordered items and also sends it to the customer.

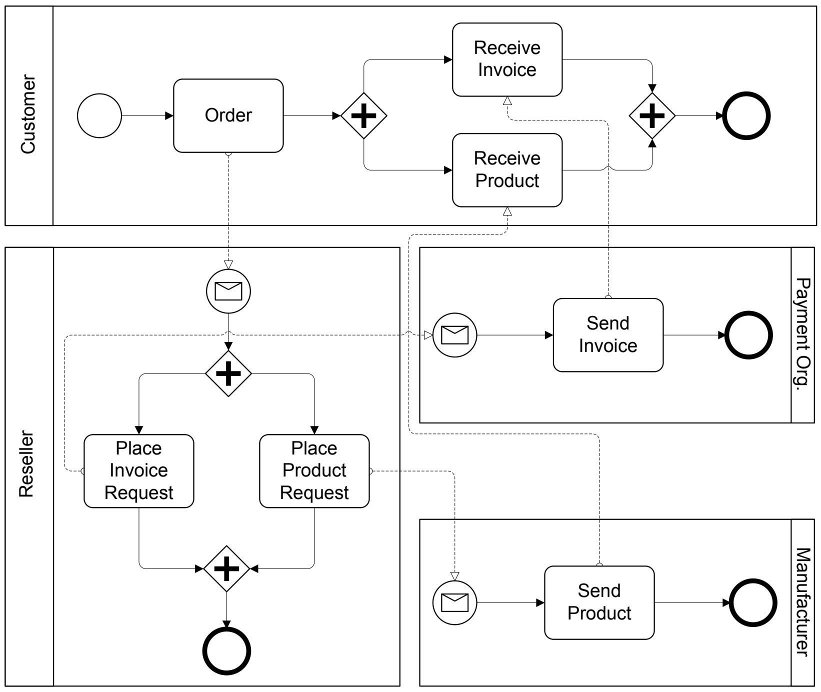


Figure 2.3: Reseller Example modeled in BPMN

Problems arising from this model are that no accurate specifications regarding the flow of the customer information or the number of actual instances of processes and their connections with each other are inferable. The first issue aims, for example, at clearly defining how information like the customer's address as the address where he wants to receive the products travels from the initial order to the manufacturer process. Seemingly, the manufacturer process has a stable link to the customer process for sending messages from the beginning on, even before the customer tells the reseller about his order. There is no indication that this link is established between a concrete manufacturer and customer during the process. The second point challenges exactly this aspect that BPMN only shows processes at the type level. During actual execution many reseller processes with a structure as described in the business process diagram may exist forwarding orders of various customers to a variety of manufacturers and payment organizations. Links between the last are established only between the specific instances. Such dynamic linking behavior can be modeled using the π-calculus. As a conclusion, the models of BPMN show the static structure of processes with all tasks and links that occur during process execution, and π-calculus processes show the structure evolving over time with snapshots of the systems structure at specific moments.

2.1.3 Business Processes and the Pi-Calculus

As the π-calculus is discussed as a formal foundation for BPM [47, 55], and the target of this thesis is to develop an environment for simulating π-calculus systems, one problem surfaces that has to be solved. It regards the modeling of business processes in π-calculus notation, which is needed as input for the simulation environment. Directly modeling processes in this notation is not easy, since no graphical editors exist and directly writing the process definitions can become very confusing. This problem is currently targeted in the context of the development of a tool chain for the π-calculus [46]. A high level overview of the architecture of the tool chain modeled as a block diagram of the FMC (Fundamental Modeling Concepts) notation [30] is depicted in Figure 2.4. Rectangles in this notation present active components, which are capable of communicating with each other. Rounded shapes represent storages and documents that can be read or written to.

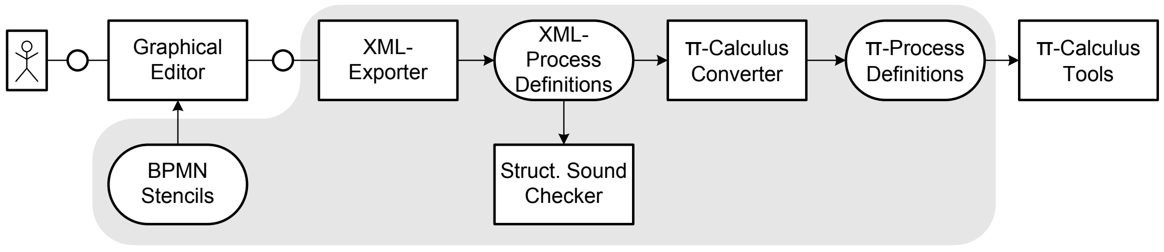


Figure 2.4: Tool chain [46]

The objectives of this tool chain are the application of π-calculus reasoning on business processes modeled with BPMN. Components of the tool chain are a graphical editor for designing business process diagrams (BPD) in BPMN, an XML exporter, a structural soundness checker, and a BPMN to π-calculus converter. The editor uses a special set of BPMN stencils that is annotated with additional information. With the help of these annotations the business process diagrams modeled by a user are translated to an intermediate XML format by the XML exporter component. The resulting XML process definitions can be checked for structural soundness of the BPD. A BPD is structurally sound if it contains exactly one start event without incoming sequence flows or outgoing message flows and exactly one end event with no outgoing sequence flows or incoming message flows. In case of the BPD being structured into pools, this restriction has to apply to each pool. Furthermore, the BPD should not contain any "dangling" activities or edges. Such activities are those that are not on a path from the start to the end event. Dangling edges are edges that lack a source or target node. A structurally sound diagram will be translated to π-calculus process definitions by the converter. The components within the shaded area are developed as part of the tool chain.

The implementation of such a converter from BPMN to the π-calculus provides a convenient way for the creation of π-calculus processes representing business processes that can be used as input for the simulation environment. Some more details about the converter and the resulting process definitions will be given in the following section after the π-calculus has been introduced.

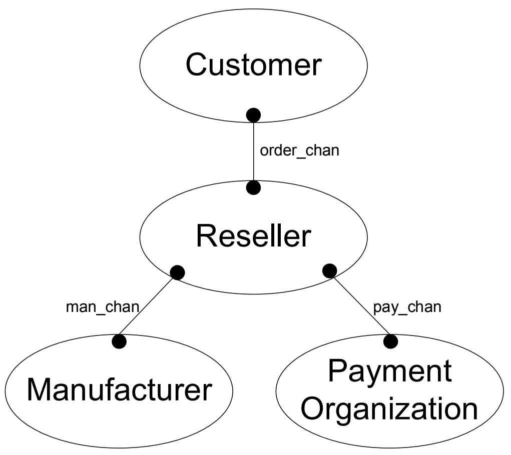
2.2 The Pi-Calculus

In this section an introduction to the π-calculus and its reduction semantics will be given. The behavior of π-calculus systems as results from the reduction semantics will be implemented in the environment for enacting and monitoring π-calculus systems. Furthermore, some tools already working with systems based on the π-calculus will be introduced and finally the idea of using the π-calculus as a basis for the specification of business processes is discussed.

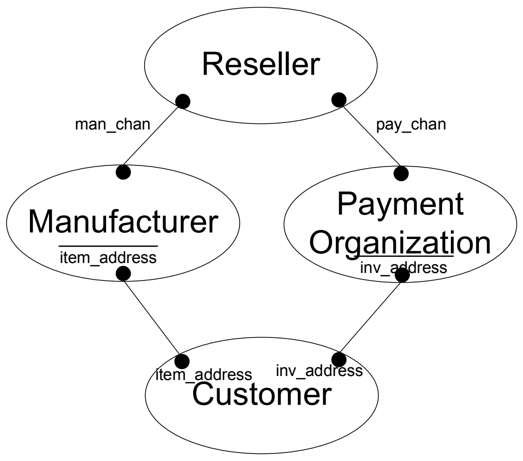
The π-calculus is a calculus for mobile systems. It is a model of the changing connectivity of interactive systems as defined in [35]. This statement already gives an idea of the kind of mobility used in the π-calculus: mobility is expressed by links moving in the virtual space of linked processes. Another real world scenario, besides the reseller example introduced above, is a cellular phone moving within the network. The cellular phone network is supported by a number of antennas, each of them covering a certain area. For a phone to be reachable, it is always connected to the antenna covering its current position. When a cell phone is moved away from one antenna into the area of another one, a new connection between the cell phone and the other antenna needs to be established while the old connection is lost. Such a system of moving links between entities can be modeled with the π-calculus. Hence, the π-calculus describes systems containing agents, where agents are the actors within the system that are running concurrently, communicating with each other in a continually changing environment. Communication in this case means sending messages along links between those agents. The links, also called channels, as well as the objects sent along are denoted as names.

Names are the basic notion in the π-calculus. Because names can be used for sending and receiving as well as themselves can be sent or received the π-calculus gains its power of dynamically establishing and omitting links between existing agents.

The reseller example introduced in 2.1.2 shows the issue: Besides informing the reseller about the desired items in the order, the customer gives the reseller information about the addresses where he wants to receive the items and the invoice. The reseller forwards this information appropriately and thereby gives the manufacturer the customer's address to deliver the item directly. Additionally, the reseller forwards the payment information of the customer to the payment organization, who takes over the business of retrieving the money for the order. The channels used by the customer to place the order and the reseller to communicate with the manufacturer and payment organization are publicly known. The manufacturer and payment organization can now communicate directly with the customer using the newly established links based on the customer's address information. Those links are private to the according agents, because the given address information is not publicly known. Since the customer is not going to order other items in this example and therefore has lost his capability of sending an order, the link between the reseller and the customer has disappeared. The situation before and after establishing the channels is displayed in Figure 2.5.



(a) Customer without connection to manufacturer and payment organization



(b) Customer with connection to manufacturer and payment organization

Figure 2.5: Reseller example in flow graph notation

The notation used in these Figures is called *flow graph*. Flow graphs were introduced in [33] and adapted to the π-calculus in [35]. A flow graph illustrates the structure of a system in a certain state, i.e. the linkage among its agents in this state. In this notation the four agents customer, manufacturer, payment organization, and reseller are depicted as oval nodes. A communication link is shown as an edge between two nodes with ports at both endings marked as dots. If the channel name of the communication is a free name, which means the name is publicly known, it is denoted as a label to the edge. Alternatively, if the channel name is restricted to the communicating agents, which means that it is private to those agents, the edge label will be positioned at both endings of the edge within the nodes representing the agents. In this case the direction of the communication can be inferred from the channel name labels. The port label of the agent sending the message is emphasized by an additional line on top of the label, e.g. $\overline{item\_address}$.

By continuously establishing new links and discarding old links between existing agents the structure of the system changes over time - the system evolves. But before going deeper into the rules by which π-calculus systems evolve, the structure of these systems will be illuminated. As already mentioned a basic notion of the π-calculus are names, which can be links as well as parameters sent over links. The second basic notion are agents. Agents (inter-)act by using names. In this context two kinds of actions have to be distinguished. The first is called interaction and happens if one agent communicates with another agent using and maybe exchanging names. The second is called intraaction and is used if the parts of one (composite) agent communicate with each other. In this case the action is not observable, since it happens within an agent and does not have any effect on its environment. Two important aspects regarding sending and receiving names are name substitution and name binding. A closer look at these aspects will be taken below, but first the grammar defining the correct structure of π-calculus systems will be given.

2.2.1 The Pi-Calculus Grammar

The π-calculus grammar as specified in [34] consists of agents, summations, and prefixes.

Definition 1 (π-Calculus Grammar)**.**

$$P ::= \quad M \mid P|P' \mid \mathbf{v}zP \mid !P \mid A(\tilde{y}) \tag{2.1}$$
$$M ::= \quad \mathbf{0} \mid \pi.P \mid M + M' \tag{2.2}$$
$$\pi ::= \quad \overline{x}\langle y\rangle \mid x(z) \mid \tau \mid [x = y]\pi \mid [x \neq y]\pi \tag{2.3}$$

Equation 2.3 specifies the prefixes within the π-calculus. Prefixes are an agent's potential for inter- or intraactions. There are four kinds of prefixes: positive prefix, negative prefix, tau prefix, and match prefix.

The negative prefix "$\overline{x}\langle y\rangle$" is an output prefix. Using x as a channel or link the name y is sent as a parameter. An example agent defined as $\overline{x}\langle y\rangle.P$ evolves to P after the action of sending. Concerning name binding, the negative prefix does not bind any new names. If the names x and y are not already bound by a restriction, which will be explained below, they both belong to the set of free names of the agent. In the polyadic π-calculus more than one name can be sent as a parameter during one action. In this case y in "$\overline{x}\langle y\rangle$" is replaced by a vector of names $\tilde{y}$ containing all the names sent.

The positive prefix "$x(z)$" is an input prefix. If x is seen as a channel an agent containing this prefix listens on this channel and waits for receiving an arbitrary name over it. The name z is a placeholder for the name to be received. Upon receiving a name all z's within the receiving agent are replaced by the received name. This operation is known as *name substitution.* An agent P defined as $x(z).P'$ receiving the name y will evolve to $P'\{^y/_z\}$. This notation provides the information that the agent P continues as the agent P' with all names z replaced by y within P'. When using the polyadic π-calculus the placeholder z is replaced by a vector $\tilde{z}$ with each component of this vector being a placeholder for a name to be received which will then be substituted. The components of the vector $\tilde{z}$ have to be pairwise distinct or otherwise a substitution like $P'\{^{ab}/_{zz}\}$ might arise that can not be resolved. Another requirement which has to be fulfilled is that matching sending and receiving prefixes have to have the same arity. The names being the placeholders in the receiving prefix are bound by it and their scope is confined to the agent the prefix belongs to. In the example the name z, or the names contained in the vector $\tilde{z}$ are bound by the prefix and their scope is P. In the special case of receiving a name already restricted to another agent, the scope of this name is expanded to agent P, now containing all the agents the name has been restricted to before with P added to it. This concept is called *scope extrusion.* Positive and negative prefixes are blocked as long as there is no counterpart using the same name as a channel being part of the same scope regarding that name.

τ is called the silent or unobservable prefix. It performs an unobservable action within an agent. An agent defined as $\tau.P$ behaves like P after performing the silent action.

Match prefixes "$[x = y]\pi$" or "$[x \neq y]\pi$" give some kind of conditional flow to the π-calculus. If the comparison of the names x and y with respect to the comparison type evaluates to `true`, execution of the agent can continue with the subsequent prefix action. Otherwise the part of the agent including the match becomes inactive "$\mathbf{0}$". The grammar above supports two types of match: equality and inequality check of names.

Agents have the syntactical form specified in Equation 2.1. They can be summations or compositions of agents, can be defined by other agents, may be agents containing restricted names, or are being replicated.

Compositions "$P \mid P'$" have the meaning that the agent is composed of two or more agents. In this case P and P' are running in parallel. Concurrent agents may interact with each other if they share a channel and one has a negative prefix on this channel and the other one having a positive prefix. If P and P' are part of the same agent the action between the two would be an intraaction of the composed agent and therefore an unobservable action τ, because no communication of the composite agent with agents in its environment has taken place.

Restrictions "$\mathbf{v}zP$" have the semantic of creating a new unique name for the placeholder z and replacing all z's within P with the new name. The name created is restricted to the agent it was created for. Intraactions between components of P along the new link are possible if P has the corresponding capabilities. These are unobservable actions. However, initially no interactions with other agents may take place using the new name as a channel. This may be changed by scope extrusion if the restricted name is sent as a parameter to another agent and the receiving agent as well as the sending agent have further capabilities of using this name afterwards. To sum up the concept of *name binding*, a name can be bound either by restriction or by an input prefix. All other names within an agent that are not bound by either one belong to the set of free names of this agent.

A replicated agent "$!P$" is the equivalent of an infinite number of agents of the same kind P running in parallel $P \mid P \mid \ldots \mid P$. The replication construct can be useful if for example a resource of a system has to answer multiple incoming requests at the same time without blocking while one request is processed. In the case of replication an instance of this agent will be assigned to each incoming request, so no waiting period has to be expected.

The agents in a system will be defined via the defined agent construct $A(\tilde{y})$. Every defined agent needs a definition like $A(\tilde{y}) \overset{def}{=} P$, with the vector $\tilde{y}$ containing pairwise distinct names that are placeholders for the free names in P and the process definition P containing the capabilities and control structures of the agent. If a defined agent $A(\tilde{z})$ is replaced by its process definition, the agent goes on behaving like $P\{^{\tilde{z}}/_{\tilde{y}}\}$, with $\tilde{y}$ containing all free names included in P. Using defined agents recursion may be represented in the π-calculus.

Summations are a set of process definition parts with only one of them being actually executed. Upon choosing the one that will be executed, the other choices are rendered void. Which of them will be chosen depends on the process definition part's current possible actions. If only one of the choices is capable of an action which is not blocked, this one is chosen. In case more than one process definition part is capable of executing, the choice depends on the current capabilities of other agents in the environment or is made non deterministically. Equation 2.2 specifies the syntax for summations. An inactive agent "$\mathbf{0}$" or a prefix "$\pi.P$" is a summation with only one choice. Summations only containing the inaction can be omitted.

Regarding the operator precedence of the grammar the convention is used that prefixing, restriction, and replication bind more tightly than summation and summation binds more tightly than composition.

Now with the knowledge of the syntax of the π-calculus the reseller example from above can be specified as π-calculus agents. Four agents need to be defined. These are the customer C, reseller R, manufacturer M, and the payment organization P. The customer, as can be seen in the example (Figure 2.5), shares a name (*order_chan*) with the reseller using it as a channel to place orders. The reseller expects three parameters on this channel. The parameters specify the desired item, the address the item is supposed to be sent to, and the address where the customer wants to receive the invoice. Upon receiving the order, the reseller forwards the order and the payment information to the accordant agents. For this the reseller shares a name with the manufacturer (*man_chan*) and one with the payment organization (*pay_chan*). The manufacturer waits for any request on its channel, then produces the requested item and sends the item to the customer using the address it received over *man_chan*. Accordingly, the agent for the payment organization can be specified. After ordering the item, the customer waits for the invoice and the item to arrive. The agent for the system S and the associated other agents are the following:

$$\begin{aligned}&\mathtt{S(order\_chan,man\_chan,pay\_chan)}\\ &\overset{def}{=}\ \mathtt{C(order\_chan)} \mid \mathtt{R(order\_chan,man\_chan,pay\_chan)}\\ &\quad \mid \mathtt{M(man\_chan)} \mid \mathtt{P(pay\_chan)}\end{aligned}$$

$$\begin{aligned}&\mathtt{C(o)}\\ &\overset{def}{=}\ (\mathbf{v}\ \mathtt{item,item\_addr,inv\_addr})\overline{\mathtt{o}}\langle\mathtt{item,item\_addr,inv\_addr}\rangle.\\ &\quad (\mathtt{item\_addr(man\_item)}.\mathbf{0} \mid \mathtt{inv\_addr(invoice)}.\mathbf{0})\end{aligned}$$

$$\begin{aligned}&\mathtt{R(o,\ m,\ p)}\\ &\overset{def}{=}\ \mathtt{o(it,it\_a,in\_a)}.\tau.\overline{\mathtt{m}}\langle\mathtt{it,it\_a}\rangle.\ (\mathbf{v}\ \mathtt{pay\_info})\ \overline{\mathtt{p}}\langle\mathtt{pay\_info,in\_a}\rangle.\ \mathtt{R(o,\ m,\ p)}\end{aligned}$$

$$\mathtt{M(m)} \overset{def}{=} \mathtt{m(i,c\_a)}.\ (\tau.(\mathbf{v}\ \mathtt{product})\ \overline{\mathtt{c\_a}}\langle\mathtt{product}\rangle.\mathbf{0} \mid \mathtt{M(m)})$$

$$\mathtt{P(p)} \overset{def}{=}\ \mathtt{p(info,c\_a)}.\ (\tau.(\mathbf{v}\ \mathtt{invoice})\ \overline{\mathtt{c\_a}}\langle\mathtt{invoice}\rangle.\mathbf{0} \mid \mathtt{P(p)})$$

The example shows one possibility where other defined agent constructs within agents are useful. Here they are used to reset the reseller, manufacturer and payment organization, so they are able to serve this customer or other customers again. Different ways of resetting are presented, e.g. the reseller is reset after all his tasks, which are triggering the manufacturer and the payment organization, are completed. Thus, he can not receive any new requests as long as he is not finished with processing the one request currently received. The manufacturer and the payment organization can process their received requests concurrently to receiving new requests. The customer agent in this system does not contain a recursive definition. It will become inactive after receiving the bill and the ordered item, hence has a once-only capability to place an order.

Apart from the grammar as specified above a different grammar exists producing slightly different constructs, which is for example used in [36] and [43]. In this grammar summations may contain arbitrary process definitions unlike the grammar as specified above where summations can only contain other summations. Not making this restriction has the effect that summations are not guarded by a prefix, so constructs like $(A \mid B) + P$ are possible. Generally, such constructs make it harder to decide what choices are available for execution, because of the possible nesting that may have a depth of several levels.

2.2.2 The Reduction Semantics

The reduction semantics of the π-calculus defines how a system represented by a π-calculus term can develop over time. The reduction semantics is specified by two relations on agents. The first is the reduction relation defining the actual communication behavior of agents and the second is a structural congruence relation. By using the structural congruence relation the term of an agent can be rewritten, so any matching input and output prefixes can be syntactically juxtaposed. Accordingly, the structural congruence relation works like a term rewriting system bringing together two agents capable of acting with each other. This simplifies the definition of the reduction relation so axioms caring about the order of components or summands of an agent can be omitted since any order can be adapted using the rules of structural congruence. Components in this context means the parallel parts of the agent in a composition. The reduction relation consists of two axioms of reduction as described in [54]. They let an agent P evolve to an agent P' as the result of either a reduction within P or a communication between P and another agent in the system. If an agent has a prefix action, which is not blocked currently, the agent has a reduction. The reduction semantics only specifies the behavior within the system independent of its environment. How a system may interact with its environment is further defined by a *labeled transition system* on terms, which will not be discussed here. For the simulation a completely specified system including all interacting agents is needed. The evolution behavior of such closed systems is completely specified by the reduction semantics and therefore the labeled transition system is not needed. Further information about the labeled transition system can be found in [54].

The first axiom of reduction defines an action with an explicit origin:

$$\textsc{Axiom 1:}\ (\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \rightarrow P_1 \mid P_2\,\{^y/_z\} \tag{2.4}$$

In this case the agent P consists of two components capable of communicating with each other via the channel x. The first component sends the name y over this channel and the second component will receive the name. According to the axiom, P can be reduced. As an effect the name y is passed between the components of P from P_1 to P_2 whereas the placeholder z in P_2 is replaced by the received name y. Both, the sending and the receiving prefix are consumed and the choices M_1 and M_2 are removed. To avoid, for example, the order of parallel components in this case to be of importance a general structural rule exists that makes it possible to infer similar reductions in combination with the axioms of structural congruence. The general rule is:

$$\text{from } P_1 \equiv P_2 \text{ and } P_2 \longrightarrow P_2' \text{ and } P_2' \equiv P_1', \text{ infer } P_1 \longrightarrow P_1', \tag{2.5}$$

with $\equiv$ as the relation of structural congruence defined in Table 2.1. To reorder the components of an agent, so the axiom 2.4 can be applied, the axiom of structural congruence SC-COMP-COMM has to be used here.

The second axiom of reduction defines the actions taken concerning the τ prefix expressing an intraaction without explicit origin:

$$\text{AXIOM 2: } \tau.P + M \to P \tag{2.6}$$

After reduction the τ prefix is consumed and the choice M is removed. Again to abandon the importance of the order of the summands within a summation the axiom SC-SUM-COMP from Table 2.1 can be applied to rewrite the terms as needed.

To conclude the inference system two more rules are defined. The first rule declares that a reduction within one component of an agent P does not affect other components of P running in parallel:

$$\text{from } P_1 \longrightarrow P_1' \text{ infer } P_1 \mid P_2 \longrightarrow P_1' \mid P_2 \tag{2.7}$$

The second rule indicates that a reduction can take place under the restriction of a name:

$$\text{from } P \longrightarrow P' \text{ infer } \mathbf{v}zP \longrightarrow \mathbf{v}zP' \tag{2.8}$$

Furthermore, the relation of structural congruence contains axioms of associativity for the parallel (SC-COMP-ASSOC) and choice operators (SC-SUM-ASSOC). Axiom SC-REP denotes that an agent $!P$ can be thought of as an infinite number of instances of P running in parallel $!P \equiv P \mid P \mid \ldots$ With the help of the axioms SC-RES and SC-RES-COMP any restriction that is not blocked by a prefix can be brought to the top of the term, so a communication might take place under this restriction. Using the axioms SC-RES-INACT, SC-COMP-INACT and SC-COMP-COMM one can prove that restricting a name not free in P has no impact, meaning if $z \notin fn(P)$ then $\mathbf{v}zP \equiv P$. This is especially useful when a term containing a restriction of a name needs to be rewritten so one of the reduction axioms can be applied.

