Interaction protocols in paradigm : extensions to a modeling language through tool development

Stam, A.W.

Interaction Protocols in PARADIGM

Extensions to a Modeling Language through Tool Development

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof.mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 8 december 2009
klokke 10.00 uur

door

Andries Wouter Stam

geboren te Vlaardingen
in 1975

Promotiecommissie

Promotor:	Prof.Dr. J.N. Kok
Co-promotores:	Dr. E.P. de Vink *Eindhoven University of Technology*
	Dr. M.M. Bonsangue
Overige leden:	Prof.Dr. R. Heckel *University of Leicester, UK*
	Prof.Dr. B.R. Katzy
	Dr. L.P.J. Groenewegen

ISBN 978–90–9024918–6

Contents

Chapter 1

Introduction

The central concept of this thesis is *interaction* – mutually influencing behavior of two or more entities. It is present in every aspect of our daily life. Human beings communicate, work together, move around, create and exchange things, operate machines. With each action they take, they enable and constrain the behavior of numerous people and things around them, often without realizing. *We interact, therefore we are.*

... and we are not alone. A similar lively community of interacting entities lives inside computers, in the virtual world of software. These entities have many different names, depending on the context in which we speak about them, or the way in which we like to think about them: active objects, actors, agents, processes, services. They communicate in many direct and indirect ways, with data units, signals, messages, via mail boxes, channels, shared data spaces, pipes. They enter, they leave, they send and receive, they store and retrieve. This community, virtual but ubiquitous, is alive and still growing, and will surely continue to grow considerably in the next decades.

In the design, analysis and implementation of software systems, interaction plays a role of increasing importance. There are at least two reasons for this. Firstly, the software construction process continuously changes towards more openness, distribution and heterogeneity. Nowadays, systems are no longer built as closed monolithic entities, but rather as open distributed systems. They are not created by a single team of developers, but rather composed out of a large pool of software services, built by multiple independent software developers, running concurrently on different physical machines somewhere on the globe, changing their functionality over time while they are in use. Clearly, in order to let these heterogeneous distributed evolving software services work together as a system, the study of their interaction is of crucial importance. As a second important trend, hardware becomes an increasingly smaller cost factor in the implementation of information and communication technology. So-called *multicore* processors are the norm nowadays. Software developers can and should exploit this trend by designing their software with full concurrency as a major design principle. This would inherently lead to more concurrent entities, and an urging need for methods and techniques to design, analyze and implement their interaction.

This thesis is about interaction in software systems. We have investigated the applicability of a *general* modeling language for interaction to the *specific* field of software development. The modeling language, called PARADIGM, has been developed earlier at the Leiden Institute of Advanced Computer Science (LIACS) and has been shaped via its application to several systems and modeling domains throughout the last decades [44, 95, 71, 32, 33, 41, 34, 45, 42, 43, 46, 92, 40]. Its interesting features are the particular *world view* it adopts, the fact that it is an *executable language* (models written in the language can be executed, e.g. by a computer), and its ability to model systems whose behavior *evolves* at runtime (i.e. while they are executed). With PARADIGM, we have performed three complementary activities in our research: the development of *tools* for the language, the definition of *conceptual extensions* to the language based on insights gained by the tool development, and a series of *case studies* in order to validate the extensions and the tools, and to assess the applicability of the resulting language for software modeling and software design.

Throughout our research, we have found interesting results in two directions. Firstly, the development of tools for PARADIGM, in particular the creation of a distributed interpreter for PARADIGM models, has cast a new light on the modeling language. Secondly, the case studies carried out in this thesis show the usefulness of PARADIGM and its world view in the context of software modeling and design. Hence, in this thesis, the knife cuts both ways: through designing and implementing software tools for PARADIGM, we improve upon the concepts of the modeling language, and through applying the concepts of the modeling language, we improve upon software design and implementation. In other words, this thesis is the outcome of an interactive process between the development of a modeling language and the development of tools for it.

1.1 Outline

An overview of the thesis is shown in Figure 1.1. The interaction between conceptualization (language development) and tool development is reflected in the structuring of the *Foundations* part into two tracks: Chapters 2 and 4 cover the PARADIGM language and its extensions, respectively, while Chapters 3 and 5 report on the development of a distributed interpreter framework for PARADIGM and its embedding in a set of tools. The *Case Studies* part contains three chapters, each reporting on a particular concrete modeling effort. In Chapter 6 to Chapter 8, validation gradually moves from a focus on the language extensions (chapter 4) to a focus on the tool features (chapter 5). Below, we briefly outline the contents and contributions of each chapter.

Part I: Foundations

We start with a technical introduction to PARADIGM in Chapter 2. We introduce the modeling principles and the concepts of the language, and address its executability in terms of an informal description of the operational semantics of the language. Additionally, a discussion about the general strengths and weaknesses of PARADIGM is provided.

In Chapter 3, we move to the tool development track and introduce PARADISE, a framework for creating distributed interpreters for PARADIGM models. Key objectives of the framework are maximal concurrency and a simple means to experiment with the PARADIGM language. The framework consists of several interpreters for the individual concepts of PARADIGM, which can be composed to form a distributed interpreter for a specific PARADIGM model.

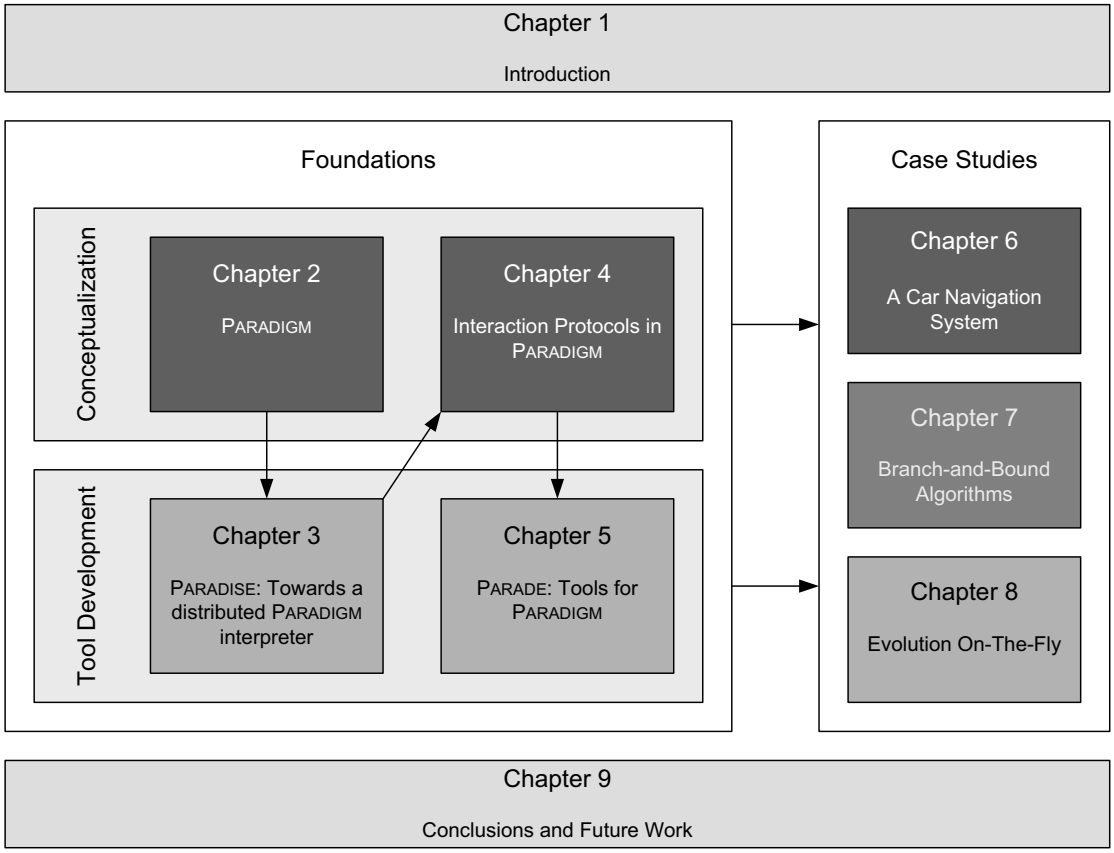


Figure 1.1: Overview of the Thesis Chapters

Based on insights gained in Chapter 3, we reflect on the PARADIGM language and introduce a series of extensions for it in Chapter 4. The major extension discussed in this chapter is the introduction of the *interaction protocol* concept, which can be used to explicitly structure the interaction in a PARADIGM model. With the introduction of the interaction protocol concept, we also contribute to the operational semantics of the language by precisely defining when interactions may take place concurrently. Another extension consists of a generalization of the *consistency rule* concept in PARADIGM, the concept which forms the basis for the specification of interactions in the language.

Finally, we combine the results of Chapters 3 and 4 in the development of the PARADE tools, the subject of Chapter 5. These tools can be used to specify, execute and visualize models specified in the extended PARADIGM language. The core of the tool set is formed by a *distributed runtime environment* for the execution of PARADIGM models based on PARADISE, which directly supports the evolution of PARADIGM models *on-the-fly*. Additionally, the distributed runtime environment offers two extension models to PARADIGM for visualization of the runtime state of component models during execution, and for extending a PARADIGM model with software functionality. Especially the latter extension plays an important role in the case studies of Chapters 7 and 8.

Part II: Case Studies

The case study presented in Chapter 6 focuses on *interaction modeling* in a non-trivial context, for purposes of *software analysis*. We create a PARADIGM model of a multi-component car navigation system, in which the components offer several modes of operation. The challenge in this case study is formed by the fact that the coordination of these modes of operation interferes with the regular communication between the components – two aspects of interaction which interact with each other. The case study is meant primarily to validate the extensions introduced in Chapter 4.

In Chapter 7, we model a generic branch-and-bound algorithm in a sequential, componentized, and parallel shape. In this case study, we do not only validate the concepts of Chapter 4, but also the PARADE tools introduced in Chapter 5: we combine the PARADIGM models with the implementation of a branch-and-bound solver, which enables a more elaborate analysis of the relationship between the generic PARADIGM model and the characteristics of a specific branch-and-bound problem. In doing so, we also show how the principles underlying PARADIGM, i.e. its world view, is useful for *software design*.

Finally, in Chapter 8, we use the extended PARADIGM language and the PARADE tools for modeling *evolution on-the-fly*: the application of changes to a PARADIGM model while it is being executed. This case study validates the extensions of Chapter 4 in the context of modeling *software evolution*. Furthermore, we apply the PARADE tools to *execute* models which evolve on-the-fly. A final contribution presented in this case study is the introduction of a new technique called *scaffolding*, a general technique to temporarily observe and restrict a specific part of the behavior of a running model while changes are applied to it.

1.2 Publications

The contents of this thesis are based on earlier publications, as follows.

In [92], we reported on experience with an implementation of the PARADIGM language in the TOOLBUS architecture [58, 61, 10]. Although not as elaborate as in this thesis (Chapter 4), a primary concern in [92] was to establish a way to symmetrize the relationship between manager components and employee components, which has been realized in this thesis through generalization of the consistency rule concept. The formal notions we developed in [43] have been used as the starting-point for setting up the informal explanation of the operational semantics of PARADIGM in this thesis (Chapter 2).

Many insights of the ArchiMate project [66, 59, 60, 24, 25, 93, 23] have been reused at various places in this thesis. The development of a new language for enterprise architecture, which we published in [60], in particular the work on roles and collaborations in this language, has played an important role in the decision to extend PARADIGM with the interaction protocol concept (Chapter 4). The separation between semantic and symbolic models for enterprise architecture, about which we published in [24] and [25], played a role in the design of the PARADISE framework (Chapter 3) and the PARADE distributed runtime environment (Chapter 5): the distinction between types and instances in the language, the separation of modeling artifacts from semantic interpretations of these artifacts by means of concept interpreters, and the distinction between the symbolic transition labels and their semantics (interpretation) in terms of operation invocations on action classes.

The runtime visualization tools for PARADIGM presented in Chapter 5 have been based on insights from work which we published in [93]. In this publication, we showed how to effectively use XML [77] and RML (the Rule Markup Language, [55]) for modeling, visualizing and analyzing enterprise architectures. Although we did not use XML transformations in PARADE, the rigid distinction between XML specifications of PARADIGM models and their visualizations in terms of *extension models* is based on the principles presented in [93]. Finally, insights from our work on impact-of-change analysis, which is part of a book chapter in [66] and which has also been published in [23], have been implicitly reused in the algorithms adopted in PARADE for the creation, update and deletion of PARADIGM modeling artifacts during evolution on-the-fly.

The case study in Chapter 7 is based on earlier work published in [91], in which we reported on the development of generic framework for coordinating parallel branch-and-bound algorithms with the exogenous coordination language MANIFOLD and the IWIM coordination model [3, 4, 11]. We used the principle of separating control flow and data flow, as present in MANIFOLD, in [90] to extend CMT, a modeling technique for component based software design.

1.3 Available Tools

The PARADE tools have been developed as part of the research presented in this thesis. The tool set consists of a distributed runtime environment for the execution of PARADIGM models, a runtime viewer and a (non-graphical) editor for all types of PARADE models. All tools are available for download at [78]. Installation instructions, concise documentation, examples and the PARADE models from the case studies are included.

Part I

Foundations

Chapter 2

PARADIGM

We introduce PARADIGM, a modeling language specifically meant for modeling the interaction between components in a system. Illustrated by the example of a client/server model, we introduce the modeling principles and the concepts of the language. Furthermore, we explain how models written in PARADIGM can be executed, thereby providing an informal description of the operational semantics of the language. Finally, we discuss the general strengths and weaknesses of PARADIGM.

2.1 Introduction

PARADIGM is an *interaction modeling language*: it is especially useful for the modeling of interaction and coordination problems in human and computer settings. The name PARADIGM is an acronym: PARallelism, its Analysis, Design and Implementation by a General Method. PARADIGM started as a language to model parallel phenomena by the use of Markov decision processes extended with several constructs for the modeling of parallelism and communication [44, 95]. Since then, the context of application changed from parallel phenomena to object-oriented modeling (mainly as part of SOCCA [33, 41]), and coordination problems in general (e.g., [42]). However, the basic ideas of PARADIGM persisted without considerable modification. In 2002, operational semantics for a restricted class of PARADIGM models were published in [45], followed in 2005 by an article in which the operational semantics were generalized for a broader class of models [43]. Recently, interesting results have been established pertaining to the evolution of PARADIGM models during their execution [46, 40].

PARADIGM has been applied to many different systems in many different domains, ranging from operating systems [95] to business processes [43]. In all cases, its purpose is to get insight into the *interaction* between components in a system. To this end, both the behavior of individual components as well as the interaction between components are modeled. Models specified in the PARADIGM language are executable – they provide an operational model of a system which can be executed by a computer.

PARADIGM is based on two modeling principles: the *multiple views* principle and the *manager/employee* principle. According to the *multiple views* principle, a distinction is made between the *detailed behavior* of a component, and views on this detailed behavior relevant to each of the roles it plays in interaction with other components. These views are called *global behaviors*. The actual interaction in the system is modeled by applying the *manager/employee* principle: the detailed behavior of a *manager* component manages one or more global behaviors of a set of *employee* components. PARADIGM allows components to be a manager and an employee at the same time. The multiple views principle enables the possibility to study the interaction between the components separately from their detailed behavior. The manager/employee principle ensures that interaction is modeled according to a standard pattern (one manager, multiple employees) and enables interaction modeling at multiple management levels.

We have structured this chapter as follows. In Section 2.2, we explain the core concepts and structuring principles of PARADIGM. After that, we focus on the operational interpretation of PARADIGM models in Section 2.3. Throughout sections 2.2 and 2.3, we illustrate our explanation with a small PARADIGM model of a system consisting of three clients being served by a server. In Section 2.4, we shortly discuss the general strengths and weaknesses of the PARADIGM language. Finally, in Section 2.5, we conclude.

2.2 Language Concepts

The PARADIGM language consists of three basic concepts: *process*, *partition* and *consistency rule*. The *process* concept is used to model behavior. For each component in the system, we model one *detailed process*, and zero or more *global processes* which represent different views on the detailed process in interaction with other components. Global processes are based upon a partitioning of the detailed process into phases, modeled using the *partition* concept, which embodies the multiple views principle. The relation between components is modeled using the *consistency rule* concept. A consistency rule synchronizes one step of a single detailed process with one step of zero or more different global processes. According to the manager/employee principle, the component containing the detailed process is called the *manager*, while the components containing the global processes are called *employees*. In the next subsections, we zoom in onto each of the three concepts by providing a detailed informal description and an example to illustrate how a concept is used.

Processes

A process in PARADIGM is modeled as a state transition diagram, a simplified version of a UML state machine diagram [12, 37, 73, 74]. It consists of states, transitions between states, and transition labels (also called actions). A state represents a certain situation the process is in. One or more *starting* states and zero or more *final* states can be defined. A state can be both a starting and a final state, and final states may have outgoing transitions. An example of a PARADIGM process can be found in Figure 2.1. This process models the detailed behavior of a (human) client, being served by a (human) server. Within the model, one starting state is defined, indicated by a black dot and an incoming transition. No final state is specified. Intuitively, we interpret this model as follows. Starting in state *NoNeeds*, the client is supposed to enter a building or room via transition *enter*. He/she then arrives at a desk and reaches state *AtDesk*. The client is then expected to explain his or her needs, after which state *NeedClear* is entered. After the client asks for the service, he/she is supposed to be under service in state *Service*. At a certain moment, the client performs action *thank* and enters state *Satisfied*, indicating that the service is to be stopped. Finally, the client leaves the building, whereafter the same behavior is repeated.

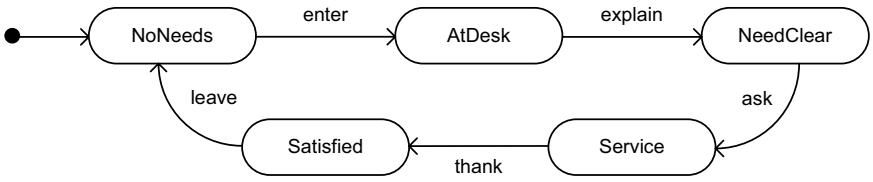


Figure 2.1: A PARADIGM process *Client*

Processes are used in PARADIGM to model the detailed behavior of a component as well as the role-specific views on this detailed behavior, called global behaviors. Before we show an example of a process modeling global behavior, we first introduce the *partition* concept, on which the modeling of global behavior is based.

Partitions

Partitions are a means to define views on the detailed process of a component. They are not the views themselves, but merely embody the relation between a detailed process and a view on it. It is possible to define many partitions on top of a single detailed process, which reflects the idea that a process can be viewed from different viewpoints, each being relevant for a specific role in the interaction with other components. A partition of a process consists of *subprocesses* of that process, each with one or more *traps*. Subprocesses can be seen as overlapping phases which the process goes through during execution, while traps are milestones achieved in a certain phase which are relevant for the purpose of interaction and which possibly enable the entrance of a following phase. A subprocess is a subset of the states and transitions of a process. Subprocesses are defined without starting or final states. A subprocess acts as a dynamic constraint on the behavior of the process: only the states and transitions within the *current subprocess* are allowed (the process is said to be in this subprocess). A trap is a subset of the states of a subprocess, such that there is no transition in the subprocess from one of the states within the trap to a state outside of the trap. For this reason, it is called a trap – once entered, it cannot be left as long as its subprocess is the current subprocess. A trap is called *trivial* if it contains the entire set of states of a subprocess. A trap is an enabler for a change from one subprocess to another subprocess. Once the underlying process, being constrained by a certain subprocess, enters the trap, this trap acts as a *connecting trap* to other subprocesses, provided that these subprocesses each contain all states included in the trap. A change from one subprocess S to a subprocess S' via a connecting trap θ is called a *subprocess change*.

An example of a partition is shown in Figure 2.2. This partition, called *ClientAsObjectOfService* (or *AsOOS* for short), consists of three subprocesses of process *Client*: *WithoutService*, *Orienting* and *UnderService*. In each subprocess, one trap has been defined, indicated by a polygon and a name in italics. The traps of the subprocesses are connecting traps: Trap *asking* is a connecting trap from *WithoutService* to *Orienting*, but also a connecting trap from *WithoutService* to *UnderService*. Trap *serverClear* is a connecting trap from *Orienting* to both *WithoutService* and *UnderService*, while trap *ready* is a connecting trap from *UnderService* to *WithoutService*. Intuitively, partition *ClientAsObjectOfService* splits process *Client* into three phases: *WithoutService*, where the client does not have the attention of the server, *Orienting*, where the client is clarifying his/her service needs, and *UnderService*, where the client is being served.

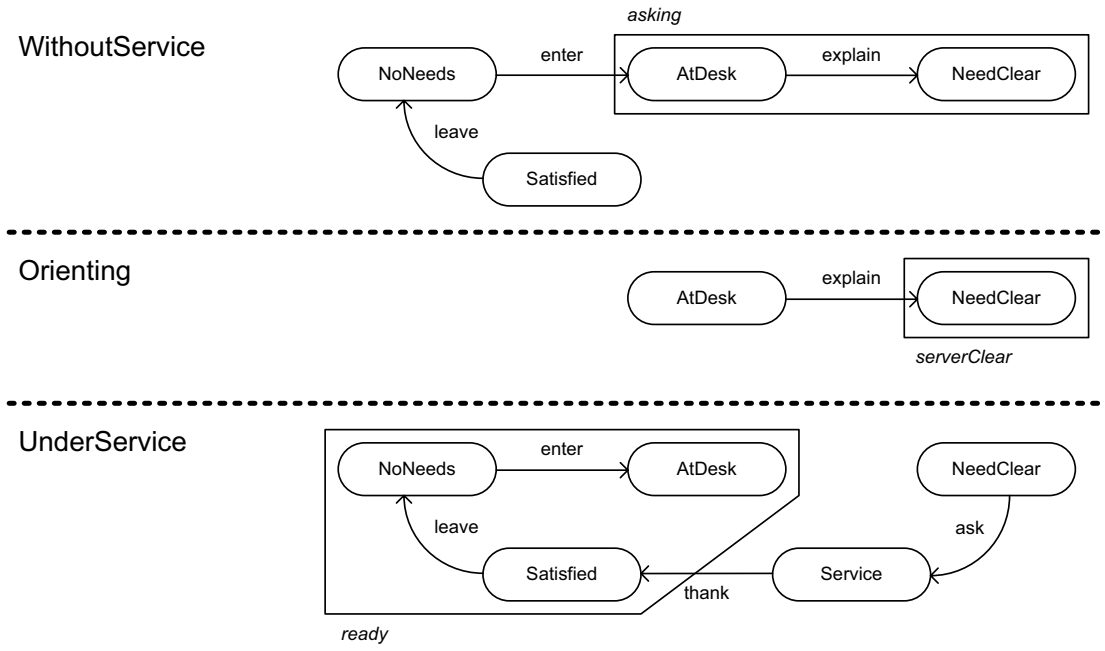


Figure 2.2: A PARADIGM partition *ClientAsObjectOfService* (*AsOOS* for short)

Global Processes

As described above, a partition of a process acts as a set of dynamic constraints on that process. The current subprocess of a partition restricts the process to a subset of states and transitions. A change from one subprocess to another is only possible whenever a connecting trap between these subprocesses has been entered. Based on a partition, one can construct a process in which each state is mapped one-to-one on each subprocess of the partition, the state name corresponding to the name of the mapped subprocess. Additionally, transitions between the states of this process can be defined whenever there is a connecting trap between the corresponding subprocesses. The transition labels then correspond to the names of the connecting traps. In PARADIGM such a process is called a *global process at the level of that partition*. A global process is a process, just like a detailed process: it has one or more starting states and zero or more final states. It represents a *view* on a detailed process, thereby modeling the (global) behavior relevant to a specific role that the component is able to play. PARADIGM does not require that global processes contain transitions for each connecting trap in the partition.

As an example, a global process at the level of partition *ClientAsObjectOfService* of process *Client* is shown in Figure 2.3. We call this process *Client[AsOOS]*, denoting the detailed process *Client* viewed "as an object of service", or playing the role "object of service", at the level of partition *AsOOS*. A state exists for each subprocess in the partition. Note, there is no corresponding transition for the connecting trap *asking* from *WithoutService* to *UnderService*, nor for the connecting trap *serverClear* from *Orienting* to *WithoutService*.

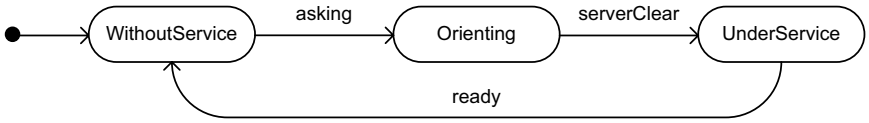


Figure 2.3: PARADIGM process *Client[AsOOS]*

A global process establishes a dynamic constraint on the detailed process that belongs to it by the fact that its current state corresponds to a current subprocess of a partition. Many different partitions and corresponding global processes can be defined on top of a single detailed process. The detailed process is constrained by the current subprocesses of all its partitions together, that is, its behavior is limited to the intersection of those current subprocesses. At the same time, a transition in a global process can only be taken if the trap corresponding to the transition label has been entered within the current subprocess. Hence, one could perceive the relation between the detailed process and the global processes of a component as a mutual dynamic constraint which maintains the components' inner consistency, i.e. the consistency between its detailed behavior and the global behaviors relevant to its roles in the interaction.

With detailed processes, partitions of these processes and global processes at the level of these partitions, we are able to model the behavior of the individual components of a system in PARADIGM. Each component consists of one detailed process together with zero or more partitions and corresponding global processes, being the views on the detailed process relevant for interaction. So far, we have not introduced a concept with which we are able to model the *interaction between different components*. This is where the *consistency rules* of PARADIGM come into play.

Consistency Rules

A *consistency rule* relates the transition of a single detailed process, called a *manager process*, to zero or more transitions of global processes at the level of partitions of *employee processes*. In general, a consistency rule is written as:

$$P : s \xrightarrow{a} s' \quad * \quad P_k(\pi_{k,\ell}) : S_{k,\ell} \xrightarrow{\theta_{k,\ell}} S'_{k,\ell}, ..., P_v(\pi_{v,w}) : S_{v,w} \xrightarrow{\theta_{v,w}} S'_{v,w}$$

where P is a detailed process and $P_k(\pi_{k,\ell}), ..., P_v(\pi_{v,w})$ are global processes. $P_k(\pi_{k,\ell})$ denotes a global process at the level of a particular partition $\pi_{k,\ell}$ of a particular process P_k. We call the part on the left side of $*$ the *manager part* of the consistency rule and the part on the right side its *employee part*.

Consistency rules synchronize transitions of detailed manager processes with those of global processes. If, upon execution of the model, a transition of a detailed process is taken, a consistency rule with this transition as its manager part must be *applied*. The result of such application is that all transitions mentioned in the consistency rule are taken simultaneously. This naturally requires that all these transitions can be taken, i.e. that the constraints with regard to partitions inside the components are satisfied. We explain the conditions for the application of consistency rules more explicitly in Section 2.3.

In PARADIGM, transitions of both detailed and global processes can only be taken as the consequence of applying a consistency rule. Therefore, PARADIGM requires that, for each transition in each detailed process in the model, at least one consistency rule is defined which has this transition as its manager part. This ensures that in principle each transition of each *detailed* process can be taken. Note, however, that this requirement is not imposed on transitions of *global* processes: if a transition of a global process does not occur in the employee part of any consistency rule, it can simply never be taken.

PARADIGM allows detailed processes to be both manager and employee, even within a single consistency rule: in terms of the general consistency rule mentioned above, this means that manager process P is equal to one of the employee processes P_k, ..., P_v. In case zero global processes are mentioned in a consistency rule, i.e. the employee part is empty, it is written as:

$$P : s \xrightarrow{a} s'$$

This means that the detailed transition $s \xrightarrow{a} s'$ is not bound to any transitions of global processes. A detailed process for which only consistency rules with empty employee parts are defined, is called a *pure employee process*.

Consider the following example. A server is serving three clients, as introduced earlier. Process *Server* is shown in Figure 2.4. The entire PARADIGM model consists of three processes *Client(i)*, each with a partition *AsOOS* and global process *Client(i)[AsOOS]*, and one process *Server* without any partitions. The processes are related to each other by means of the example consistency rules of Table 2.5.

Intuitively, the first three rules R1,2,3 relate transitions *address(i)* of manager process *Server* to transition *asking* of global processes *Client(i)[AsOOS]*. That is, it is possible to take transition *address(i)* of process *Server* if this process currently is in state *Checking*, process *Client(i)[AsOOS]* currently is in state *WithoutService* and process *Client(i)* is in one of the states of trap *asking*. If process *Server* takes transition *address(i)*, it enters state *ListeningTo(i)*, and transition *asking* of global process *Client(i)[AsOOS]* therefore must be taken as well. Now process *Client(i)* is being restricted to subprocess *Orienting*.

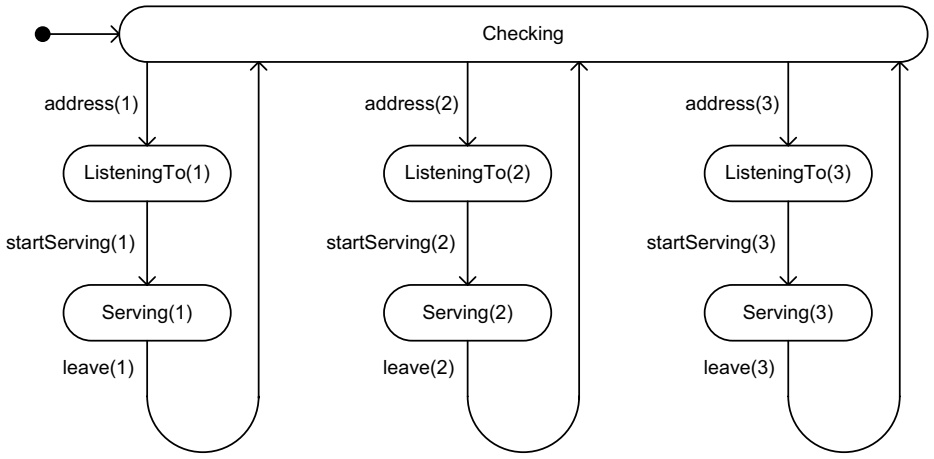


Figure 2.4: PARADIGM process *Server*

(R1,2,3)		Server:	Checking	—address(i)→	ListeningTo(i)
	*	Client(i)[AsOOS]:	WithoutService	—asking→	Orienting
(R4,5,6)		Server:	ListeningTo(i)	—startServing(i)→	Serving(i)
	*	Client(i)[AsOOS]:	Orienting	—serverClear→	UnderService
(R7,8,9)		Server:	Serving(i)	—leave(i)→	Checking
	*	Client(i)[AsOOS]:	UnderService	—ready→	WithoutService
(R10,11,12)		Client(i):	NoNeeds	—enter→	AtDesk
(R13,14,15)		Client(i):	AtDesk	—explain→	NeedClear
(R16,17,18)		Client(i):	NeedClear	—ask→	Service
(R19,20,21)		Client(i):	Service	—thank→	Satisfied
(R22,23,24)		Client(i):	Satisfied	—leave→	NoNeeds

Table 2.5: Consistency rules relating the *Server* and *Client* processes

In rules R4,5,6 and R7,8,9, the same principle is followed for transitions *startServing(i)* and *leave(i)* of process *Server*, related to subprocess changes for processes *Client(i)* from *Orienting* to *UnderService* and from *UnderService* to *WithoutService*, respectively. The remaining consistency rules R10 to R24 are defined for the transitions of the pure employee processes *Client(i)*.

PARADIGM Models

A PARADIGM model consists of a set of detailed processes, partitions of those processes, global processes at the level of each partition, and a set of consistency rules, at least one for each transition in each detailed process. The structure of an entire PARADIGM model can be visualized by means of a *component diagram*, of which we show an example for the client/server model in Figure 2.6.

For the component diagram of Figure 2.6, we adopt the notation of UML 2.0 component diagrams and composite structure diagrams [73, 74]. Processes are shown as active objects, indicating the fact that they have an individual thread-of-control. The processes run concurrently, as far as they are not constrained by their relationships with other processes. Each detailed process, together with its global processes, is put in a separate component. A detailed and a global process are related to each other by a ≪partition≫ connector between ports, which is given the name of the partition that relates them. The consistency rules are shown as a collaboration with roles for each process mentioned in the rules. These roles are played by components via ports. The connectors from these ports to the collaboration have been decorated with ≪manager≫ and ≪employee≫ stereotypes for clarity. A delegation connector

is used between a port of a component and a port of an (internal) process of this component. The consistency rules for each of the transitions of the detailed *Client* pure employee processes are not considered part of the collaboration: these consistency rules only have a manager part and hence do not define any interaction.

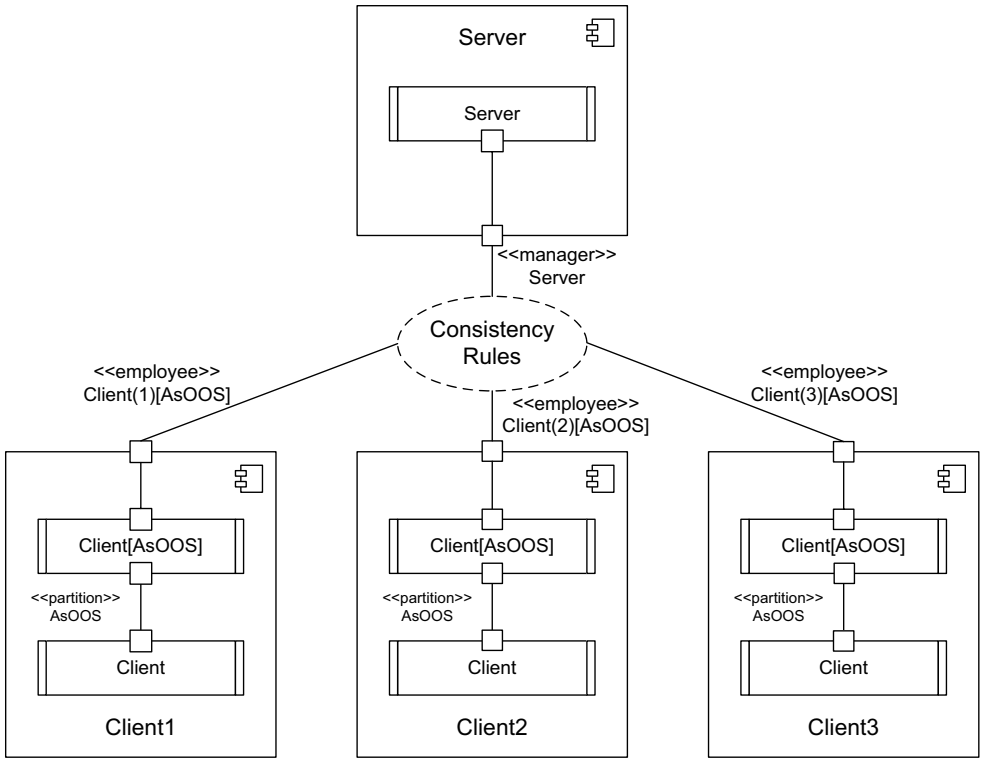


Figure 2.6: A component diagram of the PARADIGM client/server example

2.3 Model Execution

A PARADIGM model defines the behavior of a system in terms of the combined behavior of its interacting components. In this section, we specify precisely *which* behavior is defined by a PARADIGM model, i.e. how we interpret such a model operationally. We split this topic into the following parts. We start with showing how PARADIGM processes can be executed by sequentially taking transitions. After that, we address the execution of detailed and global processes of an isolated component in a PARADIGM model, where the global processes and detailed processes are related via *partitions*. Thirdly, we zoom in onto the role of *consistency rules* in the execution of processes, by showing the detailed process of a manager component together with the consistency rules which relate it to the global processes of employee components. Finally, we address the execution of a PARADIGM model in its entirety.