SC-COMP-COMM:	$P_1 \mid P_2$	$\equiv$	$P_2 \mid P_1$
SC-COMP-ASSOC:	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$
SC-COMP-INACT:	$P \mid \mathbf{0}$	$\equiv$	P
SC-SUM-COMM:	$M_1 + M_2$	$\equiv$	$M_2 + M_1$
SC-SUM-ASSOC:	$M_1 + (M_2 + M_3)$	$\equiv$	$(M_1 + M_2) + M_3$
SC-SUM-INACT:	$M + \mathbf{0}$	$\equiv$	M
SC-MAT:	$[x = x]\pi.P$	$\equiv$	$\pi.P$
SC-MATU:	$[x \neq y]\pi.P$	$\equiv$	$\pi.P$
SC-RES:	$\mathbf{v}z\mathbf{v}wP$	$\equiv$	$\mathbf{v}w\mathbf{v}zP$
SC-RES-COMP:	$\mathbf{v}z(P_1 \mid P_2)$	$\equiv$	$P_1 \mid \mathbf{v}zP_2$, if $z \notin fn(P_1)$
SC-RES-INACT:	$\mathbf{v}z\mathbf{0}$	$\equiv$	$\mathbf{0}$
SC-REP:	$!P$	$\equiv$	$P \mid !P$
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 [54]

To facilitate the understanding of the reduction semantics, the axioms will be applied to the reseller example defined above showing some steps during the evolution of the system. As a start, the agents contained within the process definition of agent $S(order\_chan, man\_chan, pay\_chan)$ will be unfolded using the axiom of structural congruence SC-UNFOLD:

$$
\begin{aligned}
&\mathtt{C(order\_chan)} \mid \mathtt{R(order\_chan,man\_chan,pay\_chan)}\\
&\mid \mathtt{M(man\_chan)} \mid \mathtt{P(pay\_chan)}\\
\overset{Sc-Unfold}{\equiv}\quad & (\mathbf{v}\ \mathtt{item,item\_addr,inv\_addr})\overline{\mathtt{order\_chan}}\langle\mathtt{item,item\_addr,inv\_addr}\rangle.\\
& (\mathtt{item\_addr(man\_item)}.\mathbf{0} \mid \mathtt{inv\_addr(invoice)}.\mathbf{0})\\
& \mid \mathtt{order\_chan(it,it\_a,in\_a)}.\tau.\overline{\mathtt{man\_chan}}\langle\mathtt{it,it\_a}\rangle.\\
& (\mathbf{v}\ \mathtt{pay\_info})\overline{\mathtt{pay\_chan}}\langle\mathtt{pay\_info,in\_a}\rangle.\\
& \mathtt{R(order\_chan,\ man\_chan,\ pay\_chan)}\\
& \mid \mathtt{man\_chan(i,c\_a)}.\ (\tau.(\mathbf{v}\ \mathtt{product})\overline{\mathtt{c\_a}}\langle\mathtt{product}\rangle.\mathbf{0} \mid \mathtt{M(man\_chan)})\\
& \mid \mathtt{pay\_chan(info,c\_a)}.\ (\tau.(\mathbf{v}\ \mathtt{invoice})\overline{\mathtt{c\_a}}\langle\mathtt{invoice}\rangle.\mathbf{0} \mid \mathtt{P(pay\_chan)})
\end{aligned}
$$

Before the customer and the reseller can be reduced using the first axiom (Equation 2.4), the restrictions of the names *item_addr*, *item* and *inv_addr* have to be pulled up using SC-RES-COMP. This is feasible, since none of these names has a free occurrence within the other agents. Additionally, an inactive agent will be inserted as a summand to the reseller and customer, so the structure of the system's definition is adapted for applying AXIOM 1.

$$\overset{Sc-Res-Comp,Sc-Sum-Inact}{\equiv}$$

$$\begin{aligned}
&(\mathbf{v}\ \texttt{item,item\_addr,inv\_addr)(}\\
&\texttt{(}\overline{\texttt{order\_chan}}\langle\texttt{item,item\_addr,inv\_addr}\rangle\texttt{.}\\
&\texttt{(item\_addr(man\_item).}\mathbf{0}\mid\texttt{inv\_addr(invoice).}\mathbf{0}\texttt{) + }\mathbf{0}\texttt{)}\\
&\mid\texttt{(order\_chan(it,it\_a,in\_a).}\tau\texttt{.}\overline{\texttt{man\_chan}}\langle\texttt{it,it\_a}\rangle\texttt{.}\\
&(\mathbf{v}\ \texttt{pay\_info)}\ \overline{\texttt{pay\_chan}}\langle\texttt{pay\_info,in\_a}\rangle\texttt{.}\\
&\texttt{R(order\_chan, man\_chan, pay\_chan) + }\mathbf{0}\texttt{)}\\
&\mid\texttt{man\_chan(i,c\_a). (}\tau\texttt{.(}\mathbf{v}\ \texttt{product)}\ \overline{\texttt{c\_a}}\langle\texttt{product}\rangle\texttt{.}\mathbf{0}\mid\texttt{M(man\_chan))}\\
&\mid\texttt{pay\_chan(info,c\_a). (}\tau\texttt{.(}\mathbf{v}\ \texttt{invoice)}\ \overline{\texttt{c\_a}}\langle\texttt{invoice}\rangle\texttt{.}\mathbf{0}\mid\texttt{P(pay\_chan)))}
\end{aligned}$$

Now the axiom can be applied reducing the customer and the reseller regarding the communication on channel *order_chan* and substituting the reseller's names by the ones received from the customer.

$$\xrightarrow{Axiom\ 1}$$

$$\begin{aligned}
&(\mathbf{v}\ \texttt{item,item\_addr,inv\_addr)(}\\
&\texttt{(item\_addr(man\_item).}\mathbf{0}\mid\texttt{inv\_addr(invoice).}\mathbf{0}\texttt{)}\\
&\mid\tau.\overline{\texttt{man\_chan}}\langle\texttt{item,item\_addr}\rangle\texttt{.}\\
&(\mathbf{v}\ \texttt{pay\_info)}\ \overline{\texttt{pay\_chan}}\langle\texttt{pay\_info,inv\_addr}\rangle\texttt{.}\\
&\texttt{R(order\_chan, man\_chan, pay\_chan)}\\
&\mid\texttt{man\_chan(i,c\_a). (}\tau\texttt{.(}\mathbf{v}\texttt{product)}\ \overline{\texttt{c\_a}}\langle\texttt{product}\rangle\texttt{.}\mathbf{0}\mid\texttt{M(man\_chan))}\\
&\mid\texttt{pay\_chan(info,c\_a). (}\tau\texttt{.(}\mathbf{v}\texttt{invoice)}\ \overline{\texttt{c\_a}}\langle\texttt{invoice}\rangle\texttt{.}\mathbf{0}\mid\texttt{P(pay\_chan)))}
\end{aligned}$$

The next steps that can be taken are transforming the term of the reseller by introducing an inactive summand (SC-SUM-INACT) and reducing the reseller concerning the τ action by applying AXIOM 2 (Equation 2.6). These and the other steps afterwards will not be exemplified here any further.

2.2.3 Related work

Since this thesis is about developing a tool working with π-calculus systems, executing them, and giving a visual representation of these systems, a short look-out into the world of existing work and tools concerning this field will be taken.

The most work has been done in the field of equivalence checking. Systems of π-calculus agents are compared using bisimulation. Bisimulation is an equivalence relation between state transition systems associating systems behaving in the same way in the sense that one system simulates the other and vice versa. Different kinds of bisimulation equivalences exist. Bisimulation can be distinguished in weak and strong bisimulation depending on the possibility of internal transactions of a system being ignored or taken into account. Further differentiation of bisimulation is explored in [37], introducing the concepts of *late* and *early* bisimulation. A third kind of bisimulation, the *open* bisimulation, was formulated in [53]. Practical work on the topic of bisimulation can for example be found in the tools *Mobility Workbench*(MWB) [57], *Another Bisimulation Checker*(ABC) [14], and *Open Bisimulation Checker*(OBC) [20]. These are tools intended for manipulating and analyzing mobile concurrent systems based on the π-calculus. Their main feature is checking for open bisimulation equivalences. Although these tools are able to give the trace of a step by step execution of π-calculus terms, the user is presented with a command line interface without visual representation of the systems. Such command line output, as it might be convenient for communicating the results of a bisimulation check, is insufficient for easily understanding and comprehending the structural changes of π-calculus systems during evolution.

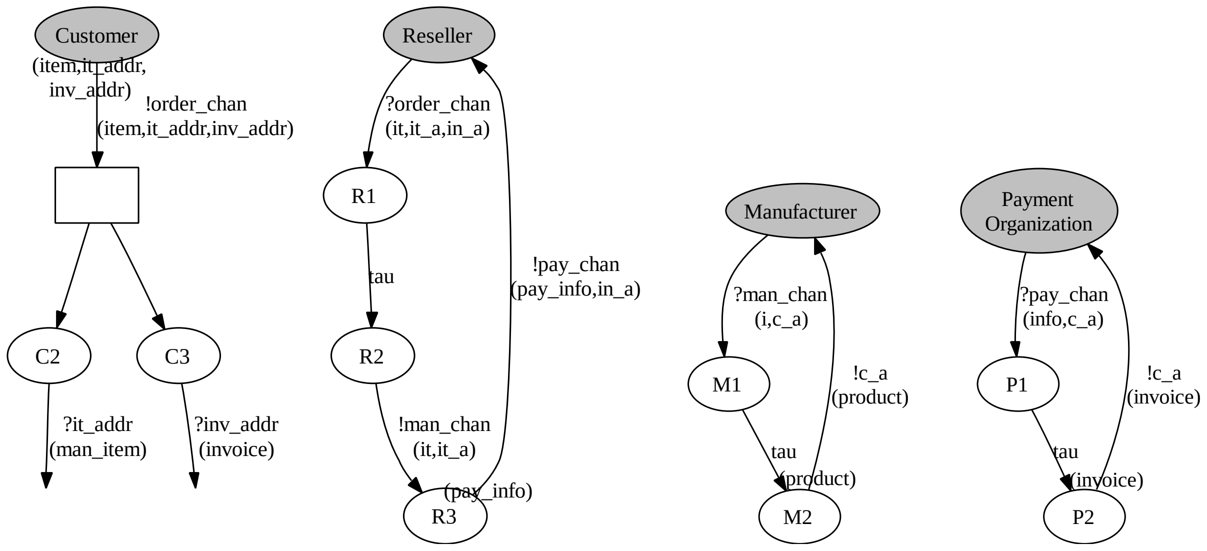


Figure 2.6: Reseller example in the proposed graphical representation [45]

Some work on graphical representations of the π-calculus has been done in [45] proposing a state machine like approach of visualizing the agents. The graph of an agent consists of nodes and edges. Edges represent an agents capabilities. Nodes represent the different states of the agent with certain capabilities available, which means that they are not blocked by another prefix at the moment the agent is in this state. Since this is a static representation of the entire system's agents, a labeling function has been introduced marking all nodes in the graphs of the agents that represent states, which are currently active. More than one node of an agent's graph may be labeled and the sum of all labeled nodes corresponds to a state denoting all currently available capabilities. The edge labels preceded by an exclamation point represent output prefixes and the ones preceded by an interrogation point represent input prefixes. Edges that have no target node represent inaction after the according capability has been executed. Moreover, the names in brackets annotated near a node are those names that are newly created in this state if it is active. Although giving a graphical representation of the agents, this approach does not visually show the interaction behavior including establishing and discarding links between them. The reseller example in the proposed graphical representation with the initial states shaded is depicted in Figure 2.6.

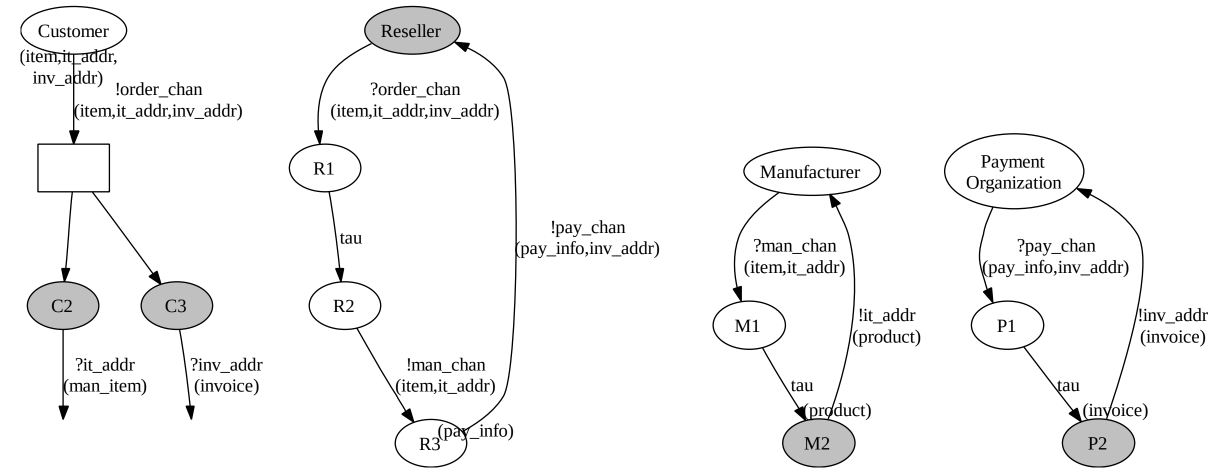


Figure 2.7: Reseller example in the proposed graphical representation during execution

Lets assume that the customer has sent his order to the reseller and is waiting for the product and the invoice to arrive. The reseller has forwarded the relevant information and the manufacturer as well as the payment organization have done their work being ready to send the product and invoice back to the customer. Utilizing the proposed graphical representation, the state of the system after these actions and name substitutions is depicted in Figure 2.7. The example shows that the representation, as it might be convenient to show the structure of the agents, does not directly present the possible interactions. For analysis of interacting agents another graphical representation has to be provided directly showing the connections between and within the agents that have the ability to (inter-)act.

2.2.4 Converting BPMN Processes to Pi-Calculus Processes

After introducing the π-calculus, some more specific details regarding the BPMN to π-calculus converter of the introduced tool chain can be given now. First of all the ASCII π-agents resulting from conversion can not only be used as input for the simulation environment developed in this thesis, but also for the bisimulation checking tools Mobility Workbench and Another Bisimulation Checker already introduced in 2.2.3.

Since business processes are modeled using the workflow patterns, a mapping of them to the π-calculus is needed as a basis for the conversion. Mappings of the behavioral patterns of business processes to the π-calculus have been proposed in [41, 48] and the conversion algorithm for the converter is described in [49]. A short overview with an example of the conversion algorithm will be given in the following.

By the converter each flow object contained in the modeled business process diagram is interpreted as a π-calculus agent and the connecting objects between them are represented by the input and output prefixes of those agents. The incoming and outgoing sequence flows of a flow object are interpreted as its pre- and post-conditions. A π-calculus agent is blocked until its pre-conditions are fulfilled, which is until reductions are available concerning the appropriate input prefixes. After the agent is finished with its internal actions and possibly processing incoming and outgoing message flows, it will notify its succeeding agents by activating the appropriate output prefixes that represent its outgoing sequence flows. Since the π-calculus agents of the flow objects all run in parallel, this is the way of synchronizing them according to the structure described by the BPD. As a conclusion, all of the π-calculus agents have a similar basic structure, which is defined by a sequence consisting of input prefixes for the incoming sequence flows, the specific actions regarding the flow object type, and the output prefixes for the outgoing sequence flows. Their specific structure is defined according to the π-calculus mappings of the behavioral patterns. Exceptions to this structure are start and end events. Start events do not have any incoming sequence flows and therefore do not have the according input prefixes as pre-conditions. End events do not have any outgoing sequence flows. At the end an additional agent as a composition of all the agents created for the flow objects within the pool is created for each pool contained in the business process diagram. If no pools are contained in the BPD, just one agent containing all created agents representing the flow objects is generated. This composite agent restricts the names that are used for sequence flows within the pool exclusively to those agents contained in the pool. This ensures that no inter pool communication takes place using sequence flows.

As an example, the conversion of the customer process as part of the reseller business process depicted in Figure 2.3 will be shown. First of all, unique π-calculus agent identifiers are assigned to the flow objects of the customer process and unique π-calculus names are assigned to its connection objects representing sequence flows. These assignments are done automatically by the converter. The assignments as annotations to the objects of the BPD are illustrated in Figure 2.8.

Additionally, the message flows have to be extended with the information about the data to be sent and received. This information has to be specified explicitly by the

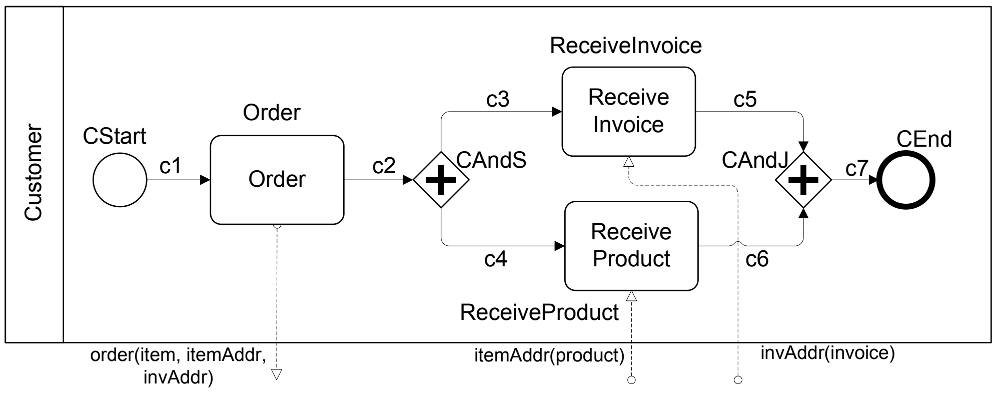


Figure 2.8: Customer process with assignments for conversion

modeler, since the converter has no notion of what is going to be sent or received via message flow. In this case, for example, the customer tells the reseller the name of the item he orders and also informs him about the location where he wants to receive the item and the invoice. Now the agents for the flow objects can be defined as follows:

$$\mathtt{CStart(c1)} \overset{def}{=} \tau.\overline{\mathtt{c1}}.\mathbf{0}$$

$$\begin{aligned}&\mathtt{Order(c1,c2,order)}\\ \overset{def}{=}\ &\mathtt{c1}.\tau.\mathtt{(v\ item,itemAddr,invAddr)(}\overline{\mathtt{order}}\langle\mathtt{item,itemAddr,invAddr}\rangle.\\ &\mathtt{(}\overline{\mathtt{c2}}\langle\mathtt{itemAddr,invAddr}\rangle.\mathbf{0} \mid \mathtt{Order(c1,c2)))}\end{aligned}$$

$$\begin{aligned}&\mathtt{CAndS(c2,c3,c4)}\\ \overset{def}{=}\ &\mathtt{c2(itemAddr,invAddr)}.\tau.\\ &\mathtt{(}\overline{\mathtt{c3}}\langle\mathtt{invAddr}\rangle.\mathbf{0} \mid \overline{\mathtt{c4}}\langle\mathtt{itemAddr}\rangle.\mathbf{0} \mid \mathtt{CAndS(c2,c3,c4))}\end{aligned}$$

$$\begin{aligned}&\mathtt{ReceiveInvoice(c3,c5)}\\ \overset{def}{=}\ &\mathtt{c3(invAddr)}.\tau.\mathtt{invAddr(invoice).(}\overline{\mathtt{c5}}.\mathbf{0} \mid \mathtt{ReceiveInvoice(c3,c5))}\end{aligned}$$

$$\begin{aligned}&\mathtt{ReceiveProduct(c4,c6)}\\ \overset{def}{=}\ &\mathtt{c4(itemAddr)}.\tau.\mathtt{itemAddr(invoice).(}\overline{\mathtt{c6}}.\mathbf{0} \mid \mathtt{ReceiveInvoice(c4,c6))}\end{aligned}$$

$$\mathtt{CAndJ(c5,c6,c7)} \overset{def}{=} \mathtt{c4.c5}.\tau.\mathtt{(}\overline{\mathtt{c7}}.\mathbf{0} \mid \mathtt{CAndJ(c5,c6,c7))}$$

$$\mathtt{CEnd(c7)} \overset{def}{=} \mathtt{c7}.\tau.\mathtt{CEnd(c7)}$$

As can be seen in these process definitions all agents, except the one representing the start event, reset themselves. This is needed if loops are part of the business process, so each activity can be executed again. Since end events can not be part of loops, as they are not supposed to have any outgoing sequence flows, the resetting of end event agents serves a different reason. For bisimulation or soundness checking of a π-calculus system the number of times the end event agent is triggered might be of importance. By resetting the end event agent it can be triggered more than once. Additionally, the flow of data, here the address information being routed from the order activity to the receive product and receive invoice activities, can be observed. Finally, the agent C for the customer as the composition of all the agents representing its flow objects can be specified:

```
C(order)
 def
  =   (v c1,c2,c3,c4,c5,c6,c7)(CStart(c1) | Order(c1,c2,order) |
      CAndS(c2,c3,c4) | ReceiveInvoice(c3,c5) | ReceiveProduct(c4,c6) |
      CAndJ(c5,c6,c7) | CEnd(c7))
```

With the definition of the agent C all the names used as sequence flows $c1 - c7$ within the pool are restricted to the agents contained in it. The processes in the other pools included in this example are converted accordingly.

The example shows the applicability of the π-calculus for representing business processes. Furthermore, the fundamental framework of knowledge for working out the concepts for a π-calculus simulation environment are laid out. The next chapter will start with the definition of the requirements for the simulation environment.

Chapter 3

Technical Analysis

In the following section the functional requirements for the environment for enacting and monitoring π-calculus process definitions will be elaborated. These can be differentiated into requirements regarding the execution of π-calculus systems that realize their evolution behavior, user interaction requirements, and visualization requirements. Furthermore, the usage of existing tools and libraries reduces the effort of implementation. As an example, the generation of the parsing component needed for input reading or the creation of the graph representing the π-calculus system's linking structure in a special state can be assisted. Hence, different tools and libraries capable of supporting the implementation will be considered and discussed. Finally, the architecture for the prototypical implementation of the simulation environment is described.

3.1 Functional Properties Elicitation

Basically, the simulation environment being developed should offer a graphical user interface. Fundamental interactions of the user are the selection of π-calculus process definitions from the file system and the selection of the next executable step of the π-calculus system. The π-calculus process definitions contain the agents for execution and the selection can be realized with the help of a file choosing dialog. An overview of user interactions is given in the use case diagram shown in Figure 3.1.

The first step to do is reading the input definitions selected by the user. After a file has been selected, it is parsed and the internal data structure representing the π-process definitions is built. Subsequently, the initial state of the π-calculus system can be graphically presented to the user. For this, a π-execution engine able to execute agents and match counterparts of communications considering scopes and further constraints, e.g. the communication partners being choices of the same summation, is needed. The execution engine has to have different execution modes. On the one hand it has to be

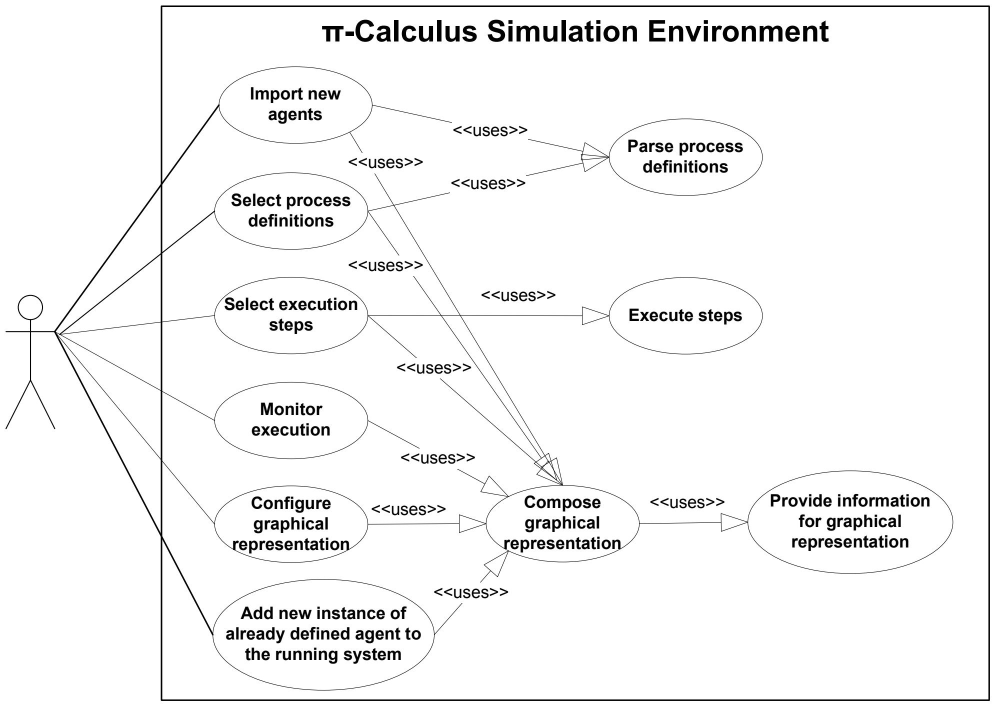

π-Calculus Simulation Environment
Import new agents
Select process definitions
Select execution steps
Monitor execution
Configure graphical representation
Add new instance of already defined agent to the running system
Parse process definitions
Execute steps
Compose graphical representation
Provide information for graphical representation
<<uses>>

Figure 3.1: Use cases of the environment for enacting and monitoring π-calculus systems

able to execute single communications selected by the user. On the other hand it has to automatically execute parts of agents in some cases that will be explained later.

Not all of the agents contained in a process definition file are always supposed to be executed initially. The user has the choice of predefining the agents he wants to start the execution with by marking them directly in the process definition file. If no agents are predefined for execution, the user is prompted with a dialog where he chooses the agents for execution from a list containing all agents defined in the process definition file. The second option also allows the user to add already defined agents to the running system during its evolution. As an example, remember the simplified reseller business process defined in section 2.2.1. The user might want to add a fresh copy of the customer agent to the running system, e.g. after the initial customer agent has finished execution. Thus, the system is revived and the interplay of the agents can be observed a second time.

For the representation, the existing notation of the flow graphs will be extended. Basically these graphs show the current linking structure of a system. For a user to choose the next action to take place, we have to differentiate between selectable and non-selectable actions. An action in this context is represented by prefixes and can either be a τ prefix or a pair consisting of a positive and negative prefix using the same name that also belongs to the same scope. Selectable actions, called active actions further on, are all actions, whose associated prefixes are not preceded by other prefixes. Non-selectable actions, called blocked actions further on, are all actions, whose associated prefixes are preceded by other prefixes. Actions that are represented by a pair consisting of a positive and negative prefix are communications. Communications are displayed as links between nodes in the graphical representation. Positive or negative prefixes that currently have no counterpart are not taken into account at all, because they would have no link in the structure of the system and therefore not be visible to the user. The set of blocked actions results from subtracting the set of active actions from the set of all actions included in the system. Referring to the extension, blocked actions need to be represented differently compared to the active actions, because the user may only select active ones for execution.

The user should be able to manipulate the systems structure by directly selecting the next action for execution in the graphical representation. This entails executing the appropriate parts of the data structure as well as updating the graphical representation to show the succeeding state. Furthermore, the scoped names should be presented to the user so he can choose, which scopes he wants to track. A suitable presentation for the representation of scopes has to be found.

Another feature is to display the process definitions of the systems current state in π-calculus syntax, which means extracting the process definitions from the internal data structure. Additionally, the original process definitions can be displayed in a different view so the user can compare the changes of the system during its evolution to the initial state.

Since π-calculus systems may be complex and the user might not be interested in the internals of each agent, or a group of agents within the system, some kind of abstraction has to be realized. By using abstraction agents can on the one hand be configured to either show or hide their internal actions and on the other hand are grouped together visually. For the prototype to graphically arrange agent groups and provide visual feedback about components in the π-calculus system being available for hiding the definition of the agents that are grouped together has to be realized beforehand in the process definition file. The configuration of closing or opening agent groups during execution should be available interactively, e.g. the selection of a predefined group of agents toggles the group's state between showing and hiding the internals. Moreover, the treatment of agent groups as black boxes leads to the requirement of automatic execution of internal actions of the closed down groups so unintended blocking does not occur.

As can be seen from the grammar in section 2.2.1 the tool will support replication as well as defined agent constructs. Defined agents can be expressed as replication using message-passing to model the passing of parameters. For example a process definition $P(\tilde{x}) \overset{def}{=} Body_P$ with $Body_P$ possibly containing calls $P(\tilde{z})$ can be formulated as replication in the following way: First a new name n is defined whereas sending names $\tilde{z}$ along n represents the call $P(z)$. Furthermore, the process definition of agent $P(\tilde{x}) \overset{def}{=} Body_P$ is refined to $!n(\tilde{x}).Body_P$ with all further recursive calls to P within $Body_P$ exchanged by sending over the channel n. This definition ensures that the agent P receives the names as parameters over the newly defined channel n. Now the system S containing the agent P and the agent Q calling this agent is redefined to

$$S \overset{def}{=} (\mathbf{v}n)\ (Q[P(\tilde{z}) := \overline{n}\langle\tilde{z}\rangle.\mathbf{0}] \mid !n(\tilde{x}).Body_P[P(\tilde{z}) := \overline{n}\langle\tilde{z}\rangle.\mathbf{0}])$$

Although defined agent constructs can be rewritten as replications, the implementation of the π-engine will support both constructs, since it might be more convenient for a user to use the one or the other in certain cases. Thereby, the choice of using replications, defined agents, or both is left to the user.

The process definitions loaded into the tool at the beginning of the execution contain agents of a system and specifications, which agents are initially executed. During execution the need of adding additional agents to the executed system may arise. A dialog can be offered to the user where he can specify the agents to be added to the running system. For adding another agent its process definition has to be loaded into the execution engine. Different options exist where to find its process definition. Firstly its definition might already be loaded, because it was defined in the process definition of the π-calculus system, but the agent was not set for execution. Secondly the user may select another definition file containing the definition of this agent. As a last option the new agent's name, parameter list, and process definition can be inserted directly via text input. The last two options require an additional parsing step to build the appropriate data structure for the added agent's process definition. Name clashes of the newly defined or imported agent and agents already existing in the system have to be avoided. Additionally, if the newly defined agent contains calls to other agents in its process body definitions of these agents have to be provided to the system, too, if not already included. If such definitions of agents contained in the process body of an added agent are not provided, a runtime error is raised in the moment the execution engine tries to resolve the agent, but is not able to find it.

Finally, the requirements for the tool can be grouped in requirements regarding the user interface, visualization of the π-calculus system, and the execution engine. User interface requirements are those directly concerned with the user, e.g. what interactions are possible. Visualization requirements refer to the graph representing the π-calculus system especially regarding what information of the π-calculus system is displayed and how interaction with it is realized. Lastly, the execution engine requirements define all functionality needed in the background, invisibly to the user, that facilitate the execution of π-agents. In the following list the requirements are named and sorted by group.

- User interface requirements
  - Selection of π-process definition file
  - Automatic selection of multiple steps for execution if invisible to the user
  - Selection of scoped names to be displayed
  - Import dialog for additional agents
  - Representation of the current system in π-calculus syntax
  - Representation of the original system in π-calculus syntax
  - Dialog for selecting agents for execution if no pre-definitions are contained in the process definition file
  - Dialog for adding new instances of already defined agents to the running system
- Visualization requirements
  - Graphically display current state of π-calculus system
  - Graphically display selected scoped names
  - Provide visual differentiation between blocked and active communications
  - Direct interaction with the graphical representation
  - Selection of single steps to be executed
  - Configuration of display mode (open/closed) for groups of agents
- Execution engine requirements
  - Same syntax for process definitions as MWB and ABC tools
  - Parsing of π-process definitions
  - Execution of π-calculus systems
  - Execution of single communications
  - Automatically select steps for auto-execution
  - Parsing of imported agents
  - Addition of imported agents to the current system
  - Consistency check of imported agents (e.g. name collisions with existing agents)

3.2 Usage of External Tools and Libraries

For the implementation of the prototype a survey of tools and libraries that can be useful to support the realization of the desired functionality has been taken. Components of the prototype, whose implementation can be assisted by using existing tools and libraries are the parser, needed for input reading, and the module for composing the graphical representation. In the following a selection of available tools to be considered for accomplishing subtasks will be stated.

3.2.1 The Parsing Component

Lots of open source parser generating tools exist and can be of use for creating the component that reads the π-process definitions from the input files. Since the prototype will be implemented using the Java programming language a parser generator producing output in the same language is of avail, but still the variety of options is considerable. Some popular choices are for example ANTLR (Another Tool for Language Recognition) [42], Coco/R [39] or JavaCC (Java Compiler Compiler) [2].

ANTLR is a language tool providing a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. It offers operations for the construction and traversal of trees. Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for it. This special compiler generator does not provide for tree building functionality. JavaCC is a Java application itself only offering Java output. By means of the tree building preprocessor JJTree, JavaCC offers powerful tree construction functionality. JJTree by default constructs parse tree nodes for each nonterminal in the language, which can be modified so nodes are not generated for every nonterminal. Generated nodes implement a defined interface offering methods for setting the parent of the node, adding children, or retrieving them.

JavaCC as well as ANTLR have two modes of tree creation. Either homogeneous trees or heterogeneous trees can be created. Homogeneous trees consist of nodes all having the same type differing only in a type field, e.g. holding a value out of an enumeration of types, which influences the node's behavior. Heterogeneous trees consist of different node types being implemented by different classes. One reason for having heterogeneous trees is for storing radically different kinds of data in the tree nodes or for the tree nodes showing radically different behavior. In the heterogeneous tree mode JJTree derives the type of the node to be created from the node's name in the grammatical description. It generates sample implementations for the needed tree nodes or uses provided implementations of the classes with the appropriate name. The sample implementations can also be customized to fit the desired behavior.

Since the tree building capability is of great importance as the proposed internal data structure is a tree structure, the choice of tools comes down to either using ANTLR or JavaCC. As can be seen from the argumentation both tools are equally powerful concerning the functionality aimed for. In the prototypical implementation JavaCC as the tool for generating the parser component is used.

As already stated above, the process definitions as input of the parser are exclusively composed of ASCII characters. The complete specification of the input grammar in extended Backus-Naur form (EBNF) [1] can be found below. The initial production for the specification of π-process definitions is `InputFormat`.

```
/**** Terminal Symbols ****/
INACTION  = "0";
AGENT_ID  = (["A"-"Z"])+ ("_" | ["0"-"9"] | ["a"-"z"]|["A"-"Z"])*;
AGENT_TOK = "agent";
TAU       = "t";
EXEC      = "exec";
POOL      = "pool";
NAME      = ("_" | ["0"-"9"] | ["a"-"z"])
            ("_" | ["0"-"9"] | ["a"-"z"]|["A"-"Z"])*};

/**** Productions ****/
InputFormat     = PoolDefinition* AgentDefinition+;
PoolDefinition  = POOL AGENT_ID
                  "{"AGENT_ID (","AGENT_ID)* "}";
AgentDefinition = EXEC? AGENT_TOK AGENT_ID
                  "(" ( NAME (","NAME)+ ) | NAME? ")"
                  "=" Composition;
Composition     = P ("|" P)*;
P               = Restriction | Replication | DefinedAgent
                  | Summation | "(" Composition ")";
Summation       = (INACTION | Pi) ("+" (INACTION | Pi))*;
Restriction     = "(" "^" NAME ("," NAME)* ")" P;
Replication     = "!" P;
DefinedAgent    = AGENT_ID "("(NAME (","NAME)+) | NAME?")";
Pi              = Send | Receive | Tau | Match;
Send            = "'" NAME ("<"NAME? | (NAME (","NAME)+)">")? "." P;
Receive         = NAME ("("NAME? | (NAME (","NAME)+)")")? "." P;
Tau             = TAU "." P;
Match           = "["NAME ("=" | "!=") NAME"]" Pi;
```

Additional tags dissentient from the input format used by the Mobility Workbench and Another Bisimulation Checker have to be introduced. They will allow for pre-configuration of the visual representation. Pre-configuration includes pointing out agents that are going to be executed directly and defining pools for the abstraction of agents. Therefore, the newly introduced tags are `exec` and `pool`. For the specification of initially executed agents the tag `exec` is used. This tag tells the parser to flag the agent for execution. Definitions of agents belonging together being available for abstraction are defined via the EBNF rule starting with the tag `pool`. However, the definition of pools does not have any influence on the execution behavior of the π-calculus agents, since this concept is unknown to the π-calculus and is only needed for the visual representation.

Knowing this, the example from section 2.2.1 has to be rewritten. The resulting process definitions are illustrated below and can directly be used as input for the tool. In this example the agent S is started initially, letting the customer C, reseller R, manufacturer M, and payment organization P run in parallel.

```
exec agent S(order_chan,man_chan,pay_chan)
        = C(order_chan) | R(order_chan,man_chan,pay_chan)
          | M(man_chan) | P(pay_chan)

agent C(o) = (^item,item_addr,inv_addr)'o<item,item_addr,inv_addr>.
             (item_addr(man_item).0 | inv_addr(invoice).0)

agent R(o,m,p) = o(it,it_a,in_a).t.'m<it,it_a>.
                 (^pay_info)'p<pay_info,in_a>.R(o,m,p)

agent M(m) = m(i,c_a).(t.(^product)'c_a<product>.0 | M(m))

agent P(p) = p(info,c_a).(t.(^invoice)'c_a<invoice>.0|P(p))
```

3.2.2 The Visualization Component

The visualization component has the task to build a graph that expresses the structure of the π-calculus system. For this the agents within the system and all links between them are needed. Furthermore, the representation should be interactive in the sense that a user may select a node or edge he wants to execute as the next step directly within the image, instead of from a list next to the graphical representation.

A special issue in dynamically composing visual representations of a permanently changing system regarding the number of nodes and the changing link structure is finding an adequate layout. Concerning this matter, a tradeoff has to be made. It manifests itself in the choice of keeping all the graph nodes representing agents at roughly the same position in the graphical representation during the entire execution of the π-calculus system, or having the graph nodes take their best position concerning the link structure of the system after each execution step. The first option may end up in a confusing link structure for large systems with many links, since paths between nodes are not kept as short as possible. This disadvantage is overcome by the second solution always offering the best representation. However, the second option has the drawback of switching the positions of nodes, which may be confusing to the user once more, but can be compensated for by appropriate naming of agents.

Several libraries exist that can be used for building the desired graphs. A selection of these are for example the Piccolo Toolkit [4], Java Universal Network/Graph Framework (JUNG) [3], Prefuse Information Visualization Toolkit [5], and Grappa (A Java Graph Package) [38]. All of the mentioned toolkits and libraries provide graph drawing capabilities with layouting options and user interaction support.

For the prototypical implementation "PiVizTool" the grappa library will be used. Grappa ports a subset of functionality provided by the graph visualization software Graphviz [23] to Java. It provides methods for building, manipulating, traversing, and displaying graphs of nodes, edges and subgraphs as well as interactive means for the selection of graph elements. Concerning the layout of the graphs, grappa can use the engines provided by Graphviz, e.g. "dot" [22] or "neato" [40]. Graphviz "dot" allows for drawing of hierarchical layered and directed graphs. Its algorithm aims at edges pointing in the same direction, reducing the lengths of edges and it tries to avoid edge crossings. Using such a layout engine keeps the graphical representation concise and easier to comprehend.

3.3 Architecture

Figure 3.2 shows the architecture of the simulation environment for π-calculus systems as a block diagram of the FMC notation. Main components in this Figure are the user, the file system containing files with π-calculus process definitions, and the π-calculus simulator. Process definition files are created by the user, e.g. using the tool chain described in section 2.1.3.

User interaction with the π-calculus simulator takes place via a graphical user interface (GUI) letting the user choose π-calculus process files for execution and waiting for the selection of executable steps by the user. In each step of execution a graphical representation of the current state of the π-calculus system is displayed. The visual representation is composed by the π-visualizer component. Actions chosen by the user for execution are signaled to the controller that relays the appropriate commands to the execution engine.

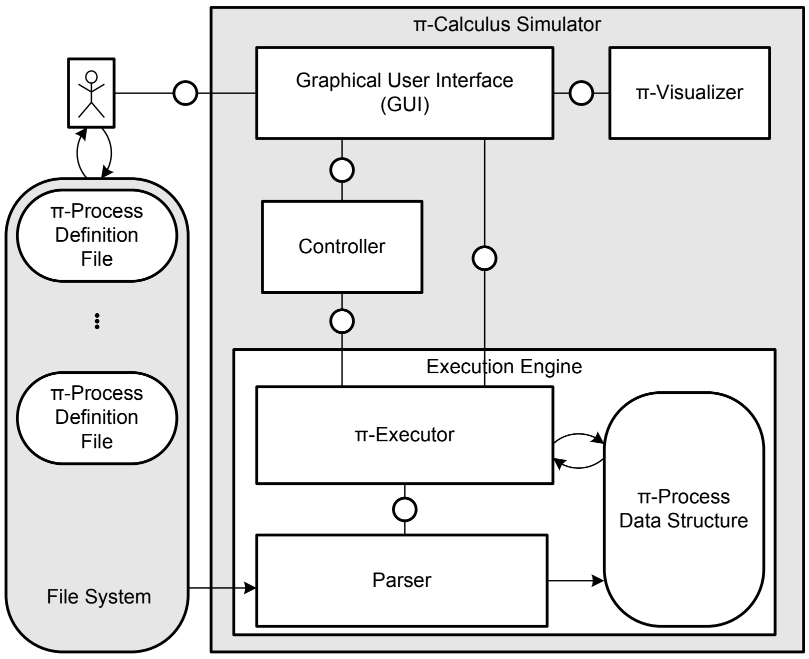


Figure 3.2: General architecture of the π-calculus simulator

The execution engine consists of a parser and a π-executor component. After a file is chosen by the user, it needs to be parsed. During parsing the internal data structure for execution is built. It is used by the π-executor to provide the information needed by the visualizer to create the graphical, interactive representation of the initial state of the π-calculus system. For the execution component the provision of the information for visualization means to figure out the link structure of the system, which will be displayed. The updated information for visualization is pulled from the execution engine upon receiving the event that the state of the π-calculus system has changed.

User interactions concerning selections of actions (τ or communication) trigger further execution steps in the execution engine. Choosing agents to be displayed as abstract agents (black box agents) causes automatic execution of all actions not being blocked within these agents. In both cases the π-executor handles the execution by modifying the data structure and providing updated information for visualization.

The above described architecture obeys the model view controller design pattern (MVC) [21]. MVC proposes an architecture that separates an application's data model, user interface, and control logic into three distinct components, so that modifications to one component can be made with minimal impact on the other components. In this case the view component accords to the GUI and the π-visualizer, and the model component is implemented by the execution engine. Through decoupling of the model, view and controller components, each unit can be changed irrespectively of the implementation of the other units. Each, the view and the execution engine implement an own interface, keeping the coupling between view and controller as well as controller and execution engine weakly typed and therefore enforcing effortless exchangeability of the components.

For the parsing component as well as the visualization component the use of external tools is expedient. The interactions with the parsing component are relatively simple, only consisting of the command for parsing a file and producing the appropriate data structure. However, the interactions of the GUI with the π-visualizer component are more complicated, since user interaction within the graphical representation triggers execution and a range of information has to be provided for composing the graphical representation. For that reason the π-visualizer implements an interface, which is used by the GUI, so exchanging the visualization component, e.g. using another external tool as the basis for visualization, has no effect on the GUI. The π-visualizer provides feedback of user interactions within the graphical representation by signaling to its registered listeners, thus implementing the observer design pattern [21]. The essence of this pattern is that one or more objects (called observers or listeners) are registered (or register themselves) to observe an event, which may be raised by the observed object (the subject). The object which may raise an event generally maintains a collection of the observers. In this case the GUI is the observer and the π-visualizer is the observed object.

After the overall requirements are now elaborated and the components within the architecture of the π-calculus simulator have been worked out, the concepts driving the implementation will now be explained in the following chapter.

Chapter 4

Design

This chapter is divided into two parts, the first part going into detail about concepts concerning the π-execution engine and the second part bringing out details about the visualization of π-calculus systems. For the execution engine an underlying data structure needs to be defined. For this data structure a lossless mapping from the processes defined in π-calculus syntax has to be specified. The data structure also has to be capable of serving as the foundation for implementing the behavioral rules for the evolution of π-calculus systems based on the reduction semantics. How the behavior of the reduction semantics is reproduced by execution rules specified for the proposed data structure will be addressed in the succeeding section. To conclude the first part, an overview of the static structure of the prototypical implementation PiVizTool is given.

The second part focuses on visualization aspects for π-calculus systems and therefore includes a description of the flow graph extension utilized in the π-calculus simulator. Since the visualization is directly connected with user interaction, corresponding ideas will be brought up. Finally some implementation details concerning the implemented visualization in the PiVizTool will be given, likewise.

4.1 The Execution Engine

An execution engine for π-calculus agents needs a data structure, which is modified during the execution holding all necessary information about the agents. The underlying data structure will be defined in the following. For the execution, rules have to be specified determining the behavior of the agents based on the reduction semantics described in section 2.2.2. A further task for the execution engine is the provision of the information necessary to compose the graphical representation of the system. For this a matching of prefix actions has to be implemented. The matching of prefix actions, for example, regards constraints like the scopes of names, being choices of the same summations or being part of a replicated agent. Besides providing information about the linking structure of the π-calculus system, the execution engine has to handle the execution of communications and τ actions. After such an execution, the information about the linking structure has to be refreshed and preprocessed for redisplaying.

4.1.1 Specification of the Data Structure

The proposed structure is a tree structure. It is applicable to the π-calculus since the π-calculus consists of splits and decisions, but no joining constructs. Synchronization between multiple agents or the components of an agent is done using positive and negative prefixes that block further execution until the respective counterpart becomes available.

Definition 2 (Tree)**.** A tree is a directed acyclic graph (N, E), where N is a finite set of nodes and $E \subseteq (N \times N)$ is a finite set of directed edges. For a directed edge two functions are defined as $source : E \rightarrow N$ and $target : E \rightarrow N$, where $source$ returns the node the edge originates from and $target$ returns the node the edge leads to. The following conditions have to apply to this tree:

i: $N \neq \emptyset$

ii: $\forall e \in E : source(e) \neq target(e)$

iii: $\exists! n \in N : \nexists e \in E : target(e) = n$. This node n is called the root node of the tree: $root$.

iv: $\forall n \in N \backslash \{root\} : \exists! e \in E : target(e) = n$

Definition 3 (Ancestor Function)**.** The ancestor function $anc : N \rightarrow \mathcal{P}(N)$ assigns each node $n \in N$ a finite set of nodes $A \in \mathcal{P}(N)$. The set A contains all nodes $n_i \in N$ that lead from the root node to n through a sequence of directed edges. A root node's ancestor set is always the empty set: $anc(root) = \emptyset$.

Every construct of the π-calculus needs a mapping to the tree structure to obtain a tree from the agents defined in π-calculus syntax. The π-calculus consists of the following constructs that need to be mapped:

- Prefixes
  - Send Prefix or Negative Prefix: $\overline{x}.\langle y\rangle$
  - Receive Prefix or Positive Prefix: $x(z)$
  - Tau Prefix: τ

  - Match Prefix: $[x = y]\pi$ or $[x \neq y]\pi$
- Sequence as denoted by "." within the process definitions
- Summation: $M + M'$
- Composition: $P \mid P'$
- Defined Agents: $A(\tilde{y})$
- Restriction: $\mathbf{v}zP$
- Replication: $!P$
- Inaction: $\mathbf{0}$

For most of these constructs special tree node types will be created, which store special properties like for example the list of parameters and the name of a defined agent or the two names that are compared during a match. Additionally, a dedicated root node is needed and a node of type "AgentDefinition". The "AgentDefinition" node is the root node for the process definitions of defined agents that have to be stored for unfolding. Hence, the set of tree node types can be defined as follows:

Definition 4 (Tree Node Types)**.**

$$\begin{aligned} T \;=\; & \{Send, Receive, Tau, DefinedAgent, Match, Restriction, Replication, \\ & Summation, Composition\} \\ & \cup\{Root, AgentDefinition\} \end{aligned}$$

Definition 5 (Type Function)**.** The function $type : N \rightarrow T$ assigns a node type to each node.

All of the tree nodes have two properties in common. These are their process identifier (pid) and an attribute storing the process name. The process identifier provides the information if nodes belong to the same component of an agent or are part of parallel components. It will furthermore be needed for the scoping of restricted names within and between agents. The process name holds the information, which agent a subtree belongs to. This will be important during visualization, when the capabilities of parallel components of an agent are assigned to its graph node. In the following the mapping to tree nodes and special properties of the tree nodes are defined.