Processes

Execution of a process starts with selecting one of its starting states as the *current state*. After that, the following procedure is applied. If the current state is a final state, it is possible (but not obligatory) to stop the execution of the process. If execution is continued, however, one of the transitions of the process is selected non-deterministically; this must be a transition which has the current state as its source state. The selected transition is taken, which means that the current state of the process becomes the target state of the transition. This small procedure is applied iteratively.

Take for example the server process from Figure 2.4. Since there is only one starting state *Checking*, it will be selected as the first current state. Now, there are three transitions that can be selected: *address(1)*, *address(2)* and *address(3)*. Suppose for example that transition *address(1)* is selected, then taking this transition results in the current state being *ListeningTo(1)*. Since no final state has been defined, execution of this process never stops.

Partitions

In a PARADIGM component, a detailed process is related to zero or more global processes by means of partitions. This relation influences the execution of the processes. Consider as an example the visualization of Figure 2.7. Here, four consecutive states of global process *Client(i)[AsOOS]* are shown. In each state the corresponding subprocess for *Client(i)* is depicted. Each state in process *Client(i)[AsOOS]* corresponds to a subprocess in partition *ClientAsObjectOfService*, by which process *Client(i)* is restricted. The "lightning"-like arrows indicate the global transitions, starting at connecting traps (the "enablers" of the transition), and ending in the next state of the global process. The figure clearly shows the degree of freedom with regard to taking transitions of the processes. For example, consider trap *ready* for subprocess *UnderService*. Regardless whether the current state of detailed process *Client(i)* is *Satisfied*, *NoNeeds* or *AtDesk*, it is possible to take the transition in global process *Client(i)[AsOOS]* from state *UnderService* to state *WithoutService*.

In this example, we consider only a single partition on top of a detailed process. For multiple partitions, the mutual influence between the detailed process and each of the global processes is similar. Because the detailed process is constrained by each of the current subprocesses at the same time, its behavior is restricted to the intersection of all current subprocesses. Therefore, given a detailed process P with partitions π_j and corresponding global processes $P(\pi_j)$ ($j \in J$), in general, the following holds (for now, we do not take into account consistency rules):

- A transition $s \xrightarrow{a} s'$ of detailed process P can only be taken if, for all $j \in J$, subprocess S_j corresponding to the current state of $P(\pi_j)$ contains transition $s \xrightarrow{a} s'$.
- A transition $S_j \xrightarrow{\theta} S_j'$ of global process $P(\pi_j)$ at the level of partition π_j can only be taken if the current state of detailed process P lies in trap θ of subprocess S_j, i.e. the current state of $P(\pi_j)$.

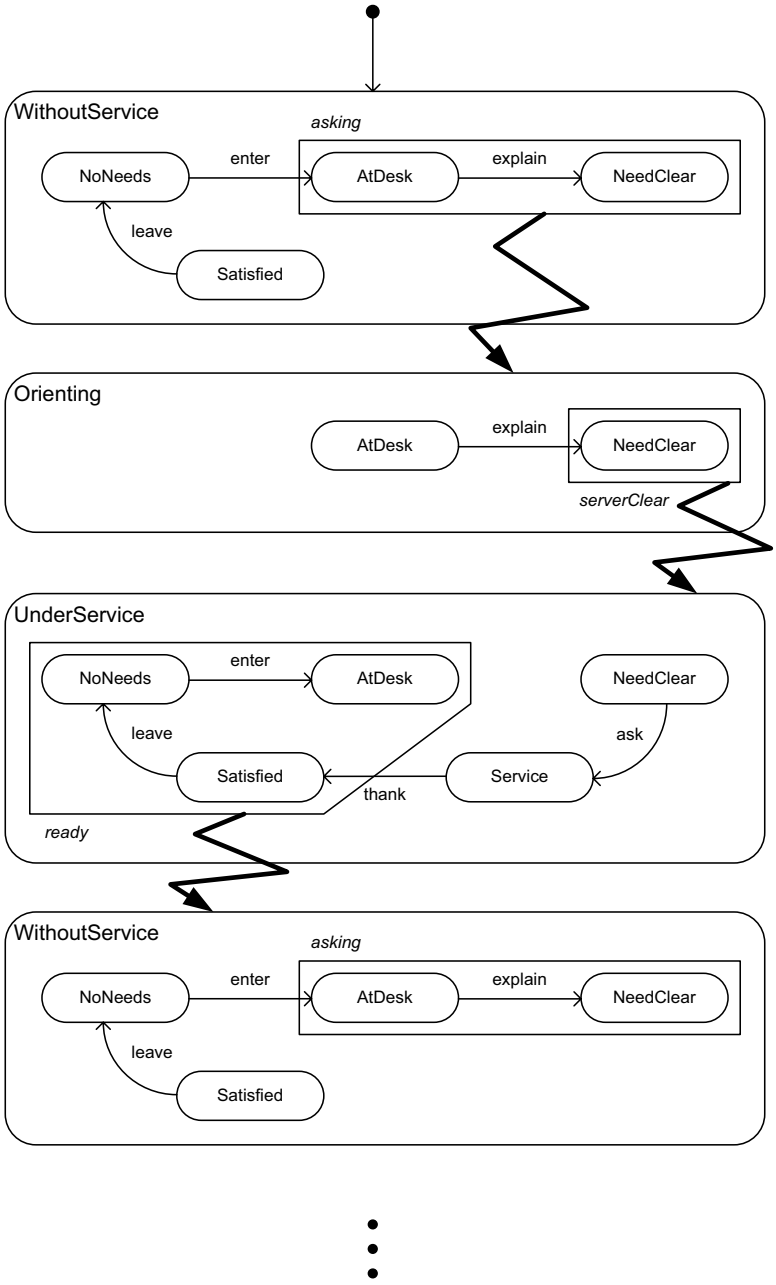


Figure 2.7: A *Client(i)* and *Client(i)[AsOOS]* process in combination

For a meaningful execution of a detailed process and its global processes, the PARADIGM model should incorporate some initial consistency: the chosen starting state of the detailed process should lie in each of the subprocesses corresponding to the chosen starting states of the global processes.

We illustrate the mutual constraints of detailed and global processes of a component in Table 2.8, where the states for processes *Client(i)* and *Client(i)[AsOOS]* from Figure 2.7 have been shown. State names have been abbreviated. From top to bottom, the states of process *Client(i)* are shown, while from left to right, the states of corresponding global process *Client(i)[AsOOS]* are depicted. Assume that process *Client(i)* starts in state *NoNeeds* (NN) and process *Client(i)[AsOOS]* starts in state *WithoutService* (WS). Seen this way, each partition in a PARADIGM model defines a restriction on two processes in that model, one process being the detailed process of a component, the other being a global process of the same component.

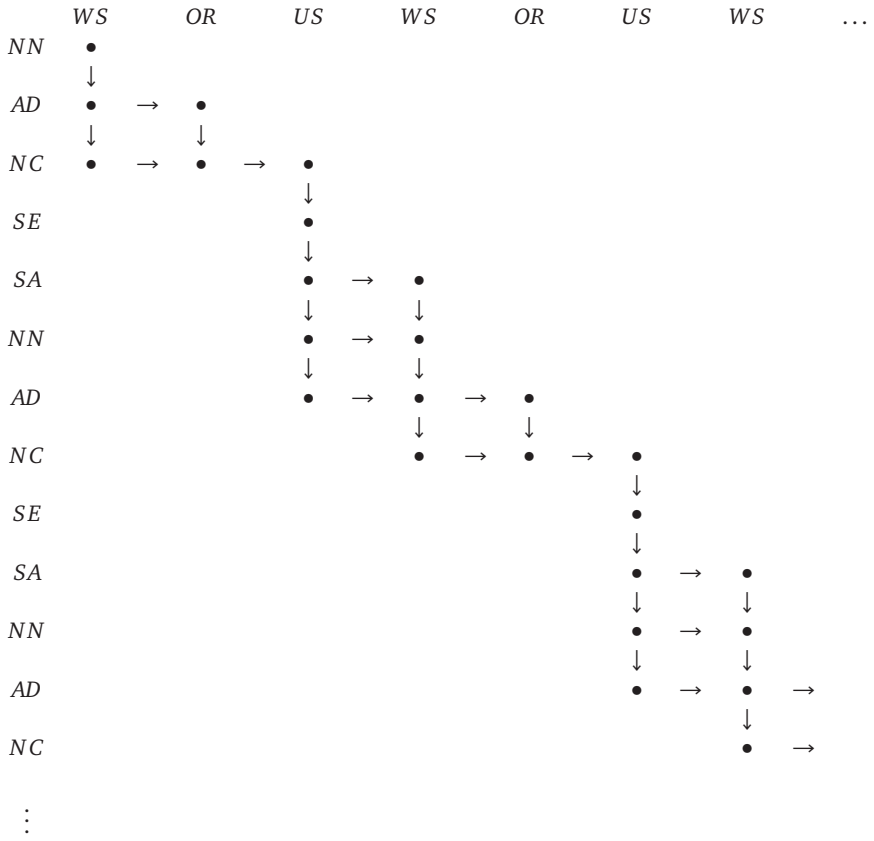
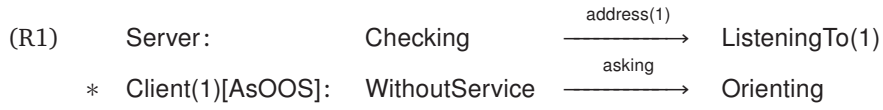


Table 2.8: Mutual constraints of *Client(i)* (↓) and *Client(i)[AsOOS]* (→) in combination

Consistency Rules

Consistency rules bind a single transition of a detailed process (the manager part) to zero or more transitions of global processes (the employee part). This is illustrated in Figure 2.9 for the client/server example. Part of a behavior of the *Server* process is shown in relation to the three processes *Client(i)[AsOOS]*. We also visualize the subprocesses corresponding to the states of processes *Client(i)[AsOOS]*, like shown before in Figure 2.7. At the right side of the figure, the rules from Table 2.5 are mentioned that play a role in the *Server*'s transitions. All processes start in their starting state. In total, seven processes are running: *Server*, *Client(i)* and *Client(i)[AsOOS]* ($1 \leq i \leq 3$).

In the previous section, we already noted that transitions in PARADIGM can only be taken as the consequence of the application of consistency rules. In the starting situation of our example, only consistency rules $R10$, $R11$ or $R12$ (which all have an empty employee part) can be applied, that is, only the three processes *Client(i)* can take a transition. The execution of the other processes is constrained either by the current subprocess of a partition, or by the fact that none of their consistency rules can be applied. Suppose that, in this situation, a transition is taken in process *Client(1)* from its first state *NoNeeds* to state *AtDesk* by applying consistency rule $R10$. As we have seen in the previous example of Figure 2.7, since *Client(1)* now enters trap *asking* of subprocess *WithoutService*, it is possible for global process *Client(1)[AsOOS]* to take transition *asking* from state *WithoutService* to state *Orienting*. Thereby, consistency rule $R1$ comes into play:



This consistency rule states the following: If *Server* is in state *Checking* and transition *address(1)* can be taken, and if *Client(1)[AsOOS]* is in state *WithoutService* and transition *asking* can be taken, then if the consistency rule is *applied*, both transition *address(1)* in process *Server* and transition *asking* in process *Client(1)[AsOOS]* are taken *simultaneously*. In this example, rule $R1$ can now be applied, since both transition *address(1)* of process *Server* and transition *asking* of process *Client(1)[AsOOS]* can be taken. The consequence of applying rule $R1$ is a simultaneous transition in both process *Server* and process *Client(1)[AsOOS]*, as shown in the Figure. The example continues in a similar way.

The application of consistency rules is further illustrated in Table 2.10 for the combinations of the first four states of Figure 2.9 for process *Server* and process *Client(1)[AsOOS]*. As is shown by this figure, the consistency rules enforce *synchronization* of the taking of transitions by these two processes. The reader may also notice the resulting twofold mutual restrictive influence between employee process *Client(1)* and manager process *Server* via global process *Client(1)[AsOOS]*. In one direction, the taking of transition *address(1)* in manager process *Server* depends, because of consistency rule $R1$, on the possibility of taking transition *asking* of global process *Client(1)[AsOOS]*, as this depends on the state of employee process *Client(1)* which must be inside trap *asking* of subprocess *WithoutService*. In the other direction, employee process *Client(1)* is restricted by subprocess *WithoutService* determined by the state of global process *Client(1)[AsOOS]*, which in turn can only be left by applying rule $R1$, which depends on the taking of transition *address(1)* within process *Server*, the manager of this rule.

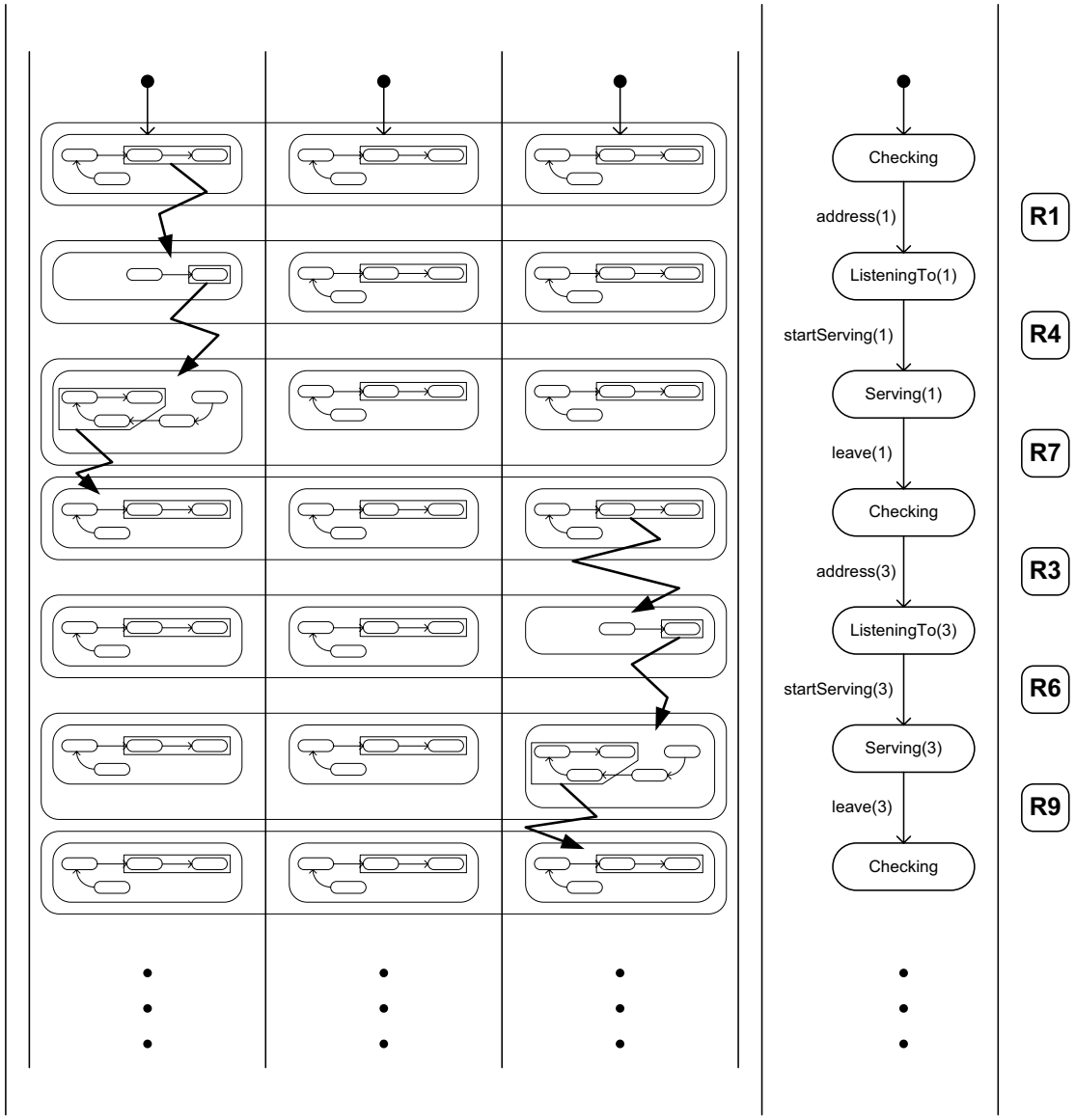


Figure 2.9: Three *Client(i) [AsOOS]* processes in relation with the *Server* process

In our client/server example, precisely one consistency rule has been defined for each transition of each detailed process. Note, however, that PARADIGM allows the definition of more than one consistency rule with the same manager part. In that case, one of these consistency rules is selected non-deterministically for application.

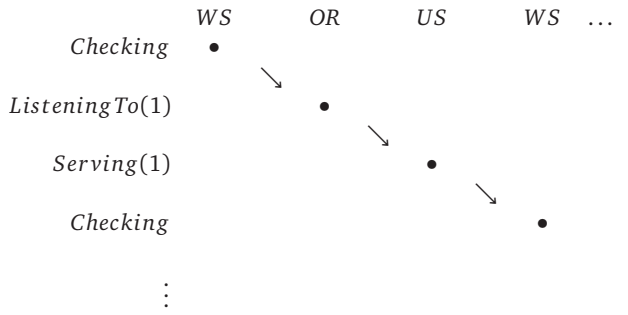


Table 2.10: Mutual constraints of *Server* (↓) and *Client(1)[AsOOS]* (→) in combination

PARADIGM Models

The execution of an entire PARADIGM model is equal to the concurrent execution of all its processes, taking into account the mutual restrictive influence they have on each other via the partitions and the consistency rules. In this respect, the execution of global processes differs from the execution of detailed processes.

Global Transitions

A transition $S_j \xrightarrow{\theta} S'_j$ of a *global* process $P(\pi_j)$ at the level of partition π_j can be taken if:

- the current state of process $P(\pi_j)$ is S_j;
- the current state of P lies in trap θ of subprocess S_j, i.e. the current state of $P(\pi_j)$.

In that case, transition $S_j \xrightarrow{\theta} S'_j$ is said to be *enabled*, otherwise it is said to be *disabled*. Because the transition belongs to a global process, it is only taken as the *result of the application of a consistency rule*, which happens as the consequence of a detailed manager process taking a transition.

Detailed Transitions

A transition $s \xrightarrow{a} s'$ of a *detailed* process P can be taken if:

- the current state of process P is s;
- for all $j \in J$, subprocess S_j corresponding to the current state of $P(\pi_j)$ contains transition $s \xrightarrow{a} s'$;
- there exists at least one consistency rule R with a manager part corresponding to $s \xrightarrow{a} s'$ which is *enabled*: that is, the set of global transitions in the employee part of R can all be taken.

If the above conditions hold, transition $s \xrightarrow{a} s'$ is said to be *enabled*, otherwise it is said to be *disabled*. If transition $s \xrightarrow{a} s'$ is indeed taken by process P, then one of the corresponding enabled consistency rules is non-deterministically selected and *applied*: all global transitions in the employee part of the selected consistency rule are taken simultaneously. Thus, a consistency rule defines the effect of taking the transition of a *detailed* process in terms of the taking of zero or more transitions of *global* processes. Thereby, detailed processes in a PARADIGM model are able to interact with each other only *indirectly*, via global processes.

2.4 Discussion

We presented PARADIGM as an *interaction modeling language*: its two basic principles, *multiple views* and *manager/employee*, aim at providing structure to the interaction of multi-component systems. Multiple partitions and global behaviors per component allow the modeler to concentrate on specific parts of the interaction separately. The global behaviors themselves abstract away from internal details of the component. They can be compared to e.g. *interface automata* [21, 22] and UML 2 *protocol state machines* [74, 73]. These notions specify, for individual components, the order in which they accept messages and in which they themselves send messages to other components. Clearly, PARADIGM does not incorporate the notion of a message as used in the UML. PARADIGM also does not specify input or output actions on transitions like in [21]. An interesting approach is presented in [70], in which multiple *protocol machines* (comparable to global processes) are combined in order to specify a repertoire of accepted events. Abstract behavioral descriptions on top of more detailed component behavior also simplify to a certain extent the process of analyzing and model-checking the interaction within a system, primarily because of state space reduction at the level of interaction. In PARADIGM, for example, the interaction between components can be completely understood by only considering the detailed behavior of a set of managers together with the relevant global behaviors of their employees.

The particular way in which views on top of the detailed behavior of components are modeled in PARADIGM, differs considerably from the way in which abstractions are created in e.g. statecharts [51, 52], on which UML state machines are based. PARADIGM does not contain the notion of a region; at the detailed level, a component has one single thread of control. Merely, global processes in PARADIGM can be regarded as behavioral models describing the possible orderings of *phases* through which the component moves during its detailed execution. Typically, these phases overlap: a single detailed state can be contained in multiple subprocesses of the same partition. In our client/server model presented earlier, we already showed an application of these overlapping phases on top of process *Client*. The detailed behavior and global behaviors of a component are thereby, behaviorally speaking, "loosely coupled". An interesting application of this idea can be found in [46] and [40], where it has been applied for the modeling of *evolution on-the-fly*. The evolution of a component can in fact be regarded as the transition to a new, *previously non-existing* phase in its execution. Appropriate application of the PARADIGM concepts and careful coordination allow for the precise modeling of controlled evolution of a system's behavior. Apart from allowing parts of a PARADIGM model to be updated, created or deleted, the concepts of the language are sufficient for the creation of PARADIGM models which evolve during their execution.

The manager/employee principle of PARADIGM deserves some attention in this discussion as well. Clearly, managers in PARADIGM are "in charge" of establishing the interaction between components: their behavior can be regarded as a sequentialization of the application of a set of consistency rules. They can be compared with some of the *behavioral patterns* found in literature on design patterns [38], like the *Mediator* pattern, which encapsulates how a set of objects interact. Such a mediator pattern is actually apparent in PARADIGM in the combination of a set of consistency rules on one hand, and the sequentialization imposed on them by one or more managers on the other hand. In fact, managers in PARADIGM models often combine their role as a "consistency rule sequentializer" with their role as a representative of an entity in the system. E.g., in our client/server example, the *Server* process manages the interaction between itself and the clients, but it also represents the (real-world) server itself, i.e. the entity which serves the clients. This choice is in itself quite arbitrary: one could also have modeled the three clients as managers of a single server employee, or even a separate manager process managing the interaction between one server employee and three client employees.

Finally, we draw a note on the application of consistency rules in PARADIGM. As we explained above, a consistency rule is applied once a transition of a detailed process corresponding to the manager part of that rule is taken and the rule is selected for application. Since all processes run concurrently, it is therefore possible for two consistency rules managed by different manager processes to be applied *simultaneously*. This causes no problems as long as these consistency rules are concerned with different global processes in their employee parts. However, if the employee parts overlap, conflicts in their parallel application could arise: only one of them must be chosen for application. In a distributed implementation of PARADIGM, such conflicts should be taken into account and either prohibited or solved.

2.5 Conclusions

In this chapter, we introduced PARADIGM. A model in this language consists of a set of interacting components running concurrently. The detailed behavior of each component is modeled as a *detailed process*. On top of each of those detailed processes several views can be created, according to the *multiple views* principle of PARADIGM. This is done by means of *partitions*, which have corresponding *global processes*. A global process models the relevant behavior of the component in its role in the interaction with a set of other components. The interaction within the system is modeled by means of *consistency rules*, according to the *manager/employee principle*: a transition of a detailed *manager process* is bound to transitions of a set of global processes defined on top of *employee processes*. All processes of a PARADIGM model (both global and detailed) are assumed to be executed concurrently, thereby keeping into account the restrictions on their behavior embodied in the partitions and consistency rules.

PARADIGM models typically have a structure like in the component diagram of Figure 2.11. The detailed process and zero or more global processes are drawn as active objects, while partitions are represented as connectors between a detailed and a global process. The set of consistency rules is drawn as a collaboration, to which the relevant processes are connected by means of connectors. Note in particular that the consistency rules for *pure employee processes* are not part of this collaboration. These consistency rules have an empty employee part and do not define any interaction. Therefore, they can be applied concurrently without conflicts.

As we pointed out in the discussion, we regard PARADIGM particularly useful for the modeling of interaction between components. Both the multiple views principle and the manager employee principle contribute to the structuring of interactions. The unique way in which views on detailed processes are modeled in PARADIGM, enables the elegant modeling of evolution on-the-fly, interpreting the future behavior of a process as a new, previously unknown phase in its overall behavior. The manager/employee principle in PARADIGM sometimes forces the modeler into making an arbitrary choice which components to assign manager and/or employee roles. In a distributed implementation of PARADIGM, one must be aware of potential conflicts between applications of overlapping consistency rules.

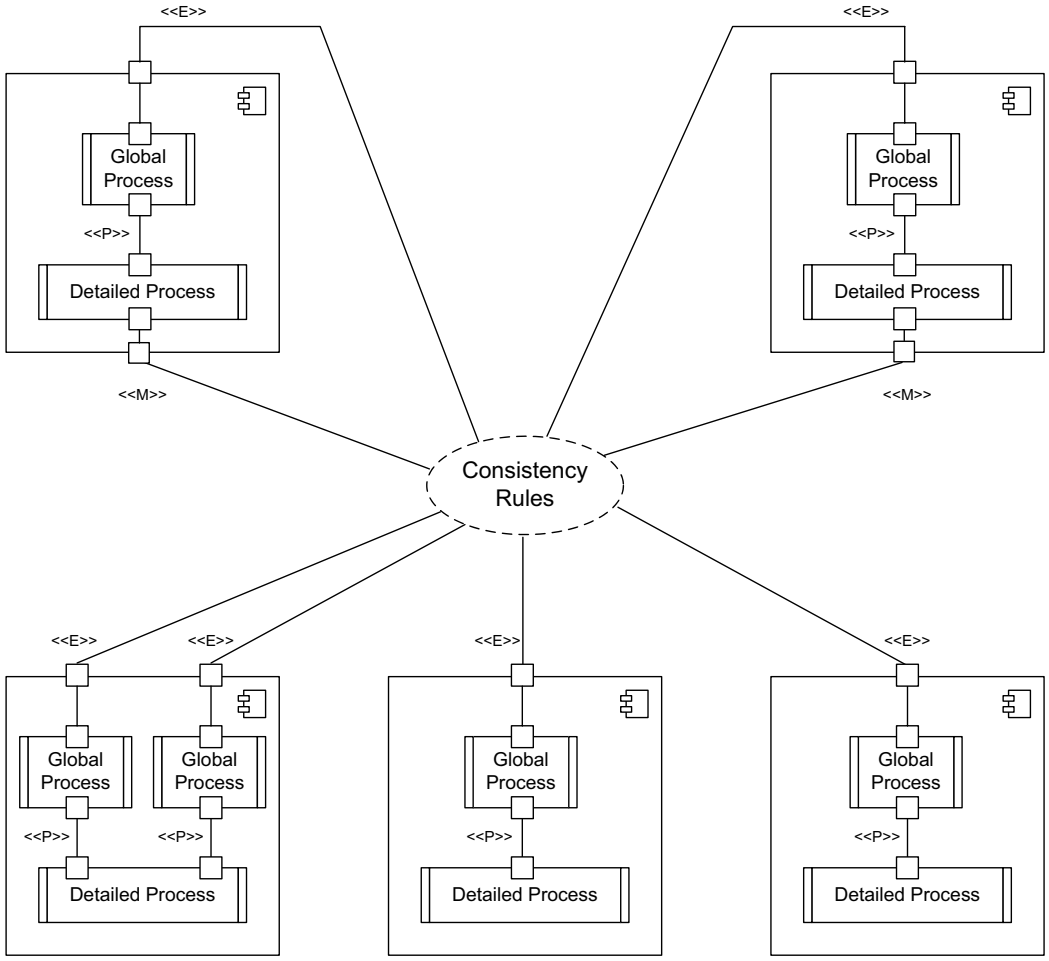


Figure 2.11: General Structure of a PARADIGM Model

Chapter 3

PARADISE

A Distributed PARADIGM Interpreter Framework

We present PARADISE, a framework for creating distributed interpreters for PARADIGM models. The design of PARADISE allows for maximal distribution of the PARADIGM processes being executed. In order to facilitate experiments with the implementation of the PARADIGM language concepts, PARADISE has been implemented as a framework with several composable interpreters for the individual concepts of the PARADIGM language. We use the client/server model introduced in the previous chapter to illustrate the implementation of the elements of the PARADISE framework.

3.1 Introduction

PARADISE is a software framework for the creation of a *distributed interpreter* for a given PARADIGM model. Such a distributed interpreter can be used to execute a PARADIGM model in a fully distributed manner, with each process running on a separate processor. The framework has been designed with two objectives in mind. Firstly, PARADISE supports maximal concurrency in the execution of a PARADIGM model. Thereto, each detailed and global process is perceived to be executed on an individual *virtual node*. Secondly, we aim at a minimum of synchronization between the process executions. Communication between the nodes, needed for adherence to the behavioral constraints in the model, is done via *asynchronous channels*. The PARADISE framework consists of small interpreters for specific concepts of the PARADIGM language, like processes, partitions and consistency rules. Elements of the framework can be easily replaced or extended, which eases experiments with the language. The framework elements can be composed in order to form a concrete distributed interpreter for a specific PARADIGM model.

In this chapter, we focus on the implementation of PARADIGM in PARADISE and on the interpretation of the concepts of the PARADIGM language by the elements of the PARADISE framework. Technological issues, like the programming language used and the execution environment, are deferred to Chapter 5, in which we present the PARADE distributed runtime environment and related tools. Pseudo code for the individual elements of the PARADISE framework can be found in Appendix A.

In Section 3.2, we give an overview of PARADISE and present the contents of the framework. In Sections 3.3, 3.4 and 3.5 we zoom into the implementation of processes, partitions and consistency rules, respectively. We illustrate these three sections with the creation of a PARADISE distributed interpreter for the client/server model introduced in Chapter 2. We discuss the framework and related work in Section 3.6 and draw conclusions in section 3.7.

3.2 Overview

A distributed interpreter built with PARADISE (a PARADISE interpreter for short) is distributed over a set of *virtual nodes*, which are connected to each other by a network of bidirectional point-to-point *channels*. The nodes are virtual: they could each correspond to a single physical computing device, or several nodes could be combined on one physical computing device. In this manner, we abstract away from the physical distribution of the interpreter and regard it as fully distributed. We visualize PARADISE interpreters similar to deployment diagrams in the UML, as shown in Figure 3.1. On each of the virtual nodes, one of the following entities in a PARADIGM model is executed: a *detailed process*, a *global process*, or *the set of consistency rules*. Communication between the nodes is possible via *asynchronous channels*. The channels are used to communicate constraints imposed by partitions or consistency rules.

In line with the intuition underlying PARADIGM that all processes in a model are executed concurrently, we use a virtual node for each process in a PARADIGM model. For convenience, we call these nodes *process nodes*. Hence, the processes executed by the PARADISE interpreter are virtually maximally distributed. On each process node, a *process handler* is "deployed", capable of executing a process by sequentially taking transitions. As we pointed out in Chapter 2, we must be aware of the fact that, in principle, the parallel application of consistency rules could introduce conflicts. In order to avoid these, we use a separate virtual node for the implementation of the entire set of consistency rules. On this *ruleset node*, a *ruleset handler* performs the application of consistency rules, *one at a time*. However, the consistency rules of *pure employee processes* are not implemented in PARADISE, since they do not define any interaction. As a consequence, pure employee processes can run concurrently with all other processes.

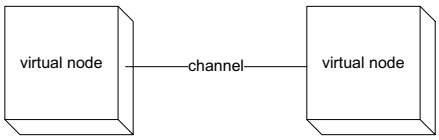


Figure 3.1: Visualization of virtual nodes and channels

Execution of the processes is constrained by partitions and consistency rules, for which communication between the nodes is needed. We have chosen to use *asynchronous* bidirectional channels, which are assumed to have *unbounded* capacity. As we will show in Sections 3.4 and 3.5, the usage of asynchronous channels instead of synchronous primitives is directly appropriate for most of the communication in PARADISE. We use three communication primitives for the channels: *send*, which sends a message via a channel and never blocks, *receive*, which receives a message from a channel and blocks if no message is available, and *isEmpty*, which tests whether a channel contains an incoming message and returns either true (no incoming message) or false (a message can be received).

The channels in PARADISE are used for three types of communication. Partition restrictions are maintained by communication between the two nodes that execute the detailed and global process of the partition, via a *partition channel*. Restrictions on manager processes and global processes of employees are maintained by communication between the corresponding process nodes and the ruleset node. This communication takes place via *manager channels* and *employee channels*, one for each process node involved. Process nodes which execute a pure employee process do not communicate with the ruleset node, since their consistency rules define no interaction.

An example of the general structure of a PARADISE interpreter is depicted in Figure 3.2. It represents the distributed interpreter for the PARADIGM component diagram of the previous chapter, Figure 2.11. As can be seen, each component in the PARADIGM component diagram is implemented in the PARADISE interpreter as a non-empty set of nodes, one for the detailed process and one for each global process. The nodes of a single component are connected by partition (≪P≫) channels. The set of consistency rules is implemented in PARADISE on a separate ruleset node, to which all manager processes and global processes of employees are connected by employee (≪E≫) and manager (≪M≫) channels. On the three nodes at the bottom of the figure, pure employee processes are executed. Therefore, they are not connected to the ruleset node.

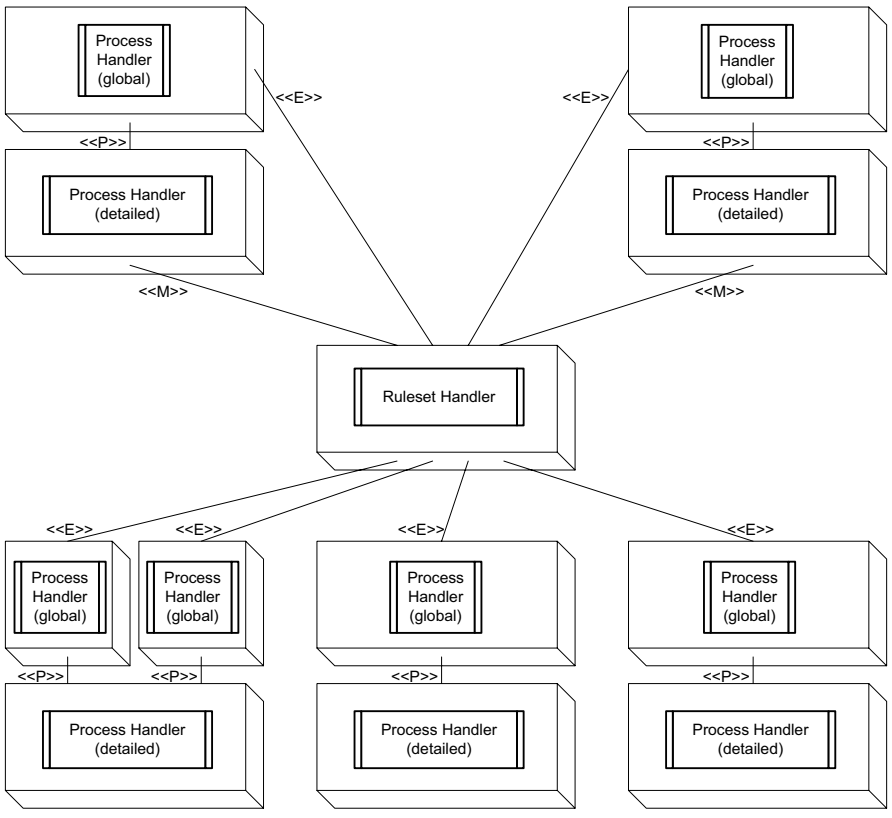


Figure 3.2: General Structure of a PARADISE Distributed Interpreter

Figure 3.2 clearly illustrates that the structure of a PARADISE distributed interpreter depends completely on the PARADIGM model that the distributed interpreter executes. The processes, partitions and consistency rules in the model determine which nodes exist in the distributed interpreter and how they are connected to each other. Consequently, a distributed interpreter which can execute any arbitrary PARADIGM model must be able to adapt its distributed structure to the model at hand. In view of this, we have implemented PARADISE as a *generic framework*, independent of a specific PARADIGM model. The elements of the framework can be used as building blocks for the creation of a distributed interpreter for a specific PARADIGM model. They can be parameterized with parts of the PARADIGM model and connected to each other in order to form a distributed interpreter for the execution of that specific model. In Figure 3.2, we already showed two elements of the PARADISE framework: the *process handler* and *ruleset handler*. If, for example, a process handler element is parameterized with the model of a PARADIGM process and deployed on a virtual node, the result is a concrete implementation of that PARADIGM process.