Prefix Nodes Four node types for the prefix nodes will be introduced. These are *Send*, *Receive*, *Tau*, and *Match*. The send and receive nodes have two additional fields. The first contains the name of the channel the interaction takes place on and the second contains a set of names sent or received during the interaction. Figure 4.1 shows a visual representation of the prefix nodes $\overline{y}\langle x\rangle, y(x)$ and τ in the tree structure. Match constructs are mapped to nodes of type "Match" with two special fields containing the

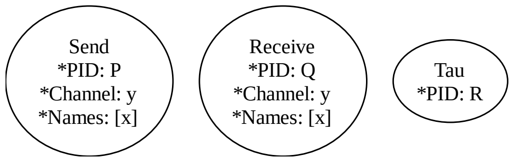


Figure 4.1: Prefix nodes

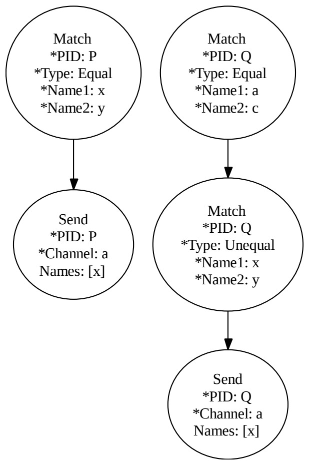


Figure 4.2: Match node

names to be compared and an additional field specifying the type of match. The type of comparison can be equality or inequality match. More than one node of this type may be encountered in sequence in case several decisions have to be made, like for example in $[a = b][x \neq y]\overline{a}\langle x\rangle.\mathbf{0}$. Because of the commutativity of conjunctions, the order of consecutive match nodes in a sequence is not of importance. The last match node in this sequence has one child node of type *Send*, *Receive*, or *Tau* that specifies the associated action. Examples for match tree nodes representing the processes below are presented in Figure 4.2.

$$\begin{aligned} P(a,x,y) &\overset{def}{=} [x = y]\overline{a}\langle x\rangle.\mathbf{0} \\ Q(a,c,x,y) &\overset{def}{=} [a = c][x \neq y]\overline{a}\langle x\rangle.\mathbf{0} \end{aligned}$$

Sequence The sequence construct "." is represented by the ancestor-descendant relationship of the tree nodes. Every node having a type different from summation and composition must either have exactly one or no child node. If an action a is sequentially done before another action b, the node representing action a will be an ancestor of the node representing action b.

Summation A summation behaves like executing only one choice of an agent out of a set of choices. Upon choosing one, the others are discarded. Which part will be picked depends on the actions available. This includes some non-determinism on the choice if more than one action is available. Summations in the π-calculus have the syntax $M_1 + M_2 + \ldots + M_n$ with $M_1, M_2, \ldots, M_n$ as possible choices. The summation

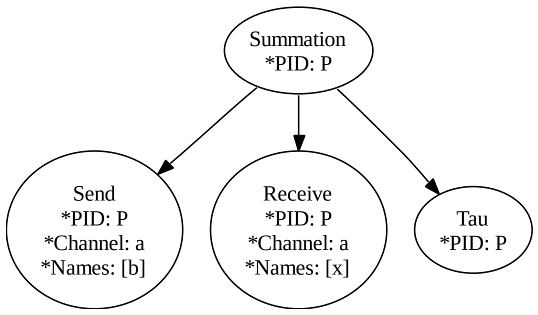


Figure 4.3: Summation Node

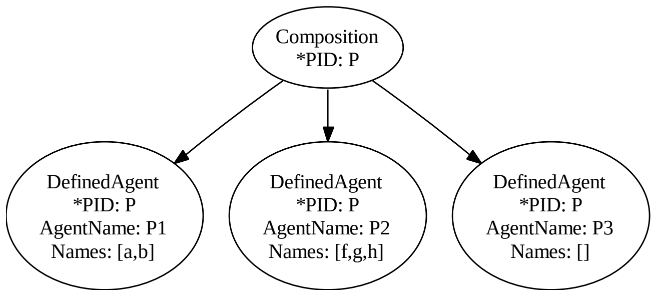


Figure 4.4: Composition Node

is represented as a node of type "Summation" with the choices as this node's children. Figure 4.3 represents the process definition

$$P(a, b) \overset{def}{=} \overline{a}\langle b\rangle.\mathbf{0} + a(x).\mathbf{0} + \tau.\mathbf{0}$$

Composition Compositions consist of components of an agent running in parallel that may interact with each other using channels to exchange names. The π-calculus notation for composition is $P_1 \mid P_2 \mid \ldots \mid P_n$ with the agents $P_1, P_2, \ldots, P_n$ running in parallel. For this construct a new node type "Composition" is needed. The concurrent components are inserted as the composition node's children. Figure 4.4 shows an example of the process definition

$$P(a, b, f, g, h) \overset{def}{=} P_1(a, b) \mid P_2(f, g, h) \mid P_3$$

Composition and summation nodes are the only node types in the tree that may or actually should have more than one child. If they have exactly one or no child node, optimization might take place, which removes the composition or summation node. This can already be done during the parsing step, since the tree is built at this time and the parser is able to detect how many children these nodes will have. If they do not have enough children, creation of these nodes will be skipped. These two node types do not have any further properties aside from the common process identifier and process name.

Defined Agents Defined agents are represented by the "DefinedAgent" node. Special properties of this node type are the name of the agent and the list of parameters that will substitute the according parameters of the process definition body the agent is related to. Figure 4.5 depicts the defined agent node for an agent with the name A

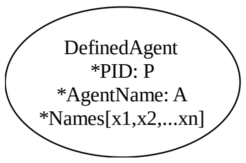


Figure 4.5: Defined agent node

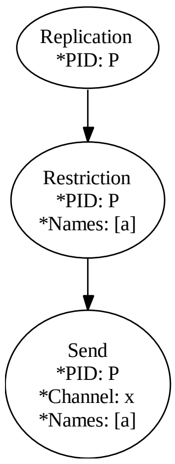


Figure 4.6: Replication and restriction nodes

and parameters $x_1, x_2, \ldots, x_n$. The list of parameters of a defined agent has to include all of the corresponding process bodies' free names. Otherwise inconsistencies during name substitution might occur. To exemplify this issue we define two agents:

$$\begin{aligned} P(a) &\overset{def}{=} a(x).Q(x) \\ Q(x) &\overset{def}{=} \overline{x}.\mathbf{0} \end{aligned}$$

Referring to the structural congruence rule SC-UNFOLD from Table 2.1 the process definition of agent P is congruent to the process definition:

$$P(a) \overset{def}{=} a(x).\overline{x}.\mathbf{0}$$

Depending on the implementation this example with a defined agent Q not having the name x in its parameter list may lead to different results. If the agent Q is unfolded before an arbitrary name is received over the channel a, the free name x within Q would be substituted by that name. Else if the agent is unfolded after the name has been received and substitution has taken place, the x in Q would not have been replaced.

Restriction, Replication and Inaction For restrictions a node of type "Restriction" is introduced. It holds the set of restricted names. Replications are represented by a node of type "Replication" that has no further fields. Inaction does not have a representation in the tree. If a branch in the tree ends, this corresponds to the part of the agent becoming inactive. Replication and restriction tree nodes are shown in Figure 4.6 corresponding to the process definition:

$$!(\mathbf{v}\ a)\ \overline{x}\langle a\rangle.\mathbf{0}$$

4.1.2 The Execution Cycle

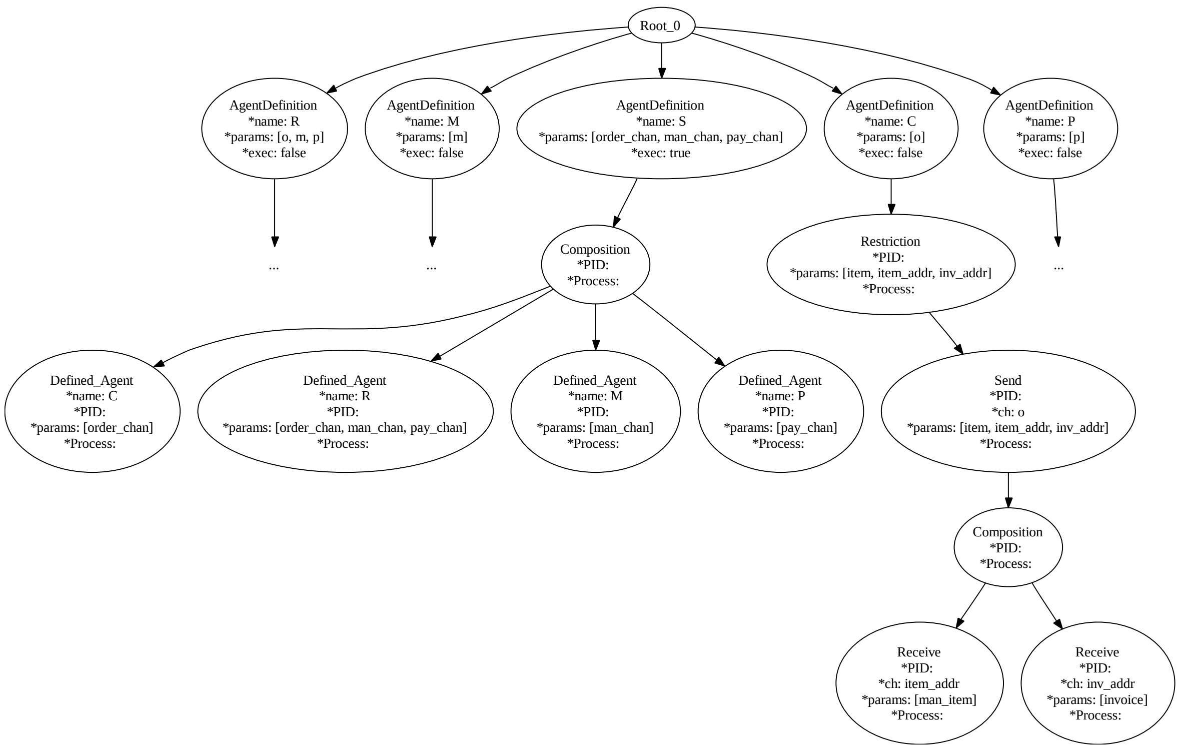
For the execution, the rules of the reduction semantics have to be mapped onto execution rules applicable to the proposed data structure. By using the reduction semantics in this work, only “closed”π-calculus systems are examined. This means that the observed agents do not interact with agents of the environment not specified in the π-process definitions. All the agents in this system interacting with each other have to be fully specified.

After the initial parsing of the process definitions, a tree containing all definitions of agents involved in the system becomes available to the execution engine. This tree will be called “definition tree” further on. The definition tree will not be changed during execution, since definitions of agents have to be stored during the entire execution time of the system, given that they might be needed at any point of time if a defined agent has to be resolved. An extract from the definition tree of the reseller example is depicted in Figure 4.7.

As can be seen, the reseller example contains the definitions of the five agents customer, reseller, manufacturer, payment organization, and the system. Besides their names and parameter list, their entire course of process in terms of prefix actions and agent resolutions is stored there. The field *exec* represents the pre configuration of agents initially being selected for execution.

For the actual execution a second tree, called “execution tree”, will be composed. All process definitions of agents in the definition tree that have the *exec* flag set to `true` will be copied to the execution tree. If several agents are set for execution, their process definitions will be inserted as siblings under the root node of the execution tree. Being direct children of the root node is equivalent to these children having a composition node as parent, which means that all children processes of the root node are executed in parallel. Referring to the reseller example, the execution tree would have exactly one child in the beginning, namely the process body of the agent S. The result is depicted in Figure 4.8.

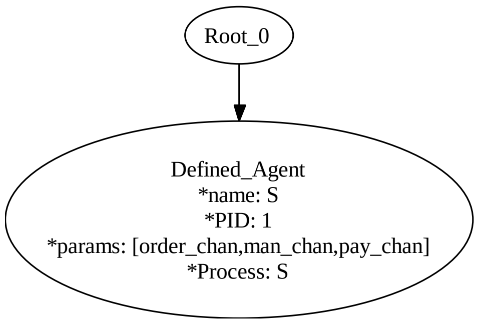
An extract of the steps that are taken by the π-calculus simulation environment through an entire execution cycle is depicted in Figure 4.9 using the UML Activity Diagram notation [51]. As can be seen, the first step after parsing and building the initial trees



Root_0
AgentDefinition *name: R *params: [o, m, p] *exec: false
AgentDefinition *name: M *params: [m] *exec: false
AgentDefinition *name: S *params: [order_chan, man_chan, pay_chan] *exec: true
AgentDefinition *name: C *params: [o] *exec: false
AgentDefinition *name: P *params: [p] *exec: false
...
Composition *PID: *Process:
Restriction *PID: *params: [item, item_addr, inv_addr] *Process:
Defined_Agent *name: C *PID: *params: [order_chan] *Process:
Defined_Agent *name: R *PID: *params: [order_chan, man_chan, pay_chan] *Process:
Defined_Agent *name: M *PID: *params: [man_chan] *Process:
Defined_Agent *name: P *PID: *params: [pay_chan] *Process:
Send *PID: *ch: o *params: [item, item_addr, inv_addr] *Process:
Composition *PID: *Process:
Receive *PID: *ch: item_addr *params: [man_item] *Process:
Receive *PID: *ch: inv_addr *params: [invoice] *Process:

Figure 4.7: Extract from the definition tree of the reseller example

will be to scan the execution tree for available and blocked prefix actions, which means finding all capabilities of the current system's agents. This phase will be called "scanning phase". By determination of all capabilities, irrespective of them being blocked or active, the linking structure of the system is discovered and can be visualized. Furthermore, determining the linking structure is a prerequisite for choosing the next communication or τ action to be executed if such is available. If only blocked communications or no more capabilities at all reside in the system, no further evolution will be possible. On the other hand after an active communication or τ action is chosen, its execution is triggered and the "execution phase" is entered. The execution phase



Root_0
Defined_Agent *name: S *PID: 1 *params: [order_chan,man_chan,pay_chan] *Process: S

Figure 4.8: Initial execution tree of the reseller example

is followed by another scanning phase determining the new linking structure of the system. This cycle of scanning, visualizing, choosing execution steps and execution is repeated until no more actions are available or the user chooses to end the simulation of the system.

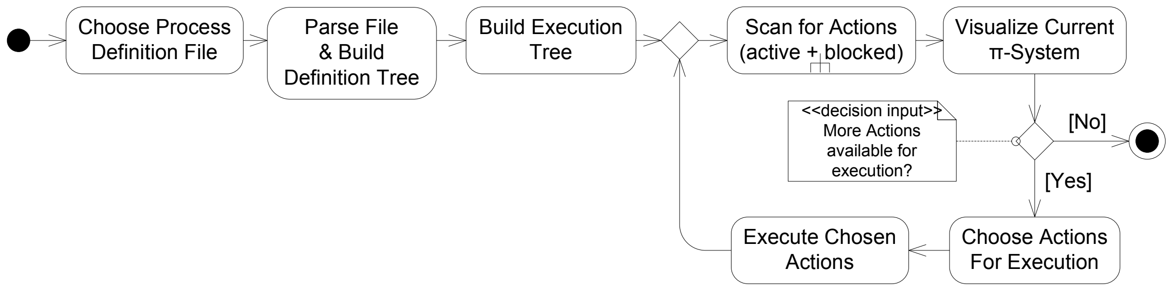

Choose Process Definition File
Parse File & Build Definition Tree
Build Execution Tree
Scan for Actions (active + blocked)
Visualize Current π-System
<<decision input>> More Actions available for execution?
[No]
[Yes]
Execute Chosen Actions
Choose Actions For Execution

Figure 4.9: Extract of the execution cycle of the π-calculus simulation environment

To find capabilities, the initial execution tree from the reseller example (Figure 4.8) has to be transformed, because it only contains defined agent nodes. A transformation will convert each subtree until a prefix node is present in each branch. In this transformation, for example, defined agent nodes will be resolved if they are encountered instead of a prefix node. Additionally, composition nodes are not allowed as direct children of the root node, since siblings under the root node are already interpreted as process bodies running in parallel. They will be executed instead. Beyond these two, restriction nodes may not become children of the root either. Consequently, the tree has to be transformed until none of the named node types are children of the root node. This transformation will be done as part of the scanning phase, because it needs to be redone after every execution. As for example defined agent nodes may again hide prefixes, because only the upmost defined agent nodes have been resolved previously.

Before taking a closer look at the scanning phase for determining the linking structure of the system and the execution phase, execution rules for each node type will be defined, since they are needed in both phases.

4.1.3 Execution Rules for the Tree Nodes

The handling of some node types depends on the phase of execution. In the scanning phase the tree is browsed for prefixes. The scanning phase is again divided into two stages. The first stage finds all prefixes that are not blocked by other prefixes and the second collects all other prefixes not regarded by the first stage. A more detailed description of these stages will be given in 4.1.4. In the first stage some node types encountered on the path are executed. In the second stage of the scanning phase no execution of nodes will take place. During execution phase all nodes on the path starting from the root to the selected prefix nodes, including these prefix nodes, are executed. Selected prefix nodes in this case are either a send and receive node, representing the communication chosen for execution, or a tau node. In the following the behavior of each node type concerning the two phases and the reduction semantics introduced in section 2.2.2 will be examined.

Summation nodes will not be changed in the scanning phase. Regarding the reduction rules Axiom 1 (Equation 2.4) and Axiom 2 (Equation 2.6), the behavior of summation constructs is defined to discard all choices, except the one containing the selected action. The axioms of structural congruence Sc-Sum-Comm and Sc-Sum-Assoc from Table 2.1 allow the choices of a summation to be placed in any order without special precedence between them. If a summation node is encountered during execution phase being an ancestor of a selected prefix node, it will be executed. For a summation node this means that the node itself and all its subtrees except the one containing the selected prefix node will be deleted from the execution tree. The subtree containing the selected prefix will be set as the child of the former summation node's parent.

As already stated, composition nodes will be executed instead of becoming direct children of the root node. Rule 2.7 of the reduction semantics declares that one component having a reduction, e.g. containing a selected prefix, does not affect the other components running in parallel. This rule will be applied in the execution phase. The other rules Sc-Comp-Comm and Sc-Comp-Assoc from Table 2.1 are directly reflected by the tree structure, which defines no order on the children of a node and implements no precedence of operation between them. A composition node can be executed in both phases if it is encountered directly beneath the root node. Execution involves the generation of a new unique process identifier for the process body of each child. After replacing the process identifiers in each subtree, the composition node will be deleted from the tree and its children will be set as the root node's children. If the current phase is the execution phase, the subtree(s) of the former composition node containing the selected prefix(es) will need further handling. Composition nodes on lower levels of the tree will not be changed at all, since the branching of the tree still needs the information if it presents a summation or composition. For the same reason, summation nodes are not removed from the tree, unless one of their descendant prefix nodes is executed.

Defined Agents will be resolved during the scanning phase only, because they hide information about prefix nodes contained within their process bodies. The process body of a defined agent may contain other defined agent nodes. Resolving of agent nodes will be done until a prefix node is encountered in each branch of the subtree representing the agents process body or until the branch becomes inactive. Some restrictions have to be imposed on the definition of agents. For example, agents containing unguarded calls to themselves will be prohibited, since this will lead to infinite loops, doing continuous agent resolving, while trying to find prefixes in each subtree. As a simple example the following definition is not allowed:

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

This agent has the behavior of sending data y along the channel x infinitely often. As a workaround this process can be defined using replication, e.g.:

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

One more restriction prohibits the parameter list of an agent definition node in the definition tree to contain duplicate names. For example the definition of an agent like the following is not allowed:

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

Such a definition causes problems if for example used in the context presented below, because no decision can be made how the substitution of each of the x's in A by c or e is realized.

$$B(a) \overset{def}{=} a(c, d, e).A(c, d, e)$$

Axiom SC-UNFOLD (Table 2.1) describes how to resolve a defined agent. Applied to the tree structure the following actions are taken: The definition tree will be scanned for an agent definition node with a name matching the agent name contained in the defined agent node of the execution tree. If no agent definition node with this name is found in the definition tree, a runtime error is raised. Beyond this the number of parameters has to match, otherwise another runtime error is raised, because the name substitution will not work accurately. If an agent definition node satisfying these constraints is found, its process body is copied from the according template in the definition tree and replaces the defined agent node in the execution tree. Afterwards all the names, as they were stated in the parameter list of the replaced node, have to substitute the names of the templates parameter list in the copied and newly inserted subtree.

Match nodes will be executed as soon as they are encountered during the first stage of the scanning phase. Relating to the axioms SC-MAT and SC-MATU (Table 2.1) the two names contained in the match node will be compared. If the match type is *equal*, axiom Sc-Mat is applied. In the event of the two names being equal, the match node is removed from the execution tree and its subtree is connected to the match node's former parent node. In case of the names not being equal, under the restriction of the match type being set to *equal*, the entire subtree including the match node will be removed and this part of the agent becomes inactive. Axiom Sc-MatU is applied in a similar manner, just the other way around. The entire subtree is removed if the two names match and only the match node is removed in case the names are not equal. Since match nodes are executed during the first stage of the scanning phase, they will not be encountered during execution of a selected prefix, as will become clear after the execution phase is explained in more detail in section 4.1.5.

Changes to replication nodes and their subtrees will only be made during the scanning phase in two cases: The first is if defined agent nodes are met in the subtree during the first scanning stage. These have to be resolved in order to acquire information about prefixes possibly hidden in the agents' process bodies. The second case is that match nodes as descendants of replication nodes being encountered in the first stage of the scanning phase will be executed. No substitution of the names being compared within these match nodes is possible any longer as they are not preceded by any input prefix nodes. All other structures in the replication subtree, including the replication node, are maintained. Additionally, such changes will only be done during the first scanning phase of the entire execution cycle, the replication subtree takes part in. Succeeding scanning phases need not make any changes, because the replication subtree will never be changed in the execution phase and therefore agents are already resolved and match nodes will not be encountered anymore. During the execution phase, instead of changing the replication subtree, a copy of the subtree excluding the replication node will be produced in case a selected prefix is contained within. The copy will be inserted in the execution tree as a sibling to the replication subtree and will be used as a basis for further handling. The user selected action/communication now points to the prefix node in the copied tree that represents the same node as was chosen in the replication subtree. Henceforth, the copied subtree will be used for the execution of the prefix. This behavior matches the axiom Sc-Rep (Table 2.1), where an exact copy of the replication process body (without the replication) is created, running in parallel and being modified.

For the handling of restriction nodes a restriction table, which keeps track of scopes of names during execution, is introduced. Each entry of the restriction table holds a set of process identifiers, a set of names, and a flag providing the information if the entry is a temporal entry or not. Hence, a restriction table entry is defined as a triple $(P, C, \textit{temporal})$, with $\textit{temporal} \in \{\texttt{true}, \texttt{false}\}$, P as a subset of all process identifiers contained in the system, and C as a subset of all names in the current system, respectively. Such an entry is interpreted in the following way: All names mentioned in the set of names share the same scope that is produced by all agents having one of the process identifiers contained in the set of process identifiers belonging to this entry. Accordingly, several names may have the same scope. Furthermore, the same name may belong to several different scopes, whereas these scopes have to be pairwise disjoint. A name associated to different scopes is interpreted as a different name for each one of them. In case of the scopes becoming intersected during execution, e.g. by scope extrusion, the resembling name of the receiving scope will be renamed to avoid name clashes. The renaming is done before the scope of the other name is extended. Equally, one process identifier may be contained in the set of process identifiers of several entries in the restriction table, but as a consequence to the constraint just mentioned, the sets of names of these entries have to be pairwise disjoint. Resulting from this, a tuple containing a process identifier and a name points to exactly one entry of the restriction table if this entry exists. This fact is important when querying the restriction table during execution. Both sets of an entry must not be equal to the empty set, otherwise the entry is useless and can be deleted from the restriction table.

Definition 6 (Scope)**.** The scope of a name within an agent is defined as a function $\mathit{scope} : (\mathit{name}, \mathit{pid}) \in (C \times P) \to \mathcal{P}(P)$.

How restriction nodes in the execution tree are actually handled depends on the phase. During the scanning phase restriction nodes will be executed in the first stage of scanning if they are encountered on the path. A special case has to be taken into account in the event of the restriction node being a descendant of a replication node. Since replication agents are copied and only their copies are executed, the names restricted to the replication agent and to the copied process body have to be distinguishable. Therefore the actual execution of the restriction node is deferred to the execution phase.

The general execution of restriction nodes includes to create a new unique name for each name mentioned in the list of restricted names of the restriction node and substituting the old names by the newly created ones in the subtree. The created names have to be scoped to the agent they are restricted to, so an entry is added to the restriction table. Since this entry is final the temporal flag will be set to `false`. Adding the new names to the restriction table includes a check if an entry for this scope already exists. If this is the case, the new names are just added to the existing entry, otherwise a new entry will be created. After finishing these steps, the restriction node is removed from the execution tree and its subtree is connected to its former parent node.

The special case execution of restriction nodes forgoes the creation of new unique names and does not delete the restriction node from the execution tree either. The only action taken is the creation of a temporal entry with the temporal flag set to `true` in the restriction table, containing all the names from the list of the restriction node.

If a restriction node is encountered on the path between the root node and the selected prefix node in the execution phase, it will be executed in the same manner as described by the general execution during the scanning phase. The only additional step taken is the removal of a possibly created temporal entry.

Executing restriction nodes is not the only action touching the restriction table. The table has to be updated during the execution of composition and replication nodes as well. While dealing with composition nodes, one agent is split into its components with all of them receiving an own process identifier. If the former agent was part of the scopes of restricted names, its components having a new process identifier should also be part of these scopes. Correspondingly, the entries in the restriction table containing the old processes identifier have to be updated by deleting the old identifier and adding the new ones to each of these entries' set of process identifiers. Replication nodes are handled similarly if they are part of a scope, except the process identifier of the replication subtree will not be deleted. Merely the new process identifier of the copied tree will be added to all entries containing the identifier of the replication agent.