The focus of this chapter is on how the PARADISE framework has been organized and how its elements implement the PARADIGM language. We start with an overview of the framework, as shown in Figure 3.3. This figure is drawn as a UML class diagram, with thick dashed lines indicating which elements are able to communicate with each other via channels in the distributed interpreter. The elements on top of the figure are meant for deployment on process nodes, while at the bottom the elements are shown which must be deployed on the ruleset node. The *process handler* and *ruleset handler* are the active elements in the framework: they steer the execution of a process and the application of a set of consistency rules, respectively. The remaining elements implement more detailed tasks in the execution of a PARADIGM model. For the implementation of partitions, the framework provides two *role handler* elements which communicate via a partition channel. The *selectors* implement the application of consistency rules on the node of a process. Three specialized selectors are provided, of which one must be chosen depending on the process at hand being a global process, a pure employee process or a manager process. The *rule handler* element handles the application of a single consistency rule, while the *proxy* elements implement, on the ruleset node, the communication between this node and the process nodes of managers and global processes of employees.

In the next three sections, we zoom in onto specific elements of the PARADISE framework. We use the client/server model of Chapter 2 to illustrate how a complete PARADISE distributed interpreter can be created for a PARADIGM model. In Section 3.3 we show how detailed and global processes are executed in the implementation, in Section 3.4 we show how partitions are implemented, and in Section 3.5 we elaborate on the implementation of consistency rules. Throughout the sections, we use an informal style of presentation for the elements of the PARADISE framework – more formal pseudo code for the elements is provided in Appendix A.

3.3 Processes

The execution of a process requires the taking of transitions, starting at a valid initial state of the process, until a final state has been entered and execution discontinues. For this purpose, PARADISE provides a *process handler* element. A process handler is deployed on each of the process nodes of the distributed interpreter. The process handler is parameterized with a specification of the process to be executed, including its states, transitions, initial states and final states. Process handlers are used for the execution of both detailed and global processes. A process handler executes a process by continuously taking the following series of steps:

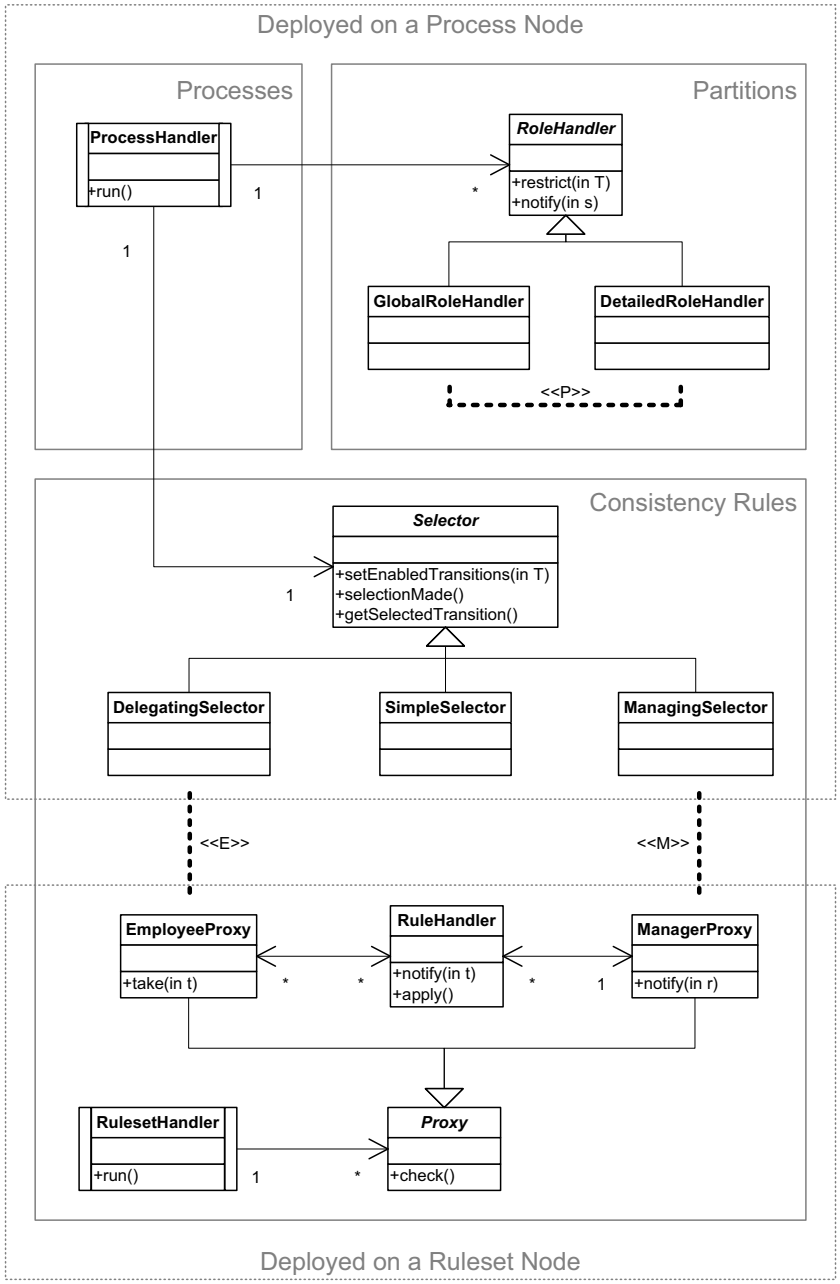


Figure 3.3: Overview of the PARADISE elements implementing PARADIGM

1. set the current state to one of the initial states;
2. if the current state is a final state, decide whether or not to stop execution;
3. determine the set of transitions which can be taken from the current state;
4. non-deterministically select one of the transitions from this set;
5. set the current state to the target state of the selected transition;
6. continue with step 2.

As we will see in the next two sections, steps 3 and 4 do not only depend on the current state and the transitions of the process, but also on the restrictions introduced by partitions and consistency rules. In step 3, the *role handlers* for partitions will be involved, and in step 4 the *selectors* for consistency rules.

The starting and stopping of processes in the implementation requires additional attention. In PARADIGM, one or more initial states can be defined for a single process. For a meaningful execution of the model, it is required that upon starting the execution, the first current states of detailed processes are contained in all of their first current subprocesses, i.e. the first current states of their global processes. In the implementation of PARADISE, this requirement holds as well. At startup, a process handler in a PARADISE distributed interpreter requires a parameter specifying which of the initial states of its process is selected as the first current state. Another issue concerns the final states of a process. In PARADIGM, these final states are states in which the execution of the process *could* stop. In PARADISE, the choice whether or not to stop in a final state is made by the process handler. By default, execution of a process always halts once a final state has been reached. An implementation in which a random selection is made between stopping or continuing the execution can be realized easily.

As an example, consider the deployment diagram of Figure 3.4. It shows an incomplete version of the distributed interpreter for the client/server example, containing only the virtual nodes and channels together with the process handlers, one for each process in the model. In the following sections, we will gradually extend this deployment diagram to a complete distributed interpreter, by deploying elements of the framework which implement the partitions and consistency rules of the client/server model.

3.4 Partitions

In a PARADISE distributed interpreter, the detailed process and global processes of a PARADIGM component are executed on separate virtual nodes. The implementation of partitions therefore obviously requires communication between these nodes. In this section, we recall the precise dynamic constraints imposed by partitions in PARADIGM, whereafter we show how these constraints are maintained through communication in the PARADISE implementation.

Analysis

In PARADIGM, a partition represents the dynamic constraints on a detailed and a global process, as follows. A state change in a detailed process can cause trap entrances in the current subprocess of a partition of that process. These trap entrances in turn enable transitions of the global process corresponding to that partition. The other way around, a state change in a global process stands for a subprocess change in the partition corresponding to the global process. This subprocess change in turn possibly causes a change in the set of enabled transitions at the detailed process. Some transitions may become enabled, in that they belong to the new current subprocess but not to the previous one, but

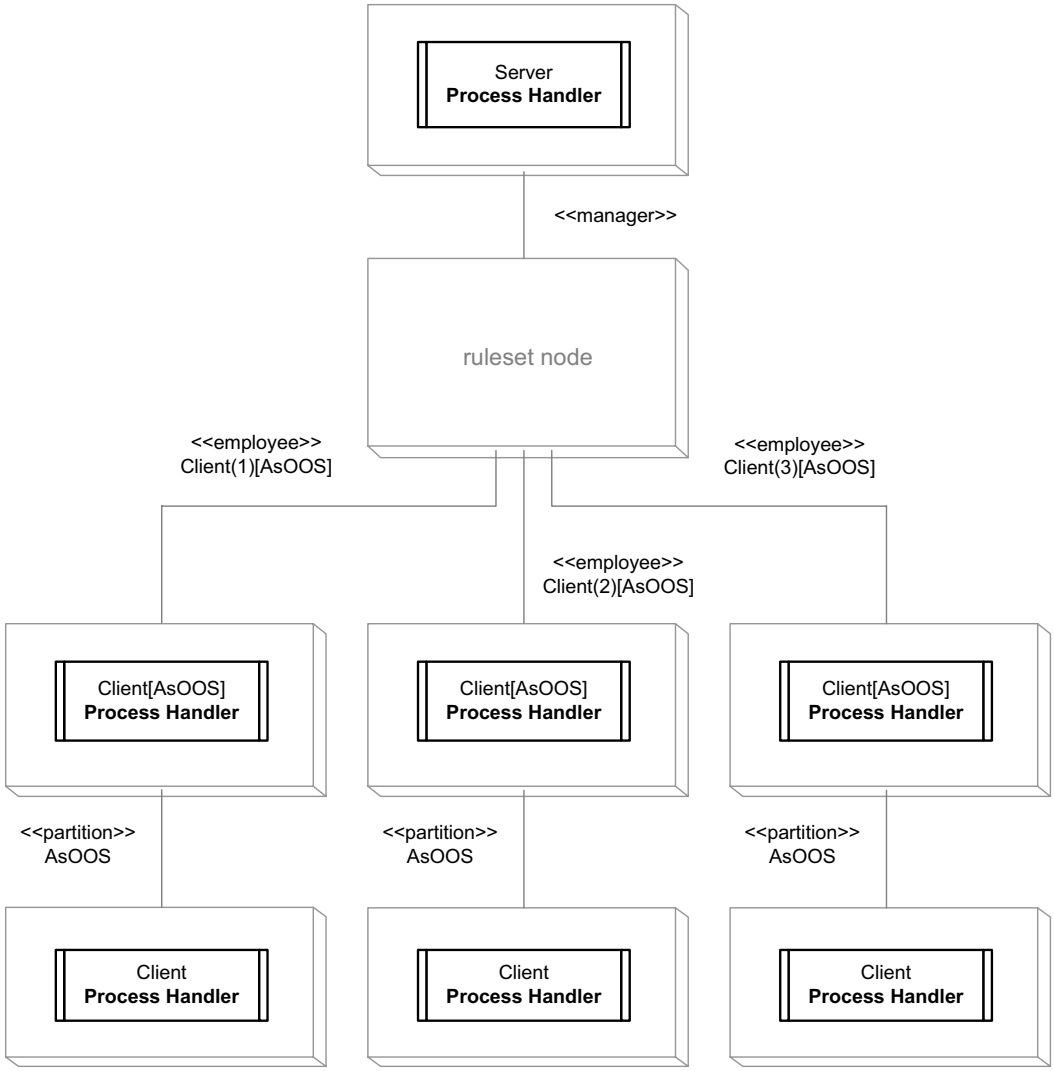


Figure 3.4: Process handlers in a PARADISE interpreter for the client/server model

transitions may become disabled as well. Whether or not transitions are enabled, depends on the current subprocesses of all partitions of the detailed process together. In the following table, we summarize the dynamic constraints of a partition on its detailed process and its corresponding global process, in terms of the effect of a state change in one process on the set of enabled transitions in the other.

detailed process		*partition*		*global process*
state reached	$\longrightarrow$	traps entered	$\longrightarrow$	transitions enabled
transitions enabled/disabled	$\longleftarrow$	subprocess changed	$\longleftarrow$	state reached

Clearly, the constraints imposed by one process on the other consist of dynamic changes to the set of enabled transitions, which are caused by local state changes. Remark that a state change in a detailed process can never cause transitions in a global process to become *disabled*, by definition of the trap concept. Hence, the amount of traps that a detailed process enters within a single subprocess of a partition is always *monotonically increasing*. This is different from the effect in the opposite direction: subprocess changes can certainly cause detailed transitions to become disabled.

Implementation

As to have a separation between the execution of processes and the adherence to partition constraints, we have implemented partitions as two additional elements in the framework: a *detailed role handler* and a *global role handler* (see Figures 3.3 and 3.5). A detailed role handler is deployed on the process node containing the detailed process, and a global role handler on the process node containing the global process. The role handlers communicate with each other via a *partition channel*.

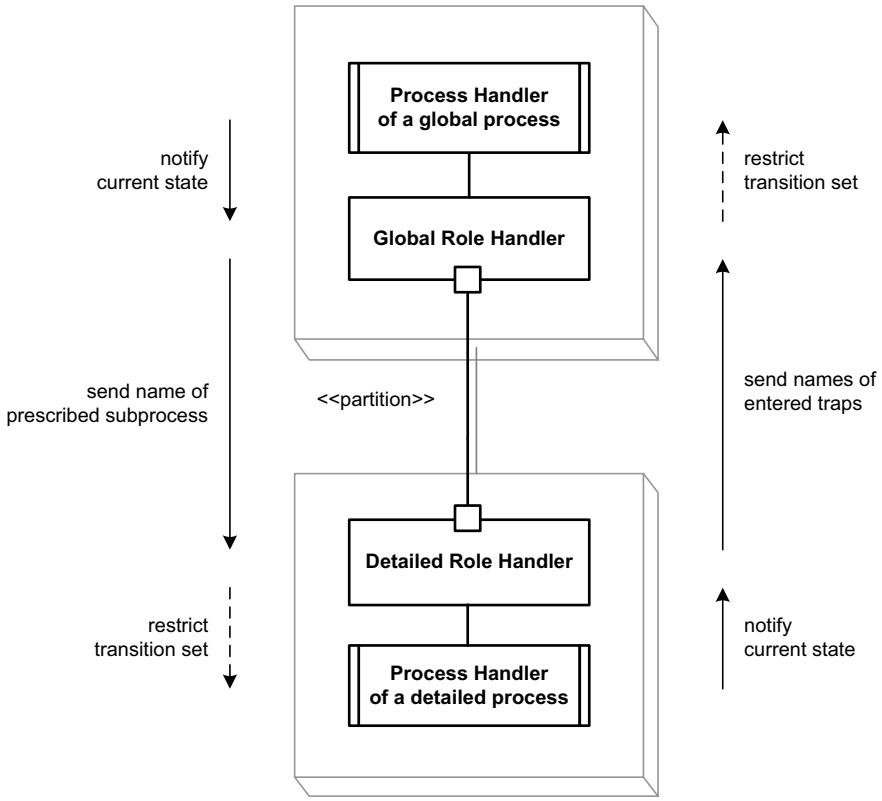


Figure 3.5: Overview of a partition between two process handlers in PARADISE

Role handlers can be dynamically *registered* at the process handlers. Once a role handler is registered at the process handler, the latter one conforms to the constraints imposed by the partition, as follows. A process handler *notifies* a role handler whenever a state change occurs in the execution of its process. In addition, it asks a role handler to *restrict* the set of transitions each time before one transition from this set is selected in order to be taken. The role handler restricts the set by simply removing transitions which are not allowed according to the current constraints of the partition. We minimize the amount of communication between two role handlers by letting them exchange the names of traps and subprocesses *only*. As a consequence, global role handlers need no partition specific information, since they only compare and exchange trap names and subprocess names. Detailed role handlers, in contrast, are parameterized with all subprocess definitions of a partition, which they use to check for enabled transitions in a subprocess and to check in which traps a detailed state lies. A detailed role handler only notifies a global role handler if a trap has been entered.

An overview of an implemented partition is shown in Figure 3.5. Starting at the right bottom of the figure, the dynamic constraints of a partition are maintained as follows. Whenever a state change occurs in the detailed process, the process handler of that process notifies the detailed role handler. The latter one is parameterized with the subprocess definitions of the PARADIGM model. Hence, it knows about which states and transitions are contained in which subprocesses and which states belong to their traps. Upon being notified, the detailed role handler checks in which traps the current state of the detailed process lies, and sends a list of names of entered traps to the global role handler via the partition channel. This communication takes place asynchronously: the list is received as soon as the process handler of the global process asks its global role handler to restrict the set of transitions. If that happens, the global role handler checks for each transition in the set whether the name of the trap corresponding to the transition label is in the list. If not, the transition is removed.

In the opposite direction, similar communication takes place. The global role handler is notified as soon as a state change in the global process occurs. Since the name of the current state of the global process equals the name of the current subprocess of the partition, the global role handler does nothing more than sending this name to the detailed role handler via the partition channel. Again, communication takes place asynchronously, and the detailed role handler receives the name of the newly prescribed subprocess as soon as it is asked by the process handler of the detailed process to restrict the set of transitions. The detailed role handler, prior to restriction, sets the current subprocess to the newly prescribed subprocess and checks whether the current state of the detailed process lies in some traps of this subprocess. If this is the case, it sends a list of entered traps to the global role handler. After that, it restricts the set of transitions by removing all transitions which do not belong to the current subprocess.

In a PARADIGM model, more than one partition may be defined on top of a detailed process. In that case, many detailed role handlers are registered at the detailed process handler. Notification and restriction then works as follows. In case of a state change, the detailed process handler notifies all detailed role handlers in some order. The set of transitions is restricted by providing the initial set to the first role handler and passing the result to the following role handler, etc., until all role handlers have removed the transitions that do not belong to the current subprocess of their partition. The resulting set contains only those transitions which are contained in every current subprocess. Note furthermore that, after the registration of a new detailed role handler at a detailed process handler, the latter one could ask this role handler to restrict the set of transitions before it has received the first current subprocess name from the corresponding global role handler. In that case, the detailed role handler returns an empty set of transitions. In other words, transitions of a detailed process are taken only *after* it is possible to adhere to the restrictions of all partitions.

Example

We show the implementation of partitions for the client/server example in Figure 3.6. Between the process handlers for the *Client(i)* and *Client(i)[AsOOS]* processes, role handlers have been deployed, which are parameterized with the subprocess definitions of the *AsOOS* partition. They communicate via the partition channels between the virtual nodes. As an example of the resulting constraints on the behavior of the processes, consider the sequence diagram shown in Figure 3.7. The process handler at the left side executes a detailed *Client* process, the one at the right side a global *Client[AsOOS]* process. In between, a detailed and a global role handler represent the *AsOOS* partition. The role handlers communicate asynchronously via the partition channel between the process nodes. The two process handlers continuously notify the role handlers about state changes (indicated with notify()) and ask them to restrict the set of transitions before selecting and taking one of them (restrict()).

We take a closer look at the communication scenario in Figure 3.7. At startup, the process handlers enter their initial state and notify their role handlers about this. After that, they determine the set of transitions that can be taken from the current state, and ask the role handlers to restrict this set. Initially, the detailed role handler returns the empty set, because it does not know the current subprocess yet. Upon applying the restriction a second time, however, it receives the current subprocess from the global role handler and returns a non-empty set of transitions. The process handler takes the transition and notifies the detailed role handler that the process is now in state *AtDesk*. Upon concluding that thereby trap *asking* has been entered, the detailed role handler sends a list of trap names to the global role handler. The list is accompanied with the name of the current subprocess, which is used by the global role handler to determine whether the traps in a trap list refer to traps of the current subprocess (since communication is asynchronous, it could happen that a list is received with traps referring to a previously prescribed subprocess). Meanwhile, the global role handler returns empty sets of transitions until it receives a list of traps containing trap *asking*. Hence, transition *asking* of the global process is enabled and taken. Thereby, the current state of the global process changes to *Orienting* and, upon notification, subprocess *Orienting* for the detailed process is prescribed. Once it is received by the detailed role handler, the detailed process is already in state *NeedClear*, hence in trap *serverClear* of subprocess *Orienting*. Therefore, a set of current traps is sent back directly. The example continues with a similar pattern of taking transitions and exchanging trap and subprocess names.

3.5 Consistency Rules

The primary issue in the implementation of consistency rules in PARADISE is to organize the communication between the virtual nodes involved. In this section, we recall the dynamic constraints imposed by consistency rules in PARADIGM, whereafter we show how these constraints are maintained through communication in the PARADISE implementation.

Analysis

A consistency rule in PARADIGM represents a dynamic constraint on one manager process and zero or more global processes of employees. We can express such a constraint operationally in terms of transitions being *enabled* and transitions being *taken*, as follows.

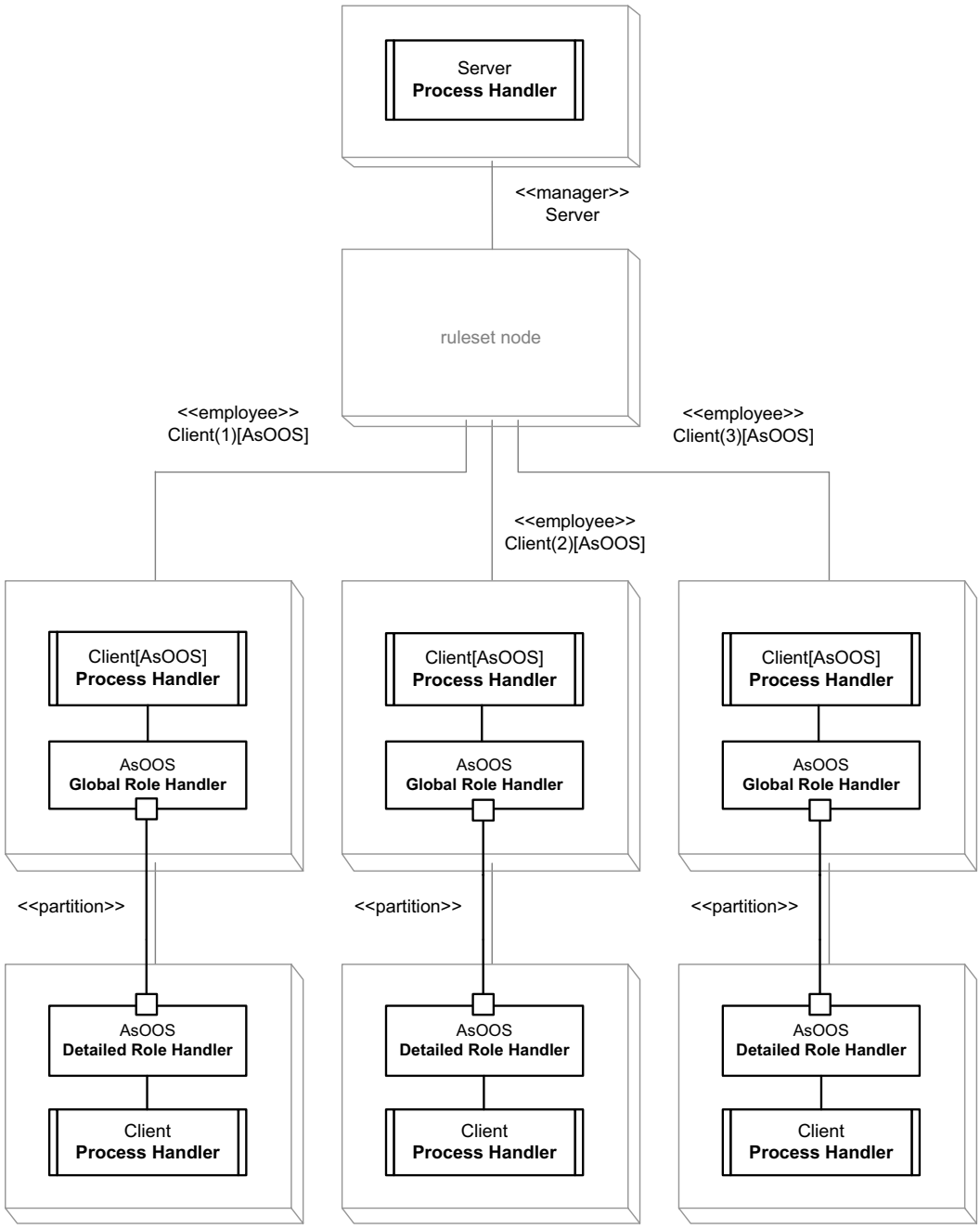


Figure 3.6: PARADISE client/server interpreter with processes and partitions

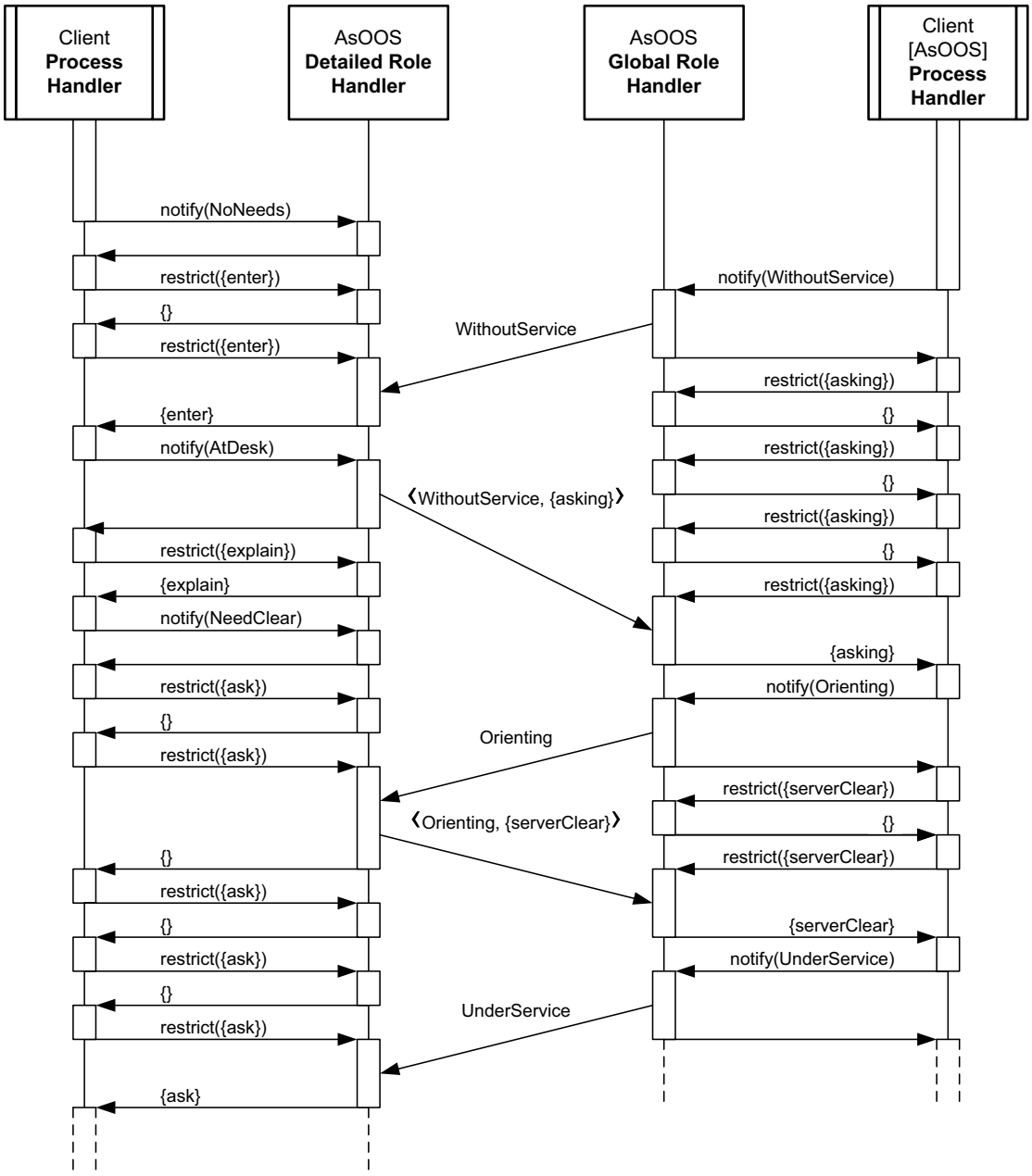


Figure 3.7: Communication for a partition in the PARADISE client/server interpreter

A transition of a global process is enabled as the consequence of the related detailed process having entered a trap of the relating partition. This enabling of transitions of global processes can cause consistency rules to become enabled (recall from Chapter 2 that we defined a consistency rule to be *enabled* if all transitions in its employee part are enabled). Consequently, if a consistency rule becomes enabled, the manager transition mentioned in its manager part could become enabled, depending on the current subprocesses of partitions which are possibly defined on top of the manager process. In the opposite direction, if in the execution of a manager process a transition is taken, a consistency rule is applied, which causes all transitions in the employee part of that consistency rule to be taken. Hence, the dynamic constraints of a consistency rule in terms of the effect on the enabling and taking of transitions in its related global processes and manager process can be summarized as follows.

global processes		*consistency rule*		*manager process*
transitions enabled	$\longrightarrow$	consistency rule enabled	$\longrightarrow$	transition enabled
transitions taken	$\longleftarrow$	consistency rule applied	$\longleftarrow$	transition taken

Clearly, some notification of transitions being enabled must take place between global processes and manager processes, via consistency rules. In the opposite direction, a manager process triggers the application of a consistency rule and consequently the taking of global transitions. If a consistency rule has an empty employee part, no global processes are involved in the communication. By definition, such a consistency rule is always enabled and its application has no consequence. Note that if two enabled consistency rules have different manager parts, two manager processes running concurrently could decide to apply both consistency rules. If the employee part of both consistency rules contain different enabled transitions of a *single* global process, the latter one is obliged to take two different transitions. Such a situation corresponds to the potential conflicts in the application of consistency rules in a truly concurrent setting, which we indicated in Chapter 2.

Implementation

In view of the above, we distinguish between consistency rules of pure employee processes and those of all other processes. For pure employee processes, the consistency rules have empty employee parts, hence they do not define actual behavioral constraints and do not need to be implemented. The consistency rules of all other processes are implemented on a single separate *ruleset node*. Thereby, we reflect the precise structure of PARADIGM models in PARADISE, while at the same time, we have a way to prevent conflicts in the application of consistency rules. We extend the PARADISE framework such that all issues related to consistency rules are addressed in separate elements. Consistency rules themselves are implemented with *rule handler* elements, while the communication of the enabling and taking of transitions is done by *selector* elements on the process nodes and *proxy* elements on the ruleset node.

An overview of the communication involved in the implementation of a consistency rule is given in Figure 3.8. On top, the node of a manager process is depicted. The nodes at the bottom of the figure execute global processes of employees. The communication *upwards* is about the enabledness of global transitions and the enabledness of consistency rules, while the communication *downwards* is about the application of consistency rules and the taking of global transitions. We follow the four parts in the communication as indicated with numbers in the figure.

Firstly, in order to determine whether a consistency rule can be applied, information about enabled transitions in the employee part of the rule is needed. This information comes from nodes executing global processes. Secondly, the actual application of a consistency rule is done on a node executing a manager process. Such a node then needs to be informed about which consistency rules may be applied. Thirdly, now from top to bottom, as soon as a transition of a manager process is taken, a consistency rule needs to be applied as well, about which the ruleset node must be informed in order to apply the employee part of the rule. Finally, the application of a consistency rule requires communication of the transitions in its employee part to each of the global processes.

As can be seen, the interface between the process handlers and the consistency rule "machinery" is formed by the *selectors*. These elements select one transition from a given set of enabled transitions, thereby taking into account the enabledness of consistency rules. The selected transition is then taken by the process handler. One selector element is deployed on each process node. The way in which a selector is implemented, depends on the type of process that the process handler runs. For pure employee processes, we use *simple selectors*, which simply select a transition non-deterministically (the fact that we use a selector for pure employee processes *at all* is merely for reasons of uniformity). The selectors of global processes must forward their enabled transitions to the ruleset node and receive a single selected transition back from it. We have implemented *delegating selectors* for that purpose. Finally, the selector of manager processes must perform the selection of a transition together with the selection of a consistency rule. This is done by *managing selectors*.

The generic structure of a ruleset node implementing a set of consistency rules is depicted in Figure 3.9. For each global process in the system, the ruleset node contains an employee proxy, and likewise, for each manager process, a manager proxy is deployed. For each consistency rule in the model, the ruleset node contains a rule handler. These rule handlers are always connected to a single manager proxy (because the manager part of a consistency rule precisely contains one manager transition), and to zero or more employee proxies (likewise, reflecting the fact that the employee part of a consistency rule contains zero or more global transitions). The thread of control of the ruleset node is implemented in the *ruleset handler*, which ensures fairness in the application of its consistency rules. It cycles through all employee proxies and manager proxies in a round-robin fashion. In each cycle, each proxy checks for an incoming message from its channel. If there is a message, the proxy takes the appropriate actions: employee proxies drive the communication *upwards*, manager proxies *downwards*, as we will explain in the following paragraphs.

Upwards communication is shown in Figure 3.10. This sequence shows the behavior of an employee proxy at the moment it notices the arrival of a message from the delegating selector at the opposite side of its employee channel. The received message contains a set of enabled transitions of a global process. The employee proxy now notifies the rule handlers of all consistency rules which have transitions from this global process in their employee part. In the sequence diagram, we have shown a single rule handler which is notified. Upon receiving this notification, each rule handler directly checks whether its consistency rule is enabled or not. If the state of its consistency rule (enabled or disabled) is swapped, the rule handler informs its manager proxy about this, after which the latter one sends an update of the set of enabled consistency rules for its manager process to the managing selector. All channel communication in upwards direction takes place asynchronously.