Prefix execution is defined by Axiom 1 (Equation 2.4) and Axiom 2 (Equation 2.6) of the reduction semantics. Axiom 1 describes the reduction by communicating via input and output prefixes using the same channel and Axiom 2 describes the reduction of an unobservable action τ. Prefix nodes are only executed during the execution phase. The execution of prefix nodes of type τ is relatively simple and includes just the removal of the respective τ node from the execution tree as well as connecting its subtree to its former parent node. For the execution of a communication between an input and an output prefix both nodes in the execution tree are needed. The send prefix node will just be removed and its subtree will be connected to its former parent node, so the main part of the execution will be done during the execution of the receive prefix node. Information needed by the receive prefix node are the sent names in the right order and the process identifier of the sending agent. Before the old names in the receiving agent are substituted by the received names, a check to avoid possible name clashes has to be performed. Two kinds of name clashes have to be distinguished: firstly a collision of a received name and a restricted name of the receiving agent and secondly the collision of a received name and a bound name that is bound by one of the input prefixes of the receiving agent. In the first case, the restricted name of the receiving agent has to be renamed within the subtree of the receiving agent and within all other subtrees of agents belonging to its scope. This will be realized by finding the appropriate entry in the restriction table, which contains the process identifier of the receiving agent and the aforesaid name, and renaming it in all agents' process bodies, whose process identifiers are additionally contained in the found entry. Then, the corresponding entry of the restriction table will be updated, also exchanging the name. In the second case, the colliding bound name of the receiving agent will be renamed in its subtree only. Afterwards, the name substitution of the received names can take place if no more name clashes are detected. After the substitution has been carried out, a further check is needed if one of the received names was restricted to the sending agent. If so, scope extrusion has to be handled, which is done by adding the process identifier of the receiving agent to the entry of the restriction table containing the process identifier of the sending agent and the restricted name. Subsequently, the receive prefix node will be deleted from the execution tree and its subtree is connected to its former parent node.

This concludes the specification of the execution rules of the single tree nodes.

4.1.4 Determining the Linking Structure of the Pi-Calculus System

As was stated already, all agents currently active in the system have to be scanned for their capabilities in order to discover the link structure of the π-calculus system, which is needed for the visualization. Blocked and active actions have to be differentiated for the visualization, so the user is able to identify selectable and non-selectable communications. These two kinds of actions will be filtered out of the execution tree in two separate stages. An overview of the steps taken in the scanning phase is given by the UML Activity Diagram depicted in Figure 4.10 refining the "Scan for Actions" activity from Figure 4.9.

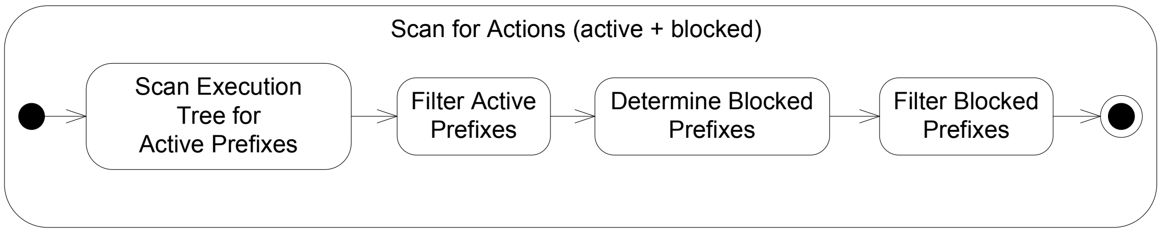

Scan for Actions (active + blocked)
Scan Execution Tree for Active Prefixes
Filter Active Prefixes
Determine Blocked Prefixes
Filter Blocked Prefixes

Figure 4.10: Steps of the scanning phase

In the first stage of the scanning phase all active actions will be searched for.

Definition 7. [Criteria for active actions] Active actions have to fulfill the following criteria:

1. Active prefixes are those prefixes not being sequentially behind other prefixes.
2. Input and output prefixes need a matching counterpart concerning the name of the channel, so the communication is not blocked.
3. Input and output prefixes having a matching counterpart, have to belong to the same scope as their communication partner regarding the utilized channel.

4. Input and output prefixes fulfilling the previous two conditions must not be choices of the same summation. This condition is subject to some special cases that will be explained later.

The first criterion of Definition 7 will be accomplished by the way of scanning the execution tree. Starting from the root each branch of the tree is visited. A branch will be traversed downward. As soon as a prefix node is found, this node is added to a set and the execution continues with the next sibling branch. If a branch splits into sub-branches via a summation or composition node and no prefix node has been visited yet within this branch, each of the sub-branches has to be scanned for the first prefix. In case a defined agent is encountered instead of a prefix node, this defined agent node will be resolved and the sub-branches of the inserted process definition will be scanned. Some of the nodes come across during downward traversal can already be executed. This can happen to match nodes without any restrictions, because they are not preceded by another prefix node anymore that is able to change one of the names within the match node. Alternatively if the match node was preceded by another prefix node, the scanning of this branch in the first stage would have been completed before finding the match node. Further node types that can be executed when come across, but have to yield to some conditions, are restriction nodes and composition nodes. If they are met as descendants of a replication node, only temporal entries for the restriction node will be created in the restriction table and the composition node will not be executed at all. Otherwise both node types will be executed as usual.

In the reseller example scanning the tree leads to firstly resolving the agent S, which leads to the intermediate execution tree depicted in Figure 4.11. It shows the composition of the reseller, customer, manufacturer and payment agent. Still this tree is not sufficient for finding prefixes and as mentioned before, composition nodes are not allowed as children of the root node, since siblings under the root node are already interpreted as parallel components. Hence, the composition node is executed and the defined agent nodes as its children receive own process identifiers. Additionally, the defined agent nodes have to be resolved, leading to the tree depicted in Figure 4.12. During scanning some more nodes are executed, which are sequentially before prefix nodes. In this example the restriction node of agent C (shaded in Figure 4.12) is executed, creating new unique names for the names mentioned in the restriction node and creating or updating restriction table entries for this agent.

Figure 4.13 shows the execution tree of the reseller example after the first scanning step. The prefix nodes, which are returned from the tree and will undergo further processing are shaded. Furthermore, the restriction table contains exactly one entry consisting of the restricted names and process identifier of the component of agent C that previously contained the restriction node. The current state of the restriction table is illustrated in Table 4.1.

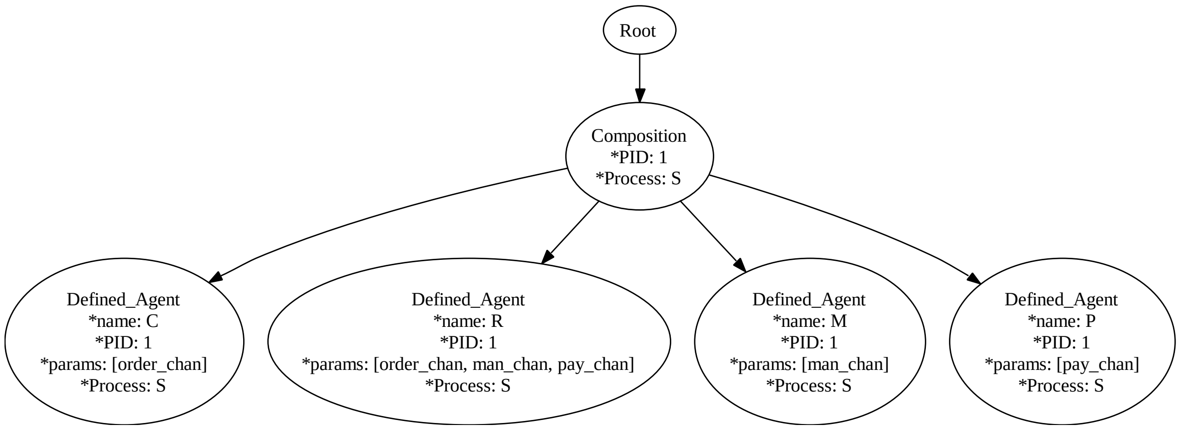


Figure 4.11: Intermediate execution tree of the reseller example

Process Identifiers	Names	Temporal Entry
$pid0\#0$	$item\#0, item\_addr\#1, inv\_addr\#2$	`false`

Table 4.1: Restriction table of the reseller example after the first scanning step

Succeeding the collection of all first level prefix nodes in the execution tree, the resulting set of nodes has to be checked for compliance with the other criteria mentioned in Definition 7. To ease this check an optimization can be introduced during the scanning phase, already sorting the nodes with respect to their node types and the channel names used for communication. Considering this, the scanning step yields two sets as a result. The first set contains tau nodes and the second set contains subsets of input and output prefix nodes with each of them only containing prefix nodes using the same channel name.

The set of active tau nodes does not have to be submitted to more checks, as tau nodes only have to yield to criterion 1 of Definition 7, which is already fulfilled after the scanning step. It is defined as follows:

Definition 8 (Active Tau Node Set)**.** The active tau node set AT is defined as $AT = \{t \in N \mid type(t) = Tau \ \wedge \ \forall n \in anc(t) : type(n) \in T \setminus \{Send, Receive, Tau\}\}$

In the reseller example the set of active tau nodes is currently empty.

$$AT = \emptyset$$

For the definition of the active send/receive prefix node set two more functions are needed:

Definition 9 (Pid Function)**.** The function $pid : N \to P$ returns the process identifier of a tree node.

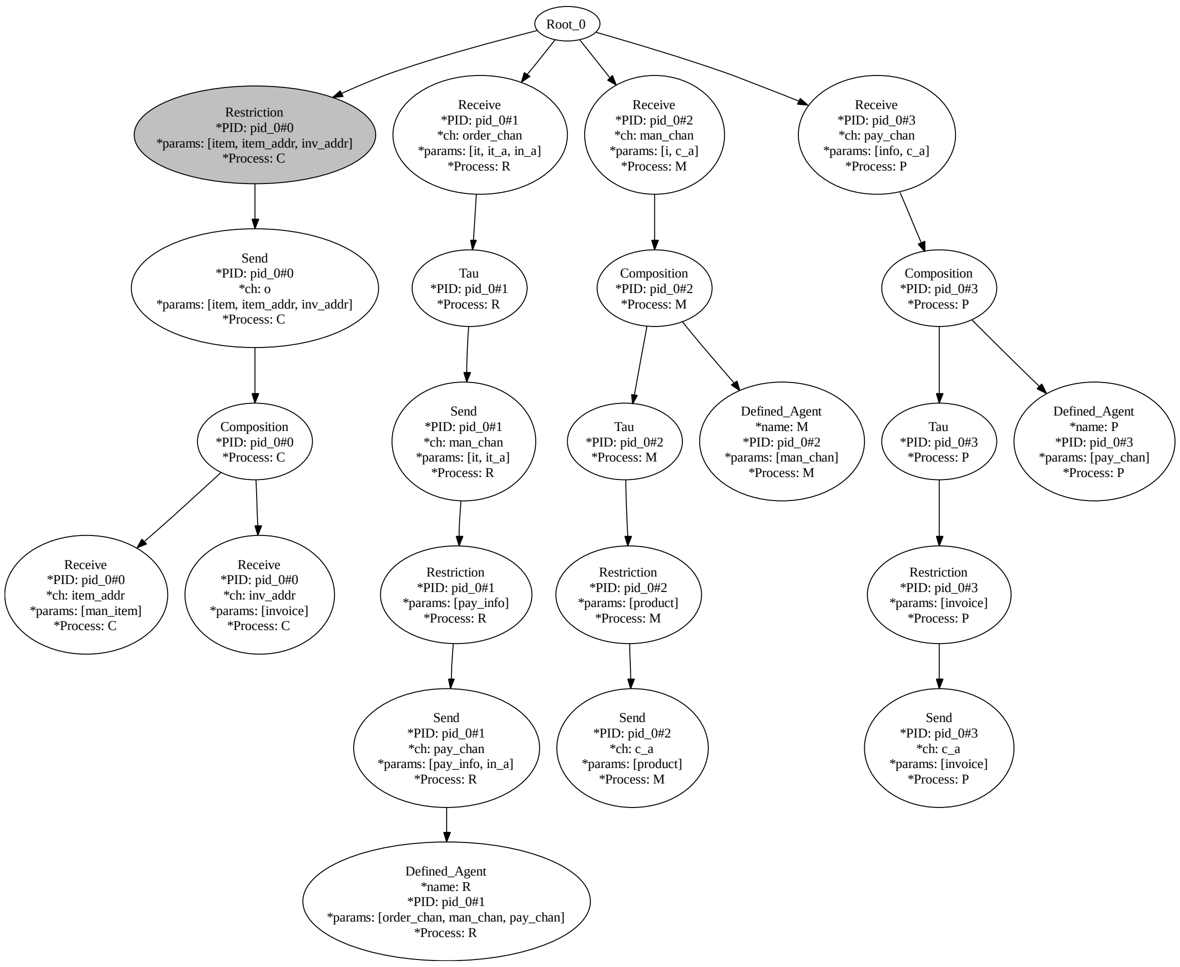

Root_0
Restriction *PID: pid_0#0 *params: [item, item_addr, inv_addr] *Process: C
Send *PID: pid_0#0 *ch: o *params: [item, item_addr, inv_addr] *Process: C
Composition *PID: pid_0#0 *Process: C
Receive *PID: pid_0#0 *ch: item_addr *params: [man_item] *Process: C
Receive *PID: pid_0#0 *ch: inv_addr *params: [invoice] *Process: C
Receive *PID: pid_0#1 *ch: order_chan *params: [it, it_a, in_a] *Process: R
Tau *PID: pid_0#1 *Process: R
Send *PID: pid_0#1 *ch: man_chan *params: [it, it_a] *Process: R
Restriction *PID: pid_0#1 *params: [pay_info] *Process: R
Send *PID: pid_0#1 *ch: pay_chan *params: [pay_info, in_a] *Process: R
Defined_Agent *name: R *PID: pid_0#1 *params: [order_chan, man_chan, pay_chan] *Process: R
Receive *PID: pid_0#2 *ch: man_chan *params: [i, c_a] *Process: M
Composition *PID: pid_0#2 *Process: M
Tau *PID: pid_0#2 *Process: M
Restriction *PID: pid_0#2 *params: [product] *Process: M
Send *PID: pid_0#2 *ch: c_a *params: [product] *Process: M
Defined_Agent *name: M *PID: pid_0#2 *params: [man_chan] *Process: M
Receive *PID: pid_0#3 *ch: pay_chan *params: [info, c_a] *Process: P
Composition *PID: pid_0#3 *Process: P
Tau *PID: pid_0#3 *Process: P
Restriction *PID: pid_0#3 *params: [invoice] *Process: P
Send *PID: pid_0#3 *ch: c_a *params: [invoice] *Process: P
Defined_Agent *name: P *PID: pid_0#3 *params: [pay_chan] *Process: P

Figure 4.12: Intermediate execution tree of the reseller example with resolved agents

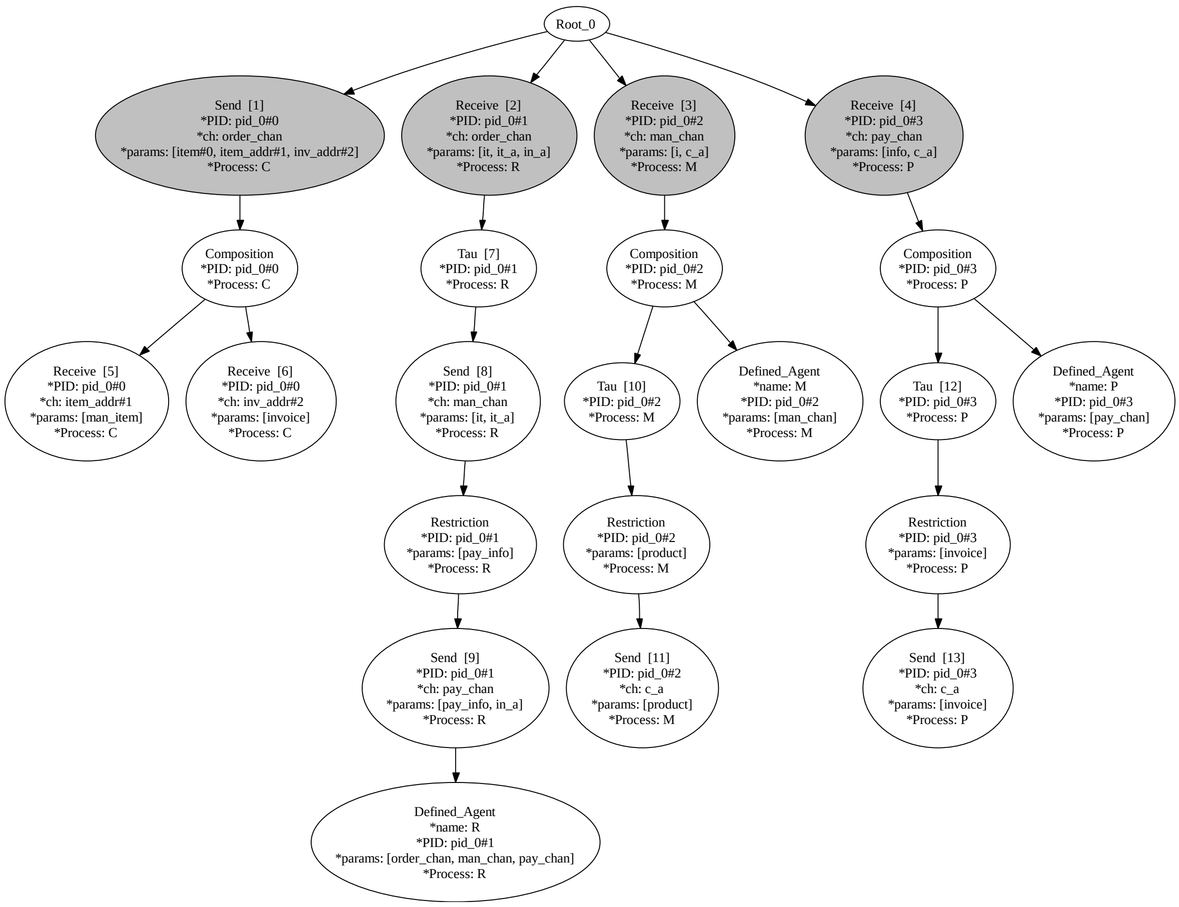

Root_0
Send [1] *PID: pid_0#0 *ch: order_chan *params: [item#0, item_addr#1, inv_addr#2] *Process: C
Receive [2] *PID: pid_0#1 *ch: order_chan *params: [it, it_a, in_a] *Process: R
Receive [3] *PID: pid_0#2 *ch: man_chan *params: [i, c_a] *Process: M
Receive [4] *PID: pid_0#3 *ch: pay_chan *params: [info, c_a] *Process: P
Composition *PID: pid_0#0 *Process: C
Tau [7] *PID: pid_0#1 *Process: R
Composition *PID: pid_0#2 *Process: M
Composition *PID: pid_0#3 *Process: P
Receive [5] *PID: pid_0#0 *ch: item_addr#1 *params: [man_item] *Process: C
Receive [6] *PID: pid_0#0 *ch: inv_addr#2 *params: [invoice] *Process: C
Send [8] *PID: pid_0#1 *ch: man_chan *params: [it, it_a] *Process: R
Tau [10] *PID: pid_0#2 *Process: M
Defined_Agent *name: M *PID: pid_0#2 *params: [man_chan] *Process: M
Tau [12] *PID: pid_0#3 *Process: P
Defined_Agent *name: P *PID: pid_0#3 *params: [pay_chan] *Process: P
Restriction *PID: pid_0#1 *params: [pay_info] *Process: R
Restriction *PID: pid_0#2 *params: [product] *Process: M
Restriction *PID: pid_0#3 *params: [invoice] *Process: P
Send [9] *PID: pid_0#1 *ch: pay_chan *params: [pay_info, in_a] *Process: R
Send [11] *PID: pid_0#2 *ch: c_a *params: [product] *Process: M
Send [13] *PID: pid_0#3 *ch: c_a *params: [invoice] *Process: P
Defined_Agent *name: R *PID: pid_0#1 *params: [order_chan, man_chan, pay_chan] *Process: R

Figure 4.13: Execution tree of the reseller example with shaded first level prefix nodes

Definition 10 (Channelname Function)**.** The function $channelname : N \rightarrow C$ returns the name used by a prefix node to send or receive. For other nodes the result of the function is not defined.

Finally, the active send/receive node set is defined as follows:

Definition 11 (Active Send/Receive Node Set)**.** For each π-calculus name i existing in the defined system a set L_i is defined as: $L_i = \{n \in N \mid channelName(n) = i \wedge type(n) \in \{Send, Receive\} \wedge \forall a \in anc(n) : type(a) = T \setminus \{Send, Receive, Tau\}\}$. The active send/receive node set AP results as $AP = \{L_i \mid |L_i| \geq 2 \wedge \forall n_1, n_2 \in L_i : scope(channelName(n_1), pid(n_1)) = scope(channelName(n_2), pid(n_2)) \wedge \exists n_1, n_2 \in L_i : (type(n_1) = Receive \wedge type(n_2) = Send)\}$

The set of send and receive prefix nodes, after the scanning step and before the filter active prefixes step, contains the following items with the numbers referencing the shaded nodes in Figure 4.13:

$$\{order\_chan \longrightarrow \{1, 2\}, man\_chan \longrightarrow \{3\}, pay\_chan \longrightarrow \{4\}\}$$

Each subset of the active send/receive node set has to be checked for fulfillment of criteria 2 - 4 of Definition 7. The following steps of processing the input and output prefix nodes set will be done during the “Filter Active Prefixes” step. Starting with a scoping check, each subset containing nodes communicating on differently scoped channels has to be split. New subsets that only include the nodes of the same scope regarding their communication channel are created and added to the set. The old subset is removed from the active send/receive node set. Input and output prefix nodes being siblings under the same summation node have to be filtered from the subsets. They are not able to interact, because only one choice can be taken. For each of the summation siblings a new subset will be created from the old one they were part of, including all the remaining nodes from the old subset that are no summation siblings. The old subset is again removed after the newly created ones are added. A special case has to be handled, where summation child nodes are not filtered out of the subset: If the input and output prefixes are siblings under a summation node, which again is a descendant of a replication node, a communication is possible between different instances of the replicated agent. To clarify this issue we will have a look at the following example agents A and B defined as:

$$A(n) \overset{def}{=} !(\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0})$$

$$B(n) \overset{def}{=} (\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0})$$

Agent B may either have a reduction sending a name on channel a or receiving a name on channel a if an appropriate counterpart is supplied by another agent in the system. If the system contains only the agent B, it would have no reduction at all. However, agent A has reductions in this case, as can be seen after rewriting its term, using the axiom Sc-Rep twice and then applying Axiom 1 to the two rightmost components:

$$\begin{aligned}
&!(\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0}) \\
&\overset{Sc-Rep*}{\equiv} \quad !(\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0}) \mid (\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0}) \mid (\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0}) \\
&\xrightarrow{Axiom1} \quad !(\overline{a}\langle n\rangle.\mathbf{0} + a(b).b.\mathbf{0}) \mid \mathbf{0} \mid n.\mathbf{0}
\end{aligned}$$

This also leads to a special case when handling compositions as descendants of replication nodes. Two interpretations of the possible communications are available. Either the two parallel components of a replicated agent communicate with each other, which will result in an unobservable action for the agent, or components of different instances of the replicated agent can communicate. Using the agent A from the example above, but replacing the summation symbol by a composition symbol, the two interpretations lead to different results after exactly one reduction has taken place:

$$\begin{aligned}
&A(n) \overset{def}{=} !(\overline{a}\langle n\rangle.\mathbf{0} \mid a(b).b.\mathbf{0}) \\
&\xrightarrow{\text{unobservable communication}} \quad !(\overline{a}\langle n\rangle.\mathbf{0} \mid a(b).b.\mathbf{0}) \mid n.\mathbf{0} \\
&\xrightarrow{\text{observable communication}} \quad !(\overline{a}\langle n\rangle.\mathbf{0} \mid a(b).b.\mathbf{0}) \mid a(b).b.\mathbf{0} \mid \overline{a}\langle n\rangle.\mathbf{0} \mid n.\mathbf{0}
\end{aligned}$$

After all these checks have completed, the subsets only containing one kind of prefix node, e.g. only send prefix nodes, will be deleted, because if no counterpart is available for a certain channel, no communication takes place.

Consequently, the two sets, active tau node set and active send/receive node set, contain only nodes that take part in communications and actions that are not blocked and thereby are selectable by the user for execution. The sets AT and AP specify all of the currently executable steps and communications of the π-calculus system.