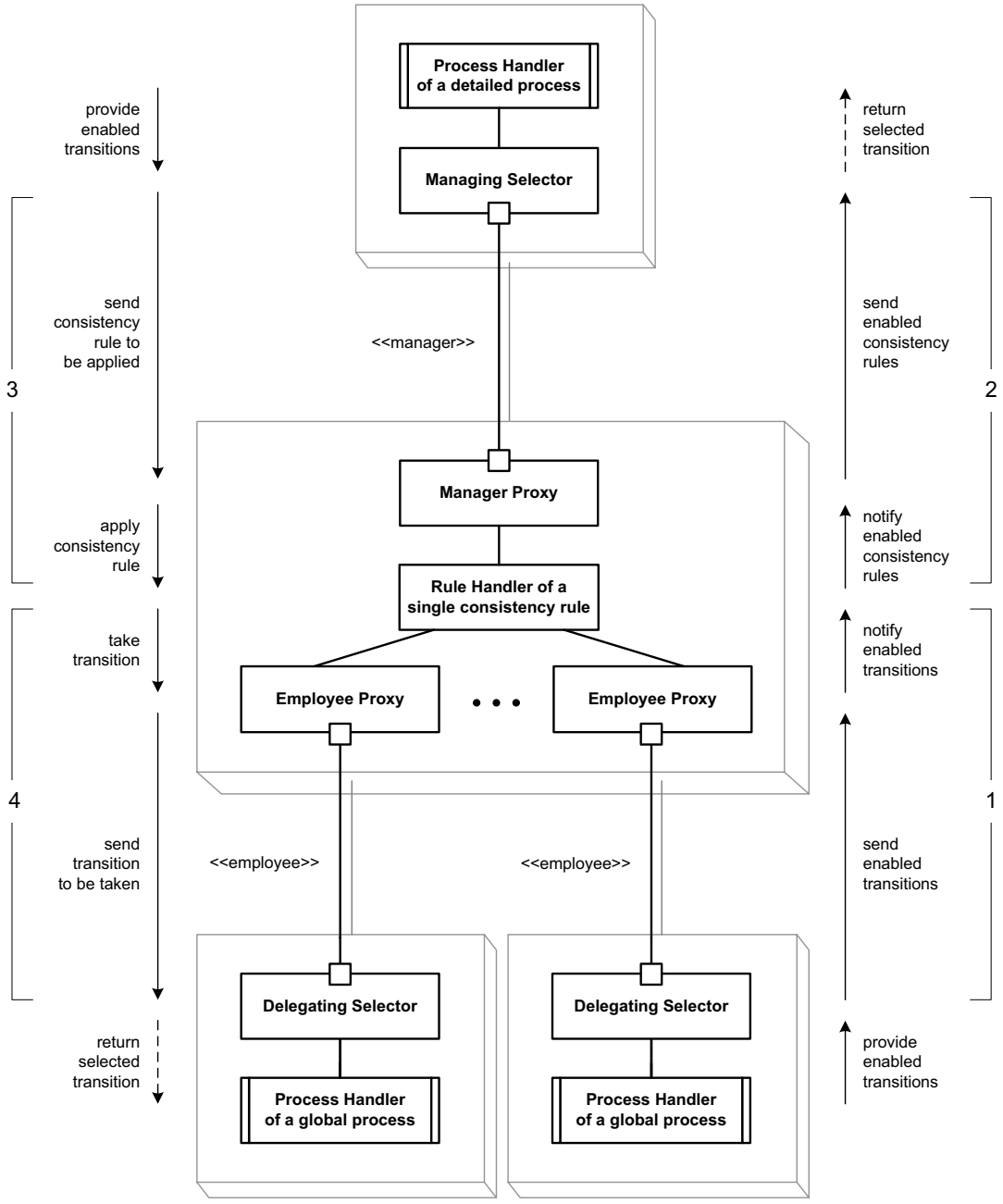


Figure 3.8: Handling consistency rules of global and manager processes in PARADISE

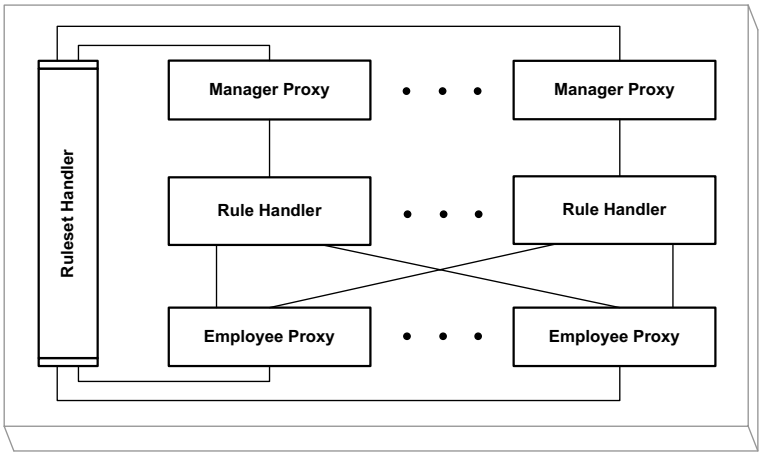


Figure 3.9: Generic structure of the ruleset node in PARADISE

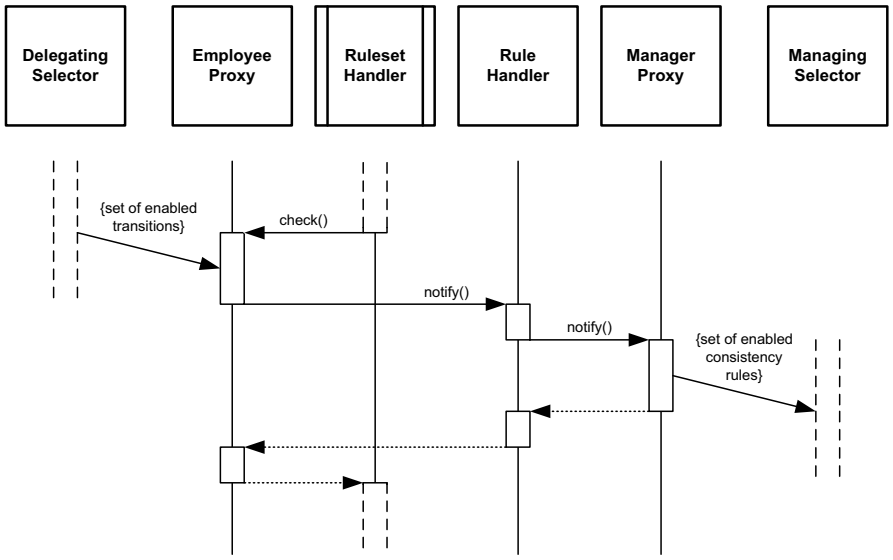


Figure 3.10: Actions of an employee proxy receiving a message from a delegating selector

Downwards, communication is slightly more complicated, since this establishes the actual application of a consistency rule. We must therefore guarantee that the information on which we decide to apply a consistency rule is up-to-date. A usual approach to this is to use a synchronization mechanism. For example, we could apply a *two-phase commit* protocol to ensure that all involved processes agree upon the application of a consistency rule. In the case of PARADIGM, however, there is no necessity to use

such a protocol over *all* processes involved. As we explained in Section 3.4, transitions of global processes can never become disabled after they have been enabled. By definition of the trap concept, the amount of enabled transitions from a certain source state in a global process is always monotonically increasing. Only when a transition is *taken*, it may become disabled. Hence, synchronization between the ruleset node and the global process nodes is only needed *after* the application of a consistency rule. We exploit this fact in our implementation, as follows.

Consider the actions performed by a manager proxy upon receiving a message, as shown in Figure 3.11. If a manager proxy receives a message from a managing selector, this effectively means that the latter one has selected a transition and a corresponding consistency rule, and now asks for the application of this consistency rule. Since upwards communication takes place asynchronously, the manager proxy and managing selector must now *synchronize* upon the set of enabled consistency rules in order to make sure that the consistency rule can indeed be applied. To this end, the manager proxy sends a sync message to the managing selector, which allows it to check whether all updates of enabled and disabled consistency rules have been received via the manager channel: it processes all incoming messages until sync is received, checks whether the previously selected consistency rule can still be applied, and either sends a *commit* or a *rollback* message to the manager proxy, indicating that the consistency rule must be applied or that the previous request was invalid, respectively. In case of a rollback, the manager proxy finishes directly (not shown in the scenario of the sequence diagram). Otherwise, it informs the rule handler of the selected consistency rule that its rule must be applied. The rule handler, in turn, directly informs all appropriate employee proxies that the transitions in the employee part of the consistency rule must be taken, whereafter these proxies send a message to their delegating selectors via their employee channels. Now, we need to synchronize upon the set of enabled transitions, since transitions could have been disabled. To this end, the delegating selectors simply send a sync message. We can be sure that, eventually, the global processes will take the prescribed transitions and change state. We therefore assume that all previously enabled transitions must be regarded as disabled, unless the source state and the target state in the global process are the same – in that case, the set of enabled transitions will be eventually resent by the delegating selector. The set of enabled transitions of each global process involved in the application of the consistency rule is now empty. The employee proxies notify the rule handlers and, via them, the manager proxies, in order for them to update the status of the consistency rules (enabled or disabled). Finally, control flow returns to the ruleset handler in order to continue with the next proxy.

Example

We conclude with a discussion of the complete PARADISE interpreter for the PARADIGM client/server example, as shown in Figure 3.12. The PARADIGM model contains 24 consistency rules in total, of which only *R1* to *R9* have been deployed on the ruleset node. The remaining 15 consistency rules are for pure employee processes, hence they are contained within the three simple selectors on the process nodes for processes *Client(1)*, *Client(2)* and *Client(3)*. The *Client[AsOOS]* processes are global processes. For that reason, they each have a delegating selector which communicates with a corresponding employee proxy on the ruleset node. The *Server* process is a manager process and therefore has a managing selector, communicating with a manager proxy at the side of the ruleset node. The rule handlers on the ruleset node are all connected to the same manager proxy, but to different employee proxies, according to the contents of the employee part of their consistency rule. The entire interpreter is executed by running all process handlers and the ruleset handler concurrently. The process handlers execute their processes, meanwhile being constrained in their execution by partitions and consistency rules.

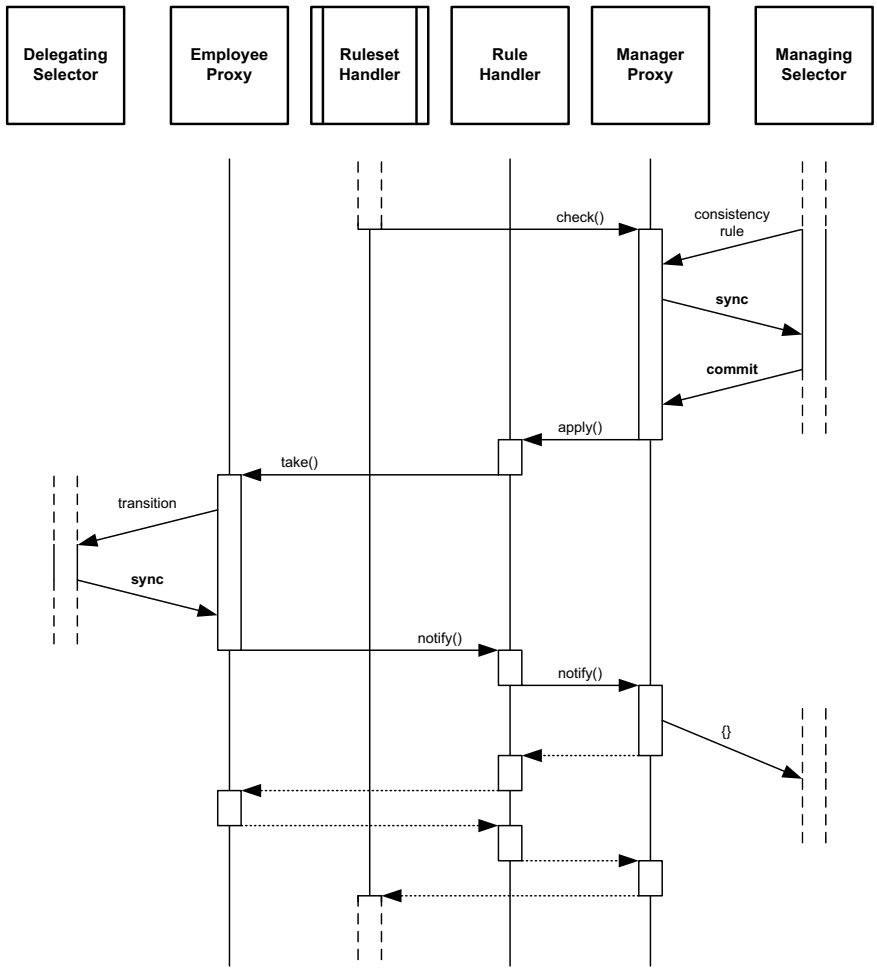


Figure 3.11: Actions of a manager proxy receiving a message from a managing selector

An example of the execution of two processes constrained by consistency rules is provided by Figure 3.13. This sequence diagram contains the first part of an execution of process *Client(1) [AsOOS]* and process *Server*. Both processes communicate with the ruleset node via their selectors. We have depicted four elements of the ruleset node: the ruleset handler, the involved proxies, and the rule handler for consistency rule *R1*, which plays a role in this fragment of the execution. When the distributed interpreter is started, the ruleset handler cycles through the set of proxies, while the process handlers of process *Client(1) [AsOOS]* and *Server* start the execution of their processes by entering the starting state (not shown). After that, both process handlers inform their selectors about the set of enabled transitions of their processes, from which the selectors are expected to select one transition.

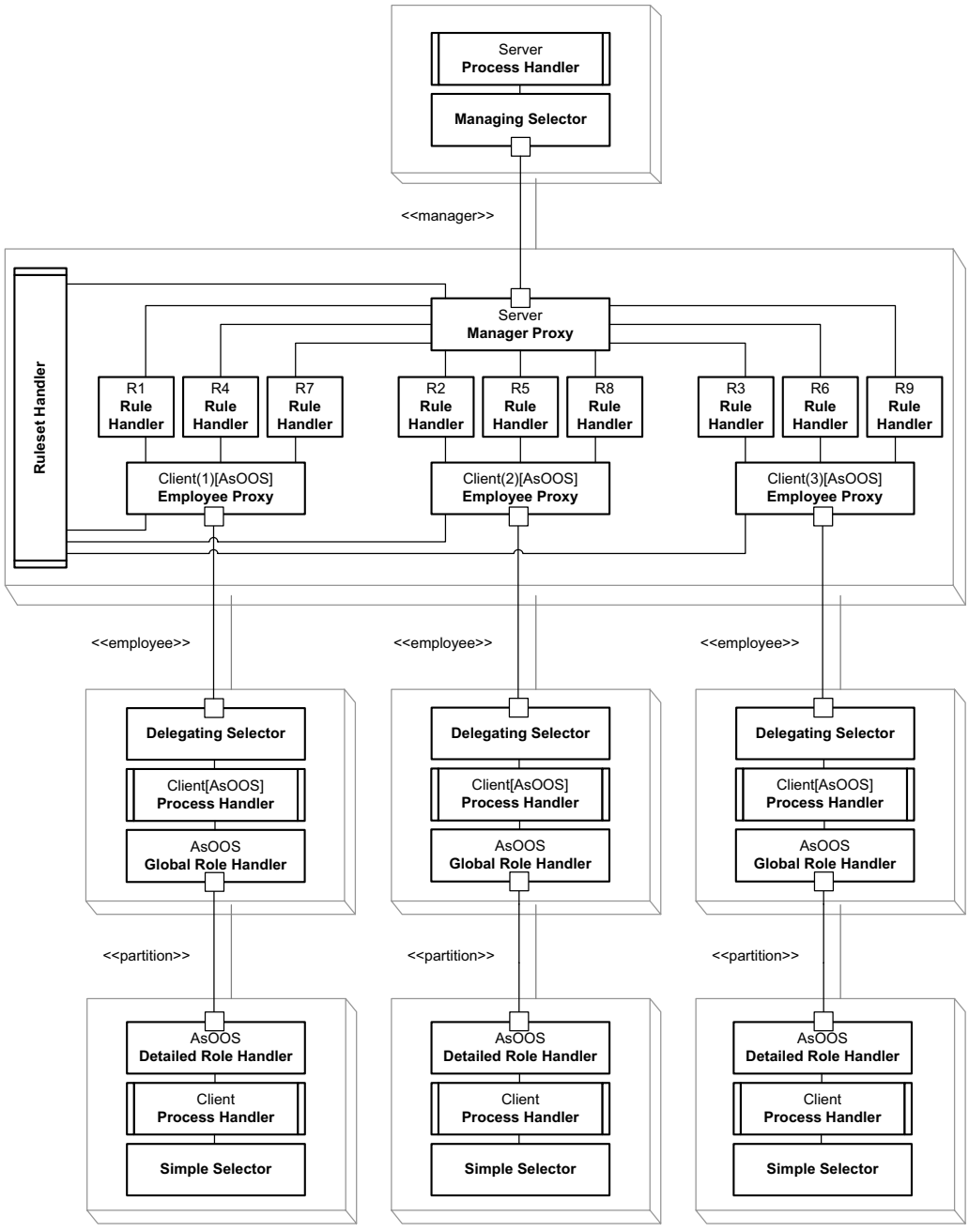


Figure 3.12: Complete PARADISE client/server interpreter

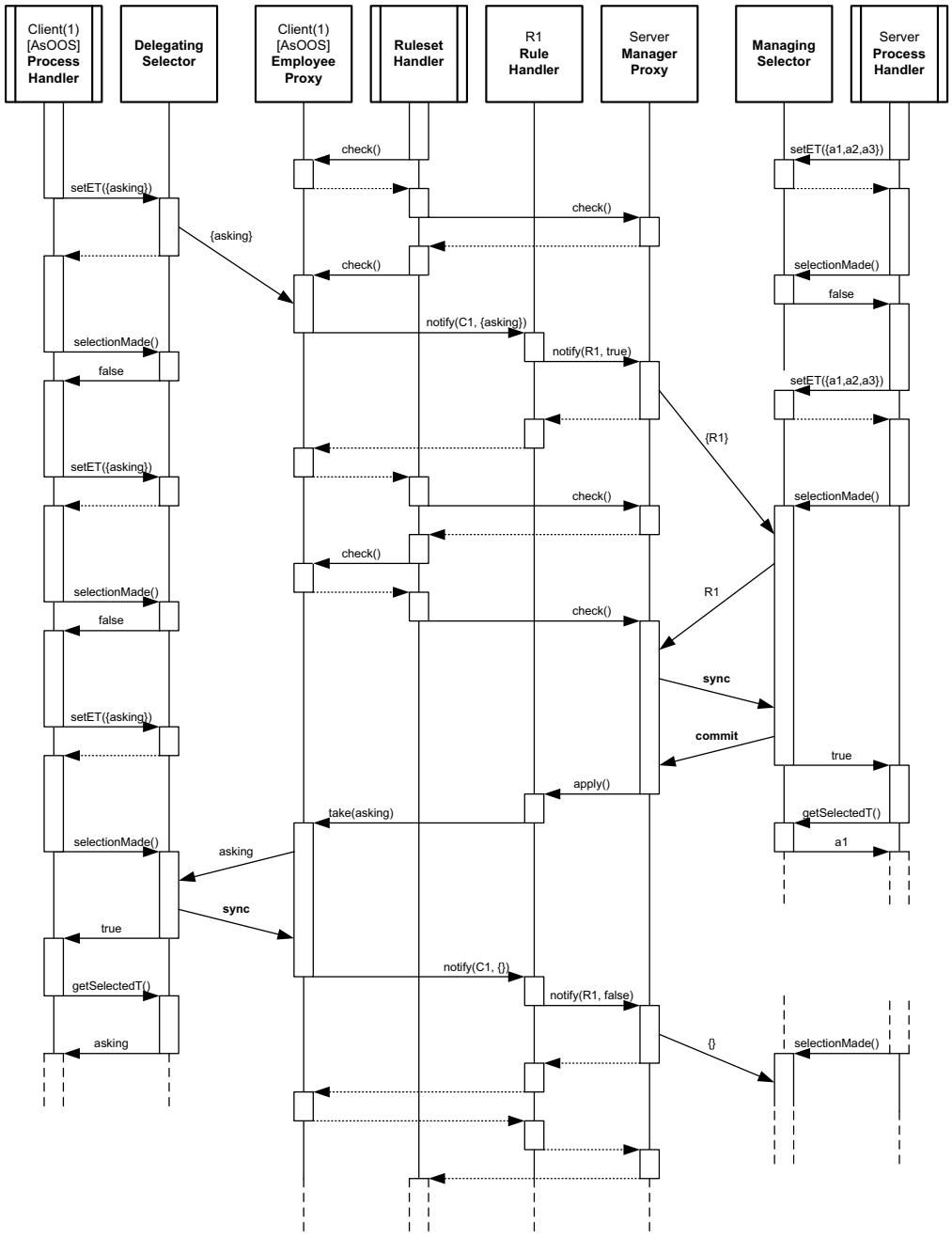


Figure 3.13: Comm. for a CR application in the PARADISE client/server interpreter

At the side of the *Server* process, three transitions *address(i)* (a_i for short) are enabled, but no selection can be made, since the managing selector does not know of any enabled consistency rule corresponding to the enabled manager transitions. At the side of the *Client(1)[AsOOS]* process, the delegating selector delegates selection by passing the set of enabled transitions to the ruleset node via its employee channel.

Now, as soon as the ruleset node asks the employee proxy of *Client(1)* to check for incoming messages, a set of enabled transitions is received containing transition *asking*: apparently, detailed process *Client(1)* has entered trap *asking*. Hence, the rule handler of consistency rule *R1* is notified, which in turn notifies the manager proxy that its consistency rule is enabled as far as the employee part is concerned. The set of enabled consistency rules, now containing rule *R1*, is sent to the managing selector of the *Server* process. This enables the *Server* process to take a transition and select a consistency rule to be applied. The managing selector of the *Server* process, at receiving the set of enabled consistency rules containing rule *R1*, finds a matching enabled manager transition *address(1)*, selects rule *R1* and sends this rule to the manager proxy. Eventually, the manager proxy receives *R1* and answers with a sync, in order for the managing selector to check whether rule *R1* can be indeed applied. This is the case, therefore the managing selector answers with commit. Hence, the employee part of the consistency rule is to be applied by the rule handler. It asks each of its employee proxies (in this case just one) to take its transition in the employee part. The proxy sends transition *asking* to the delegating selector, which answers with sync. After this, the employee proxy informs all rule handlers that the set of enabled transitions for the *Client(1)[AsOOS]* process is empty. Because thereby rule *R1* becomes disabled, the rule handler of rule *R1* informs the manager proxy: the empty set of enabled consistency rules is sent to the managing selector.

3.6 Discussion

With PARADISE, we have created the first distributed interpreter for the PARADIGM modeling language. The interpreter supports maximal distribution of the processes in a PARADIGM model, hence maximal concurrency in the execution of that model. Processes, partitions and consistency rules have been implemented as separate interpreter elements, in order to allow for the creation, updating and deletion of these elements while the interpreter executes a PARADIGM model. The primary purpose of PARADISE is to act as an *experimental framework* for the PARADIGM language, both regarding its concepts and their operational semantics. The particular choice we made to design PARADISE as a *framework* eases the creation of extensions and adaptations.

The choice to create an implementation of PARADIGM in terms of an *interpreter*, in contrast with a compiler, was primarily made in order to enable detailed inspection of a running PARADIGM model. We exploit this possibility in the PARADE runtime viewer, which we present in Chapter 5. Another reason is to achieve maximal flexibility, especially during the *execution* of a PARADIGM model. In [92], we reported on an implementation of PARADIGM for the TOOLBUS architecture [58, 61, 10]. This work showed us that runtime adaptivity is an important property of component-based systems, not only with respect to individual components (replacement or updating), but also with respect to their interaction (changes in their composition and coordination). The runtime flexibility of PARADE is exploited in the PARADE distributed runtime environment (see Chapter 5), which supports the application of changes to a PARADIGM model while it is being executed in a distributed manner.

The PARADISE framework, as it is presented in this chapter, has one major drawback: the usage of a *single* ruleset node for the consistency rules introduces a severe reduction of concurrency in the execution. This is caused by the fact that the ruleset node allows for only one consistency rule to be applied at a time. Thereby, all global processes and all manager processes run alternately, even if they are distributed. A possible solution is to allow for *multiple ruleset nodes*, which each implement a proper subset of the consistency rules in the model. Consistency rules of one ruleset node could then be applied concurrently with consistency rules on other ruleset nodes. In Chapter 4, we will introduce an extension to the PARADIGM language based on this idea.

A less obvious but useful extension is to generalize the usage of *partitions*. In PARADIGM, partitions are solely used to define views, in terms of global processes, on top of detailed processes. In PARADISE, detailed and global processes are treated equally. As a consequence, the framework readily allows for an extension to define partitions on top of *global processes*, i.e. views on views. Such an extension has been used in the case study about modeling *evolution on-the-fly*, which we will present in Chapter 8. In that chapter, we use a partition on top of a global process in order to coordinate its evolution. However, this extension comes at a certain cost. As we have seen in Section 3.4, subprocess changes of a partition on top of a process may cause transitions of this process to become disabled, by which the amount of enabled transitions from a global source state is no longer monotonically increasing. This has consequences for the application of consistency rules, which then must be performed via a two-phase commit over the manager process and all global processes involved. In Section A.6 of Appendix A, pseudo code for the PARADISE framework elements can be found in which these consequences are taken into account.

3.7 Conclusions

In this chapter we introduced PARADISE, a distributed interpreter framework for PARADIGM models. A PARADISE distributed interpreter consists of virtual nodes connected to each other by asynchronous bidirectional channels of unbounded capacity, on which elements of the framework are deployed which are parameterized with parts of the PARADIGM model. Each node executes a single process. One of the nodes, the *ruleset node*, manages the consistency rules. The implementation of PARADISE has been designed such that it allows for maximal distribution of the processes in a PARADIGM model. The amount of communication in the PARADISE distributed interpreter can be kept to a minimum by exploiting the trap concept. This way, almost all communication needed for adherence to partition constraints and consistency rules can be carried out in an asynchronous manner. A drawback of the framework is the usage of a single ruleset node for the implementation of consistency rules. Nevertheless, since the framework allows for easy experimentation with the implementation of the PARADIGM concepts, appropriate solutions to this problem can be integrated easily.

Chapter 4

Interaction Protocols in PARADIGM

We present the interaction protocol concept, an extension to the PARADIGM language. Interaction protocols are entities which serve as an anchorage for consistency rules and ensure that the consistency rules anchored to them are applied in a conflict-free manner. We show that this extension to PARADIGM is valuable for modeling, analysis and implementation purposes. Additionally, we generalize the consistency rule concept in PARADIGM by allowing their manager parts to be empty. This way, synchronization relations over a set of global processes can be defined without the need to model an explicit manager process.

4.1 Introduction

In this chapter, we extend PARADIGM with the *interaction protocol* concept. Interaction protocols are entities which serve as an *anchorage* for consistency rules. From a modeling and analysis perspective, they can be used to structure the consistency rules in a PARADIGM model into groups that represent separate aspects of interaction. In addition, interaction protocols regulate the application of consistency rules: they ensure that only one consistency rule in their group is applied at a time. The idea for the addition of the interaction protocol concept to PARADIGM stems from our implementation of the PARADISE distributed interpreter. The *ruleset node* can be easily split up into multiple ruleset nodes for separate sets of consistency rules in a single PARADISE interpreter. As we will show in this chapter, this idea can be effectively leveraged from the level of implementation to the level of modeling. In addition, we define a more precise notion of conflict between consistency rules. Based on this, we provide guidelines for the division of consistency rules amongst a set of interaction protocols, in order to ensure that conflicting consistency rules are kept together. Furthermore, as already mentioned, we introduce in this chapter a generalized shape of the consistency rule concept, by allowing the manager part of a consistency rule to be empty. This is possible because the interaction protocols are regarded as behavioral entities: they can directly select consistency rules for application themselves and do not need to defer this selection to an explicit manager process. This simplifies the modeling of *peer-to-peer* interaction, for which the components in a system are regarded as equals and as a consequence the manager/employee principle of PARADIGM cannot be satisfactorily applied.

We have set up this chapter as follows. In Section 4.2, we provide an overview of the extensions to the PARADIGM language introduced in this chapter. In Section 4.3, we provide definitions for three different types of interaction protocols, as well as for related notions like the notion of conflict between consistency rules. After that, in Sections 4.4, 4.5 and 4.6, we show examples of the three types of interaction protocols, which are all based on an extended version of the client/server example introduced in earlier chapters. We discuss related work in Section 4.7 and conclude in Section 4.8.

4.2 Overview

An interaction protocol is an *anchorage* for a set of consistency rules in a PARADIGM model. Each consistency rule is anchored to a *single* interaction protocol. Interaction protocols also play a role in the execution of a PARADIGM model: the consistency rules anchored to them are applied *one at a time*. However, interaction protocols perform this sequential application *concurrently* with other interaction protocols. This way, interaction protocols make the concurrency in the PARADIGM model explicit.

The generic structure of PARADIGM models with interaction protocols is shown in Figure 4.1. In this figure, the consistency rules are anchored to four separate interaction protocols. A line between a process and an interaction protocol indicates that at least one consistency rule is anchored to the interaction protocol which has a transition of this process in either its manager or its employee part. We distinguish between two basic types of interaction protocols: *covering* and *non-covering*. As the name suggests, covering interaction protocols cover the entire interaction between their roles, which are played by detailed manager processes and global processes of employees. As a consequence, the set of consistency rules anchored to a single covering interaction protocol is sufficient for understanding the interaction between its roles. In Figure 4.1, the covering interaction protocol covers the interaction between one manager process and two global processes. Non-covering interaction protocols, in contrast, cover the interaction between their roles only *partially*. They allow for the separation of multiple aspects or phases in the interaction, but the consistency rules of multiple non-covering interaction protocols must be taken together in order to fully understand the interaction between their roles. The two non-covering interaction protocols in Figure 4.1 both contain consistency rules with manager transitions of a single detailed manager process. For this reason, they are called non-covering, even though they cover the interaction with their respective global processes. In addition to the two basic types, we introduce the *self-managing* interaction protocol, which has the property that all consistency rules anchored to it have an *empty manager part*. Such consistency rules are applied by the interaction protocol *itself*, rather than by a manager process. Self-managing interaction protocols are not connected to manager processes at all. They can be either covering or non-covering. In the next section, we zoom in on covering, non-covering and self-managing interaction protocols, motivate them and provide definitions for them.

4.3 Definitions

The definitions of the various interaction protocols in this section are introduced in a number of steps and interspersed with the necessary notions for consistency rules. We start with the definition of *covering* interaction protocols. Next, we introduce a precise notion for *conflict* between consistency rules, which we use to define the ways in which we allow consistency rules to be divided among a set of interaction protocols. Based on this, we provide definitions of *sound* interaction protocols, *non-covering* interaction protocols and *minimal* interaction protocols. After that, we introduce the notion of consistency rules

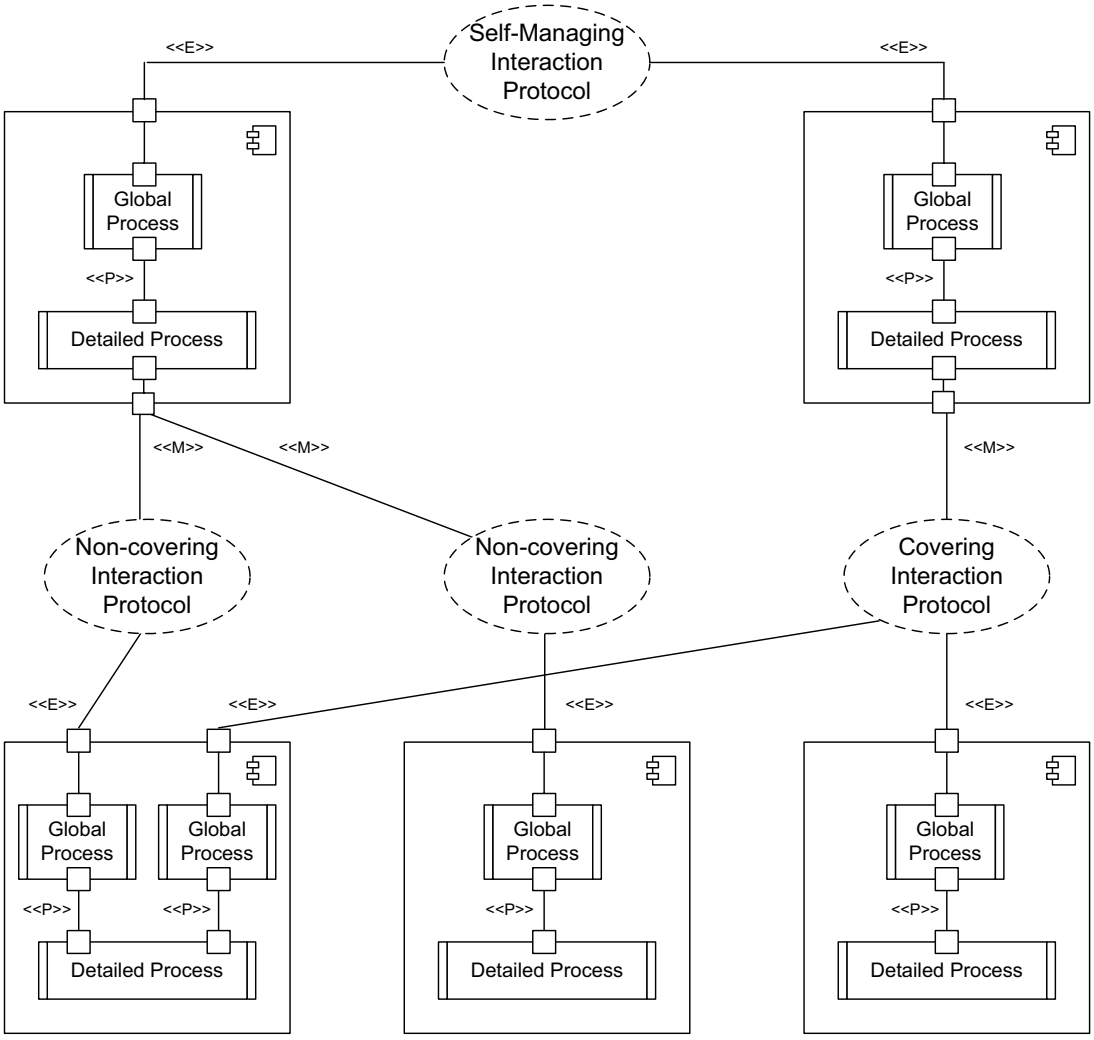


Figure 4.1: General Structure of a PARADIGM Model with Interaction Protocols

with *empty manager parts*, which leads to the definition of *self-managing* interaction protocols. Finally, we introduce our *operational* definition for PARADIGM models with interaction protocols.

Covering Interaction Protocols

The most straightforward type of interaction protocol is *covering*. Such a protocol covers the entire interaction between the manager processes and the global processes of employees that play a role in it. Hence, in order to understand and analyze the interaction between these processes, it is sufficient

to analyze the consistency rules of this single interaction protocol. Consider for example the covering interaction protocols shown in Figure 4.2. Example A illustrates the situation in which m manager processes and n global processes of employees are involved in a single interaction protocol. In fact, if m and n correspond to the entire amount of manager and global processes in a PARADIGM model, this situation corresponds to a model with a single interaction protocol, like we depicted in the component diagrams of Chapter 2. In example B, several global processes are combined with a single manager process. It represents the situation in which the set of consistency rules is split up along the manager processes in a model. Consider for example a model which contains m manager processes, each managing the interaction for a *distinct* set of global processes. In that case, the sets of consistency rules applied by each manager process can be anchored to an individual interaction protocol, which entirely covers the interaction between a manager process and its global processes. Finally, example C illustrates the situation in which the consistency rules of two manager processes are combined, but these consistency rules define the interaction with a *single* global process. An example of such a situation can be found in [42], in which one global process of a client is managed by both a broker manager process and a server manager process. In such cases, a *combination* of the appropriate consistency rules from both manager processes is needed in order to understand and analyze the interaction with the client. This combination can be anchored to a single interaction protocol.

Definition 4.1 *An interaction protocol I is* covering *if the following properties hold:*

- *I has at least one consistency rule anchored to it.*
- *If I is the anchorage for a consistency rule with a transition of a manager process P as its manager part, then I is the anchorage for* all *consistency rules which have a transition of manager process P as its manager part.*
- *If I is the anchorage for a consistency rule with a transition of a global process $P(\pi)$ in its employee part, then I is the anchorage for* all *consistency rules which have a transition of global process $P(\pi)$ in their employee part.*

These two properties constitute the "coverage" of the interaction protocols with regard to the processes related to them. Thus, if the consistency rules of a PARADIGM model are split into covering interaction protocols, the interaction with each manager process or global process can be understood through a *single* protocol.

Conflicting Consistency Rules

Before we continue with non-covering interaction protocols, we first establish a more precise notion of *conflict* for consistency rules. In the remainder of this chapter, we will use this notion for determining whether two consistency rules must be anchored to a single interaction protocol. As a consequence, the notion is instrumental in determining a maximal division of consistency rules over interaction protocols.

Definition 4.2 *Two consistency rules R and R' in a* PARADIGM *model M are* conflicting on global process $P(\pi)$ *if:*

- *The employee part of both R and R' contains a global transition of $P(\pi)$;*
- *There exists an execution sequence of M in which, at some point in this sequence, R and R' are applied simultaneously.*

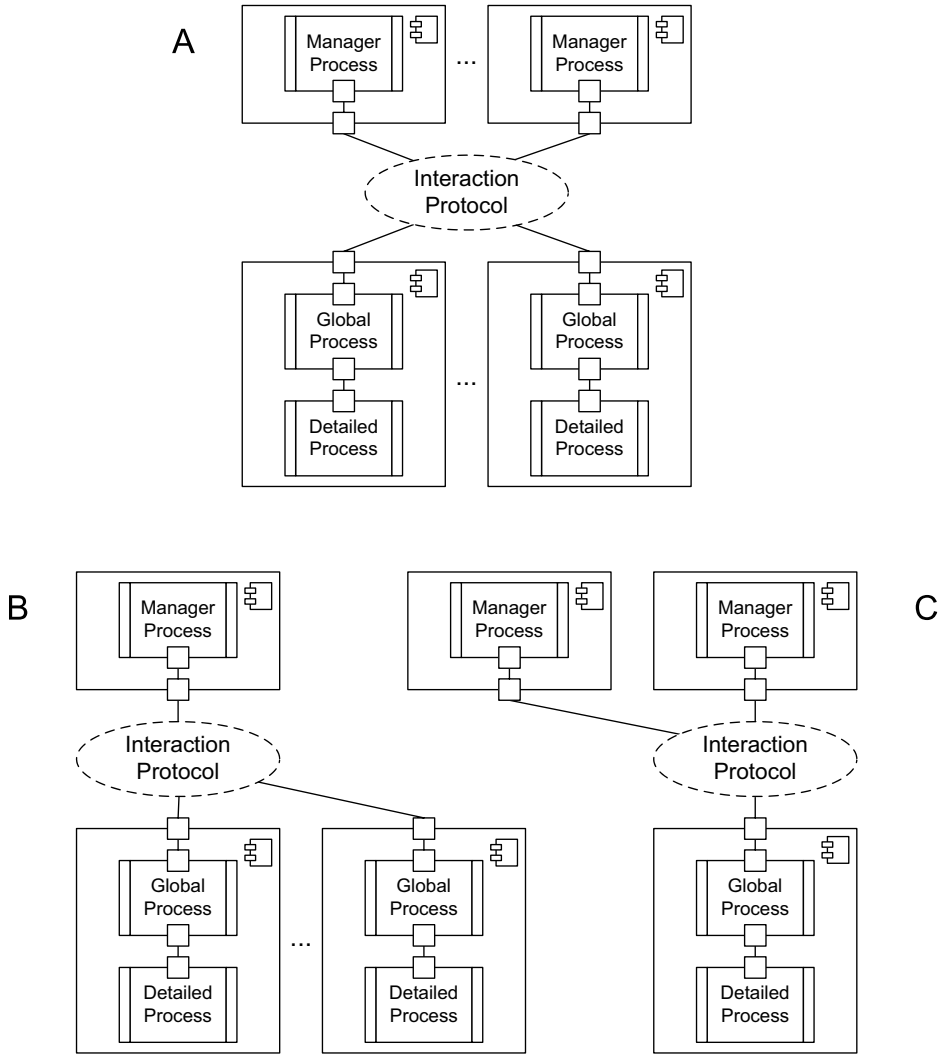


Figure 4.2: Covering Interaction Protocols

The second condition implies that these consistency rules can be *enabled* at the same time, which implies that the transitions in both rules from the same global process at least have the same source state. In general, in order to determine whether the second condition holds, a global analysis of the execution sequences of the PARADIGM model is required, which is not always easy to be carried out in practice. We therefore adopt a more practical definition for *potentially conflicting* consistency rules, as follows.

Definition 4.3 *Two consistency rules R and R' in a* PARADIGM *model are* potentially conflicting on state S of global process $P(\pi)$ *if the employee part of both R and R' contains a transition from $P(\pi)$ with the same source state S.*

As the name says, if two consistency rules are potentially conflicting, this merely indicates that they may be conflicting, i.e. a global analysis of the model may reveal that there is a reachable state in which both rules can be applied simultaneously. A practical benefit of the definition of potential conflict is that its condition can be checked by only considering information within the consistency rules themselves. Thereby, modelers are able to easily prevent conflicts by anchoring potentially conflicting consistency rules to a single interaction protocol.

Non-covering Interaction Protocols

Modeling cases exist in which a single manager process manages several separate interaction aspects. Examples of such cases can be found in [42, 43, 92] and in Chapter 2 of this thesis: a manager process *Server* manages multiple unrelated employee processes *Client(i)* ($1 \leq i \leq 3$). These cases would benefit from a *division* of the consistency rules for a single manager process into multiple separate subsets. For global processes, the same situation may occur. Take for example situation C in Figure 4.2. It could well be the case that the individual manager processes coordinate separate parts of the global process. However, the use of covering interaction protocols does not allow the split-up of their consistency rules into two separate interaction protocols. In view of these two situations, we introduce *non-covering* interaction protocols: the interaction of a single manager or global process can be split-up into separate parts, managed via separate interaction protocols. Three examples of non-covering interaction protocols are depicted in Figure 4.3. In example D, the consistency rules of a single manager process are contained within two separate interaction protocols, while example E shows a single global process which is managed via two separate protocols. Finally, situation F depicts two separate interaction protocols for a single manager process and a single global process. Non-covering interaction protocols enable us to distinguish between different *aspects* in the interaction with a single process. Naturally, non-covering interaction protocols can be defined as interaction protocols which do not have the properties of covering interaction protocols.

Definition 4.4 *An interaction protocol is* non-covering *if it is not covering.*

However, since each interaction protocol in a model applies its consistency rules concurrently with other interaction protocols, we must be careful not to put two conflicting consistency rules in different non-covering interaction protocols. Therefore, we define the notion of a *sound interaction protocol*. Based on this notion, we can define *minimal interaction protocols*, called this way because we do not allow to reduce the set of consistency rules anchored to it.

Definition 4.5 *An interaction protocol I is* sound *if the following properties hold:*

- *I has at least one consistency rule anchored to it.*
- *If a consistency rule R is anchored to I with a transition $S \xrightarrow{\theta} S'$ of a global process $P(\pi)$ in its employee part, then I is the anchorage for all consistency rules which are potentially conflicting with R on state S of global process $P(\pi)$.*

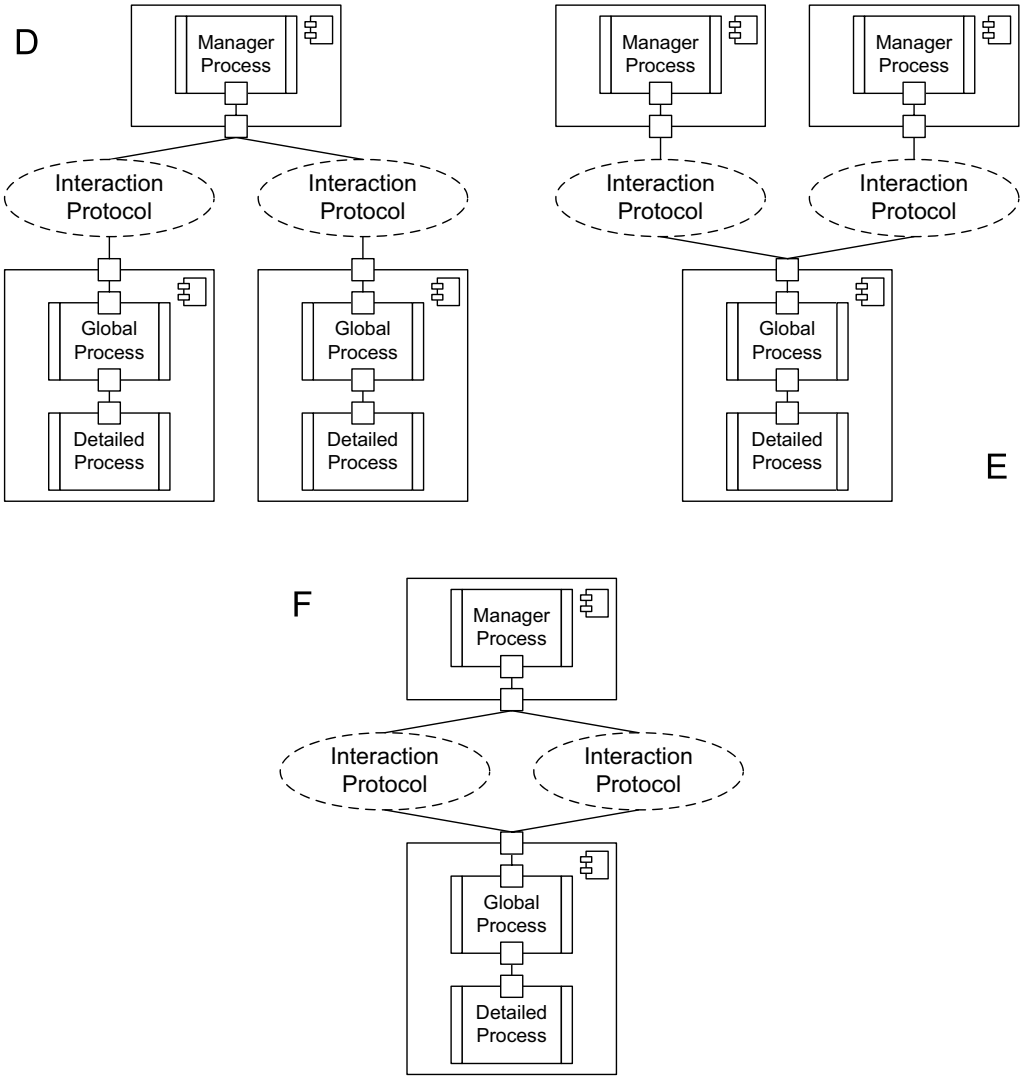


Figure 4.3: Non-covering Interaction Protocols

The first property ensures non-emptiness of the interaction protocol, the second property ensures that if two consistency rules are conflicting, they are anchored to a single interaction protocol. It can be easily argued that covering interaction protocols are always sound, since a covering interaction protocol contains *all* consistency rules for each of the processes that play a role in it. For minimal interaction protocols, we include the soundness property in their definition, as follows.

Definition 4.6 *An interaction protocol I is* minimal *if the following properties hold:*

- *I is sound.*
- *For each non-empty proper subset of consistency rules anchored to I, it holds that if this subset is removed from I, then I is not sound.*

The second property ensures the minimality of the interaction protocol: if the set of consistency rules anchored to it cannot be partitioned into two non-empty subsets from which two sound interaction protocols can be created, then the interaction protocol is minimal. Our definition of soundness and minimality imply that, from the point of view of a global process, the choice which transition from a certain state must be taken always depends on a single interaction protocol. Take for example the global process in Figure 4.4. The rectangles drawn on top of the process divide its transitions into five sets. If an interaction protocol contains a consistency rule in which one of the global transitions in one of these sets is mentioned in its employee part, then this protocol should contain all consistency rules for all transitions in that set. Hence, the global process in Figure 4.4 is managed by at most five different interaction protocols, dividing the set of consistency rules for transitions $\{a, b, c, d, e, f, g\}$ into five subsets of consistency rules for transitions $\{b\}$, $\{c, a\}$, $\{d, e\}$, $\{f\}$ and $\{g\}$.

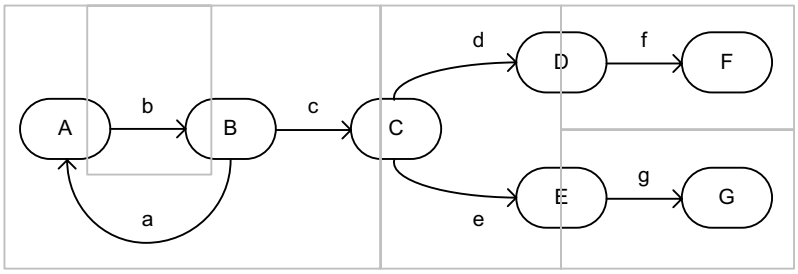


Figure 4.4: Maximal distribution of the coordination of global process transitions

Consistency Rules with Empty Manager Parts

The idea to allow *empty manager parts* in consistency rules is motivated by the following observation. The *manager/employee* principle in PARADIGM opposes two pre-defined *roles* on components in a model: the *manager* and the *employee* role. The choice how to associate these pre-defined roles with the entities in a PARADIGM model is not always obvious, sometimes even inconvenient. Examples of cases relevant in this respect are the producer/consumer problem [45] and many situations in which the components can be more or less regarded as peers.

Consider as an example the client/server model presented in the previous two chapters. We modeled the server as a manager process and the clients as employee processes. Instead, we could have decided to model the clients as manager processes, each managing one partition of a single server employee process. In either case, the division between managers and employees in the model serves only as a vehicle to model the interaction, it is not an obvious part of the coordination problem. In fact, there is a third option, in which the clients and the server are all regarded as "employees", while an *additional* manager process coordinates the mutual influence between the clients and the server. Thereby,

we "extract" the manager/employee hierarchy from the clients and server and regard them as equals, as *peers*, coordinated by an external manager process. However, this additional manager process is in fact redundant, since we originally managed to perform the entire coordination *without* it. It is only there because of the requirement in the PARADIGM language that for each consistency rule, a manager part containing precisely one manager transition exists. It potentially introduces *superfluous constraints* on the order in which consistency rules may be applied – the sequential behaviors of the employee processes pose enough constraints themselves. The only reason for having an additional manager process in this case, is that some *trigger* is needed for the application of consistency rules – a trigger which could be perfectly provided by the interaction protocol to which the consistency rules are anchored.

In view of the above, we allow consistency rules to have empty manager parts. Such consistency rules act as synchronization constraints upon a set of global processes and can be applied by interaction protocols *themselves*, one at a time. We generalize the syntax of consistency rules in PARADIGM, allowing them to have three different shapes.

1. A single manager transition and one or more employee transitions:

$$P : s \xrightarrow{a} s' \quad * \quad P_k(\pi_{k,l}) : S_{k,l} \xrightarrow{\theta_{k,l}} S'_{k,l}, ..., P_v(\pi_{v,w}) : S_{v,w} \xrightarrow{\theta_{v,w}} S'_{v,w}$$

2. A single detailed transition only:

$$P : s \xrightarrow{a} s'$$

3. One or more employee transitions but no manager transition:

$$* \quad P_k(\pi_{k,l}) : S_{k,l} \xrightarrow{\theta_{k,l}} S'_{k,l}, ..., P_v(\pi_{v,w}) : S_{v,w} \xrightarrow{\theta_{v,w}} S'_{v,w}$$

Regarding the third shape, such a consistency rule can be applied whenever all global transitions specified in it are enabled. The interaction protocol to which the rule is anchored takes care of the actual application of the consistency rules, one at a time. If multiple consistency rules are enabled at the same time, selection occurs in a non-deterministic order.

Self-managing Interaction Protocols

Having motivated and defined consistency rules with empty manager parts, we show two examples of *self-managing* interaction protocols in Figure 4.5. We call them self-managing interaction protocols because *only* consistency rules with empty manager parts are anchored to them. Therefore, they are not connected to any manager process. They facilitate the modeling of interaction between processes for which a manager/employee hierarchy is not obvious or not desired, and for which no additional sequentialization of the application of consistency rules is needed. Self-managing interaction protocols can be either *covering* (example G) or *non-covering* (example H).

Definition 4.7 *An interaction protocol I is* self-managing *if all consistency rules anchored to I have an empty manager part.*

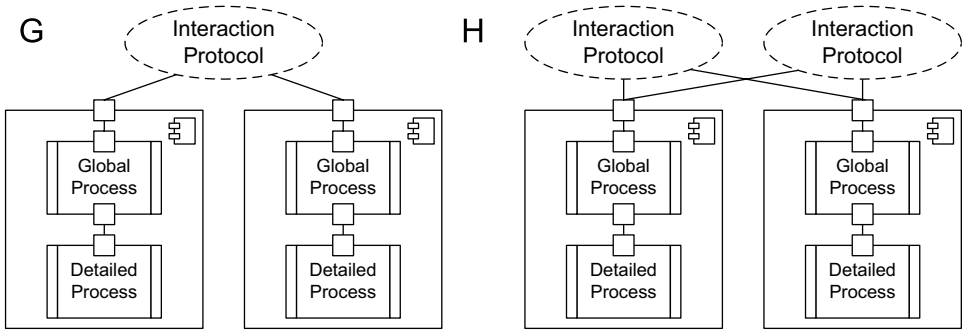


Figure 4.5: Self-managing Interaction Protocols

The Execution of Interaction Protocols

Finally, we define our general notion of interaction protocols in PARADIGM, both structurally and operationally. An *interaction protocol* in PARADIGM is a behavioral entity to which one or more consistency rules are anchored. Each interaction protocol defined in a PARADIGM model must be sound. Each consistency rule of each manager process must be anchored to one of the interaction protocols. Each consistency rule with an empty manager part must be anchored to one of the interaction protocols. The execution of an interaction protocol consists of the taking of *steps* in a sequential manner. Each step is bound to a consistency rule anchored to the interaction protocol, which must be enabled in order to take the step. We consider two types of steps: *externally managed* and *internally managed*.

- An externally managed step of an interaction protocol can only be taken in synchronization with the taking of a transition in a manager process, and only if that manager process has chosen to apply the enabled consistency rule bound to this step. If the step is taken in synchronization with the taking of the transition in the manager process, then its consistency rule is applied.
- An internally managed step of an interaction protocol can be taken whenever the consistency rule with an empty manager part anchored to that interaction protocol and bound to this step is enabled. If the step is taken, then its consistency rule is applied.

The behavior of an interaction protocol can be visualized as a *process*, as shown in Figure 4.6. Such a process consists of one state and transitions from that state to itself. The process contains one transition for each consistency rule anchored to the interaction protocol.

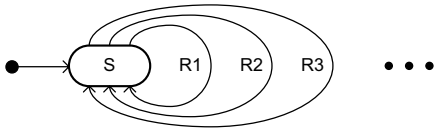


Figure 4.6: Process-like Specification of an Interaction Protocol

The execution of a PARADIGM model extended with interaction protocols is equal to the concurrent execution of all its processes and interaction protocols, with the following constraints: A transition of a global process can only be taken as the result of the application of a consistency rule. A transition of a manager process can only be taken in synchronization with an externally managed step of an interaction protocol. By this, the manager process is said to have *chosen* to apply the enabled consistency rule bound to this step. A transition of a pure employee process can always be taken directly.

In the next three sections, we illustrate the usage of covering, non-covering and self-managing interaction protocols by showing three different extensions to the client/server example.

4.4 Covering Interaction Protocols

Our starting point for an example with covering interaction protocols is the client/server model of Chapter 2, which consists of one server and three clients being served. The extension consists of allowing the clients to check whether they are satisfied with the service provided to them, and if not, to reject the result and re-request the service. In order to prevent the server from having to redo the service indefinitely, an extra process *Timer* is used to keep track of the time and, if needed, to enforce clients to accept the result and leave. Hence, two different aspects can be recognized in the interaction: the aspect of providing the service, and the aspect of enforcing the timing constraints. We separate these two aspects by modeling two interaction protocols: a *server protocol* and a *timer protocol*.

The architecture of the client/server/timer example is depicted in Figure 4.7. The two interaction protocols *server protocol* and *timer protocol* are both *covering* protocols: each manager and each global process is connected to a single interaction protocol. The *Client* processes are now coordinated via *two* partitions and corresponding global processes: *Client[AsOOS]*, which represents the view of a client as an *object of service*, and *Client[AsTE]*, which views the client as a *timed entity*. The model contains two manager processes *Server* and *Timer* for the coordination of serving and timing, respectively.

We take a closer look at the individual parts of the example, starting with process *Client*. We extend this process with the ability to reject the result of the service, as shown in Figure 4.8. After the service has been provided, the client checks whether he or she is satisfied with the result. If so, the process continues normally, but if the result is rejected, the server is urged to provide the service once more, hopefully better.

Because we applied changes to global process *Client*, we redefine partition *AsOOS* as shown in Figure 4.9. This partition, like in the original example, is meant for coordinating the service provision. Global process *Client[AsOOS]* remains the same and consists of three states *WithoutService*, *Orienting* and *UnderService*. The timing issues are coordinated via a separate partition *AsTimedEntity* or *AsTE* for short. This partition and the corresponding global process *Client[AsTE]* are shown in Figures 4.10 and 4.11. The partition consists of three subprocesses. In subprocess *Waiting*, the client is not being served and no time constraints have to be applied. In subprocess *Running*, the service is being provided and the client is allowed to re-request the service, while in subprocess *ForcedToStop*, the client is no longer allowed to reject the result. As defined in global process *Client[AsTE]*, the client is of course allowed to accept the result in time, hence the same trap *finished* in both subprocesses *Running* and *ForcedToStop* of partition *AsTE*.

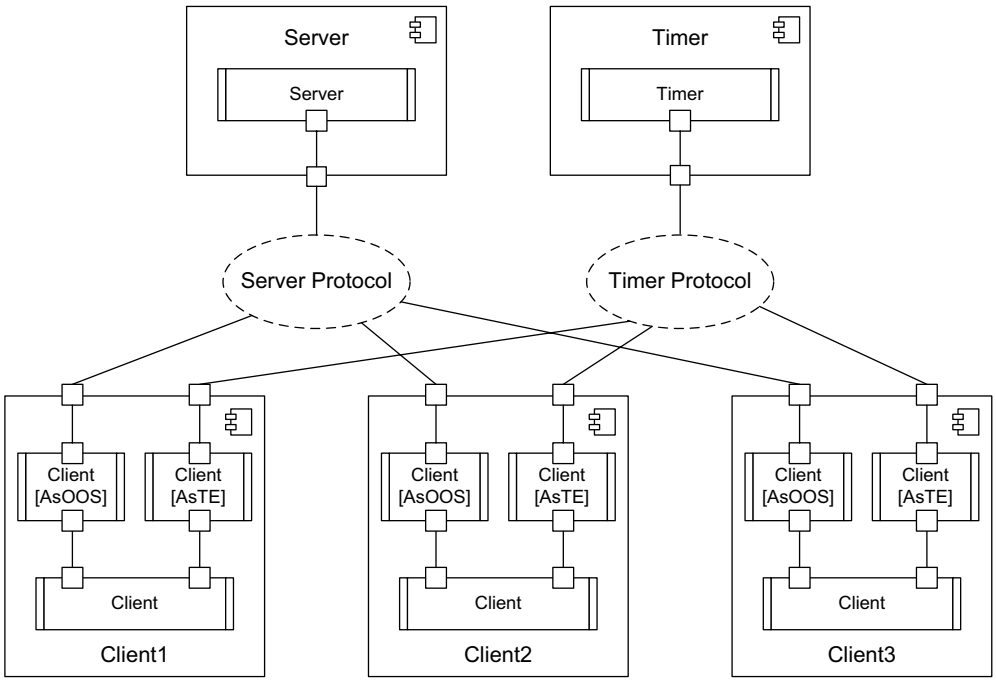


Figure 4.7: Overview of the client/server/timer example

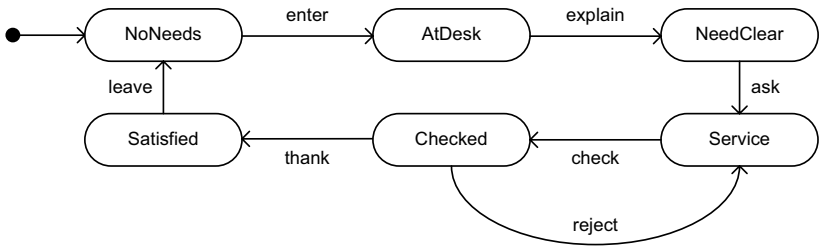


Figure 4.8: Extended process *Client*

The two global processes *Client[AsOOS]* and *Client[AsTE]* are managed by separate manager processes. For the management of global process *Client[AsOOS]*, we reuse the existing process *Server* without change, while for the management of partition *AsTE*, we introduce a new manager process *Timer*, as shown in Figure 4.12. This process coordinates the timing constraint for each of the three clients, one at a time. Once a timeout occurs, it changes the subprocess from *Running* to *ForcedToStop*. This is always possible, since the connecting trap *trivial* contains all states of both subprocesses.

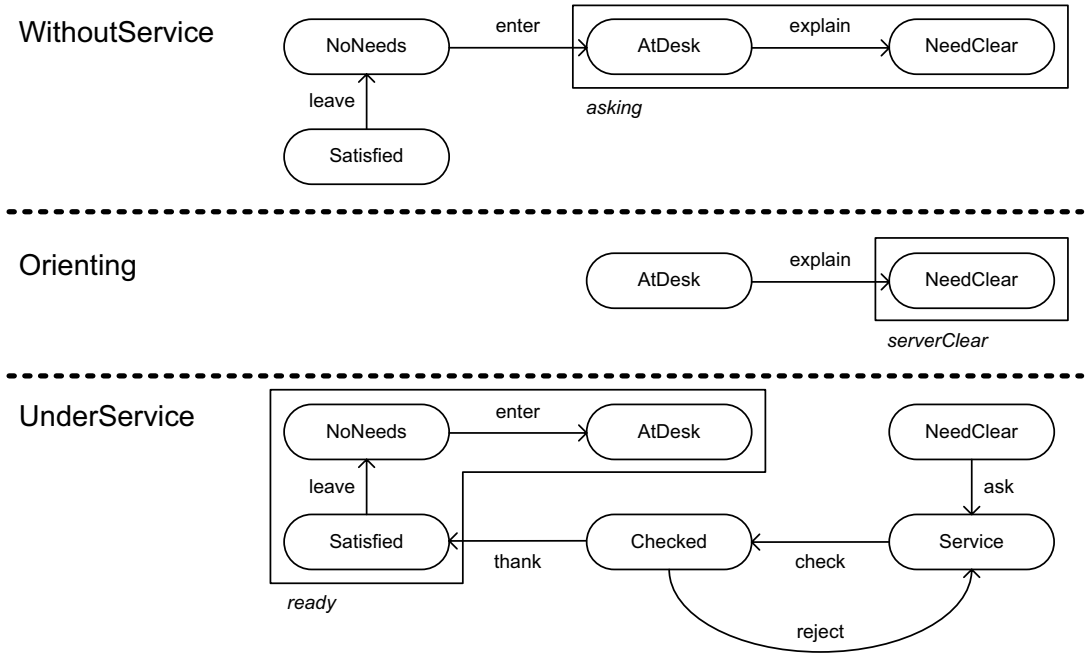


Figure 4.9: Revised partition *AsOOS* for extended process *Client*

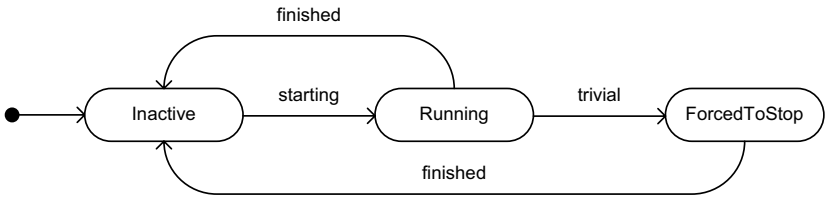


Figure 4.10: Process *Client[AsTE]*

In line with Figure 4.7, the consistency rules for the client/server/timer example are split up into two subsets, one for each interaction protocol, shown in Tables 4.13 and 4.14. We do not show the consistency rules for pure employee processes *Client(i)*. The reader is already familiar with rules Svr-R1 to Svr-R9 from the server interaction protocol: they are exactly the same as rules R1 to R9 in the example in Chapter 2. Note that this reuse is possible because of the *multiple views* principle of PARADIGM: we extended process *Client* without the need to change global process *Client[AsOOS]*. The timer interaction protocol, on the other hand, is completely new: it relates manager process *Timer* to global processes *Client(i)[AsTE]* ($i = 1, 2, 3$).

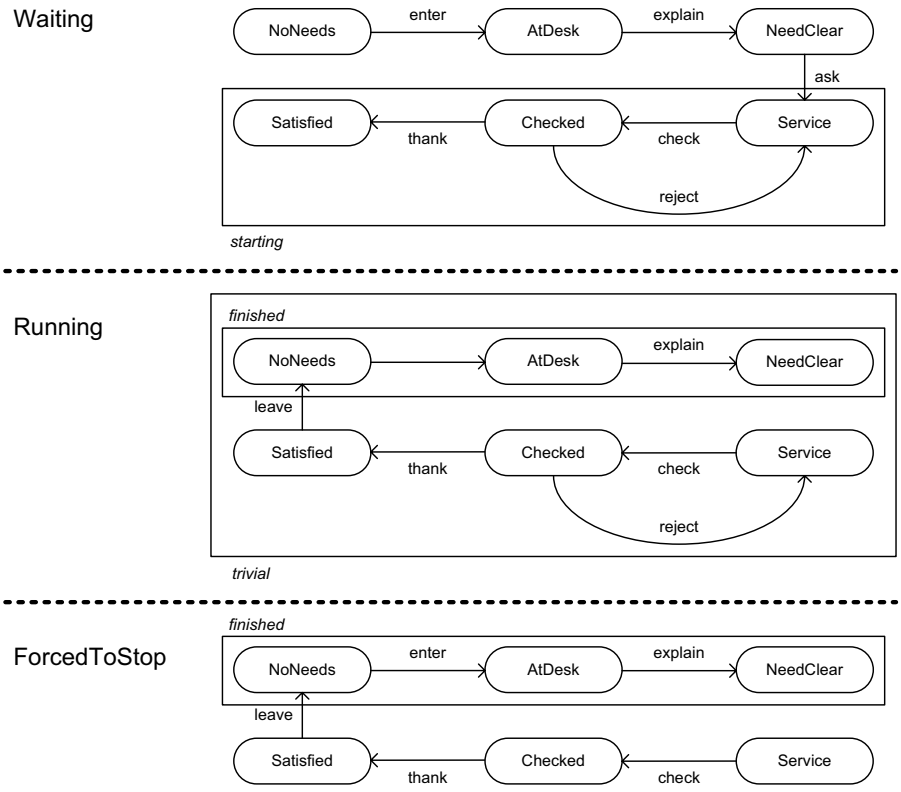


Figure 4.11: New partition *AsTE* for extended process *Client*

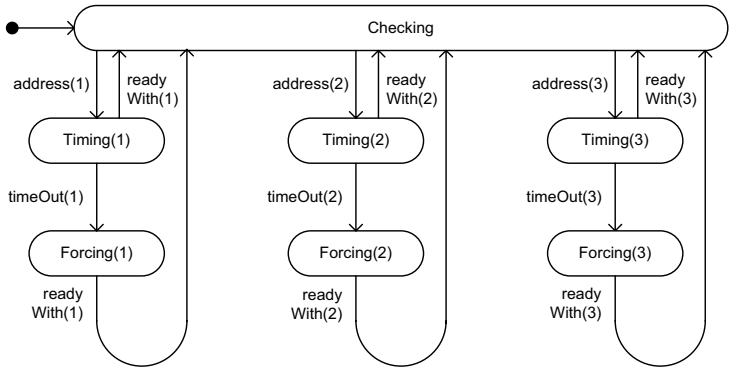


Figure 4.12: Process *Timer*

(Svr-R1,2,3)		Server:	Checking	address(i) →		ListeningTo(i)
	∗	Client(i)[AsOOS]:	WithoutService	asking →		Orienting
(Svr-R4,5,6)		Server:	ListeningTo(i)	startServing(i) →		Serving(i)
	∗	Client(i)[AsOOS]:	Orienting	serverClear →		UnderService
(Svr-R7,8,9)		Server:	Serving(i)	leave(i) →		Checking
	∗	Client(i)[AsOOS]:	UnderService	ready →		WithoutService

Table 4.13: Server Interaction Protocol

(Tmr-R1,2,3)		Timer:	Checking	address(i) →		Timing(i)
	∗	Client(i)[AsTE]:	Waiting	starting →		Running
(Tmr-R4,5,6)		Timer:	Timing(i)	timeOut(i) →		Forcing(i)
	∗	Client(i)[AsTE]:	Running	trivial →		ForcedToStop
(Tmr-R7,8,9)		Timer:	Timing(i)	readyWith(i) →		Checking
	∗	Client(i)[AsTE]:	Running	finished →		Waiting
(Tmr-R10,11,12)		Timer:	Forcing(i)	readyWith(i) →		Checking
	∗	Client(i)[AsTE]:	ForcedToStop	finished →		Waiting

Table 4.14: Timer Interaction Protocol

Note that the serving and timing interactions remain unrelated: there is no explicit synchronization relation between the two interaction protocols. Only the detailed *Client* processes with their two partitions *AsOOS* and *AsTE* constitute the indirect relation between them. The interaction protocols themselves clearly reflect the separation of concerns in the model.

4.5 Non-covering Interaction Protocols

The previous example showed a separation of the set of consistency rules into covering interaction protocols. However, individual manager processes or even global processes could be involved in multiple different aspects of interaction. In such cases, non-covering interaction protocols can be used to separate the consistency rules for each of the aspects from each other. The model in Figure 4.15 is an example of the use of non-covering interaction protocols. Our starting point is the example of Section 4.4. In this example, we combine the *Server* and *Timer* process into a single *TimedServer* process which performs serving and timing. We also combine the two partitions *AsOOS* and *AsTE* into a single partition *AsTOOS*, which stands for *TimedObjectOfService*. We separate the consistency rules for two parts of the service provision into two non-covering interaction protocols *Orientation Protocol* and *Servicing Protocol*.