Concerning the reseller example, the filter active prefixes step eliminates the subsets containing references to the receive prefix nodes of the manufacturer and payment organization, since no counterparts are available. The only references kept in this set are the ones to the send node of the customer (node 1, Figure 4.13) and the receive node of the reseller (node 2, Figure 4.13), so the only available communication in this system at this point is the customer sending an order to the reseller:

$$\{order\_chan \longrightarrow \{1,2\}\} \Rightarrow AP = \{\{1,2\}\} \tag{4.1}$$

Subsequently to the filter active prefixes step, the blocked prefixes have to be found in the second scanning stage. A scan of the entire current execution tree is done for this, returning all send and receive prefix nodes contained, again sorted by channel as mentioned above. No nodes will be executed. The only processing besides scanning for prefix nodes is the creation of temporal restriction table entries if needed. Those entries are disposed of after an execution step has been selected. Tau prefix nodes are not collected at all, because blocked τ actions have no influence on the linking structure and therefore do not need to be presented graphically at this stage. All nodes contained in the active send/receive node set and active tau node set resulting from the previous steps are subtracted from the node set acquired in this step, which leaves only references to blocked prefix nodes. From these, the linking structure concerning blocked communications will be deduced by performing the same filtering steps as stated in the filter active prefixes step above. An additional criterion is needed in the "filter blocked prefixes" step, which is not applicable to first level prefix nodes.

Definition 12. [Criteria for blocked actions] The criteria are defined as follows:

1. Blocked prefixes are those send and receive prefixes that are sequentially behind other prefixes.
2. Input and output prefixes need a matching counterpart concerning the name of the channel, so the communication is not blocked.
3. Input and output prefixes having a matching counterpart, have to belong to the same scope as their communication partner regarding the utilized channel.
4. Input and output prefixes fulfilling the previous two conditions must not be choices of the same summation. This condition is subject to some special cases that will be explained later.
5. Input and output prefixes of different agents or parallel components can not communicate on channels that are bound by an input prefix to either one of the agents or components.

Such binding of names mentioned by criterion 5 of Definition 12 is not encountered when working with first level prefix nodes, as are contained in the active send/receive node set, because they are not sequentially behind other input prefix nodes. According to criterion 5, all communications on channels with the channel name being bound within one of the counterparts by an input prefix have to eliminated. A prefix node whose channel name is bound in such a way can not communicate at all, momentarily. As an example, the agents A and B shown below should not have a blocked communication using channel a presently, because the actually received names may differ.

$$A(x) \overset{def}{=} x(a).a.\mathbf{0}$$

$$B(y) \overset{def}{=} y(a).\overline{a}.\mathbf{0}$$

Similarly to the active send/receive node set, the blocked prefix node set is defined.

Definition 13 (Blocked Prefix Node Set)**.** For each π-calculus name i existing in the defined system a set L_iB is defined as: $L_iB = \{n \in N \mid channelName(n) = i \wedge type(n) \in \{Send, Receive\} \wedge n \notin (L_i \in AP)\}$. The blocked prefix node set BP results as $BP = \{L_iB \mid |L_iB| \geq 2 \wedge \forall n_1, n_2 \in L_iB : scope(channelName(n_1), pid(n_1)) = scope(channelName(n_2), pid(n_2)) \wedge \exists n_1, n_2 \in L_iB : (type(n_1) = Receive \wedge type(n_2) = Send)\}$

In the reseller example the "determine blocked prefixes" step returns the following set of send and receive prefix nodes:

$$\{order\_chan \longrightarrow \{1,2\}, item\_addr\#1 \longrightarrow \{5\}, inv\_addr\#2 \longrightarrow \{6\},$$
$$man\_chan \longrightarrow \{8,3\}, pay\_chan \longrightarrow \{9,4\}, c\_a \longrightarrow \{11,13\}\}$$

After the subtraction of the nodes contained in the active node sets, the first subset concerning the channel name *order_chan* is erased from the set above. Due to lacking a matching counterpart (criterion 2 of Definition 12), the *item_addr*#1 and *inv_addr*#2 subsets are removed. They are blocked, but do not have a representation in the linking structure, so they can be left out. The *c_a* subset is removed, since it does not fulfill criteria 2 and 5 of Definition 12. The final set of blocked communications contains the entries stated in Equation 4.2 with the numbers referencing the tree nodes in Figure 4.13.

$$\{man\_chan \longrightarrow \{8,3\}, pay\_chan \longrightarrow \{9,4\}\} \Rightarrow BP = \{\{8,3\}\{9,4\}\} \tag{4.2}$$

Deduced from the final sets 4.1 and 4.2 the linking structure of the reseller system in the current state contains one active communication between the customer and the reseller and two blocked communications, taking place between the reseller and the manufacturer as well as between the reseller and the payment organization.

4.1.5 Execution of the Chosen Prefixes

By selecting an active communication or a node containing a τ action that is not blocked in the graphical representation of the π-calculus system, the next step for execution is chosen. In the execution phase all ancestor nodes of the selected prefixes are going to be executed, before executing the prefix node. Therefore, the nodes on the path from the root to each prefix node selected are processed as described above. The path just mentioned only contains nodes that have not been processed already in the first scanning step. For example match nodes will never be encountered in the execution phase, since prefix nodes selected for execution must not be descendants of other prefix nodes that would have the effect of blocking the match node. All information needed to process a match node is already available during the scanning step.

After the execution of the selected nodes, another scanning phase is triggered to determine the new linking structure of the system. If no active communications or τ actions remain, the system is not able to evolve any longer and thus it becomes inactive.

To finish the first execution cycle for the reseller example, the execution tree after executing the only selectable communication and scanning for the next actions is depicted in Figure 4.14. The customer agent C now consists of two components. One is waiting for the invoice and the other for the ordered product. The only agent having an active capability is the reseller being able to do an unobservable action (node 3, Figure 4.14). Referring to the previously blocked communications of the manufacturer and the payment organization, no changes have been made in the tree structure, yet. Hence, the results for the active and blocked node sets of the second scanning phase with the numbers referencing the tree nodes in Figure 4.14 are:

- $AT = \{3\}$
- $AP = \emptyset$
- $BP = \{\{4, 6\}, \{5, 9\}\}$

4.1.6 Implementation

The implementation of the simulation environment is separated into several packages with respect to the architecture described in section 3.3. A UML package diagram [26], depicting the packages and the dependencies between the packages is shown in Figure 4.15. Implemented packages are the execution engine package, visualizer, controller and helper classes packages. The execution engine package provides all the functionality needed to simulate the execution of π-calculus agents, which includes execution of steps and determining the linking structure of the π-calculus system. Other functions contained in this package are the reading of input files inclosing the process definitions and supplying information about the linking structure of the system to other components. The visualization package provides the graphical user interface and a visualization component, which is able to create a graphical representation of a π-calculus system's

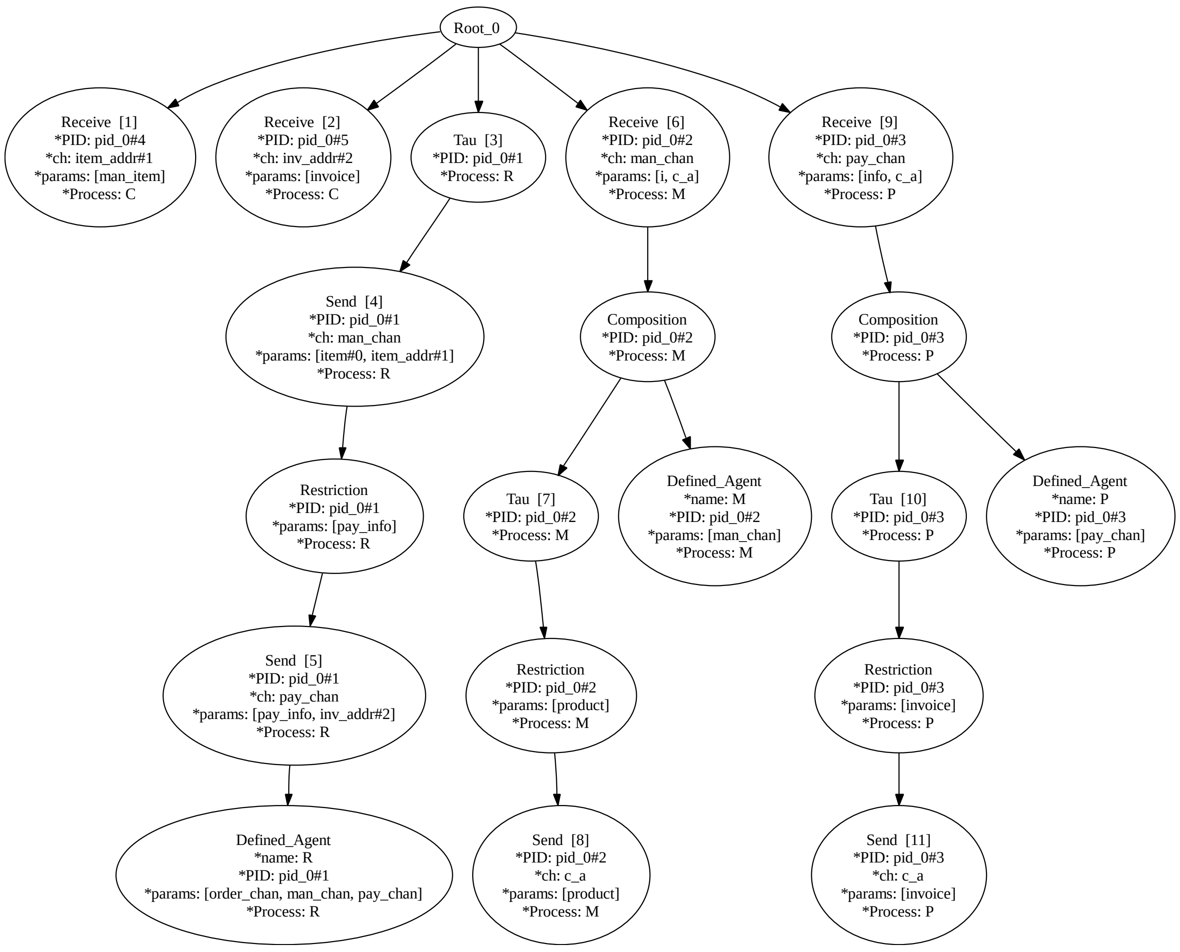

Root_0
Receive [1] *PID: pid_0#4 *ch: item_addr#1 *params: [man_item] *Process: C
Receive [2] *PID: pid_0#5 *ch: inv_addr#2 *params: [invoice] *Process: C
Tau [3] *PID: pid_0#1 *Process: R
Receive [6] *PID: pid_0#2 *ch: man_chan *params: [i, c_a] *Process: M
Receive [9] *PID: pid_0#3 *ch: pay_chan *params: [info, c_a] *Process: P
Send [4] *PID: pid_0#1 *ch: man_chan *params: [item#0, item_addr#1] *Process: R
Composition *PID: pid_0#2 *Process: M
Composition *PID: pid_0#3 *Process: P
Restriction *PID: pid_0#1 *params: [pay_info] *Process: R
Tau [7] *PID: pid_0#2 *Process: M
Defined_Agent *name: M *PID: pid_0#2 *params: [man_chan] *Process: M
Tau [10] *PID: pid_0#3 *Process: P
Defined_Agent *name: P *PID: pid_0#3 *params: [pay_chan] *Process: P
Send [5] *PID: pid_0#1 *ch: pay_chan *params: [pay_info, inv_addr#2] *Process: R
Restriction *PID: pid_0#2 *params: [product] *Process: M
Restriction *PID: pid_0#3 *params: [invoice] *Process: P
Defined_Agent *name: R *PID: pid_0#1 *params: [order_chan, man_chan, pay_chan] *Process: R
Send [8] *PID: pid_0#2 *ch: c_a *params: [product] *Process: M
Send [11] *PID: pid_0#3 *ch: c_a *params: [invoice] *Process: P

Figure 4.14: Execution tree of the reseller example in the second cycle of execution

linking structure, based on the information provided by components from the execution engine package. User input that is processed directly by components within this package is signaled to the controller component contained in the controller package, which forwards the request appropriately. Functions for creating unique names and process identifiers are included in the helper classes package. An additional functionality, which is needed in case of replications being executed, is finding a node representing a selected prefix in a second tree that is the identical copy of the first tree. Executing replications means to copy the entire subtree of the replication node and then continuing to work on the copied tree. For this a reference to the selected prefix node for execution has to be regained in the copy. Furthermore, the helper classes package contains defined signals for the communication between the graphical user interface and the controller, so the bonding of the graphical user interface to a special controller is kept on a low level.

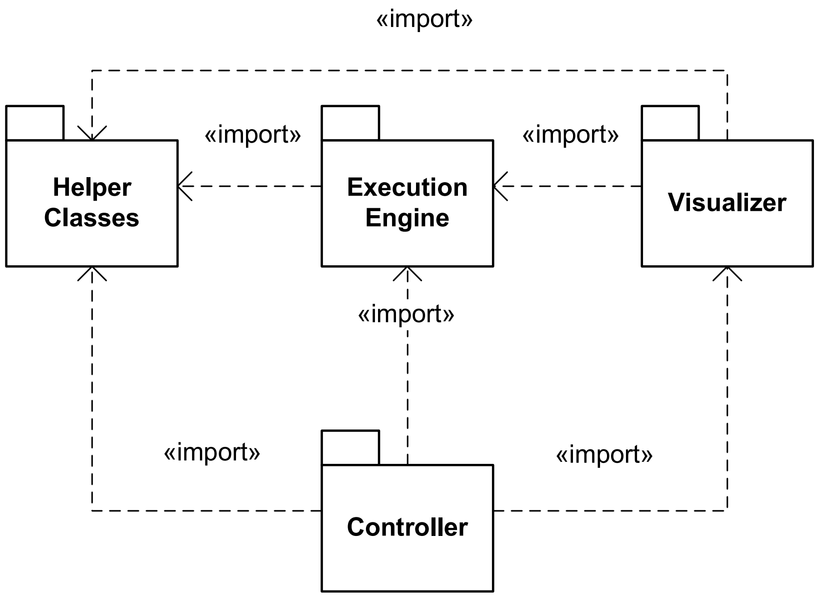


Figure 4.15: Package diagram for the simulation environment for π-calculus systems

Figure 4.16 shows a UML class diagram refining the packages discussed above. As can be seen in the execution engine package, each tree node type is implemented by an own class, specifying its special behavior. In addition to the data structure classes, a parser class having the task to create the data structure and an executor class, being able to work on the provided data structure, are defined. The tree character of the data structure is put forth by the parent and child association defined on the class *SimpleNode*. For creating the tree data structure, the parser uses the *Node* interface that supplies basic machinery for constructing the parent and child relationships between nodes. Class *SimpleNode* implements the *Node* interface and contains the basic attributes holding information about the process identifier (*pid*) and the name of the agent (*processName*) a node belongs to. All nodes that can actually be encountered within the data structure extend *SimpleNode* and thereby also posses these attributes. The special attributes each tree node type possesses, e.g. the two names being compared and the kind of comparison for a match node, are likewise enlisted in the class diagram, if existent.

The parser and the executor both have associations to the *ASTRoot* class. This class represents the root node of the tree structures and may not have a parent node. Two kinds of trees have been specified previously. These are the definition tree and the execution tree. In order to create and work with the execution tree, the executor triggers the parser to construct the definition tree from the input process definitions chosen by a user. Afterwards the executor composes the execution tree from the specifications in the definition tree and may start the execution. The executor holds two references to root nodes at all times during the execution. One of them points to the current execution tree and the other to the definition tree.

Finally the *PiExecutor* implements the *PiExecutorInterface*, which on its part should offer the only gateway for external components using the functionality implemented in the execution engine package. Using the interface decreases the coupling between the components using the functionality of the execution engine and its implementation.

In the current implementation, the *PiExecutorInterface* is used by the *Controller* and the graphical user interface class *PiMainFrame*. The controller, upon receiving signals from the user interface, triggers execution steps and the user interface fetches all information needed to visualize the π-calculus system via this interface.

The lower part of the class diagram depicted in Figure 4.16 will be taken up in the next section, explaining how π-calculus systems are represented graphically and how the information provided by the execution engine is transformed into the proposed visual representation.

4.2 Visualization Concepts

In this section the concepts for presenting π-calculus systems visually to the user will be explored. Firstly, the notation of flow graphs is extended for expressing available and blocked actions and grouping of agents for clarity and abstraction. Secondly, the way of transforming the given information about the linking structure of a system from node sets to a graph is explained. Finally, the ideas for user interaction directly with the visualized π-calculus agents in the graphical representations are described in combination with design and implementation details of the visualizing components.

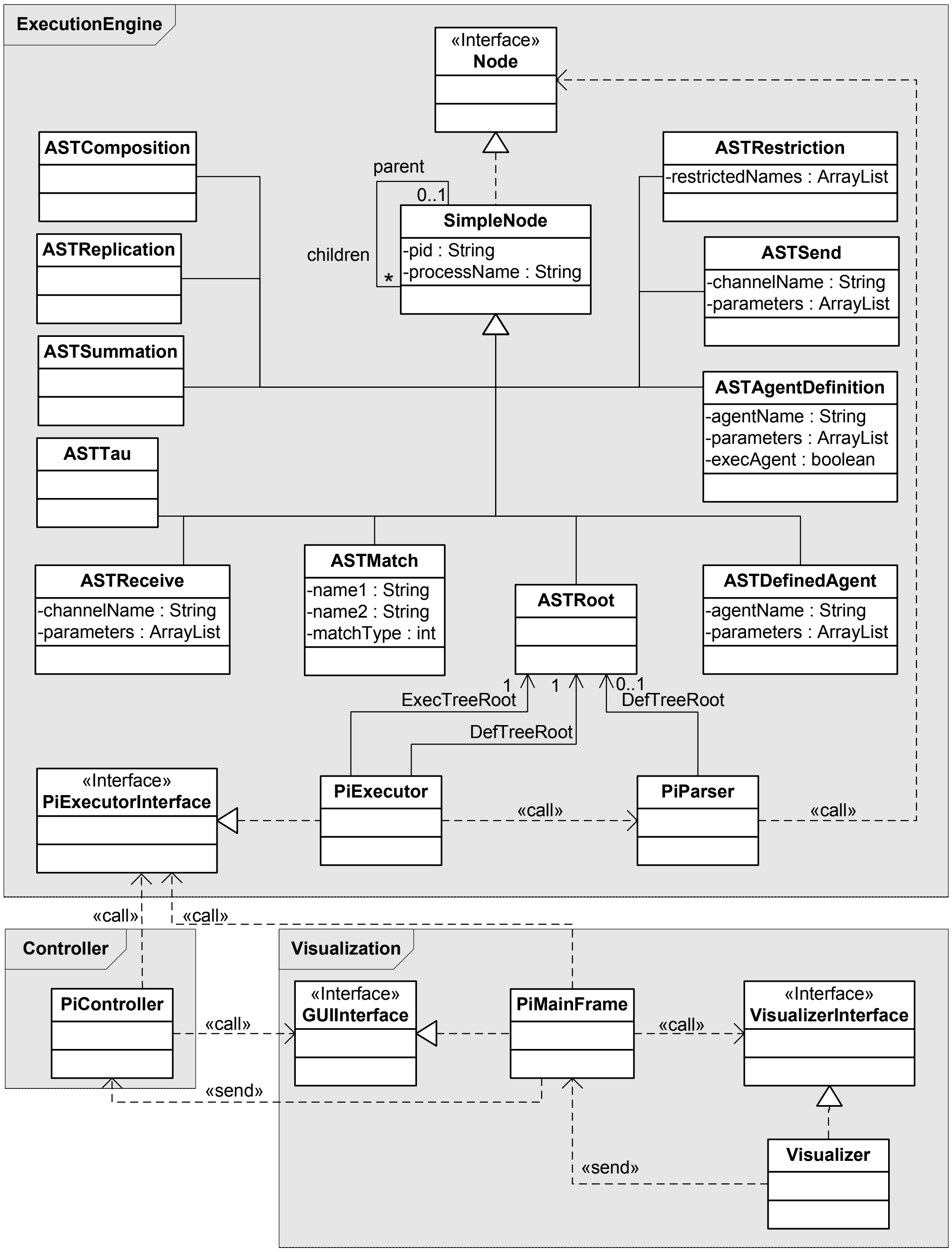


Figure 4.16: Class diagram for the simulation environment for π-calculus systems

4.2.1 Flow Graph Extension

An appropriate representation for the structure of a π-calculus system as well as the representation of its scoped names has to be found. Therefore, a notation similar to the flow graphs will be used. Differences between the two notations are, for example, that the basic flow graphs do not distinguish between active and blocked communications, which is in this case strikingly important, since users have to be able to visually differentiate between selectable and non-selectable communication steps. Additionally, the names of the channels information is sent over are the labels of the edge representing this communication. The direction of the communication is presented by a “dot” marking the end of the edge on the side of the agent receiving the information. Nodes representing agents of the system are visualized using rounded shapes. Moreover, τ actions that are not blocked have to be represented, since they are available for execution. This will be done by shading the node representing the agent with the τ capability. By selecting a shaded node, the unobservable action will be executed and previously blocked steps may become available.

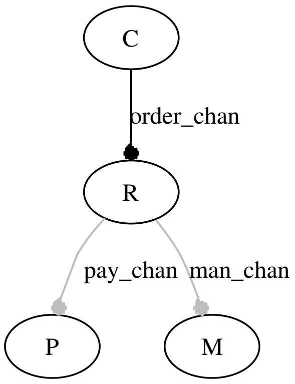


Figure 4.17: Initial state of the π-calculus system of the reseller example

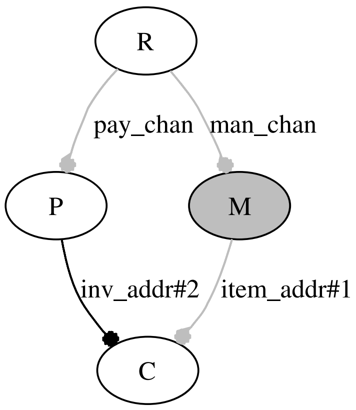


Figure 4.18: State during execution of the reseller example

Figure 4.17 depicts the initial state of the π-calculus system of the reseller example introduced in 2.2. As can be seen, the system consists of four agents, the customer C, the reseller R, the manufacturer M and the payment organization P, and has three capabilities in this state, which are communications using the channels $order\_chan, pay\_chan$ and man_chan. The communications over the last two mentioned channels are currently blocked. This can be perceived by the gray color of the links, meaning that they are not selectable for execution at the moment. The only action available for execution is the communication between the customer and reseller. Figure 4.18 shows the system's state after some steps of execution. The node of the manufacturer M is shaded, which means that a τ action is executable that furthermore blocks succeeding communication of the manufacturer. Another action currently available for execution is the communication between the payment organization and the customer.

In addition to the actions, the user may choose to display scopes of certain names. The scopes will be depicted by drawing dashed lines from the scoped name to each node representing an agent the name is restricted to. An example for displaying scopes concerning the reseller example is shown in Figure 4.19. In the example, the scopes for the names *item* and *pay_info* are displayed. The name *item* was created by the customer and forwarded by the reseller to the manufacturer for production. The name *pay_info* was created by the reseller upon forwarding the order to the payment organization, so the customer can be billed appropriately. As can be seen from the Figure, *item* is private to the customer, the reseller and the manufacturer, and *pay_info* is restricted to the reseller and the payment organization.

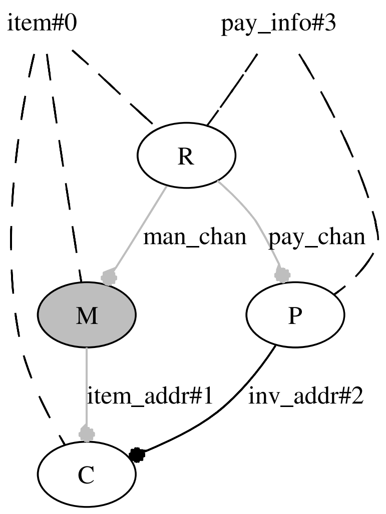


Figure 4.19: Selected scopes of the reseller example during execution

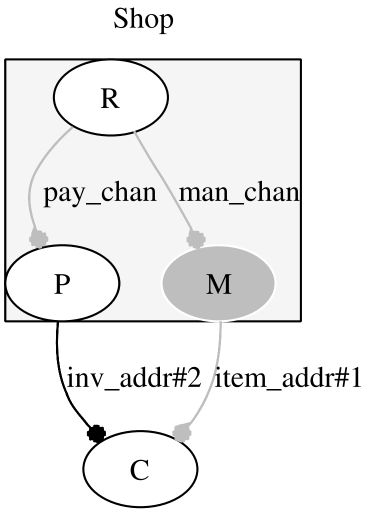


Figure 4.20: Reseller example with shop pool showing its internals

Another extension to the flow graphs, which is especially useful for business processes, are pools. Agents can be assigned to certain pools regarding a diversity of criteria, e.g. similar functionality or belonging to the same sub-domain of the entire business process. For example, a group of customers in a shopping process may be assigned to an own pool and the shops may be assigned to another pool, whereas observing the behavior between these two pools becomes easier when abstracting from the detailed internals. Pools are graphically represented by shaded boxes, grouping together the agents belonging to them. Those agents are displayed within the pool. If a pools internals are not of interest, its internals are hidden by just a shaded rectangle without visible inner life.