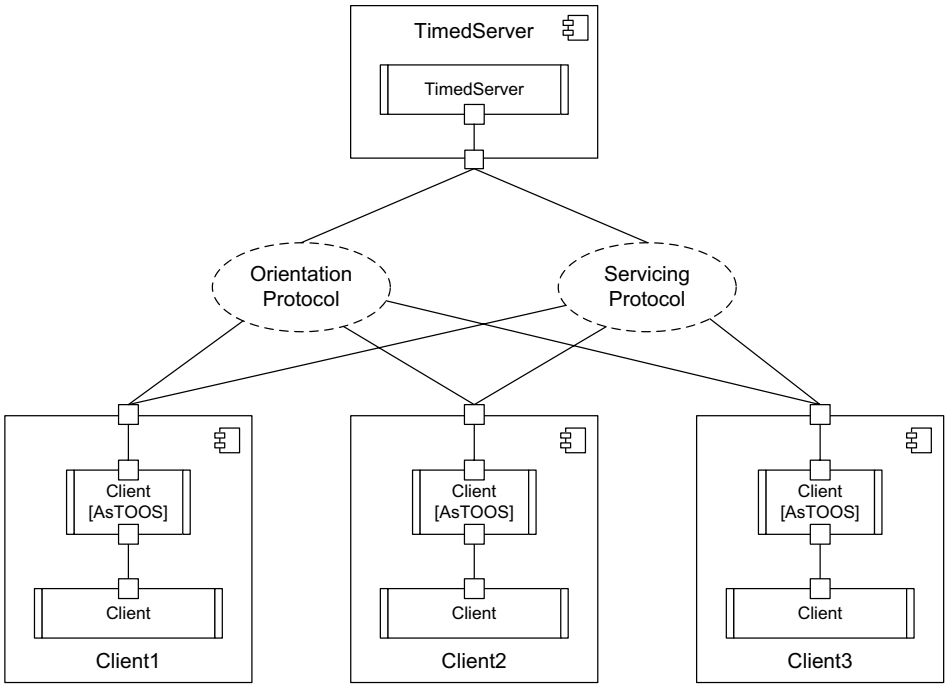


Figure 4.15: Overview of the client/timed-server example

Partition *AsTOOS* is shown in Figure 4.16. The partition consists of four subprocesses: *WithoutService* and *Orienting*, which are equal to the subprocesses of the *AsOOS* partition shown earlier, *BeingServed*, in which the client is being served and timed and *ForcedToStop*, representing the situation in which the client must accept the result of the service. Hence, the six subprocesses of the two partitions *AsOOS* and *AsTE* have now been reduced to a single partition with four subprocesses. We removed subprocess *Waiting* of partition *AsTE*, because the behavior within this subprocess is covered by the new subprocesses *WithoutService* and *Orienting*. Subprocesses *Running* of partition *AsTE* and *UnderService* of partition *AsOOS* are both combined within the new subprocess *BeingServed*.

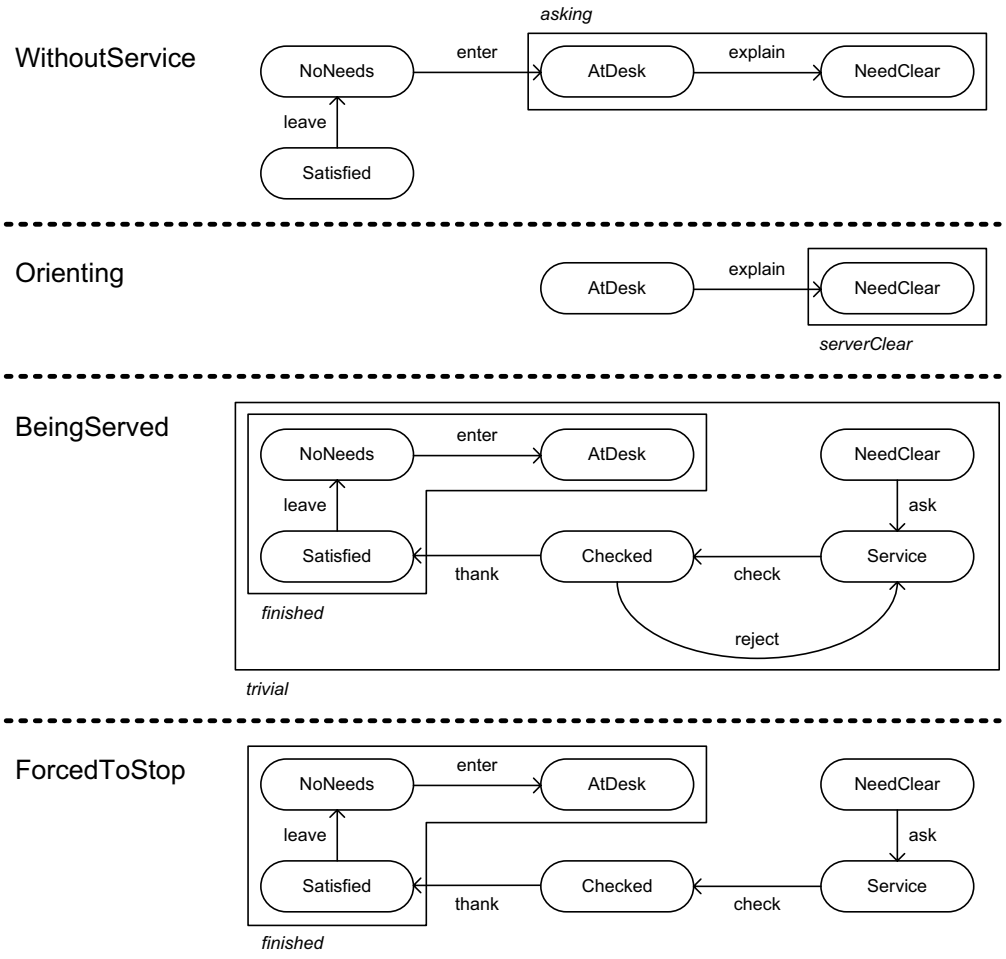


Figure 4.16: Partition *AsTOOS* for process *Client*

Global process *Client[AsTOOS]*, specifying the behavior of a *Client* process viewed as a timed object of service, is shown in Figure 4.17. If we compare this global process with the global process *OOS*, the actual difference only consists of the extra state *ForcedToStop* and transitions to and from it – the extra time-out situation which the process must obey. The combined manager process *TimedServer* is shown in Figure 4.18. A comparison with manager process *Server* reveals a difference consisting of the extra state *Forcing* plus its incoming and outgoing transitions only.

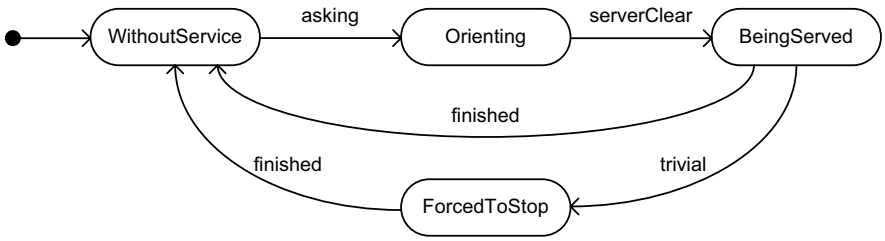


Figure 4.17: Global process *Client[AsTOOS]*

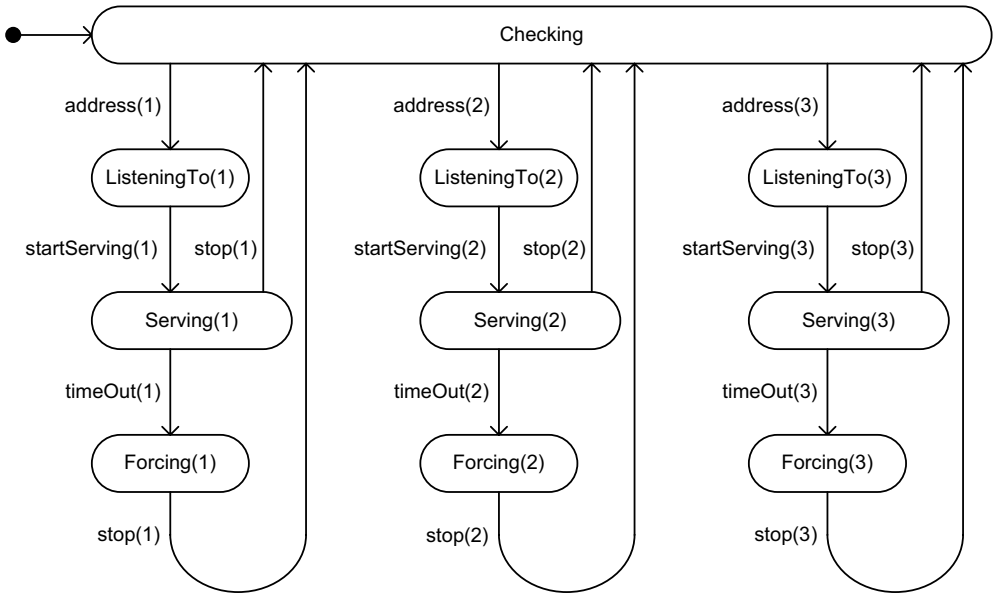


Figure 4.18: Combined process *TimedServer*

By using non-covering interaction protocols, we separate the anchorage of the consistency rules for the example into two protocols *Orientation Protocol* and *Servicing Protocol*, which are shown in Tables 4.19 and 4.20. The interaction protocols differ considerably from the interaction protocols in the previous example: the entire sequence of interactions is split into two separate non-overlapping parts. The *Orientation* protocol coordinates the part in which a client is orienting and the start-up of the service provision, while the *Servicing* protocol coordinates the actual service delivery and its finishing, enforcing the client to accept the result if a time-out occurs. These kind of splits are also useful in cases of e.g. *delegation*, where the parts are coordinated by different entities (manager processes) in a system.

(Svr-R1,2,3)		TimedServer:	Checking	address(i) →	ListeningTo(i)
	∗	Client(i)[AsTOOS]:	WithoutService	asking →	Orienting
(Svr-R4,5,6)		TimedServer:	ListeningTo(i)	startServing(i) →	Serving(i)
	∗	Client(i)[AsTOOS]:	Orienting	serverClear →	BeingServed

Table 4.19: Orientation Protocol

(Tmr-R1,2,3)		TimedServer:	Serving(i)	timeOut(i) →	Forcing(i)
	∗	Client(i)[AsTOOS]:	BeingServed	trivial →	ForcedToStop
(Tmr-R4,5,6)		TimedServer:	Serving(i)	stop(i) →	Checking
	∗	Client(i)[AsTOOS]:	BeingServed	finished →	WithoutService
(Tmr-R7,8,9)		TimedServer:	Forcing(i)	stop(i) →	Checking
	∗	Client(i)[AsTOOS]:	ForcedToStop	finished →	WithoutService

Table 4.20: Servicing Protocol

The interaction between the *TimedServer* manager process and the *Client(i)[AsTOOS]* global processes can only be fully understood by looking at the two interaction protocols together. If we combine them, this yields a *covering* interaction protocol. Both non-covering interaction protocols are *sound*: for each state in each global process, there is exactly one interaction protocol which contains all the consistency rules in which outgoing transitions of this state are mentioned. In this example, it is also clear that the potentially conflicting consistency rules are actually not conflicting: their specific relation to the single manager process inherently prevents them from being applied in parallel.

4.6 Self-managing Interaction Protocols

As a final illustration of our flavours of interaction protocols, we show the use of self-managing interaction protocols. These type of interaction protocols manage the application of consistency rules autonomously, without the use of an explicit manager process. Typically, self-managing protocols can be used in settings in which components cannot clearly be identified as managers or employees, but rather as *peers*. In those cases, additional sequentialization of the application of consistency rules through the use of a manager process is often not required.

The self-managing interaction protocol typically has only consistency rules anchored to it which do not have a manager part. It applies a consistency rule once all transitions in the employee part of that rule can be taken. If more than one consistency rule is enabled at the same time, a non-deterministic choice is made between the enabled consistency rules.

Our starting point for our illustration of the use of self-managing interaction protocols is the client-/timed-server example from Section 4.5. Instead of separating the consistency rules along the process of service delivery, we now separate them based on the individual clients in the model. The new version is depicted in Figure 4.21. In this version, process *TimedServer* has three partitions *AsEmpFor(i)* and corresponding global processes *TimedServer[AsEmpFor(i)* $(1 \leq i \leq 3)$. Each combination of two processes *TimedServer[AsEmpFor(i)]* and *Client(i) [AsTOOS]* is managed by a covering self-managing interaction protocol *Client(i)/TimedServer*. Hence, the example contains no manager processes at all: the application of the consistency rules is managed by the interaction protocols only, based on enabled transitions of global processes of their employees.

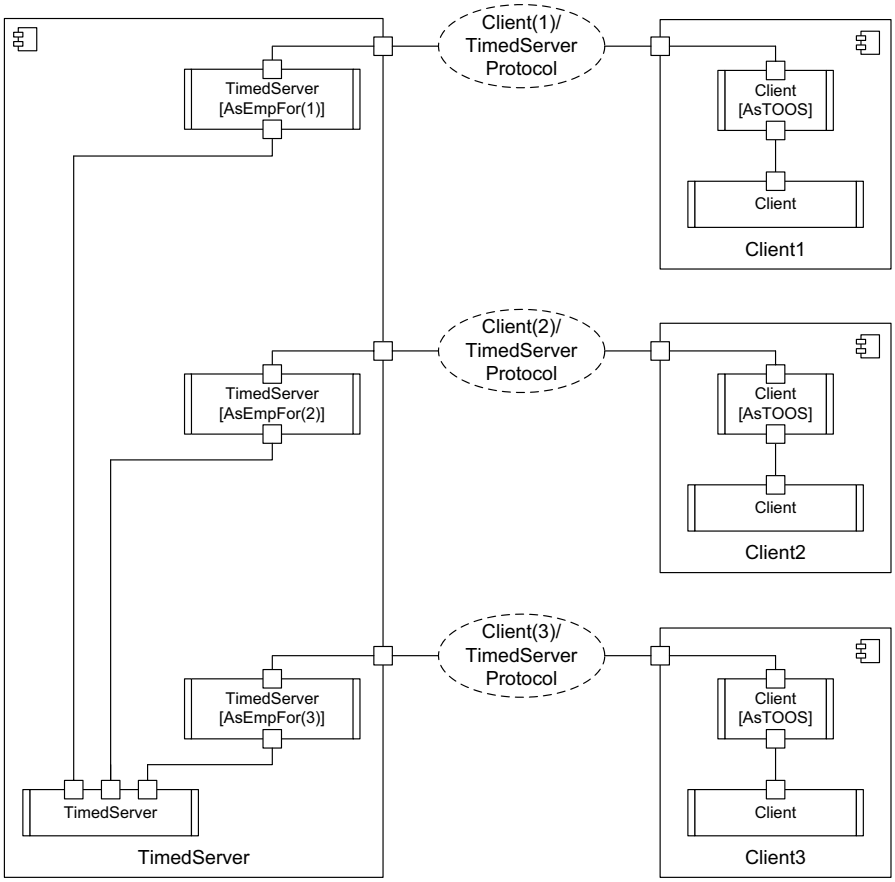


Figure 4.21: The client/timed-server example with self-managing interaction protocols

Intuitively, the *TimedServer* process and the *Client* processes are now modeled as *peers*. Their interaction is modeled in terms of synchronizations based on views on their behavior *only*. Moreover, the example shows how the consistency rules for the interaction with individual clients can be separated from each other and anchored to individual interaction protocols. Next to the addition of partition *AsEmpFor(i)* and global process *TimedServer[AsEmpFor(i)]*, which we explain in the next paragraphs, the example involves a small extension to global process *Client(i)[AsTOOS]*, which we show thereafter.

The new partition *AsEmpFor(i)* of process *TimedServer* is shown in Figure 4.22. The partition divides the process into three subprocesses: *Idle*, in which the process stops serving and/or does not serve *Client(i)*, *Attentive*, in which the process could possibly start listening to *Client(i)*, and *Serving*, in which the process is actually serving *Client(i)*. In the figure, $1 \leq i \leq 3$. Indices > 3 must be interpreted modulo 3.

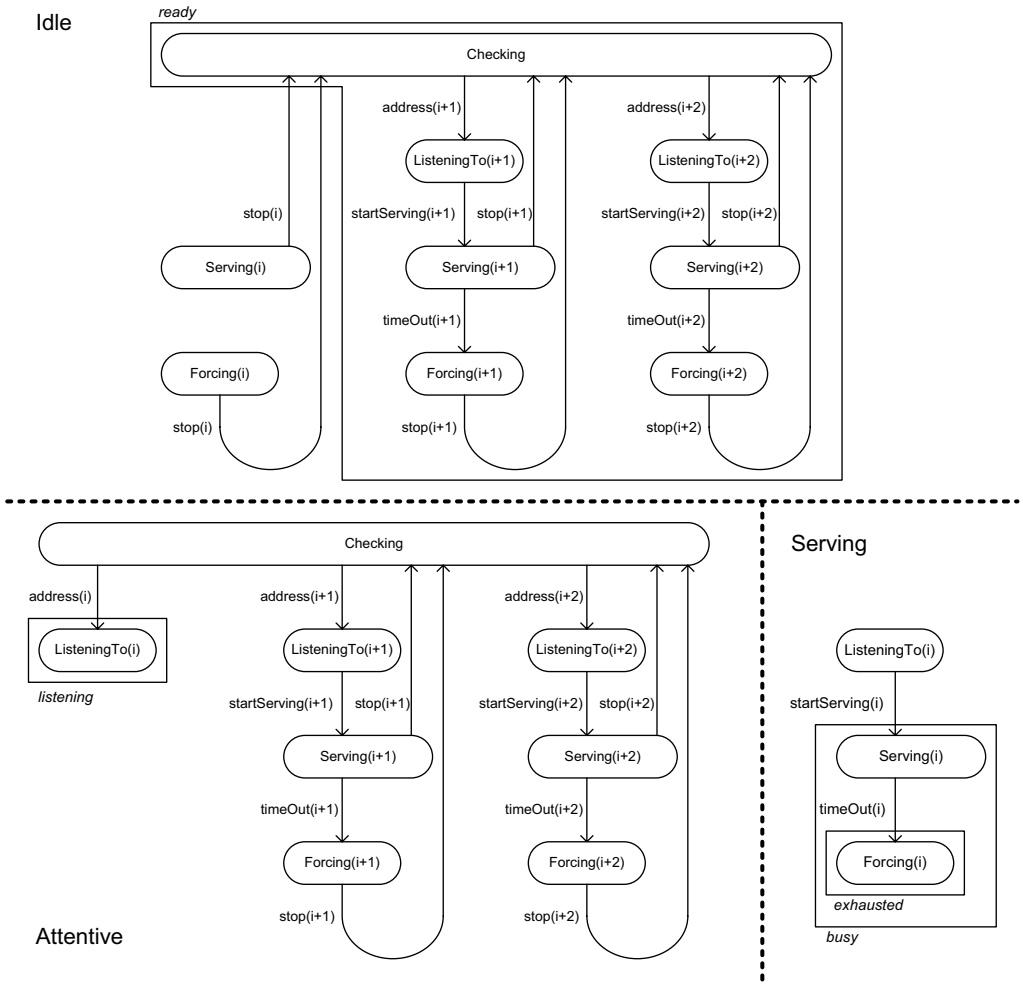


Figure 4.22: Partition *AsEmpFor(i)* for process *TimedServer*

Each of the partitions *AsEmpFor(i)* views process *TimedServer* from the perspective of the interaction with a *single Client(i)*. Process *TimedServer* actually selects which of the asking clients to serve – this is the reason why the remaining transitions of process *TimedServer* are part of subprocesses *Idle* and *Attentive*. The interaction following this selection is managed by one of the three interaction protocols, depending on the trap that process *TimedServer* enters. In subprocess *Serving*, two *nested* traps *busy* and *exhausted* are used. Intuitively, process *TimedServer* keeps track of the time while performing the service. As soon as it becomes exhausted, i.e. a time-out occurs, it takes transition *timeOut* and enters nested trap *exhausted*. The interaction protocol then acts accordingly by applying the consistency rule which forces the client to accept the result and finish.

In Figures 4.23 and 4.24, the global processes *TimedServer[AsEmpFor(i)]* and *Client(i) [AsTOOS]* are shown. These are the actual processes which interact by means of the consistency rules anchored to the interaction protocols. In particular, we draw the attention to the loops in the processes. From the perspective of the interaction protocols, these loops enable the *inspection* of a trap entrance without performing a subprocess change. Their presence can be clarified by taking a look at the revised *TimedServer* interaction protocols, which are shown in Table 4.25. Each of the three protocols *Client(i)/TimedServer* contains six consistency rules $R1(i)$ to $R6(i)$.

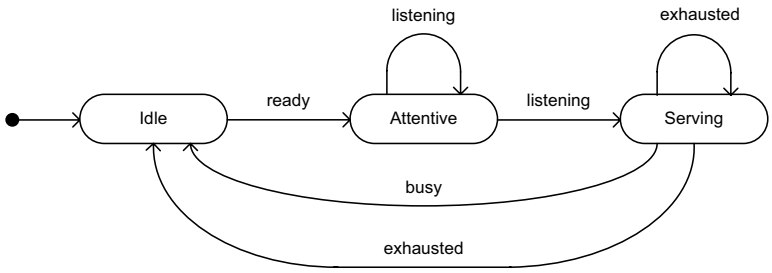


Figure 4.23: Global Process *TimedServer[AsEmpFor(i)]*

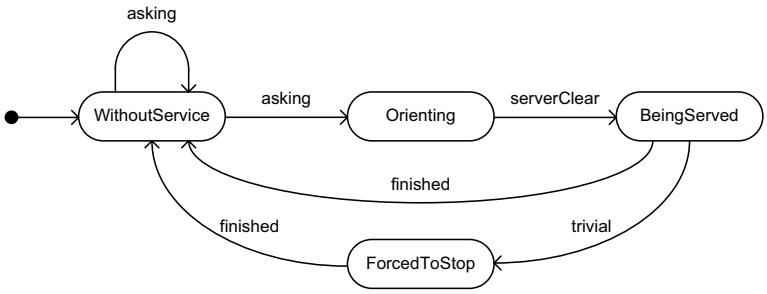


Figure 4.24: Revised process *Client[AsTOOS]*

We explain the interaction between the *TimedServer* and the *Client* processes solely in terms of views on them, i.e., in terms of the global processes of Figures 4.23 and 4.24. If a client i is asking for a service, global transition *asking* initiating from global state *WithoutService* is enabled. The timed

server, if in state *Idle* of global process *AsEmpFor(i)* and with global transition *ready* enabled, will then be changed to global state *Attentive* by application of rule $R1(i)$. Note that the client stays in global state *WithoutService*. Also, it is possible for the timed server to be in more global states *Attentive* at the same time, depending on how many clients are asking for a service. Hence, the timed server eventually selects one client by entering trap *listening* of one of its partitions, i.e., enabling global transition *listening* of one of its global processes. Rule $R2(i)$ now takes care of changing the correct client to global state *Orienting*. Both client i and the timed server change global state if rule $R3(i)$ is applied: the timed server is now serving the client and timing the service. If the client indicates it is satisfied with the service, both the client and the timed server change state through the application of rule $R4(i)$. Finally, rules $R5(i)$ and $R6(i)$ take care of the possible timeout, indicated by the timed server by enabling its global transition *exhausted*.

The resulting total interaction in this model with self-managing interaction protocols, is one in which the individual components "express their intentions" by enabling global transitions, i.e. entering traps, while the sum of these intentions results in the application of a particular consistency rule by a particular interaction protocol. The sequence in which consistency rules are applied, only depends on the employee processes and the consistency rules themselves. As can be seen, all employees are treated as equals, which makes self-managing interaction protocols particularly useful for peer-to-peer interaction modeling.

(R1(i))	∗	TimedServer[AsEmpFor(i)]:	Idle	—ready→	Attentive,
		Client(i)[AsTOOS]:	WithoutService	—asking→	WithoutService
(R2(i))	∗	TimedServer[AsEmpFor(i)]:	Attentive	—listening→	Attentive,
		Client(i)[AsTOOS]:	WithoutService	—asking→	Orienting
(R3(i))	∗	TimedServer[AsEmpFor(i)]:	Attentive	—listening→	Serving,
		Client(i)[AsTOOS]:	Orienting	—serverClear→	BeingServed
(R4(i))	∗	TimedServer[AsEmpFor(i)]:	Serving	—busy→	Idle,
		Client(i)[AsTOOS]:	BeingServed	—finished→	WithoutService
(R5(i))	∗	TimedServer[AsEmpFor(i)]:	Serving	—exhausted→	Serving,
		Client(i)[AsTOOS]:	BeingServed	—trivial→	ForcedToStop
(R6(i))	∗	TimedServer[AsEmpFor(i)]:	Serving	—exhausted→	Idle,
		Client(i)[AsTOOS]:	ForcedToStop	—finished→	WithoutService

Table 4.25: Client(i)/TimedServer Interaction Protocol $(1 \leq i \leq 3)$

4.7 Related Work

With the generalization of consistency rules as proposed in this chapter, it becomes possible to model interaction without explicit coordination by a manager process. In such a case, the interaction solely occurs between employee processes. An earlier proposal to achieve the same result has been published by us in [92], in which we presented a way to achieve *symmetric* PARADIGM models in order to ease the implementation of PARADIGM in the ToolBus architecture [58, 61, 10].

From a structural perspective, an interaction protocol in PARADIGM is comparable to a *collaboration* in the UML [73, 74]: in PARADIGM, a set of managers and employee processes play certain *roles* in the interaction protocol, like in the UML instances play roles in a collaboration. A collaboration in the UML is a purely *structural* concept [72, 62, 29], and should not be confused with a *collaboration diagram* (for this reason, this diagram type has been renamed in the UML 2.0 to *communication diagram*). We have adopted UML collaborations in the ArchiMate language for enterprise architecture [66, 60, 59].

The collaboration concept is part of the UML since version 1 [72], but it did not gain wide popularity (see e.g., [28]). In [37], UML collaborations are discussed only shortly, and their value is doubted. In [62], an example is given of how parameterized collaborations can be used for the definition of the structure of *patterns*, as is prescribed in the UML 1.3 specification [72]. In [76], an early formal approach to collaborations as realizations of use cases is presented.

Since version 2.0 of the UML, the possible ways to use collaborations and the coherence with behavioral concepts has improved (see, e.g. [86, 87]). A distinction is made between *collaborations* at the type level and *collaboration uses* at the instance level. *Ports* are introduced to distinguish between multiple roles of a collaborating component. Sequence diagrams may be used to specify *collaboration protocols* between class interfaces or to specify the interactions that may occur over a connector between ports. As a *structural* concept, UML collaborations / collaboration uses and PARADIGM interaction protocols are comparable: they both specify a set of roles to be played by components involved in a collaborative activity. Different from UML collaborations, interaction protocols have consistency rules anchored to them, which define synchronization relations over the roles. Furthermore, interaction protocols play a role in the *behavior* of the model: they sequentialize the application of the consistency rules anchored to them. The set of possible sequences of interactions can be derived if an interaction protocol in PARADIGM is combined with the global processes and manager processes playing a role in it. Such sequences are comparable to the collaboration protocols in terms of sequence diagrams mentioned earlier. In [17], the value of UML 2.0 collaborations being *composable* is pointed out: larger collaborations can be composed out of smaller ones. We did not yet investigate composability for interaction protocols. UML 2.0 collaborations have been used widely for the modeling of distributed (web) services [18, 65, 64, 35]. The interaction protocol concept added to PARADIGM can be used for the same purpose – with the addition of the self-managing interaction protocol, *choreographies* of web service peers can be modeled as well.

We remark that the motivation for the extension of PARADIGM with the interaction protocol concept stems from our effort on realizing an implementation of the PARADIGM language in Chapter 3. In line with our ideas, some authors propose to use UML collaborations at the implementation level as well. According to [15, 69, 16], the complexity of software is an emerging property of the collaborations between simple objects. The authors therefore propose to reify UML collaborations as interaction components in terms of *mediums* or *connectors*, which serve as abstractions of the interaction between objects. In their approach, mediums also play a concrete role at the implementation level. This leverages the collaboration concept in the UML from a pure modeling concept to a concept relevant for implementation. In [2], the authors propose to explicitly design the interactions within distributed systems. They define the notion of *interaction system*, a specific implementation artifact for the management of

interactions between distributed components. The role of mediums, connectors and interaction systems is comparable to the role of interaction protocols in our implementation of the PARADISE distributed interpreter framework. Of course, in our implementation, the role of interaction protocols is a limited one: they act as *arbiters* in the application of consistency rules during execution of a PARADIGM model.

Notions for the modeling of interactions at a conceptual level have been developed in the field of business process modeling, for example in terms of so-called *(process) choreographies* [26, 27, 100]. Such choreographies capture the interaction between different business entities, plus the (temporal) dependencies between these interactions. The major difference with interaction protocols in PARADIGM is that in the latter, the temporal dependencies between consistency rules are not explicitly modeled as part of an interaction protocol. Instead, these dependencies emerge from the composite behavior of an interaction protocol and the manager and global processes involved in its consistency rules.

Proposals have been done to extend the specification and description language SDL [54] with *collaboration modules* [84, 83, 82]. These modules allow for the specification of cross-cutting behavior in terms of the interactions between agents, which, according to the authors, complements the specification of behavior per individual agent. In the extended version of PARADIGM, we perceive a similar complementarity between interaction protocols and consistency rules on the one hand (an inter-component perspective), and detailed processes and partitions on the other hand (an intra-component perspective). In PARADIGM, an additional role is played by global processes, which act as the "behavioral border" between the two perspectives.

In the Multi-Agent Systems (MAS) community, the notion of *agent interaction protocol* [75, 79, 14, 63] has been developed. According to the specification in [36], agent interaction protocols are *patterns of message exchanges*, which are typically visualized like UML sequence diagrams. A primary issue in these agent interaction protocols, again, is the precise temporal ordering of message exchanges. This aspect plays no role for the interaction protocols in PARADIGM, which only ensure that consistency rules are applied *one at a time*.

4.8 Conclusions

In this chapter, we extended PARADIGM with *Interaction Protocols*, which serve as anchorage for consistency rules, thereby enabling their structuring into subsets relevant to particular modeling purposes. Moreover, interaction protocols ensure that the consistency rules anchored to them are applied one at a time. We introduced three different types of interaction protocols. Firstly, the *covering* interaction protocol, which covers the entire coordination of a set of manager and global processes. This type is particularly useful for the division of consistency rules into subsets belonging to individual manager processes in the model. Secondly, the *non-covering* interaction protocol, which coordinates a subset of the behavior of manager and global processes. Non-covering interaction protocols are useful for the separation of consistency rules which cover different aspects, but involve a single manager or global process. Our notion of *soundness* for interaction protocols defines how the consistency rules must be anchored to them in order to avoid conflicts. Finally, the *self-managing* interaction protocol, which coordinates the interaction between global processes of employees only. Protocols of this type only contain consistency rules with *empty manager parts*. They enable the modeler to avoid the manager/employee principle of PARADIGM in situations where this principle is intuitively difficult to apply. Instead, all components involved in the interaction are regarded as employees and managed by an interaction protocol. Interaction protocols strengthen the overall purpose of the PARADIGM modeling language and serve as a direct guidance for its implementation in a distributed manner.

Chapter 5

PARADE

Tools for PARADIGM

PARADE is a set of tools for the modeling, execution and visualization of PARADIGM models. The tools support the PARADIGM language and its extension with interaction protocols. PARADE consists of an *editor* for PARADIGM models, a *distributed runtime environment* for the execution of PARADIGM models based on the PARADISE distributed interpreter framework, and a *runtime viewer* for the visualization of models while they are being executed. The distributed runtime environment supports the evolution of PARADIGM models on-the-fly and the extension of PARADIGM models with software functionality: method invocations can be attached to process transitions. We describe the PARADE tool set and show that it provides a good starting point for the future development of an integrated tool suite for PARADIGM.

5.1 Introduction

The PARADIGM language has been applied in many areas to gain insight in the behavior and interaction of various types of systems. Yet, no tools are available until now to support activities like the modeling and analysis of PARADIGM models. For this reason, we have developed PARADE: a set of tools for the modeling, execution and visualization of PARADIGM models. Currently, PARADE consists of a *distributed runtime environment*, an *editor* and a *runtime viewer*. It has been developed in Java [39] using the Eclipse Modeling Framework (EMF) [31] and MOCHA [7, 50, 48, 49], an implementation of mobile communication channels. The PARADE tools incorporate our PARADISE distributed interpreter framework presented in Chapter 3, and fully support the interaction protocol concept introduced as an extension to PARADIGM in Chapter 4. Note that, for this reason, the name "PARADIGM" is used throughout this chapter to indicate the PARADIGM language including the extensions of Chapter 4. The PARADE tools and installation documentation are available for download (see [78]).

In this chapter, we present all PARADE tools and show that they are eligible for the future development of an integrated tool suite for PARADIGM. The current set of tools focuses on the *execution* of PARADIGM models. The most important tool in this respect is the PARADE *distributed runtime environment*. This environment automates the creation of an executable distributed PARADISE interpreter for a concrete PARADIGM model. Furthermore, it extends the PARADISE interpreter with two interesting features.

Firstly, it supports the *evolution* of PARADIGM models *on-the-fly*, i.e. while they are executed. Secondly, it features the possibility to extend PARADIGM processes with *software functionality*: upon the taking of a transition in the execution of a process, a method of an arbitrary Java object can be invoked and the return value can be used to steer the flow-of-control in the process execution. This way, PARADE can be used as a starting point for the development of distributed software. The *editor* and *runtime viewer* have been built primarily for the purpose of demonstrating the runtime environment. The editor can be used to specify PARADIGM processes, partitions and interaction protocols, and to combine these specifications into a PARADIGM model. The runtime viewer can be used to visualize PARADIGM models while they are executed within the distributed runtime environment.

We have set up this chapter as follows. In Section 5.2, we provide an overview of the PARADE tools and motivate their architecture and the technology used for their implementation. In Section 5.3, we show how PARADIGM models are organized in PARADE and what kinds of additional models PARADE supports. In Section 5.4, we zoom in on the automated creation of a PARADISE distributed interpreter for an arbitrary PARADIGM model, and show how our technique directly supports evolution on-the-fly. In Section 5.5, we discuss how PARADIGM models can be extended with software functionality. We present related work in Section 5.6 and provide conclusions in Section 5.7.

5.2 Overview of PARADE and Used Technology

An overview of PARADE is depicted in Figure 5.1. The primary artifacts used and manipulated in PARADE are *models*. We distinguish three categories of models: *core models*, *extension models* and *container models*. Core models specify PARADIGM processes, partitions and interaction protocols as pure syntactical constructs. Extension models extend core models with additional information (the visualization of PARADIGM entities, or their extension with software functionality). Finally, container models compose core models, possibly with their extension models, into larger structures (components and systems). We provide more details in Section 5.3.

PARADE currently consists of three *tools*: an *editor*, a *distributed runtime environment* and a *runtime viewer*. The editor can be used to edit all types of models, in a non-graphical manner. The distributed runtime environment (DRE), which incorporates the PARADISE framework introduced in Chapter 3, can be used to *execute* the models through a dynamically created and distributed PARADISE interpreter. Finally, the runtime viewer can be used to graphically visualize models while they are executed by a PARADISE distributed interpreter. The PARADE tools and the PARADISE framework have been implemented in Java [39], using the Eclipse Open Development Platform [30]. Initially, the choice for Java and Eclipse was made because of the promising Java-based code generation features of two frameworks within Eclipse, the *Eclipse Modeling Framework* (EMF) [31] and the *Graphical Modeling Framework* (GMF). Let us shortly introduce these frameworks.

The EMF is a modeling framework for structured data models. Based on a data model, Java code can be generated for the creation and manipulation of data conforming to that model. In addition, the EMF supports data persistency based on XML [77]. Finally, the EMF supports the generation of a tree-based XML editor plug-in for the Eclipse IDE. Such an editor guarantees that the contents of the created XML files syntactically adhere to a specific EMF data model. We used the EMF for the specification of meta-models for each of the model types in PARADE, using the *ECore* data modeling language (part of EMF). The code generation facilities of the EMF turned out to be particularly useful and efficient for research experiments with the PARADIGM modeling language. In addition, we were able to easily generate the Java code necessary for the PARADE *editor*.

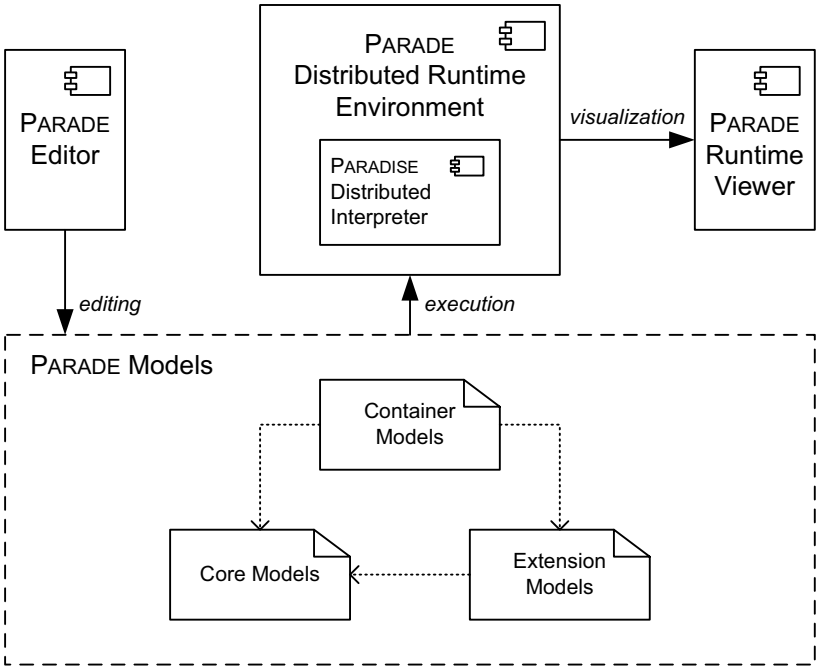


Figure 5.1: Overview of the PARADE tools and models

The GMF extends the EMF and focuses on code generation for the purpose of creating *graphical editors*. Eventually, in our research we decided not to take the step towards a graphical editing environment for PARADIGM, by which the GMF was set out of scope. The models underlying the GMF, however, are created within the ECore language. As a consequence, the existing PARADE tool-set can be easily used as a basis for the development of an integrated graphical modeling and execution environment for PARADIGM.

Distributed Runtime Environment

The primary role of the PARADE DRE is to ease the creation of a PARADISE distributed interpreter for a PARADIGM model. As we explained in Chapter 3, PARADISE provides a framework for building distributed interpreters for PARADIGM models. The creation of a distributed interpreter for a specific PARADIGM model requires that elements of the framework are composed in the appropriate way. Exactly this composition activity has been automated in the PARADE DRE. Based on the contents of a PARADE container model, the DRE automatically instantiates the appropriate elements of the framework, parameterizes them with PARADE core models, and connects them together in order to form a working distributed interpreter. As we will explain in Section 5.4, the DRE directly supports the creation, updating and deletion of PARADISE framework elements in a *running* distributed interpreter. Thereby, PARADE provides support for evolution on-the-fly.

At runtime, the PARADE DRE is structured as shown in Figure 5.2. The distributed runtime environment consists of *hosts*, which each run as a single Java console application on a single Java Virtual Machine (JVM), independent of the Eclipse IDE. The hosts automatically connect to each other in order to form a distributed runtime environment. On top of this DRE, PARADE *components* are executed, which encapsulate and manage parts of a PARADISE distributed interpreter (handlers, selectors and proxies). Each PARADE component is executed on one of the hosts in the distributed runtime environment. Obviously, each host can host multiple PARADE components.

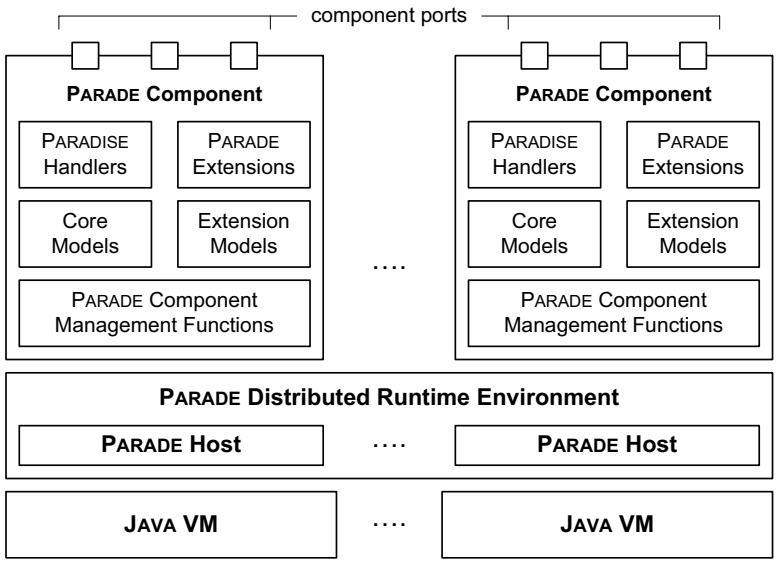


Figure 5.2: Overview of the PARADE Distributed Runtime Environment

In Chapter 3, we showed that the PARADISE framework has been implemented on the basis of *virtual nodes*, on which process handlers or a ruleset handler are "deployed" for the execution of PARADIGM processes or consistency rules, respectively. Communication between the nodes takes place via asynchronous channels with unbounded capacity. In the implementation of PARADISE in PARADE, the virtual nodes are indeed *virtual*: each handler runs as a separate Java *thread*, as if it were running on its own virtual node. In turn, each bidirectional channel between two virtual nodes in PARADISE is implemented in PARADE as a set of two unidirectional asynchronous FIFO channels, which enable the communication between two threads. These channels have been implemented with the use of MOCHA [7, 50, 48, 49].

MOCHA is an implementation of point-to-point mobile communication channels. MOCHA supports various types of channels, amongst which asynchronous channels with (virtually) unbounded capacity. Each channel in MOCHA actually consists of two *channel ends*, which can be of type *source* or type *sink*. Channel ends are residing at a MOCHA *location*, usually one per virtual machine. They are *mobile*: they can move from one MOCHA *location* to another. The Java implementation of MOCHA is based on Java Remote Method Invocation (RMI) [47].

In the current implementation of PARADE, *all* communication between threads is done via MOCHA channels. Each host in the distributed runtime environment hosts one MOCHA location. If two threads in different components need to communicate with each other, the components register a channel source end at the Java RMI Registry, which enables the two to connect and send data to each other. A channel source end which is registered at the RMI Registry is called a *port*. We did not exploit the mobility features of MOCHA in the current implementation of PARADE, but we expect that future work will address mobility of channels as well.

PARADE components also provide support for *extensions* to the PARADISE distributed interpreter. Extensions are automatically created within the components on the basis of extension models. Currently, two extensions are supported: the visualization of PARADIGM entities, and their extension with software functionality. The former extension will be shortly introduced in the next paragraphs, while the latter extension is the subject of Section 5.5.

Runtime Viewer

If a PARADISE distributed interpreter for a PARADIGM model is executed in the PARADE DRE, the interpreter simply interprets the PARADIGM model by letting the handlers take steps in accordance with the constraints of partitions and consistency rules. In order to provide visual feedback about the PARADIGM model being executed, we have developed a separate application, the PARADE *runtime viewer*, which can be dynamically attached to PARADE components running in a DRE. Based on PARADE extension models for visualization, the runtime viewer is able to visualize processes, partitions and interaction protocols in a running PARADIGM model. Each process, partition and interaction protocol is displayed in a separate window. Every window creates and maintains a dedicated connection with the process, partition or interaction protocol in the PARADE component which it visualizes. This way, the viewer can show parts of different components in separate windows on a single screen, even if the components themselves are distributed.

The PARADE runtime viewer supports three different types of windows. In a *process window* (Figure 5.3), a PARADIGM process in the system is visualized. Within this window, it is also possible to stop a running process, restart it, perform step-by-step execution or alter the speed of the process handler for the process, by setting a certain time-to-sleep between the taking of process steps.

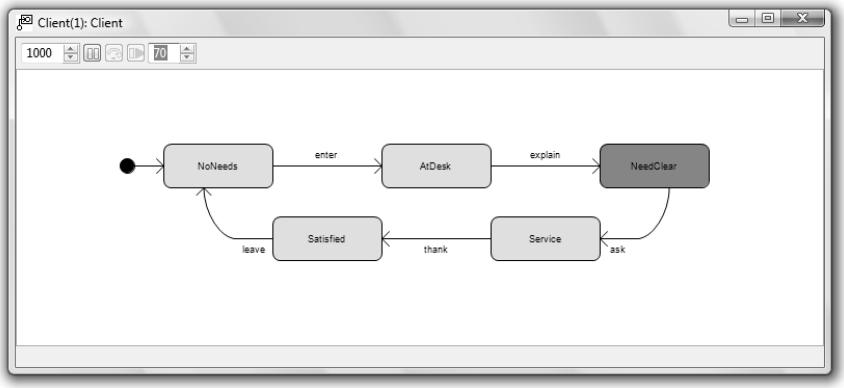


Figure 5.3: Process window in the PARADE runtime viewer

In a *partition window* (Figure 5.4), the current subprocess of a partition is shown, together with the traps defined for this subprocess. Finally, an *interaction protocol window* (Figure 5.5) shows an interaction protocol in the system as a list of consistency rules, indicating with "traffic lights" for each consistency rule which transitions are currently enabled: "red" means disabled, "green" means enabled.

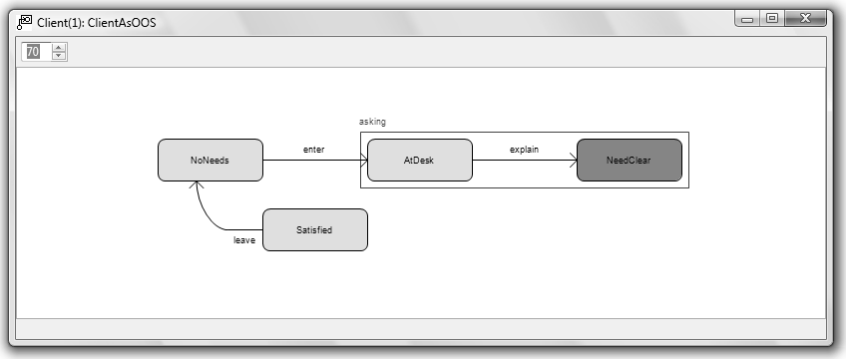


Figure 5.4: Partition window in the PARADE runtime viewer

The extension models which specify how processes and traps are to be visualized, are not known by the runtime viewer in advance. Merely, they are part of the running PARADE components. Once a window in the runtime viewer is opened, it asks the associated PARADE component to communicate the necessary visualization models. This way of working has been chosen in order to fully support evolution on-the-fly: if a running component evolves, its visualization can also be evolved.

This concludes the overview of PARADE. In the next three sections, we will zoom in onto three interesting parts of the implementation. In Section 5.3, we will address the different *models* in more detail. In Section 5.4, we will show how the PARADE distributed runtime environment is able to automatically create and evolve PARADISE distributed interpreters for a PARADIGM model. Finally, in Section 5.5 we briefly explain how software functionality can be used within the execution of a PARADIGM model.

5.3 Model Specification in PARADE

The basis for the PARADE tools is formed by the PARADE *models*, which specify PARADIGM models and extensions to them in a particular XML syntax understood by the PARADE tools. An overview of all PARADE model types is given in Figure 5.6. The arrows between the model types indicate dependencies. In the next sections, we zoom in on the individual models and explain their relationships, by showing how the client/server model of Chapters 2 and 3 can be specified in PARADE.

Core Models

The PARADE core models are used to specify the entities of a PARADIGM model. PARADE allows for the separate modeling of three different PARADIGM entities: processes, partitions and interaction protocols. We have chosen not to model separate individual consistency rules, but rather interaction protocols to which consistency rules are anchored. Core models do *not* contain information about how the PARADIGM entities are *visualized*; they only represent the PARADIGM entities as *syntactical* constructs.

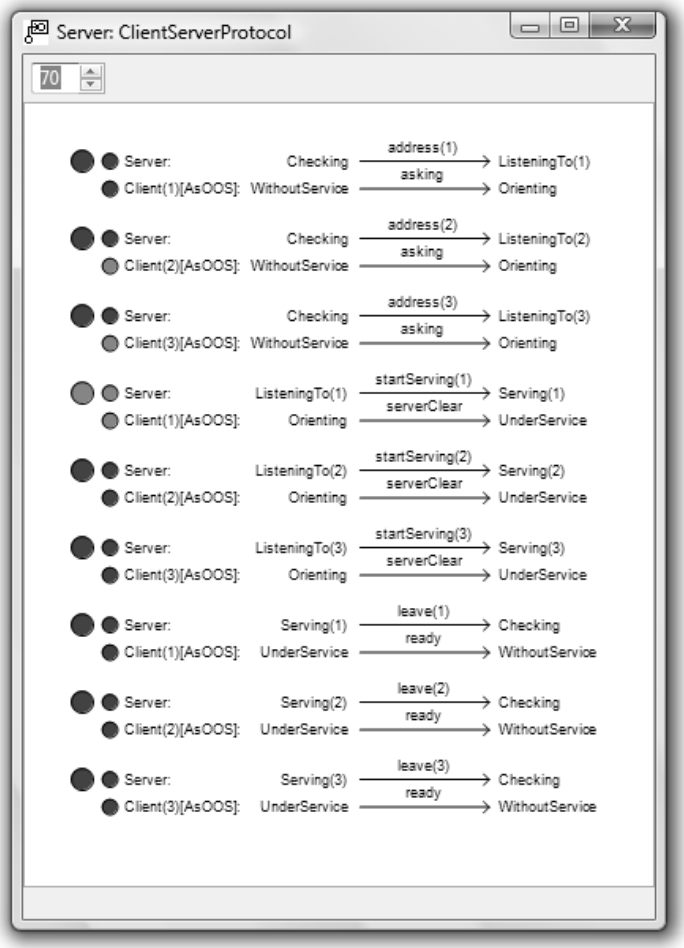


Figure 5.5: Interaction Protocol window in the PARADE runtime viewer

In order to allow for an efficient reuse of models, we regard the core models created in PARADE as *types*. An example may clarify this approach. Suppose we would like to create a PARADE model for the client/server example presented in Chapter 2. This model consists of one server and three client components. The client components are all equal, hence in PARADE we need to specify their internal processes and partitions only once. The total example requires the specification of five core models, as depicted in Figure 5.7. The process types *Client* and *ObjectOfService* specify the detailed and global process of each of the clients, respectively. Process type *Server* specifies the detailed process of the server. Partition type *ClientAsOOS* represents the partition of detailed process type *Client* with global process type *ObjectOfService*. Finally, an interaction protocol type *ServerAndThreeClients* is specified with four roles: one Server role with process type *Server*, and three roles Client1, Client2 and Client3 with process type *ObjectOfService*.

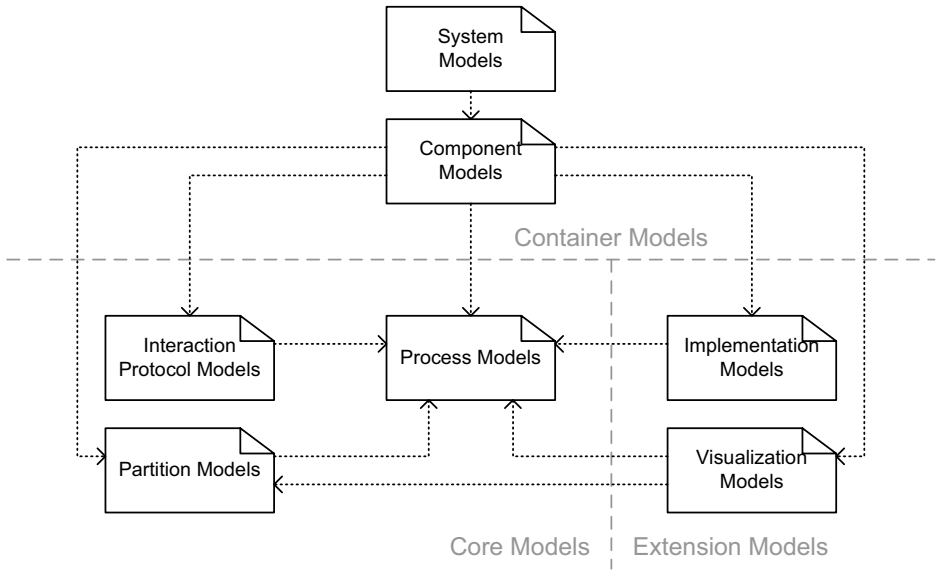


Figure 5.6: Overview of the PARADE models

We distinguish two different *process roles* for partition types: a detailed and a global role. The subprocesses of the partition need to match with the process type playing the detailed process role, while their names and trap names should match to the process type playing the global process role. By convention, we choose the names of process types in such a way that they can be used as either detailed or global process types. The names of partition types are chosen such that it is clear which detailed process type and global process type they relate to each other. For example, partition type *ClientAsOOS* can be used to view a *Client* process as an *ObjectOfService* process. Likewise, we could have defined a partition type *OOSAsClient* which allows us to view an *ObjectOfService* process as a *Client* process.

Within interaction protocol types, we define one or more *process roles*, each with a unique identifier within the interaction protocol, and a corresponding process type which we expect to play the role. A transition mentioned in a consistency rule anchored to an interaction protocol should match a transition mentioned in the process type playing the corresponding role. We also indicate for each role whether the role is a *manager role* or an *employee role*. Because the structure of interaction protocols can vary considerably, depending on the context in which it is used, we do not adopt a naming convention for the names of interaction protocols.

Extension Models

Extension models provide a way to extend the core models with additional information or functionality. With regard to the execution of PARADIGM models, these models are optional. Currently, PARADE supports two kinds of extension models: *visualization models* and *implementation models*. These models are independent from each other and can be used separately. It is possible to create many different visualization models and implementation models for a single core model.

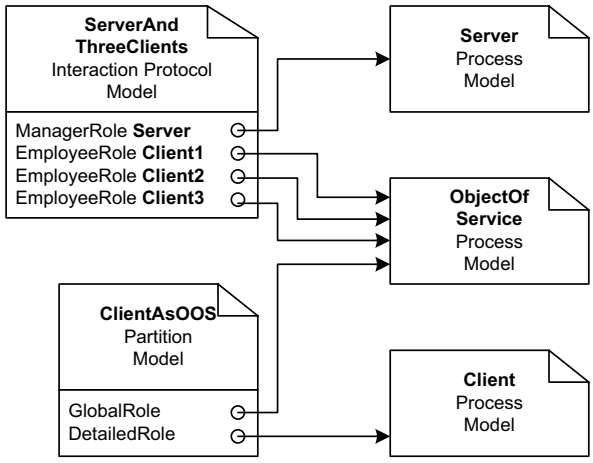


Figure 5.7: PARADE core models and their dependencies for the client/server example

Visualization models provide the information necessary to visualize processes or partitions inside the PARADE runtime viewer. For process types, they specify how their states, transitions and labels are visualized. For partition types, they specify how traps of their subprocesses are visualized. Interaction protocols form an exception and do not need a visualization model – they are always visualized in the same manner, as a list of grouped transitions for their process roles. Implementation models allow a modeler to bind the transitions in a PARADIGM process model to method invocations on Java objects. We provide more detailed information about these models in Section 5.5.

Container Models

Two types of container models are known in PARADE: *component models* and *system models*. Component models define *component types*, based on core models and extension models. System models compose PARADE components into a system by creating instances of component types. They can be used for the specification of entire PARADIGM models.

The component concept is used in PARADE as a *unit of deployment*: each component runs on a single host in the DRE. PARADE components roughly correspond to the notion of component in PARADIGM: entities containing one detailed process and zero or more views on this detailed process in terms of partitions and corresponding global processes. However, we use components in PARADE also as containers for *interaction protocols*. The primary reason for this is that interaction protocols have behavior of their own, hence they must be deployed explicitly on a host. Furthermore, we gain uniformity in the way PARADIGM models are implemented in PARADE: all processes, partitions and interaction protocols are contained in components, which in turn are composed into a system. As we will show in Section 5.4, this has direct benefits for evolution on-the-fly: at runtime, the PARADE components create, update and delete all evolvable PARADIGM artifacts in a uniform manner. An example of the two types of container models in PARADE is given in Figure 5.8, which shows the component models and system model for the client/server example. For convenience, we do not show the usage of extension models in the figure. Using the example, we shortly explain the internals of both container models.

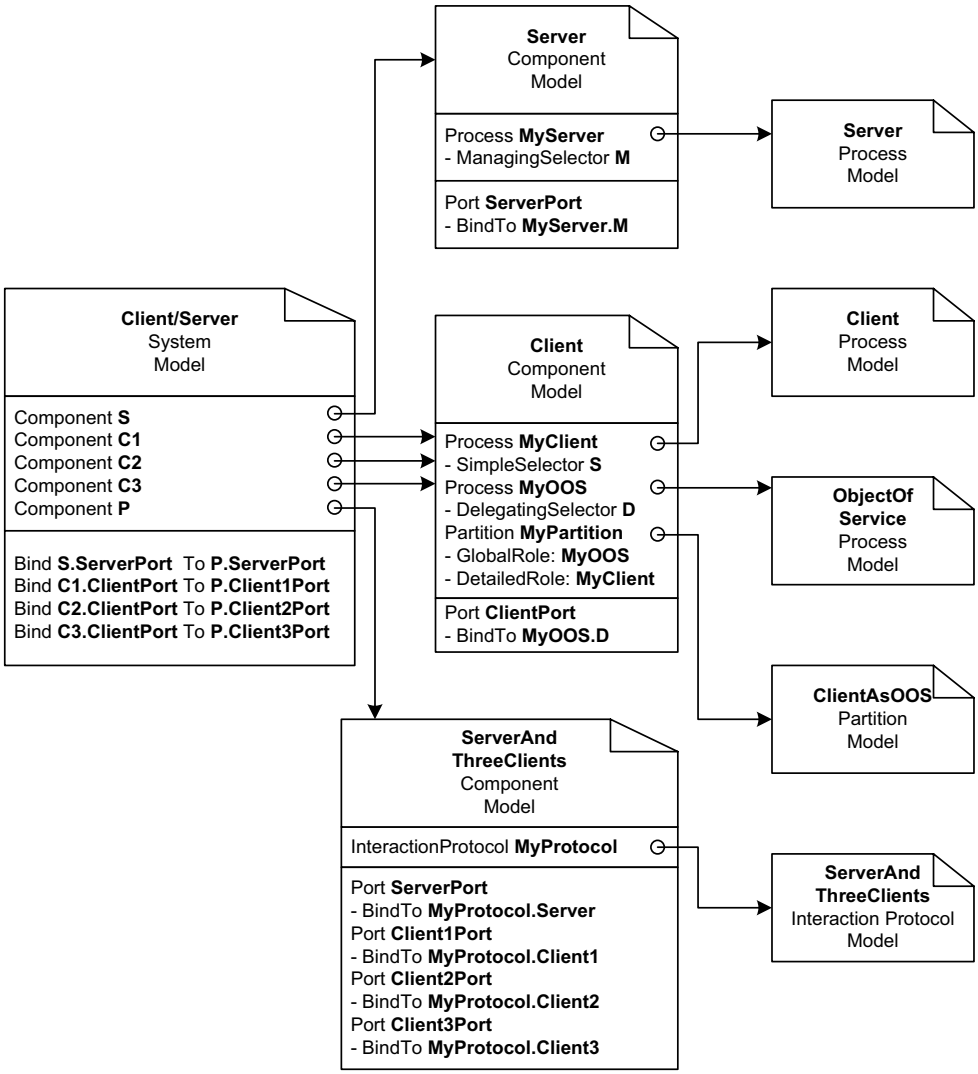


Figure 5.8: PARADE container models and dependencies for the client/server example

Component models specify *core model instances* and *ports*. The instantiated core models can be process models, partition models or interaction protocol models. Since process models only specify process *types*, they can be instantiated into a pure employee process, a manager process or a global process. This is done by using a specific type of *selector*. For example, the process instance *MyServer* created in the *Server* component type will be used as a manager process and therefore gets a managing selector *M*. For partition models and interaction protocol models, we indicate which process instances play their roles. Obviously, these instances need to be of the appropriate type. Take for example the specification of the

Client component, which contains a partition instance *MyPartition* of type *ClientAsOOS*. The detailed and global role are played by the two process instances *MyClient* of type *Client* and *MyOOS* of type *ObjectOfService*, respectively. For interaction protocol models, each role is played by a single selector of a single process. Finally, each component specifies *ports* for communication with other components. Each port can be bound internally to one of the following entities: the selector of a process, the global or the detailed role of a partition, or the role of an interaction protocol. An example is shown in Figure 5.8 of the *ServerAndThreeClients* component, which contains an instance of the *ServerAndThreeClients* interaction protocol. All roles of the interaction protocol instance are bound to separate ports. In the implementation, the ports correspond to MoCHA channel source ends.

System models specify *component model instances* and *port bindings*. The example in Figure 5.8 shows a *Client/Server* system model, which instantiates a component of type *Server*, three components of type *Client*, and a component of type *ServerAndThreeClients*. The ports of the components are connected to each other by means of port bindings. In the next section, we show how the resulting system is created and executed on top of the PARADE DRE.

5.4 Model Execution and Evolution in PARADE

In this section we show that there is essentially no difference between the creation and start-up of a PARADISE distributed interpreter in PARADE, and its evolution with updates, creates and deletes. We start with giving an overview of the changes to the PARADISE framework which were needed to incorporate the extensions of Chapter 4. After that, we illustrate how component models and system models are used to create runtime components in the PARADE DRE. Finally, we show which techniques are provided by the DRE and the runtime components for evolution on-the-fly.

Extensions to PARADISE

Throughout the development of the PARADE tools, we extended the PARADISE distributed interpreter framework discussed in Chapter 3 in order to incorporate the conceptual extensions presented in Chapter 4. The changes are relatively straightforward and mainly consist of some generalizations of the implementation presented in Chapter 3. The pseudo code in Appendix A already includes these changes.

- **Covering Interaction Protocols.** No changes to the framework were needed to support the concept of covering interaction protocols: basically, instead of using one ruleset handler for all consistency rules, we now use a ruleset handler for each interaction protocol.
- **Non-covering Interaction Protocols.** Managing selectors support the usage of separate channels to manager proxies of multiple non-covering interaction protocols. Delegating selectors support multiple channels to different employee proxies. In order to adhere to the soundness criterion formulated in Chapter 4, we require that delegating selectors use a single channel for delegating the selection between transitions with the same source state.
- **Self-managing Interaction Protocols.** The syntax of consistency rules has been generalized in order to allow for consistency rules with empty manager parts. To support self-managing interaction protocols, we use a special version of a manager proxy called a *self-manager*, which is actually not a proxy to a manager process, but a simple selector for consistency rules with empty manager parts.

From PARADE Models to PARADE Runtime Components

Each PARADE host is a Java application, which can be started up in two different modes: *passive* or *initiating*. If it is started up in passive mode, the result is an empty host which listens to *system requests* on a dedicated MOCHA system channel. If started up in initiating mode, the host is parameterized with a system model, which specifies:

- a list of passive hosts to connect to, in order to form a *distributed* runtime environment consisting of the combination of the initiating and passive hosts;
- a list of PARADE components to create, with for each component a specification on which of the hosts the component must be created;
- a list of port bindings specifying which component ports (source channel ends) must be exchanged between two components.

Typically, the distributed runtime environment is used as shown in the example of Figure 5.9. Depicted are four physical machines, each with a JVM installed. On one of the machines, a set of related core, extension and component models and a system model is available. On this machine, a PARADE host is started up in initiating mode with the available system model, while on all other machines PARADE hosts are started up in passive mode.

First, based on the system model specification, the four hosts are connected to each other, on the initiative of the initiating host, in order to form a distributed runtime environment. After that, the initiating host continues with processing the list of components to be created, as specified in the system model. This is simply done by issuing a *request to create* to the system channel of one of the hosts, with a specification of the component to be created. The creation itself involves two steps. Firstly, a runtime PARADE component class is instantiated. The resulting component acts precisely as a passive host: initially, it is an empty shell, which listens to a dedicated MOCHA system channel for incoming requests. After that, a *request to create* is sent to the system channel of the new component, with a specification of the appropriate core and extension models. After having received this request, the new (empty) component creates the necessary PARADISE handlers, selectors and proxies for the core models, following a specific procedure as shown in Figure 5.10.

An example of the creation of PARADISE elements is depicted in Figure 5.11. The example shows the *Client* component model of the client/server example and a resulting runtime component *Client(1)* with its internal PARADISE elements. As can be seen, the single component contains two process handlers, one for detailed process *MyClient* of type *Client* and one for global process *MyOOS* of type *ObjectOfService*. Each of the handlers is implemented as a separate thread object, while the remaining elements are passive objects whose methods are called by the thread object. Note also the usage of *internal* MOCHA channels between the detailed and global role handlers of partition *MyPartition*. The existence of port *ClientPort* simply means that the source end of the channel to delegating selector *D* is registered in the Java RMI registry with the name *ClientPort*.

After all components have been created in the DRE, the initiating component processes the *port bindings* as specified in the system model. Each binding between two ports *A* en *B* involves a simple procedure: send the name of port *A* to port *B*, and the name of port *B* to port *A*. The internal PARADISE elements which export the two port names, now have a reference to each other in order for them to send a receive messages. If applicable, they can move the source ends of the channels to their own location. The runtime components are started by sending to their system channel sources a *request to start*. As a final example, we show in Figure 5.12 the entire client/server model running on a PARADE DRE consisting of four JVMs.

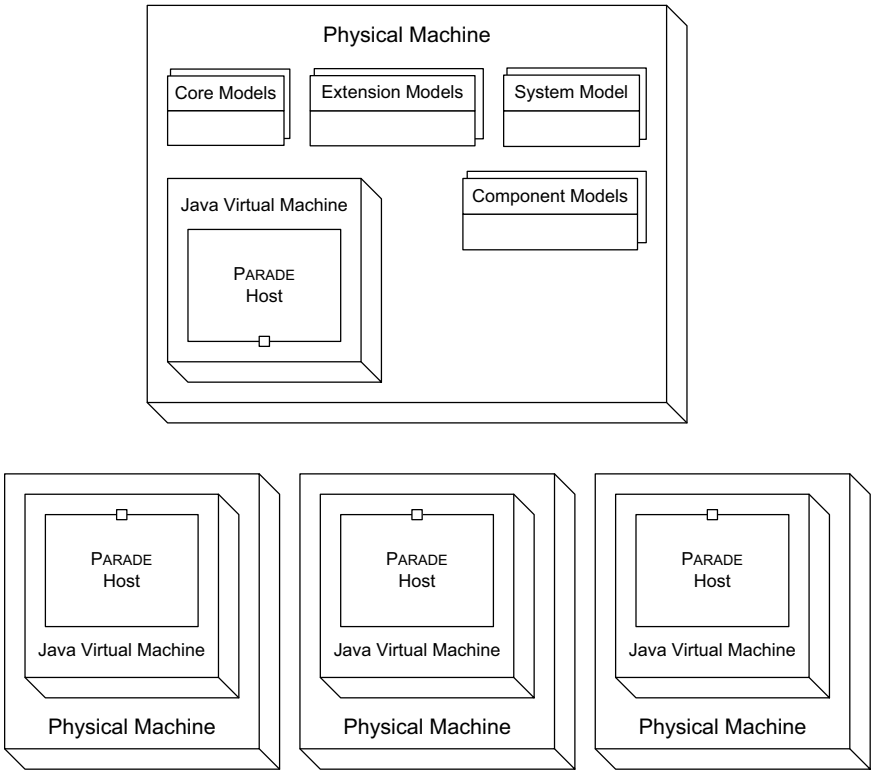


Figure 5.9: PARADE Execution: Initial situation

Evolution of PARADE Components

Runtime PARADE components can be created and deleted by using the system channel from a PARADE DRE. The internals of PARADE components can be altered by using the system channel of the components in question. This basically involves creations, updates and deletions of PARADISE handlers, selectors and proxies. The creation of processes, partitions and interaction protocols is done as explained earlier. The order in which elements are created is important: if a partition or an interaction protocol must be bound to a process or a process selector inside the same component, the latter one must already exist prior to the former one. Updates are simply done by sending new versions of core and/or extension models to the running component via its system channel and indicating which existing entities need to be updated. Finally, deletions of existing processes, partitions and interaction protocols can be done straightforwardly, again taking into account a certain order in the deletion.

In general, the creation, updating and deletion of processes, partitions and interaction protocols often requires additional coordination in order to make sure that the updates do not conflict with the running system. PARADE does not implement any checks on this point: it is the full responsibility of the modeler/developer to make sure that the running system evolves smoothly. We show techniques to do so in the case study of Chapter 8. Nevertheless, our approach completely removes the distinction between the initial creation of a system and its evolution.

1. For each process:
   (a) create a process handler, parameterized with the process model;
   (b) create the appropriate selector and bind it to the process handler;
2. For each partition:
   (a) if a detailed role binding has been specified, create a detailed role handler, parameterized with the subprocesses of the partition, and bind it to the process handler of the appropriate (detailed) process;
   (b) if a global role binding has been specified, create a global role handler and bind it to the process handler of the appropriate (global) process;
   (c) if a detailed and a global role binding have been specified:
      i. send the channel source id of the global role handler to the channel source of the appropriate detailed role handler
      ii. send the channel source id of the detailed role handler to the channel source of the global role handler
3. For each interaction protocol:
   (a) create a ruleset handler;
   (b) create a rule handler for each consistency rule, parameterized with the consistency rule;
   (c) create a manager proxy for each manager role in the interaction protocol, bind it to the protocol handler and the appropriate rule handlers;
   (d) create an employee proxy for each employee role in the interaction protocol, bind it to the protocol handler and the appropriate rule handlers;
   (e) for each specified role binding:
      i. send the channel source id of the proxy to the channel source of the bound selector;
      ii. send the channel source id of the bound selector to the channel source of the proxy;
4. For each port:
   (a) bind the channel source id of the specified PARADISE element within the component to the port name;
   (b) export the port name.

Figure 5.10: Generic procedure for the creation of PARADISE distributed interpreter elements

5.5 Software Functionality Extensions

In our PARADE implementation, a PARADIGM process is executed by a PARADISE process handler running inside a PARADE component deployed on one host of a PARADE distributed runtime environment. The process handler simply takes transitions in a sequential manner. We provide a convenient extension to this execution, namely to attach *software functionality* to the taking of transitions by process handlers. In this section, we shortly explain this extension.