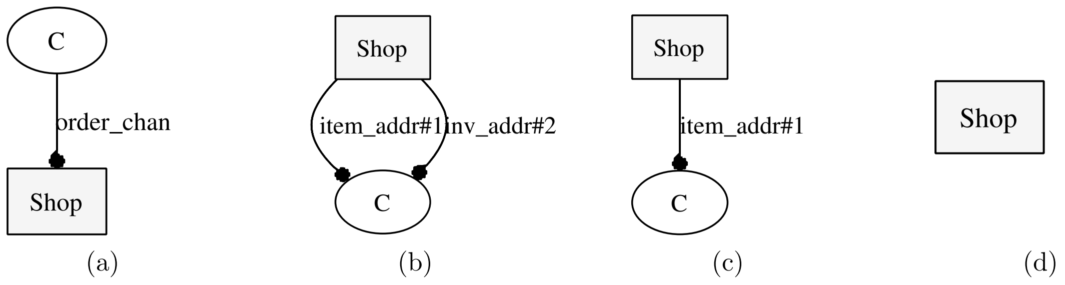


Figure 4.21: Reseller example with shop pool hiding its internals during execution

Figure 4.20 shows the reseller example π-calculus system during execution with a defined pool *Shop* grouping together the reseller, manufacturer and payment organization agents. The case of the *Shop* pool hiding its internals is depicted in Figure 4.21. Furthermore, the evolution of the interaction between the customer and the shop is shown. In Figure 4.21 (a) the customer sends his order to the shop, (b) represents the shop being finished with processing the request, ready to send the invoice and the product to the customer. In (c) the customer has received the invoice and is still able to receive the product. After receiving both, the customer is finished and its agent becomes inactive, leaving the system without further active actions (d).

4.2.2 Graph Creation

After initially parsing the process definitions or after each execution of a communication or τ action, new information about the system's linking structure is provided by the execution engine. This information is presented to the visualizing component in the format of the following node sets:

- Set containing all names of agents currently part of the system
- Set containing all active tau prefix nodes: AT
- Set containing all active input and output prefix nodes, sorted by communication channel: AP
- Set containing all blocked input and output prefix nodes, sorted by communication channel: BP

Since the π-calculus has no concept of pools, as are used in modeling business processes, the pool configuration information needs to be stored and handled by the visualizing component solely. Pools are defined as follows.

Definition 14 (Pool)**.** A pool is a triple $(poolName, A, status)$ with $poolName$ specifying the name of the pool, the set A containing a subset of all π-calculus agent names contained in the system and $status \in \{open, closed\}$ defining if the pool is hiding $(status = closed)$ or showing its internal structure $(status = open)$.

The set A of a pool has to be closed with respect to defined agent resolution meaning that all other agents an agent contained in the set of a pool may resolve to have to be contained in A, too. This restriction ensures that the only communication between pools is realized via message flow and the processing of business process tasks does not lead to a sub process of one pool switching its location to another pool by defined agent resolution. In case one pool contains a task with exactly the same functionality of a task within another pool, this task has to be defined separately as an agent for both pools. However, if the two tasks do not just have same the functionality, but are actually the same task, this can be modeled as the task being contained in a third pool or within one of two pools being used as a service utilizing message flows as means of communication.

The graph representing the current π-calculus system will be created in 4 steps: From the set containing the names of the agents in the system an oval node is drawn for each agent, except the ones that are contained in a pool hiding its internal structure at the moment, and a box shaped node is drawn for each pool specified. Nodes that are drawn for agents contained in a pool that is not closed will be positioned within the box shape representing the pool. All agents within a closed pool will not be displayed within the respective pool, but their communication links will lead to and start from the box shape representing the pool, instead. In the second step, the nodes representing agents that have a tau node contained in the active tau set AT will be shaded. If an agent that is part of a closed pool contains an executable tau action, it will be executed automatically to avoid unintended and invisible blocking. Thirdly, the active communication links are inserted into the graph by drawing edges from each node having a send capability to each node having a receive capability contained in the same subset of the active send/receive node set AT. A capability is assigned to the node representing the agent with the same process name as is stored in the prefix node. In the fourth step, the blocked communication links of the blocked prefixes set BP are drawn in exactly the same manner, but with a lighter color, so they can be distinguished from the active links.

4.2.3 Implementation

A very important feature of the implementation of the graphical representation is user interaction. Instead of listing the available actions and communications, for example in a list box next to the graph, the user should be able to directly interact with the presented graph. Such interaction is, for example, to use the mouse pointer to select a node within the graph that is currently capable of a τ action or to select an edge directly in the graph, which stands for a communication available at the moment. This kind of interaction capability calls for implementing an interactive graph, instead of just displaying an image depicting the current linking structure. As already mentioned, the grappa library provides such interactive means for graph creation, manipulation and selection of elements, e.g. nodes and edges. Observing the visual representation and waiting for user input concerning the graph is the responsibility of a dedicated listener that forwards appropriate signals to the visualizing component, which on its part either processes the signals by itself or forwards them to the controller. When receiving signals, the controller triggers the execution engine. Signals that have to be processed by the visualization component itself are for example configuration signals concerning the opening or closing of pools, since the execution engine has no notion of the pool concept.

Besides the listener, classes that realize the visualization components as are depicted in Figure 4.16 are the *GUIInterface* and the *Visualization* class. The *GUIInterface* keeps the coupling between the controller and the graphical user interface class *PiMainFrame* on a low level and the *Visualization* class contains the implementation for creating the visual output of the π-calculus system's linking structure. Besides interacting with the graphical representation, the *PiMainFrame* class supplies the main window of the application and offers all other user interface actions, e.g. opening a file that contains process definitions. As can be seen in the class diagram, the method calling structure is directed only in one way always using the interfaces and thereby keeping the implementations independent of the actual classes. An additional interface is inserted between the graphical user interface class and the class creating the visual representation of the π-calculus system. This ensures that its implementation may easily be exchanged by another one using a different graph drawing package in place of the grappa library.

As a special problem, the π-calculus visualizer has to keep track of the bonding between the graph elements and the actual π-calculus elements represented by them. It can easily be seen that after the selection of an execution step within the graph the original prefix nodes have to be extracted and reported to the controller that provokes the execution of the selected step. This is done via a table storing the identifiers of the visual elements together with the executable prefix nodes they represent. For an active tau action the identifier of the graphical node and for a communication, the identifier of the edge are stored.

After each execution step the new node sets needed for the visualization are queried from the executor and translated into the visual representation.

This concludes the design and implementation chapter, elucidating the fundamental concepts needed in the π-calculus execution engine and visualization component. A tree structure building the foundation for executing π-calculus systems has been introduced and the execution rules needed to simulate the evolution behavior according to the reduction semantics have been explained. An extension to the flow graphs has been specified. It additionally gives information about active and blocked communications in the current state and structures the agents belonging together, by utilizing the concept of pools from BPMN. Furthermore, the information needed to create a graphical representation of a π-calculus system has been described. In the following chapter those concepts will finally be put to practice in an example.

Chapter 5

Presentation of the Prototype

The elaborated concepts are implemented in the prototype PiVizTool. Implemented functionality comprises all functions stated in the analysis in section 3.1. In this section, the prototype is demonstrated. Firstly some screenshots of its user interface are shown and explained. Then the reseller business process example introduced in the beginning of the thesis in section 2.1.2 is executed.

5.1 Running PiVizTool

As a prerequisite for running the PiVizTool application, an installation of the Graphviz graph visualization software is required. Graphviz provides the graph drawing engine "dot", which is needed for arranging the graphical representation of π-calculus systems within the PiVizTool. Executable installation packages can be downloaded at:

```
http://www.graphviz.org/
```

The PiVizTool application is distributed as a Java Archive (JAR) and can be run using the Java interpreter. The command for starting the PiVizTool application is:

```
java -jar PiVizTool.jar
```

5.2 User Interface

The user interface of the prototype was developed using Java Swing [28]. In contrast to the previous Abstract Windowing Toolkit (AWT) [24] that depends on the underlying platforms windowing system, Swing components are purely written in Java, which allows the application to run on the same platforms as pure Java applications do. On the other hand, compared to AWT, the Java Swing GUI toolkit provides an extensive library of graphical user interface elements and lots of new components, e.g. trees, text-editors, and tables. Another feature is that the look and feel of a Swing application can be changed at runtime. This enables the automatic adaption of the look and feel of the application depending on the current underlying operating system, so the uniform look of different applications on the same operating system is preserved.

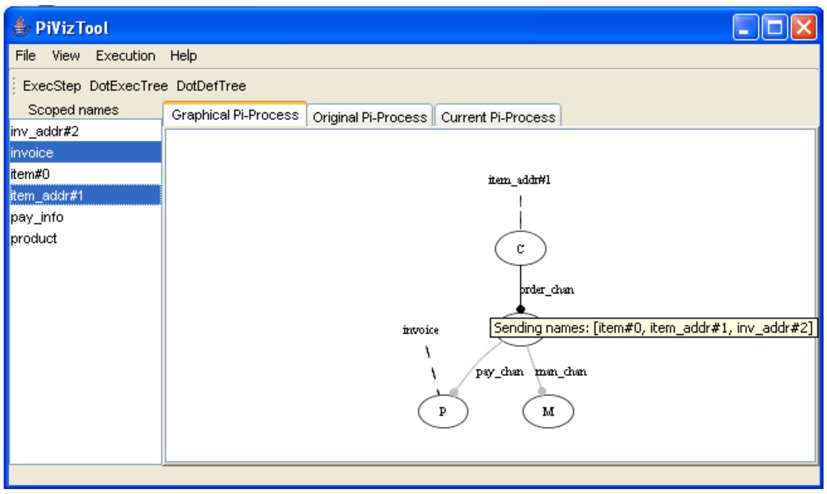
Screenshots of the PiVizTool are presented in Figures 5.1 and 5.2. In Figure 5.1(a) the main view for monitoring the evolution behavior of π-calculus system is shown. As can be seen, a graphical representation, including annotations that contain the names of the agents and communication channels, are depicted on the right hand side. Depending on the position of the mouse pointer over the graphical representation, additional information is shown via tool tips. In this case, the mouse pointer hovered above the edge between the customer and the reseller agent and the information displayed contains the names that are sent over the channel by the customer during this communication.

On the left hand side of the user interface a list of names currently under restriction in the π-calculus system is displayed. In this list several names can be selected. Their scope is going to be exposed in the graphical representation instantly upon selection. Supplementary functionality is provided by the buttons above the list of scoped names. They offer the options of executing a randomly selected step or dumping the internal data structures for analysis reasons. The Figures depicting the definition and execution trees in section 4.1.2 have been created using these dumping functions.

Figure 5.1(b) illustrates one of the process definition views of the PiVizTool. In this view the process definitions of the originally defined π-calculus system can be inspected. The last view, which is not shown in a screenshot shows the process definitions of the π-calculus system in the current state in a similar manner. The different views can be changed by selecting the appropriate tab in the user interface.

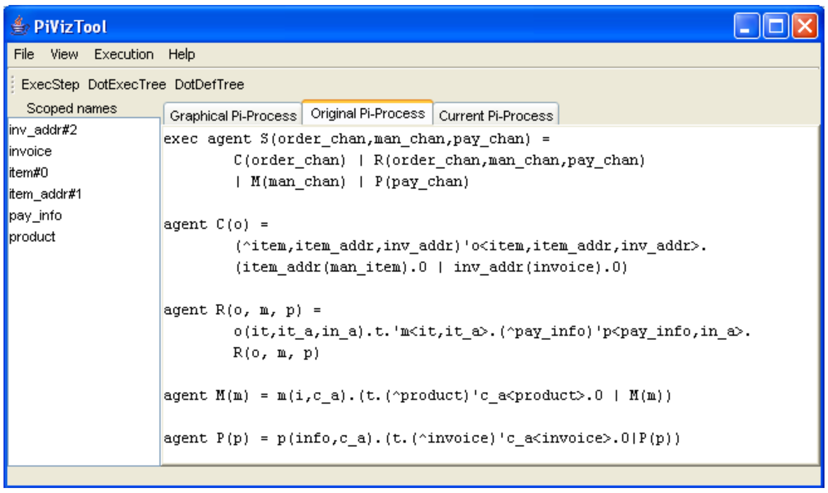
The selection of an executable step by the user is pictured in Figure 5.1(c). If a user selects an execution step with the mouse pointer, this step becomes highlighted and the user is prompted if he wants to execute this step. The option of being prompted after each selection can be turned off in the execution menu. However, regarding more complex systems with lots of communication links, this option comes in handy for selecting the desired communication, since an unintended selection does not lead to execution directly, but can be canceled.

Some more options as shown in Figure 5.2 that are directly selectable in the context menu of the graphical representation are zooming functions, which scale the graph. Such scaling is especially important for complex systems, because it ensures the readability of the graph, since adaption to the window size may let the representation of



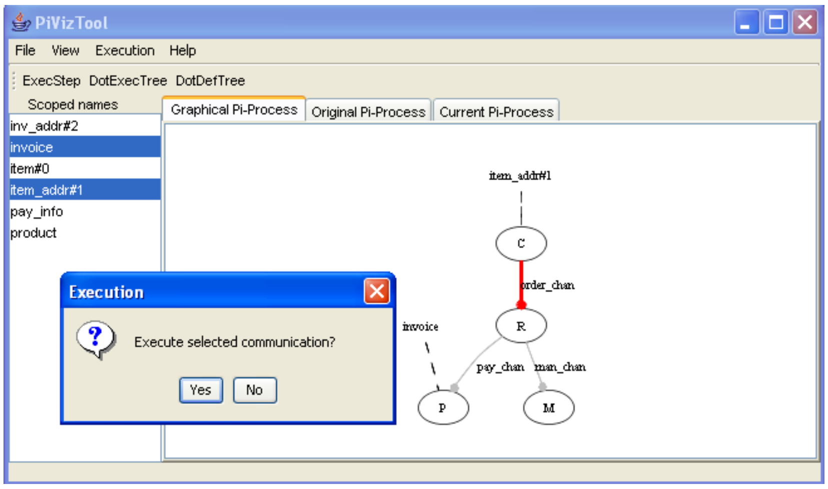
Sending names: [item#0, item_addr#1, inv_addr#2]

(a)



```
exec agent S(order_chan,man_chan,pay_chan) =
        C(order_chan) | R(order_chan,man_chan,pay_chan)
        | M(man_chan) | P(pay_chan)

agent C(o) =
        (^item,item_addr,inv_addr)'o<item,item_addr,inv_addr>.
        (item_addr(man_item).0 | inv_addr(invoice).0)

agent R(o, m, p) =
        o(it,it_a,in_a).t.'m<it,it_a>.(^pay_info)'p<pay_info,in_a>.
        R(o, m, p)

agent M(m) = m(i,c_a).(t.(^product)'c_a<product>.0 | M(m))

agent P(p) = p(info,c_a).(t.(^invoice)'c_a<invoice>.0|P(p))
```

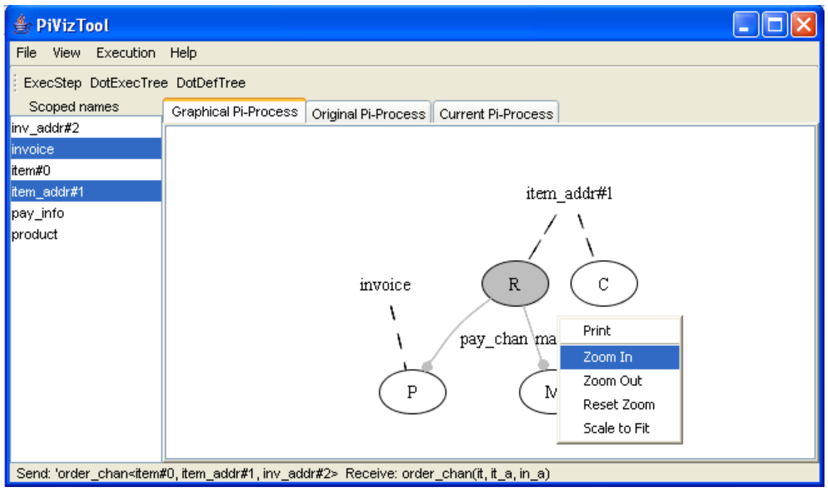
(b)



Execute selected communication?

(c)

Figure 5.1: Screenshots of the PiVizTool



Send: 'order_chan<item#0, item_addr#1, inv_addr#2> Receive: order_chan(it, it_a, in_a)

Figure 5.2: Screenshots of the PiVizTool(cont.)

the graph decrease in size. The view menu of the PiVizTool offers an option for always fitting the graphical representation to the current window size, which is convenient for small to medium sized systems. Besides zooming, an option for printing the current representation is supplied. Further information shown in this Figure is the status bar on the bottom reporting the last executed step. This information sheds light on the substitution of names that has taken place. In the example shown, the customer process has sent his restricted names *item*, *item_addr* and *inv_addr* that replace the reseller's names *it*, *it_a*, *in_a*. Additionally, their scope has been extended as can be seen for the name *item_addr* that now belongs to the reseller as well, but has been private to the customer beforehand.

5.3 Example

The complex π-calculus reseller example, resulting from the conversion of the business process diagram introduced in section 2.1.2, will be used in this section to demonstrate the functionality of the prototypical implementation for the π-calculus simulation environment. In contrast to the simple version of the reseller example used in the previous sections to exemplify the concepts needed for the simulation environment, this advanced process also illustrates the behavior within the specified pools. The complete business process diagram from section 2.1.2, including all pools and the annotations automatically generated by the conversion tool for reference to the created π-calculus agents, is depicted in Figure 5.3.

As already described, the converter creates a π-calculus agent for each of the flow objects. All the flow objects of a pool are concurrently running agents that are syn-

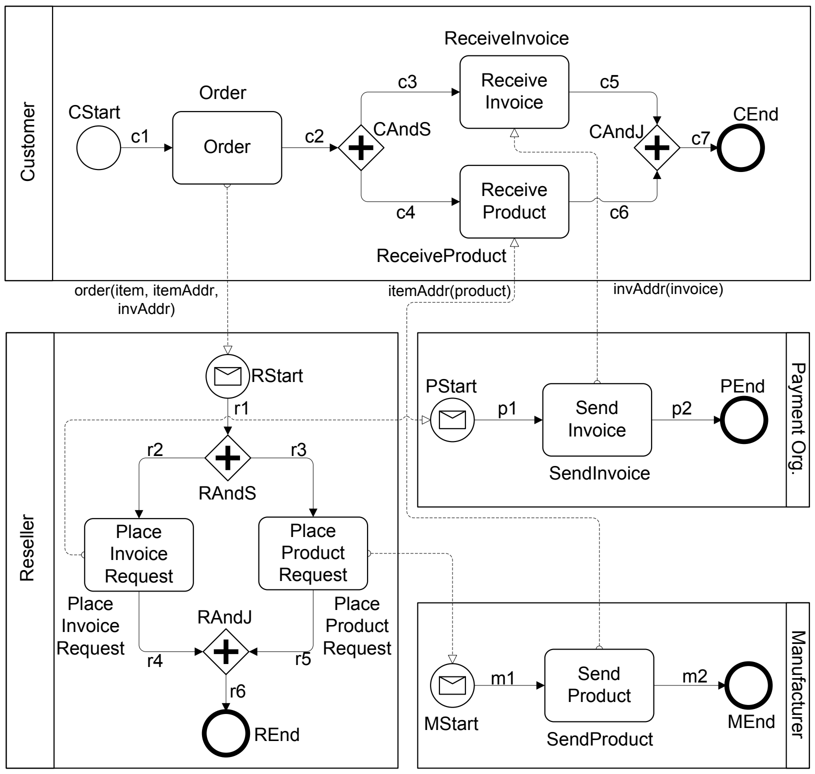


Figure 5.3: Reseller example with conversion annotations

chronized by sending messages on channel names that are restricted to this pool. The pool definitions and process definitions resulting from converting the business process diagram are illustrated below.

```
pool PaymentOrg {PaymentOrg,PStart,SendInvoice,PEnd}
pool Customer{Customer,ReceiveInvoice,ReceiveProduct,CAndJ,
       CStart,Order,CAndS,CEnd}
pool Manufacturer {Manufacturer,SendProduct,MStart,MEnd}
pool Reseller{Reseller,RAndJ,RAndS,REnd,PlaceInvoiceRequest,
       RStart,PlaceProductRequest}

agent ReceiveInvoice(c4,c5) = c4(invAddr).invAddr(invoice).t.
       ('c5.0 | ReceiveInvoice(c4,c5))
agent ReceiveProduct(c3,c6) = c3(itemAddr).itemAddr(product).t.
       ('c6.0 | ReceiveProduct(c3,c6))
agent CAndJ(c6,c5,c7) = c6.c5.t.('c7.0 | CAndJ(c6,c5,c7))
agent CStart(c1) = t.'c1.0
agent Order(c1,c2,order) = (^item,itemAddr,invAddr)c1.t.
       'order<item,itemAddr,invAddr>.
```

```
        ('c2<itemAddr,invAddr>.0 | Order(c1,c2,order))
agent CAndS(c2,c4,c3) = c2(itemAddr,invAddr).t.
        ('c4<invAddr>.0 | 'c3<itemAddr>.0 | CAndS(c2,c4,c3))
agent CEnd(c7) = c7.t.CEnd(c7)
exec agent Customer(order) = (^c1,c2,c3,c4,c5,c6,c7)
        (ReceiveInvoice(c4,c5) | ReceiveProduct(c3,c6) |
        CAndJ(c6,c5,c7) | CStart(c1) | Order(c1,c2,order) |
        CAndS(c2,c4,c3) | CEnd(c7))

agent RStart(r1,o) = o(it,it_a,in_a).t.'r1<it,it_a,in_a>.0
agent RAndS(r1,r3,r2) = r1(it,it_a,in_a).t.
        ('r3<it,it_a>.0 | 'r2<it,in_a>.0 | RAndS(r1,r3,r2))
agent PlaceInvoiceRequest(r2,r4,p) = r2(it,in_a).t.
        'p<it,in_a>.('r4.0 | PlaceInvoiceRequest(r2,r4,p))
agent PlaceProductRequest(r3,r5,m) = r3(it,it_a).t.
        'm<it,it_a>.('r5.0 | PlaceProductRequest(r3,r5,m))
agent RAndJ(r5,r4,r6) = r5.r4.t.('r6.0 | RAndJ(r5,r4,r6))
agent REnd(r6) = r6.t.REnd(r6)
exec agent Reseller(payChan,order,manChan) = (^r6,r1,r2,r3,r4,r5)
        (RAndJ(r5,r4,r6) | RAndS(r1,r3,r2) | REnd(r6) |
        PlaceInvoiceRequest(r2,r4,payChan) | RStart(r1,order) |
        PlaceProductRequest(r3,r5,manChan))

agent PStart(p1,p) = p(info,c_a).t.'p1<c_a>.0
agent SendInvoice(p1,p2) = (^invoice)p1(c_a).t.'c_a<invoice>.
        ('p2.0 | SendInvoice(p1,p2))
agent PEnd(p2) = p2.t.PEnd(p2)
exec agent PaymentOrg(payChan) = (^p1,p2)
        (PStart(p1,payChan) | SendInvoice(p1,p2) | PEnd(p2))

agent MStart(m1,m) = m(i,c_a).t.'m1<c_a>.0
agent SendProduct(m1,m2) = (^product)m1(c_a).t.'c_a<product>.
        ('m2.0 | SendProduct(m1,m2))
agent MEnd(m2) = m2.t.MEnd(m2)
exec agent Manufacturer(manChan) = (^m1,m2)(SendProduct(m1,m2) |
        MEnd(m2) | MStart(m1,manChan))
```

Loading these process definitions into the PiVizTool produces the initial graphical output illustrated in Figure 5.4. The structure of the system directly reflects the structure of the modeled business process diagram, including all start and end events, gateways and tasks, since they are all represented by an own π-calculus agent. During execution the names holding the information about the customer's addresses for receiving the product and the invoice will be monitored for showing the flow of data. They are selected in the list of scoped names and their current scope is shown in the visual representation. The numbered extensions of the names starting with "#" as in the Figures, e.g. "c1#4", are the PiVizTool's realization of unique names, achieved by using a counter.

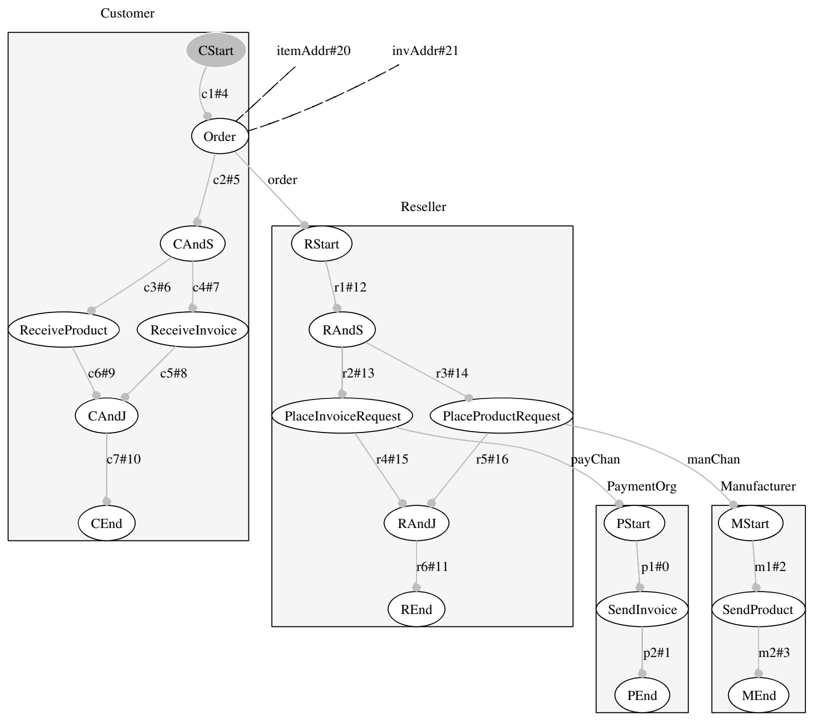


Figure 5.4: Execution of the reseller example

In the current state the names *itemAddr* and *invAddr* are only known to the customer's order agent. The only reduction currently possible is the customer's initial τ action, which means that the active tau set has only this element and the active send/receive node set is momentarily empty. To the contrary, the blocked prefix node set contains lots of entries, which are on the one hand all the sequence flows for synchronizing the agents and on the other hand the three message flows between the pools.

Upon executing the active τ-action, the sequence flow between the customer's start agent and order agent becomes active, see Figure 5.5.

After executing this communication, the order task of the customer can be processed, which means that the customer uses the public channel *order* of the reseller to send his order and further information needed for delivering the product and the invoice. Succeeding the sending of the order, the address information *itemAddr* and *invAddr* is known by the reseller, too. This circumstance is displayed by the scopes of the names that represent this information pointing to agents within the reseller pool in addition to pointing to agents within the customer pool (Figure 5.6(a)).

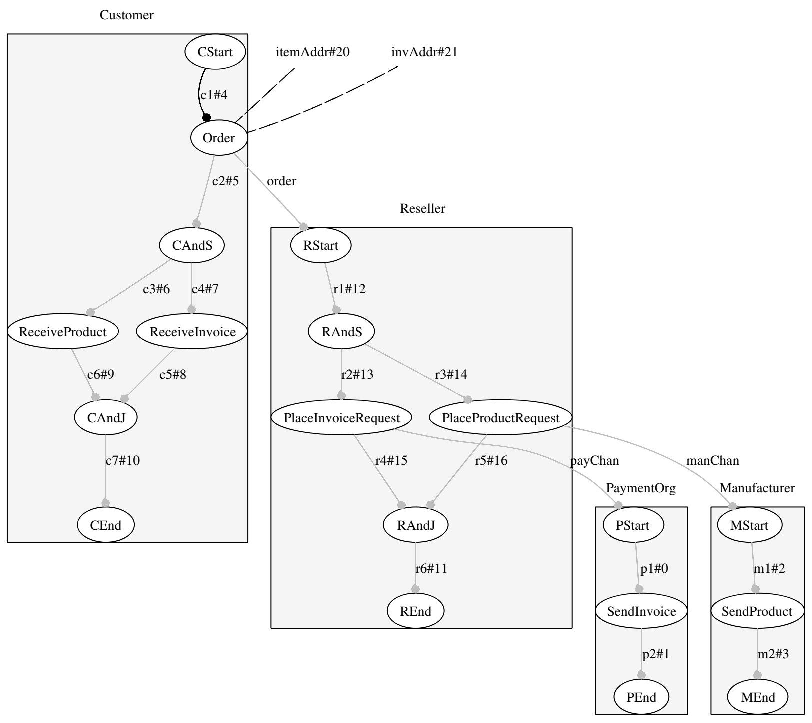
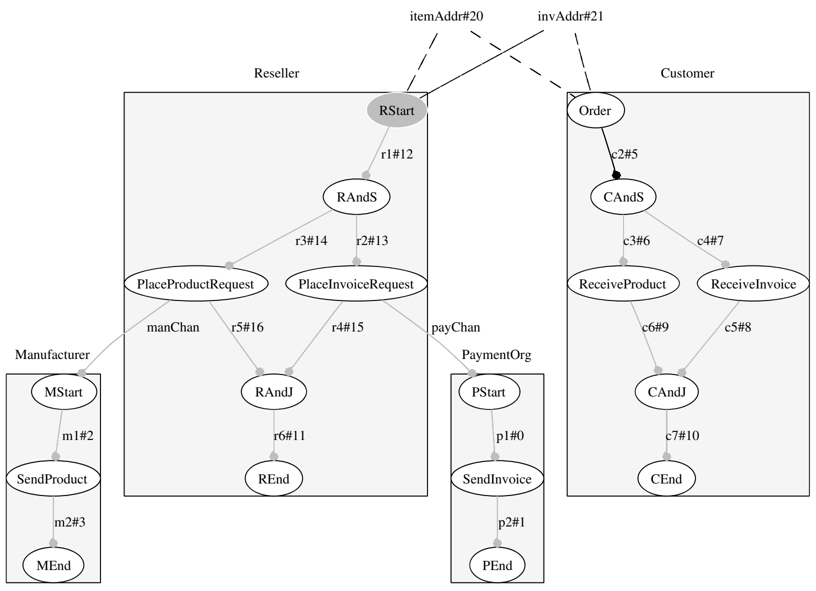


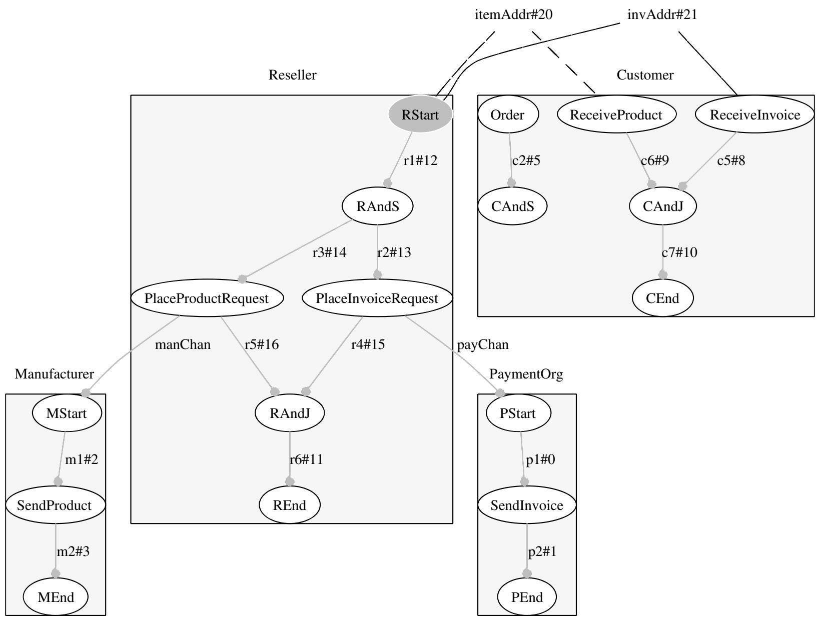
Figure 5.5: Execution of the reseller example (cont. 1)

Since the address information is needed further down by the customer's agents that are supposed to receive the product and the invoice, the information has to be forwarded within this pool to the appropriate agents. Figure 5.6(b) shows the state of the system with the customer executed as far as possible and the names being forwarded. In this state the customer process is blocked, because of waiting for the product and an invoice to arrive. As can be seen, the agents representing the customer's order task and parallel split gateway have reset themselves. This aspect is not important in the reseller example, because it contains no loops. Moreover, these agents have lost their access to the restricted names for the customer's address information. This can be inferred from their π-calculus process definitions, since they do not take along the information in their agents' parameter list during resetting.

In Figure 5.7(a) the reseller is ready to trigger the manufacturer and the payment organization. The state after the reseller process has placed the product and invoice request and moreover has finished execution, which is possible since no more dependencies to other processes exist, is depicted in Figure 5.7(b). Besides, the reseller process has lost its access to the customer's address information. The reseller process will not have any more active capabilities in the future as the agent representing its start event does not support resetting and the entire process depends on the outbound sequence flow of the start event. Start events in general are not allowed to be part of loops, because these flow objects must not have any incoming sequence flows. A similar constraint

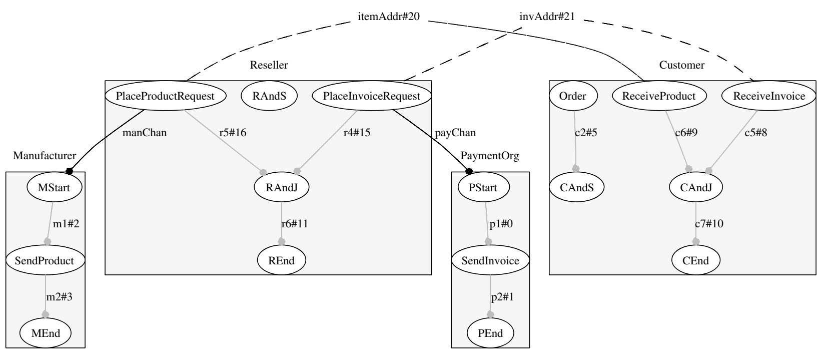


(a)

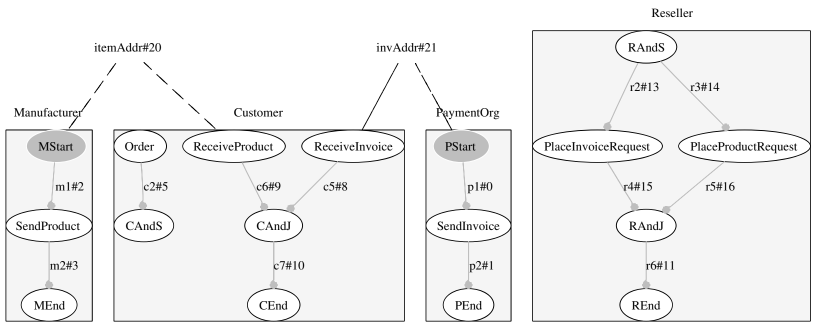


(b)

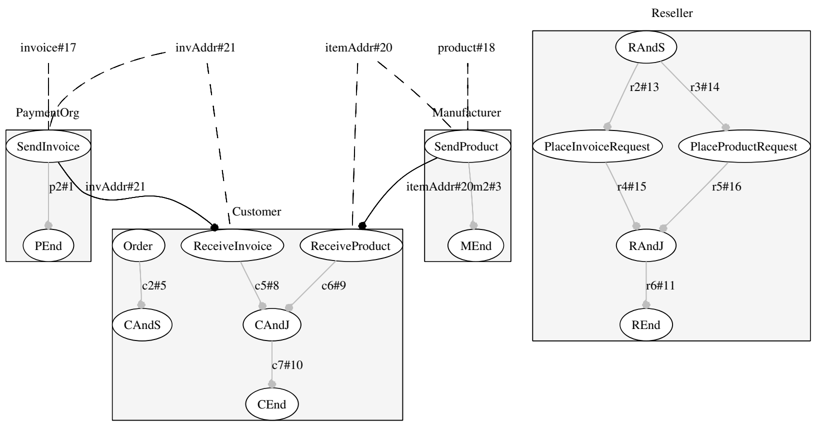
Figure 5.6: Execution of the reseller example (cont. 2)



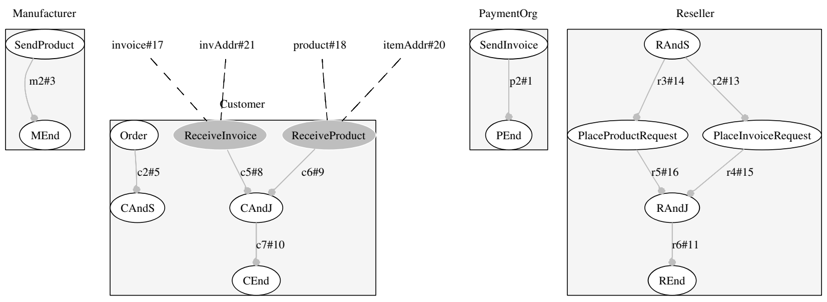
(a)



(b)



(c)



(d)

Figure 5.7: Execution of the reseller example (cont. 3)

applies to end events, which must not have any outgoing sequence flows and therefore can not be part of loops, either. However, end events support resetting, since this might be needed in bisimulation checking to observe if the end events of two systems are triggered a different number of times.

After placement of the product and invoice request by the reseller, the manufacturer and payment organization gain access to the customer's information including the ordered item and the address data. This state of the linking structure is depicted in the same Figure by the illustrated scope of the names *itemAddr* and *invAddr*.

According to the information about the ordered item, the manufacturer produces the product and the payment organization creates the invoice. The two names representing the data objects of the invoice and product are shown in Figure 5.7(c). Their current scope is confined to the manufacturer and payment organization, respectively. Succeedingly, the manufacturer and the payment organization can use the received address information of the customer for direct communication with the according agents of the customer process via message flow. New links are established by using the names representing the address information as communication channels to send the product and the invoice. As the result of sending the product and the invoice to the customer and ending their process execution, the manufacturer and the payment organization loose their access to these names and the customer gains the access, see Figure 5.7(d).

The example just demonstrated is a variation of the simple reseller example used throughout the thesis to demonstrate the concepts. In contrast to the example used in this chapter that includes a pool for each process containing numerous more agents, the simple reseller example only consists of four agents defining the four interacting processes. The advantage of the more complex example is that the internal behavior of the pools can be monitored, too. By hiding the internals of each pool and thereby only showing the interaction behavior between the pools a similar behavior as in the simple example can be observed.

Chapter 6

Conclusion & Future Work

This thesis presents the work on a simulation environment for business processes based on the π-calculus. Starting from basics on business processes and the π-calculus introducing the terminology, the underlying concepts for a π-calculus simulation engine and a graphical representation of π-calculus systems have been developed.

The π-calculus provides means for modeling processes with dynamic structures as is essential for service oriented architectures. In contrast to traditional approaches for modeling and representing business processes, the π-calculus allows for modeling actors capable of accepting and using links for communication that are established during run time. Thereby, new actors that are able to take part in the interactions of already existing actors in the system can be inserted during run time. In traditional approaches the actors and the linkages between the actors are fixedly defined in the design phase.

To attain the goal of allowing inspection of the evolution behavior of systems with changing structures, the conceptual design and a prototypical implementation of a simulation environment for π-calculus systems are the results of this work. The two most significant issues of this work are thoughts concerning the π-calculus execution engine and a sufficient interactive graphical representation of π-calculus systems. For the execution engine a tree data structure holding all the information needed to create the graphical representations and presenting the basis for the execution of π-calculus systems, has been proposed. Based on the evolution behavior specified by the reduction semantics, execution rules for the tree structure have been defined in this thesis.

For the interactive graphical representation of π-calculus systems, the flow graph notation has been extended with the concept of pools for presenting the interaction behavior of business processes as part of choreographies. In addition, graphical subtleties for helping the user in distinguishing his options for interactively exerting influence on the further evolution of the presented system have been introduced.

The prototype "PiVizTool", developed as part of this thesis, implements the entire range of functionality as specified in the requirements, section 3.1. Different directions of equipping the simulation environment with further functionality or using the acquired concepts are imaginable.

On the one hand, the simulator could be expanded with enhanced simulation functionality. As an example, the configuration of abstractions of agent groups at multiple levels could be implemented. Abstractions at multiple levels means that agents within a pool can be selected and a new sub-group can be defined for them within the pool, which again can be treated as a black box itself. This functionality might be handy providing further structuring capabilities. For instance, more clarity can be reached when using replicated agents with lots of different instances being displayed in different states, but only one of them being the matter of interest. Closing down the entire pool would keep the user from examining the instance of interest and keeping it open can be confusing, because of the many replicated instances. Therefore, adding all instances, except the one the user is interested in, to a new sub-group and hiding the new group's internals seems to be an alternative. For the availability of such a functionality, the graphical representation has to be extended with further interactive configuration options. These have to enable the selection of multiple agent nodes and creating a new group for them.

On the other hand, the simulation environment could be extended with reasoning features. As a first step providing the possibility of executing multiple systems in parallel and displaying the structures to the user on a multi-document interface is conceivable. This presents the user with the option of comparing the behavior of the systems and doing some kind of manual bisimulation. By selecting one step for execution in one of the systems and trying to reach the same state by executing one or more steps in the other system depending on the kind of actions being observable or unobservable and the kind of bisimulation check being performed, the user can compare the systems. Further on, automatic bisimulation, as is provided for example by the Mobility Workbench tool, with additional step by step visual representation for observation may be possible.

A third alternative could be to develop a workflow engine based on the π-calculus. This workflow engine may utilize the elaborated concepts of the simulation environment being able to control the execution of workflows, e.g. assigning responsibilities according to roles and triggering the automatic execution of work units.

More research can be done on the topic of memory management for running π-calculus systems. From the presentation of the prototype in section 5.3 can be seen that the agents of the reseller process will never become active again after finishing their first execution cycle, which ends after they are reset. This is the effect of all of the agents being dependent on their previous agents' outgoing messages with the utilized channels being restricted to the reseller pool. All of the depending agents are blocked forever, since the reseller's start event agent is not reset and the reseller's other agents have no defined way of sharing their restricted names through scope extrusion. In this context the detection of such "dead" agents and their elimination from the running system, comparable to the garbage collection of the Java programming language, may be an interesting subject for further studies.

Bibliography

[1] Information technology - Syntactic matalanguage - Extended BNF, International Standard, Reference Number: ISO/IEC 14977:1996(E). International Organization for Standardization and International Electrotechnical Commission, 1996.

[2] Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. https://javacc.dev.java.net/, June 2006.

[3] JUNG - Java Universal Network/Graph Framework. http://jung.sourceforge.net/index.html, March 2006.

[4] Piccolo Toolkit - A Structured 2D Graphics Framework, Human Computer Interaction Lab, University of Maryland. http://www.cs.umd.edu/hcil/piccolo/index.shtml, June 2006.

[5] Prefuse, Information Visualization Toolkit. http://prefuse.org/, June 2006.

[6] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In *Proceedings of BPM'2003*, pages 1–12, Heidelberg, Germany, 2003. Springer LNCS 2678.

[7] W. M. P. van der Aalst. Making Work Flow: On the Application of Petri Nets to Business Process Management. In J. Esparza and C. Lakos, editors, *23rd International Conference on Applications and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, Berlin, June 2002.

[8] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, Cambridge, MA, USA, 2002.

[9] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[10] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th Inter-*

national Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.

[11] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. Technical Report FIT-TR-2002-06, Queensland University of Technology, Brisbane, Australia, June 2002.

[12] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Ivana Trickovic, Sanjiva Weerawarana, and Satish Thatte (Editor). Business Process Execution Language for Web Services (BPEL4WS), Version 1.1. OASIS, May 2003.

[13] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, Reading, MA, USA, 1998.

[14] Sebastien Briais. The ABC's User Guide. Available at: http://lamp.epfl.ch/ sbriais/abc/abc.html, 2003.

[15] Roger Burlton. *Business Process Management: Profiting From Process.* Sams, 1st edition, May 2001.

[16] BPMI Business Process Management Initiative. Business Process Modeling Notation (BPMN). http://www.bpmn.org/, May 2004.

[17] James F. Chang. *Business Process Management Systems: Strategy and Implementation.* Auerbach Publications, 2006.

[18] Ann DiCaterino, Kai Larsen, Mei-Huei Tang, and Wen-Li Wang. An Introduction to Workflow Management Systems, Models for Action Project: Developing Practical Approaches to Electronic Records, Management and Preservation. Technical report, Center for Technology in Government University at Albany / SUNY, 1997.

[19] Layna Fisher, editor. *New Tools for New Times: The Workflow Paradigm, The Impact of Information Technology on Business Process Reengineering.* Future Strategies Inc, 2nd edition, June 1995.

[20] Ulrik Frendrup and Jesper Nyholm Jensen. Checking for Open Bisimilarity in the π-Calculus. http://www.cs.auc.dk/research/FS/ny/PR-pi/, February 2001.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[22] Emden Gansner, Eleftherios Koutsofios, and Stephen C. North. *Drawing graphs with dot.* Murray Hill, NJ, http://www.graphviz.org/Documentation.php, February 2002.

[23] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[24] David M. Geary. *Graphic Java 1.2, Volume 1: AWT.* Prentice Hall PTR, 1st edition, September 1998.

[25] Business Process Management Group, Steve Towers (Contributor), and Peter Fingar (Contributor). *In Search Of BPM Excellence: Straight From The Thought Leaders.* Meghan Kiffer Pr, April 2005.

[26] Object Management Group. Unified Modelling Language: Superstructure, Version 2.0. http://www.omg.org/technology/documents/formal/uml.htm, July 2005.

[27] Hugo Haas and Allen Brown, editors. *W3C Web Services Glossary.* W3C Working Group, http://www.w3.org/TR/ws-gloss/, 2004.

[28] Marc Hoy, Dave Wood, Marc Loy, James Elliot, and Robert Eckstein. *Java Swing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[29] Rashid N. Khan. *Business Process Management: A Practical Guide.* Meghan-Kiffer Pres, September 2004.

[30] Andreas Knöpfel, Bernhard Gröne, and Peter Tabeling. *Fundamental Modeling Concepts: Effective Communication of IT Systems.* Wiley, March 2006.

[31] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA : Service-Oriented Architecture Best Practices (The Coad Series).* Prentice Hall PTR, November 2004.

[32] Frank Leymann and Wolfgang Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[33] Robin Milner. Flowgraphs and Flow Algebras. *Journal of the ACM*, 26(4):794–818, October 1979.

[34] Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[35] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, New York, NY, USA, 1999.

[36] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I. Technical Report -86, 1989.

[37] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part II. Technical Report -86, 1989.

[38] John Mocenigo. Grappa - A Java Graph Package, AT&T Labs Research. http://www.research.att.com/ john/Grappa/, April 2006.

[39] Hanspeter Mössenböck, Albrecht Wöß, and Markus Löberbauer. The Compiler Generator Coco/R, University of Linz. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/, June 2006.

[40] Stephen C. North. *Drawing graphs with neato.* http://www.graphviz.org/Documentation.php, April 2002.

[41] Hagen Overdick, Frank Puhlmann, and Mathias Weske. Towards a Formal Model for Agile Service Discovery and Integration. In Kunal Verma, Amith Sheth, Michal Zaremba, and Christoph Bussler, editors, *Proceedings of the International Workshop in Dynamic Web Processes (DWP 2005)*, volume International Business Machines, 2005 IBM RC 23822, pages 25–36, 2005.

[42] Terence Parr. ANTLR, ANother Tool for Language Recognition. http://www.antlr.org/, June 2006.

[43] J. Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra.* Elsevier, 2001.

[44] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, 1962.

[45] Andrew Phillips and Luca Cardelli. A Graphical Representation for Biological Processes in the Stochastic π-calculus. In *Bioconcur'05*, August 2005.

[46] Frank Puhlmann. A Tool Chain for Lazy Soundness. In Jan Mendling, editor, *Demo Session of the 4th International Conference on Business Process Management (BPM 06)*, pages 9–16, Vienna, Austria, September 2006.

[47] Frank Puhlmann. Why Do We Actually Need the Pi-Calculus for Business Process Management? In W. Abramowicz and H. Mayr, editors, *9th International Con-*

ference on Business Information Systems (BIS 2006), volume P-85 of LNI, pages 77–89, Klagenfurt, Austria, 2006. Gesellschaft für Informatik.

[48] Frank Puhlmann and Mathias Weske. Using the π-Calculus for Formalizing Workflow Patterns. In *3rd International Conference on Business Process Management (BPM05)*, pages 153–168, Nancy, France, September 2005.

[49] Frank Puhlmann and Mathias Weske. Investigations on Soundness Regarding Lazy Activities. In S. Dustdar, J.L. Fiadeiro, and A. Sheth, editors, *Proceedings of the 4th International Conference on Business Process Management (BPM 2006), volume 4102 of LNCS, Vienna, Austria.* Springer Verlag (2006) 145-160, 2006.

[50] John Pyke, John O'Connel, and Roger Whitehead. *Mastering Your Organization's Processes: A Plain Guide to BPM.* Cambridge University Press, 2006.

[51] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[52] Wesley C. Salmon. *Scientific Explanation and the Causal Structure of the World.* Princeton University Press, November 1984.

[53] Davide Sangiorgi. A Theory of Bisimulation for the π-Calculus. In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142, Hildesheim, Germany, August 1993. Springer-Verlag.

[54] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[55] Howard Smith and Peter Fingar. *Business Process Management (BPM): The Third Wave* . Meghan-Kiffer Press, 1st edition, January 2003.

[56] Howard Smith and Peter Fingar. A BPT Column (01-04): The Third Wave. http://www.bptrends.com, January 2004.

[57] Björn Victor and Faron Moller. The Mobility Workbench — A Tool for the π-Calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[58] Workflow Management Coalition, http://www.wfmc.org/. *The Workflow Reference Model*, 1995.

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Verwendung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Potsdam, den 19. November 2006

Anja Bog