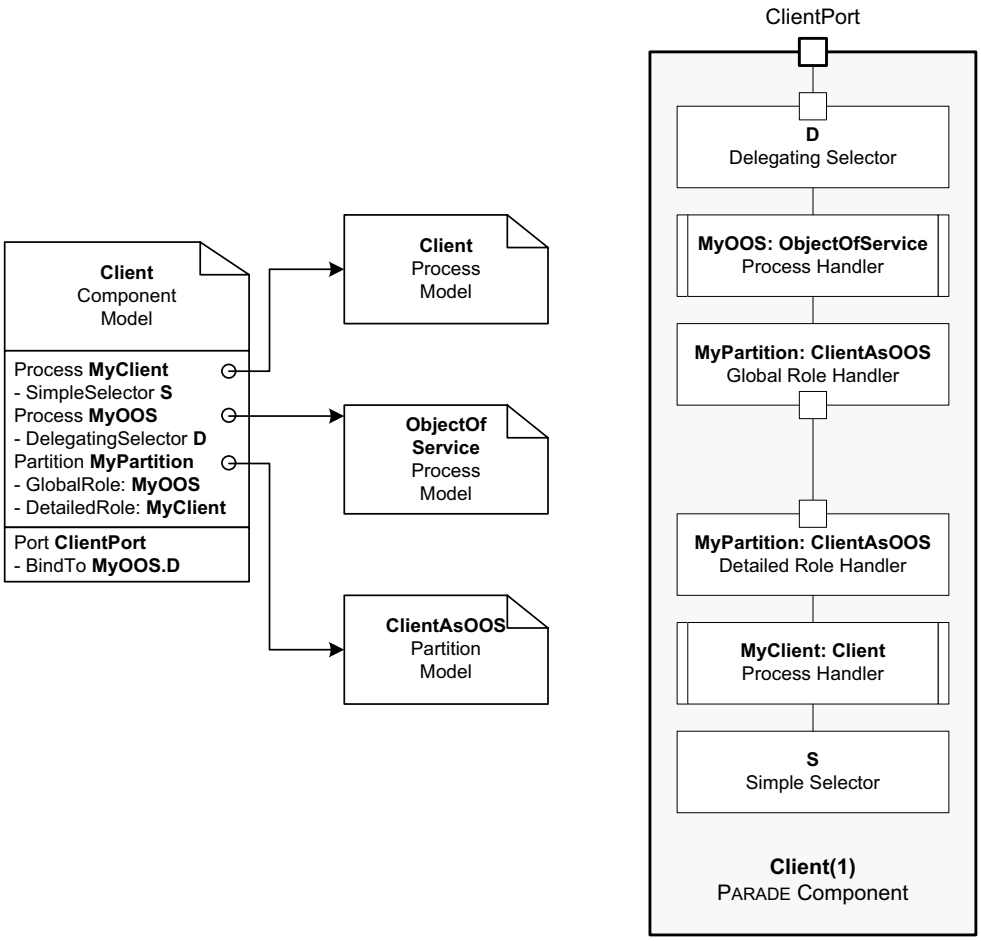


Figure 5.11: Creating a Runtime PARADE Component from a Component Model

An overview of our approach is shown in figure 5.13. On top, a fragment of a PARADIGM process is shown. Beneath, a set of Java classes is depicted, divided into *action classes* and *implementation classes*. These classes provide the actual software functionality. Action classes are used as a kind of *object adapter* or *façade* [38]. They are the bridge between the transition labels in the PARADIGM processes, which indicate a certain abstract activity, and the realization of this activity at the level of the implementation classes. This way, the actual structure of the implementation classes can be chosen differently from the conceptual structure as perceived at the level of the PARADIGM model.

In the PARADE *implementation model*, action classes can be mapped onto transition labels of PARADIGM processes. The implementation model thus provides an interpretation of the transition labels of a PARADIGM process in terms of computational activities. During execution, each time a transition of a process is selected by a process handler, the implementation model is checked whether an action class

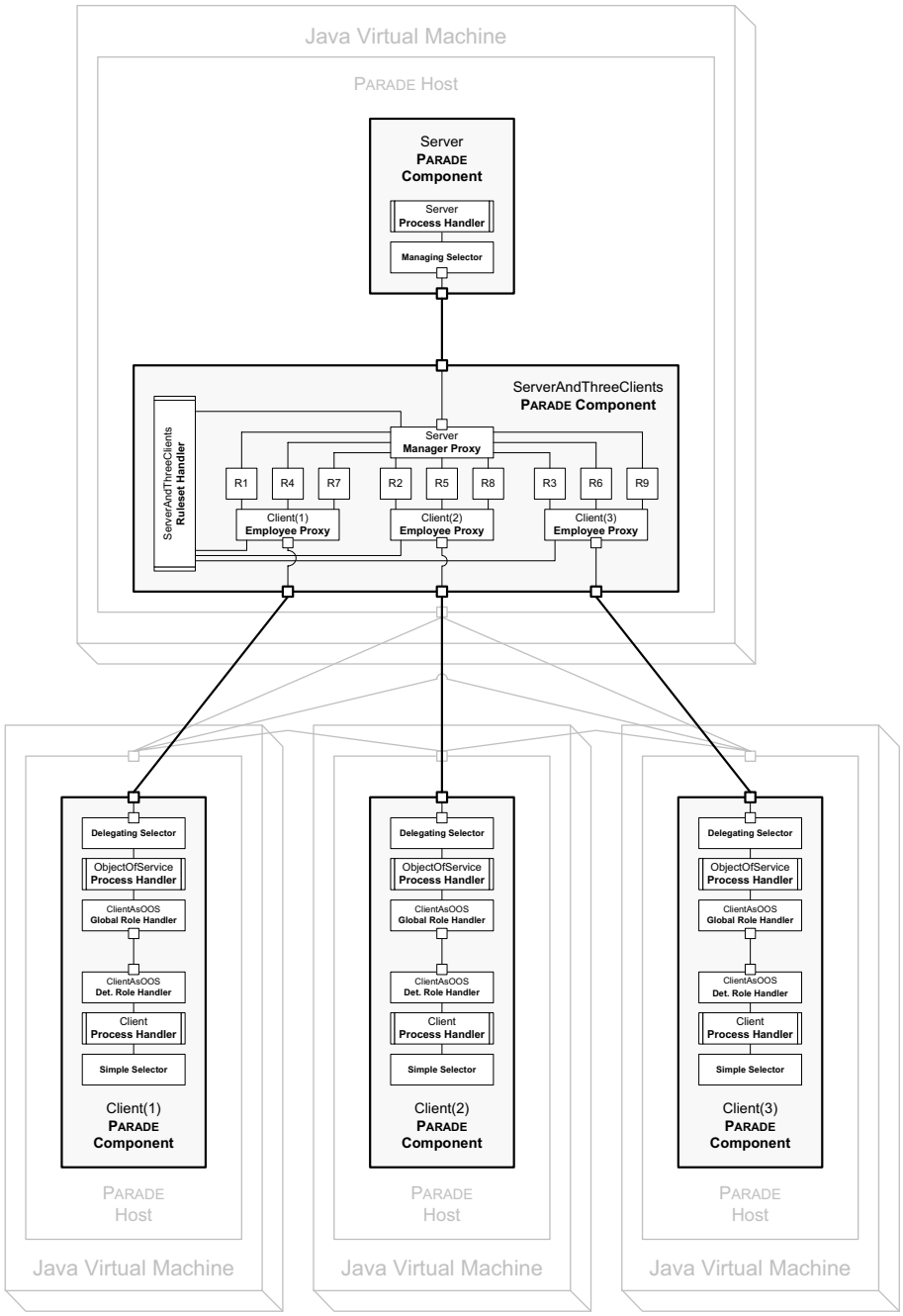


Figure 5.12: PARADE Execution: The Client/Server Example running on 4 JVMs (see also Figure 5.9)

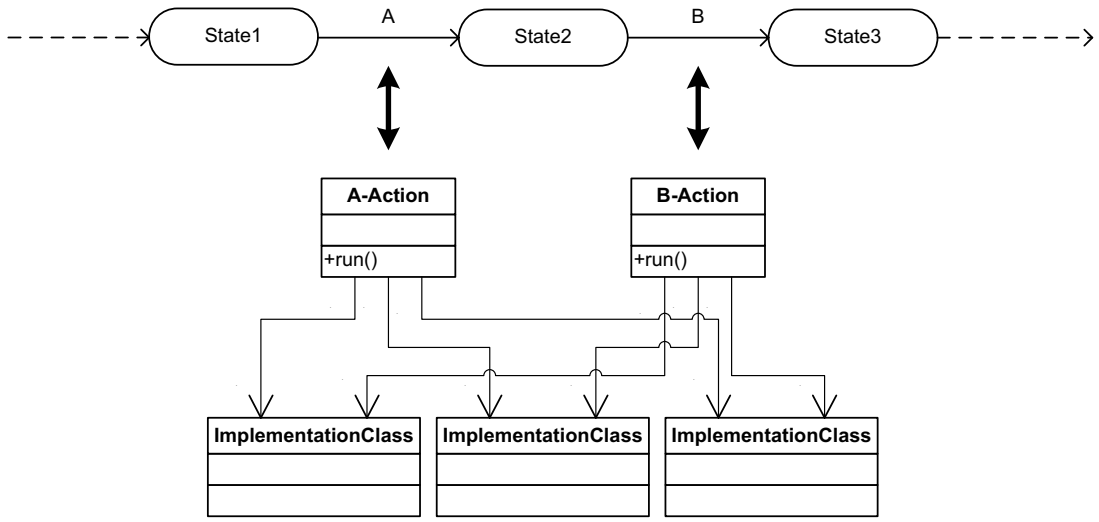


Figure 5.13: Associating transitions of a PARADIGM process to the implementation

is associated with the label of the transition. If there is an associated action class, the process handler instantiates the action class and invokes its run() operation. After the operation returns, the process handler takes the transition and enters the target state of the transition. The execution of a process in the distributed interpreter thereby could result in a sequence of software actions being executed. This way, a PARADIGM model acts as a high-level specification of the *control flow* in a distributed software system: it imposes a specific order upon the activities performed by the software.

It is possible for the software implementation to influence the (non-deterministic) transition selection mechanism in the distributed interpreter. This is especially useful for situations in which the high-level coordination relies upon choices determined by low-level computational results or external input. *How* a choice is made in the implementation, is of no importance at the conceptual level of PARADIGM: it can still be considered as a non-deterministic choice: multiple transitions with the same transition label, the same source state, and different target states. In the implementation model, however, we specify that these transitions are all mapped onto a single action class, and that the run() operation of this class has a *return value* indicating which of the transitions must be taken.

Take for example the fragment of a detailed PARADIGM process *WeatherObserver* of Figure 5.14. In this fragment, we assume that process *WeatherObserver* at some moment in time enters state *No Idea*. Being there, it checks whether it rains or not, and changes state accordingly. Suppose that we want this choice to be made within the implementation, for example based upon the input from a humidity sensor. We then indicate in the *implementation model* that both transitions *check weather* are mapped onto a single *CheckWeather-Action* class, which returns either *Raining* or *Not raining*. In the PARADISE distributed interpreter, both transitions are now treated as shown in Figure 5.15: they firstly lead to a special state, similar to the *choice pseudo state* in the UML. In this choice state, the return value of the associated action class is evaluated to determine the target state. Note that this mechanism could wrongly interfere with constraints on the transitions in the PARADIGM model. It is the responsibility of the developer to avoid this.

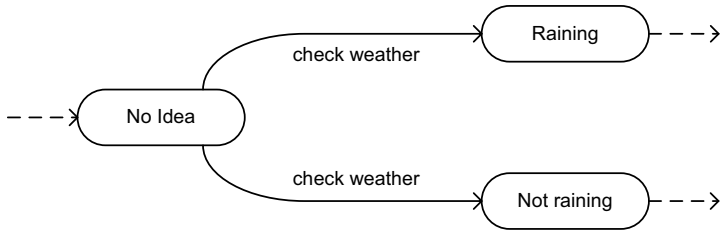


Figure 5.14: Fragment of a PARADIGM process *WeatherObserver*

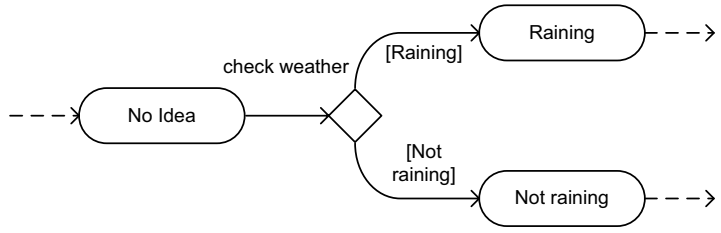


Figure 5.15: The fragment of Figure 5.14 interpreted as an implementation choice

The extension of PARADIGM models with software functionality can be used to create real software systems. In order for this approach to be effective, the level of abstraction for the PARADIGM model must be chosen carefully, preferably such that it provides insight into the conceptual organization of the distributed system as-a-whole, without containing too many technical details of the individual components. Seen this way, a *detailed process* in PARADIGM can be considered best as a *high-level, abstract control-flow model* of a software system. We show an example in Chapter 7, which contains a case study on the implementation of a branch-and-bound algorithm. In Chapter 8, we apply the software functionality extension to perform creations, updates and deletions for the purpose of evolution of a PARADIGM model on-the-fly.

5.6 Related Work

The runtime visualization tools for PARADIGM presented in Chapter 5 have been based on insights from work which we published in [93]. In this publication, we showed how to effectively use XML [77] and RML (the Rule Markup Language, [55]) for modeling, visualizing and analyzing enterprise architectures. Although we did not use XML transformations in PARADE, the rigid distinction between XML specifications of PARADIGM models and their visualizations in terms of *extension models* is based on the principles presented in [93]. Insights from our work on impact-of-change analysis, which is part of a book chapter in [66] and which has been also published in [23], have been implicitly reused in the algorithms adopted in PARADE for the creation, update and deletion of PARADIGM modeling artifacts during evolution on-the-fly.

The PARADE distributed runtime environment can be regarded as a *distributed virtual machine* for PARADIGM models. The instruction set of the virtual machine is fairly limited: the taking of transitions of detailed and global processes, in accordance with the dynamic constraints imposed by partitions and interaction protocols. The approach is comparable to work done on virtual machines for UML. In [80], the relevance of this approach is pointed out: changes to a model have immediate effects on its execution. In [85], a UML virtual machine is presented for embedded systems, showing that it is possible to achieve a fairly small footprint for its implementation, while in [99, 9], work on the Matilda distributed virtual machine is presented. These approaches, however, try to completely hide the programming layer, which leads to very detailed UML models which contain all relevant programming details. We provide a contrasting approach in PARADE with the use of software extension models: the PARADIGM model itself remains inherently abstract, but it is extended with a concrete implementation. We illustrate our approach in the case study of Chapter 7.

5.7 Conclusions

In this chapter, we presented PARADE: a set of tools for the modeling, execution and visualization of PARADIGM models. We have implemented PARADE models and editor in Java using the Eclipse Modeling Framework. The PARADE distributed runtime environment is used to ease the creation and evolution of a PARADISE distributed interpreter for a PARADIGM model, using two types of container models to specify the model's component and overall system structure. All communication in the distributed runtime environment takes place via asynchronous MOCHA channels. A *runtime viewer* allows for the visualization of parts of a PARADIGM model while it is being executed. The PARADIGM models have been clearly separated from their visualizations. Furthermore, an extension to PARADIGM models has been implemented which allows the attachment of software functionality to the taking of process transitions. This extension supports a means to influence the selection of transitions in the PARADIGM model from within the software implementation, through the application of choice states.

In our implementation approach, no distinction exists between initial construction and eventual evolution of a PARADISE distributed interpreter for a PARADIGM model: all entities are instantiated and connected to each other at runtime, based on the contents of container models. Since PARADE has been implemented with the use of the Eclipse Modeling Framework, it provides a flexible starting point for the realization of a graphical modeling environment for PARADIGM by means of the Eclipse Graphical Modeling Framework (GMF).

Currently, PARADE lacks tools for the verification and model checking of PARADIGM models. Such tools would be a valuable addition to the current tool set and relevant future work. The software functionality extension of PARADE provides an interesting starting point for future research on *model-driven* software development, in which a PARADIGM model could act as a high-level model of a distributed software system. Finally, more work should also be done on *performance optimization* of PARADE, focusing on differentiation in the applied communication mechanisms and exploitation of the mobility of MOCHA channel ends.

Part II

Case Studies

Chapter 6

A Car Navigation System

We use PARADIGM extended with interaction protocols to create a model of a multi-component car navigation system. The components of the system offer several modes of operation. These modes need to be properly coordinated in order for the system to ensure continuity of correct service. The PARADE tools are used to create and execute the model, and to gain insight into its behavior. The case study illustrates the suitability of PARADIGM for the modeling of non-trivial interaction between software components. It demonstrates the usefulness of the interaction protocol concept, which we apply to clearly separate component communication from the coordination of the modes of operation. It also shows applications of the two modeling principles of PARADIGM: we define multiple views on a single component for managing its various modes of operation, and we utilize manager/employee hierarchies for the coordination of system-level modes of operation at the level of components.

6.1 Introduction

This chapter is the first of a series of three chapters in which the PARADIGM extensions and the PARADE tools are validated by means of case studies. Each case study focuses on a different aspect of the language and/or the tools. The case study presented in this chapter illustrates the applicability of the PARADIGM concepts for modeling the behavior of a software system at multiple levels of abstraction. Partitions are used to disentangle coordination issues for individual components. The combination of interaction protocols, manager processes and partitions is used to realize hierarchy in the coordination of multiple components and to hide the coordination complexities of subsystems for the environment.

The case study of this chapter has been carried out by us in the context of the European ITEA [53] research project Trust4all [97]. This project aimed at the definition of a *trustworthiness framework* for embedded middleware, which explicitly addresses robust and reliable operation, upgrading, extension, and component trading [67]. The Trust4All project was preceded by two former ITEA-labeled projects, Space4U [89] and Robocop [81], in which the framework architecture and extensions for resource management were developed. Within the Trust4all project, the results of the former projects were extended with methods and techniques to allow for the establishment of confidence in dependable and secure operation of a (dynamically changing) system built out of components provided by multiple different parties.

One of the major issues addressed in the Trust4all project was the issue of *reliability* of software systems, defined concisely in [8] as "continuity of correct service". This issue is particularly interesting in the context of embedded software which can be upgraded and extended on the device on which it runs. Hardware resources (e.g., CPU or memory) for embedded devices are considered relatively limited and fixed, and the upgrading of existing components or the installation of new components could negatively affect CPU and memory usage and harm the reliability of the system as-a-whole. Next to this, embedded devices like mobile phones and PDAs, for which the framework developed in Trust4all is intended, can often be used in different environments, e.g., at home, at work, or while traveling. Variations in the availability of network bandwidth, power supply, etc. could affect the reliability of an embedded device.

In Trust4all, the above issue has been partly addressed by requiring from individual components that they support different *modes of operation*: ways in which they execute their intended functionality. For example, a video decoder could provide a mode of operation for decoding all layers of a video frame and a mode for decoding only the lowest layer (with lower resolution or lower frame rate). The second mode provides less quality, but also uses less resources. Ideally, the modes of operation offered by a component must be selectable at runtime and depend on the actual context and user demands. Thereby, runtime changes caused by component upgrades or occurring in the environment of the device can trigger a *trustworthiness manager* within the device to switch the modes of operation of running components, in order to realize continuity of correct service. The reliability problem can hence be reformulated as the problem of selecting the appropriate modes of operation for all running components in every possible context.

However, in case of multiple components, modes of operation cannot in all cases be selected independently from each other: components communicate and interact, hence a change to the mode of operation for one component may conflict with the behavior of other components. The mechanism adopted for the selection of modes of operation for multiple components must therefore be analyzed and designed carefully, and the communication between the components must be taken into account. It is the scope of this case study to show the suitability of PARADIGM for this analysis and design. In the case study, we model part of a system which is typically considered in Trust4all: a generic multi-component *car navigation system*. First, we model the components and their communication. On top of this, we model the selection of modes of operation, both for the individual components and for the system as-a-whole. Since PARADIGM focuses on behavior and interaction, it is especially suitable for modeling the *runtime selection* of modes of operation, where each mode of operation is modeled as a phase in a component's behavior. At component level, a set of related modes of operation is modeled as a *view* by means of a partition and a global process. At system level, modes of operation are modeled by means of manager processes in combination with interaction protocols.

Note that our aim is to show the suitability of PARADIGM and the applicability of its language concepts for modeling the *coordination* required for the selection of modes of operation for a set of *communicating components*. Our aim is not to solve the "reliability problem" as stated above, i.e. the problem of selecting appropriate modes of operation in various contexts. That issue would require a more extensive analysis including non-functional properties.

The role of the PARADE tools throughout this case study has been twofold. Firstly, the tools have been used to keep an overview of the 22 processes, 16 partitions and 3 interaction protocols in the PARADIGM model – it is hard to manage a model of this size by hand. Secondly, the tools have been used to execute the model in its entirety and parts of it in isolation, and to gain insight into the behavior of the model and the interaction between its components. We present our findings with regard to the usage of the PARADE tools in the discussion at the end of the chapter.

The remainder of this chapter is set up as follows. In Section 6.2, we introduce the car navigation system and its components. In Section 6.3, we show how the communication between the components is modeled. In Section 6.4, we model the modes of operation of both the components and the system, and the coordination required to switch between them. Finally, in Section 6.5, we discuss the modeling approach and the insights obtained from the case study.

In this chapter, we focus on the most interesting parts of the PARADIGM model and therefore do not include the PARADIGM processes, partitions and interaction protocols which are less important in this respect – these can be found in Appendix B.

6.2 The Car Navigation System

An overview of the (conceptual) car navigation system considered in this case study is depicted in Figure 6.1. The system consists of seven components, each implementing a dedicated task. Three of the components are emphasized: *Route Calculator*, *Graphics Renderer* and *Voice Synthesizer*. These are the components considered in this case study. We model their behavior, their communication and their modes of operation, while we omit the remaining components and abstract from the interaction with them. For the sake of comprehension, however, we shortly introduce all components.

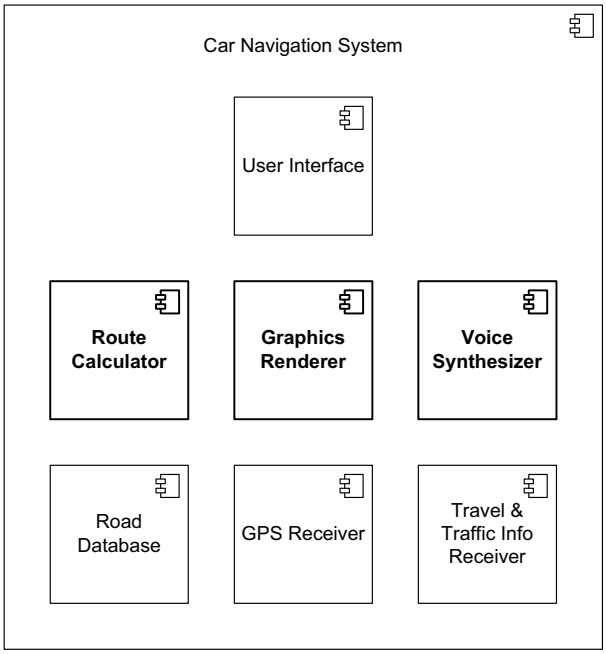


Figure 6.1: Overview of the Car Navigation System

The *Road Database* component provides access to a database containing a road map of the environment. It is a passive component that can be called to retrieve all road information about a specified region, or all road connections from a specific location. The *GPS Receiver* and *Travel and Traffic Info Receiver* components receive information about the location of the car and possible road blockades, like traffic queues and jams. These components actively manage their radio connections and can be called in order to query the actual location or information for a specific region. The *User Interface* component allows the user to enter a travel destination, or to change this destination on-the-fly. It is an active component, continuously monitoring user input and providing destination information to the Route Calculator as soon as the user has made up his mind. The *Route Calculator* component is able to calculate a route between the current location and the destination, potentially taking into account travel and traffic information. During traversal of the route, this component continuously provides information about the actions the driver should take and checks whether the driver adheres to these actions. It is able to recalculate the route if necessary. The *Graphics Renderer* component renders information from the Road Database and the Route Calculator into a graphical image on the display. The *Voice Synthesizer* component, finally, converts information about the actions to be taken into a voice sound with instructions for the driver.

The three components-in-scope provide several *component-level* modes of operation. These modes do not necessarily affect the interaction between the components and can in that case be selected independently from other modes. The following component-level modes of operation are supported:

- For the Route Calculator component, the usage of travel and traffic information for calculating an appropriate route is optional. In addition, this component can calculate either the shortest route or the quickest route.
- The Graphics Renderer component is able to render graphics in a two-dimensional or a three-dimensional layout.
- For the Voice Synthesizer component, it is possible to choose between standard voice mode or text-to-speech mode. In the former mode, only basic instructions are pronounced, like “turn left after 1 mile”, while in the latter mode, street and route names are pronounced as well.

The car navigation system as-a-whole supports four *system-level* modes of operation, in which certain components are enabled and others disabled. Mode “Fully Enabled” offers full functionality: all components are enabled. In Mode "No Voice", the Voice Synthesizer is disabled, while in Mode "No Route", also the Route Calculator is disabled (only the Graphics Renderer is active and shows the current location on a map). Finally, in Mode "No Graphics", the Graphics Renderer is disabled and the driver is only assisted by means of voice. Since these system-level modes of operation require that entire components be enabled or disabled, the communication between the components is thereby affected and careful coordination is required in order to ensure that switches from one mode of operation to another are performed correctly.

We address the communication between the components and the modeling of component-level and system-level modes of operation in Sections 6.3 and 6.4, respectively. In the remainder of this section, we introduce the detailed PARADIGM processes for the Route Calculator, the Graphics Renderer and the Voice Synthesizer. Each detailed process represents all possible behavior for a component: the internal activity, the supported modes of operation, and the communication with other components. We model modes of operation by means of *non-determinism*, to be resolved later on through the application of partitions and global processes. We model the communication between the components by including transitions with labels which indicate that certain information is to be received or sent. Note that we

do not specify the external source or target component in these labels. Finally, we explicitly model how components can be enabled and disabled at runtime, by adding transitions that skip parts of the behavior and directly lead to e.g. idle states.

Route Calculator

A PARADIGM process for the Route Calculator is shown in Figure 6.2 (as a notation convention, we combine the arrow heads of two or more transitions in a process if these transitions have the same transition label). This process behaves as follows. It starts with the acquisition of information about the destination to be reached, the current location of the car and current travel and traffic (T&T) information (this information could come from the User Interface, GPS Receiver and Travel and Traffic Info Receiver, respectively). Based on this information, the Route Calculator calculates either the shortest or the quickest route (two modes of operation, modeled via two transitions), and sends this information (to one or more unspecified components). After that, the situation at hand is checked and leads to one of five possibilities. If the destination has been reached, a message about this fact is sent and the process is repeated. If it turns out that the driver entered a different destination, the process starts again as well. In case the driver does not follow the established route for a certain amount of time, the choice is made to recalculate the route. If the driver should take some action, like turning to right or left, this information is sent (again: to one or more unspecified components). Finally, if there are no issues, no action is taken. In the last two cases, the process enters a subloop, in which it fetches the current location and potentially new travel and traffic information. Based on this new information, the current situation is checked again. This process repeats itself until the destination has been reached, the driver entered a new destination or a route recalculation is required. Note that the latter choice could also be made in case new travel and traffic information makes the established route impossible. At some states in the process, the component is able to return directly to the initial state. We will use these transitions later on in a separate *AsRunnable* partition in order to efficiently disable the Route Calculator at runtime.

Graphics Renderer

A detailed PARADIGM process for the Graphics Renderer is depicted in Figure 6.3. Its behavior is as follows. First, it gets the current location (potentially from the GPS Receiver). Then, it checks whether any route information or action information (from the Route Calculator) is available and if so, it reads this information. The route and action information is used to create visualizations, which are assumed to be reused until information about a new route or a new action becomes available. It is possible to remove these visualizations explicitly via a transition to state *Visualizations removed*. With a certain time interval, the component renders a region of the map of a predefined size, based on the current location, in either a two-dimensional or or a three-dimensional view of the map (modes of operation). Additional transitions to state *Start of rendering* are meant for disabling the component at runtime and for disabling the usage of route and action information.

Voice Synthesizer

Voice is used to instruct the driver about which actions to take in order to reach the desired destination. It is the task of the Voice Synthesizer component to synthesize a voice instruction based on information about the action to be taken. A special feature is the use of text-to-speech, which enables the pronunciation of e.g. street names.

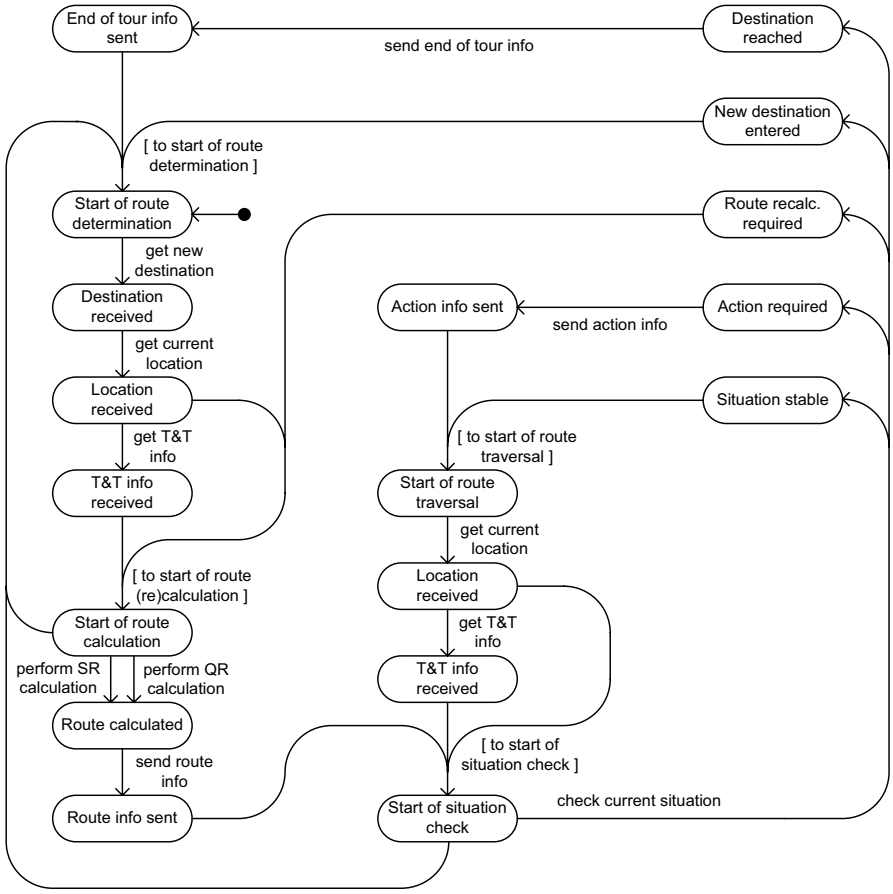


Figure 6.2: Process *RouteCalculator*

A PARADIGM process for the Voice Synthesizer component is depicted in Figure 6.4. The process behaves as follows. First, information about the action to be taken is being received. Then, this information is converted from text into speech, after which the voice message is constructed and sent to a digital-analog converter (DAC). Via transition *[to idle]* the process finally returns to the initial state. This process is repeated for each action that should be synthesized. In state *Action info received*, the process allows for skipping the text-to-speech conversion (modes of operation). Next to this, it is also possible to go from either state *Action info received* or state *TTS conversion performed* directly to state *Idle*, which will be used for disabling the Voice Synthesizer at runtime.

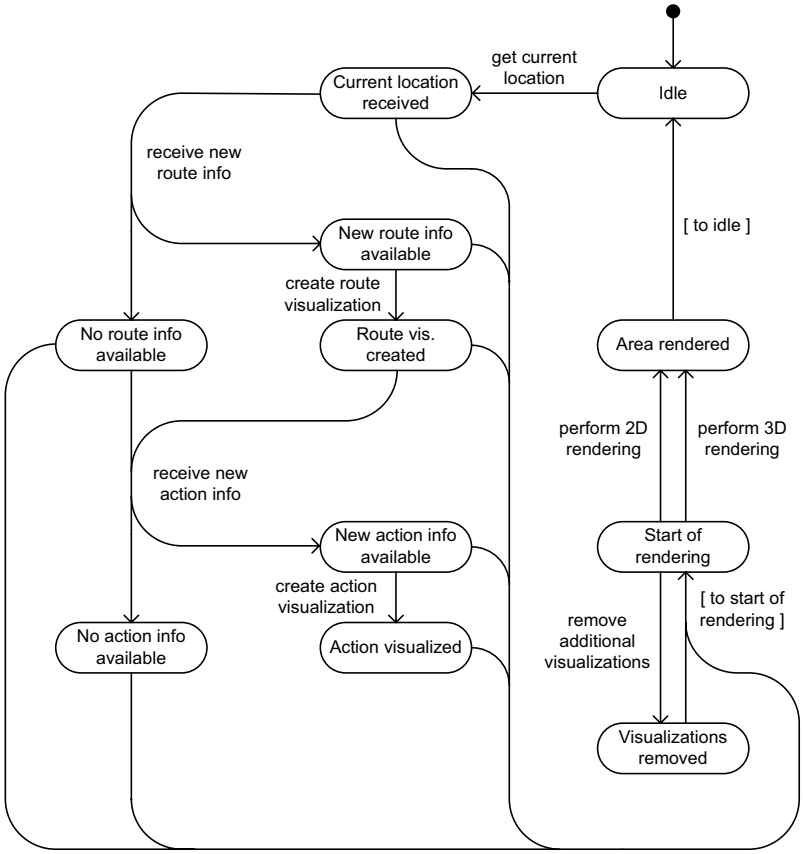


Figure 6.3: Process *GraphicsRenderer*

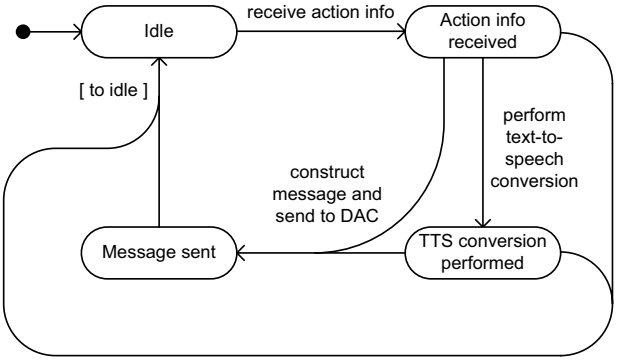


Figure 6.4: Process *VoiceSynthesizer*

6.3 Component Communication

Before we start modeling the modes of operation of the individual components and the system as-a-whole, we model how the communication between the three components takes place. The system contains three communication flows between the three components:

- The Route Calculator (transition *send route info*) provides route information to the Graphics Renderer (transition *receive new route info*);
- The Route Calculator (transition *send action info*) provides action information to the Graphics Renderer (transition *receive new action info*);
- The Route Calculator (again transition *send action info*) provides action information to the Voice Synthesizer (transition *receive action info*).

Since the communication between the three components consists of the exchange of either route information or action information, we decide to separate the coordination needed for the exchange of these two types of information by creating two separate interaction protocols. Moreover, because we do not wish to introduce any hierarchy between the three components, we decide to regard all of them as *employees* and add extra manager processes to coordinate the communication. An overview of the resulting PARADIGM model is shown in Figure 6.5. The Figure shows the Route Calculator (RC), the Graphics Renderer (GR) and the Voice Synthesizer (VS) with new partitions and global processes, which are coordinated by means of two new manager components *Route Info Manager* and *Action Info Manager* and two new interaction protocols *RouteInfoProtocol* and *ActionInfoProtocol*.

Partitions and Global Processes

On top of each of the existing components, we create partitions and global processes which abstract from internal activity and focus purely on the communication. We can easily distinguish between sender and receiver roles in the communication, since the Route Calculator only sends information and the other two components only receive information. Since we have chosen to model the exchange of route information and the exchange of action information separately, we apply this separation also to the partitions and global processes. The Route Calculator plays two sending roles in the communication, represented by partitions *AsRouteSender* and *AsActionSender*. The Graphics Renderer, which only receives information, plays two receiving roles via partitions *AsRouteReceiver* and *AsActionReceiver*. Finally, the Voice Synthesizer only receives action info and therefore plays one receiving role (partition *AsActionReceiver*). This makes a total of five new partitions. The global processes for each of the five partitions all have the same structure, as shown in Figure 6.6: they consist of two states *Active* and either *Sending* or *Receiving*, depending on whether the partition abstracts to resp. sending or receiving. Transitions are allowed from one state to the other and back.

As an example of how the partitions are defined on top of the detailed processes, we show partition *AsActionReceiver* for process *VoiceSynthesizer* in Figure 6.7. In subprocess *Active*, all detailed behavior is allowed except for the receiving of action information: transition *receive action info* to state *Action info received* is omitted. Trap *canReceive* of this subprocess, which only contains state *Idle*, indicates that the process is able to change to the phase in which it must receive action information. The second subprocess, *Receiving*, allows the process to receive action information. Trap *hasReceived* indicates that the process has finished receiving the action information and can be changed back to subprocess *Active*. Note that, within subprocess *Receiving*, the process is able to continue part of its activities after having received the action information.

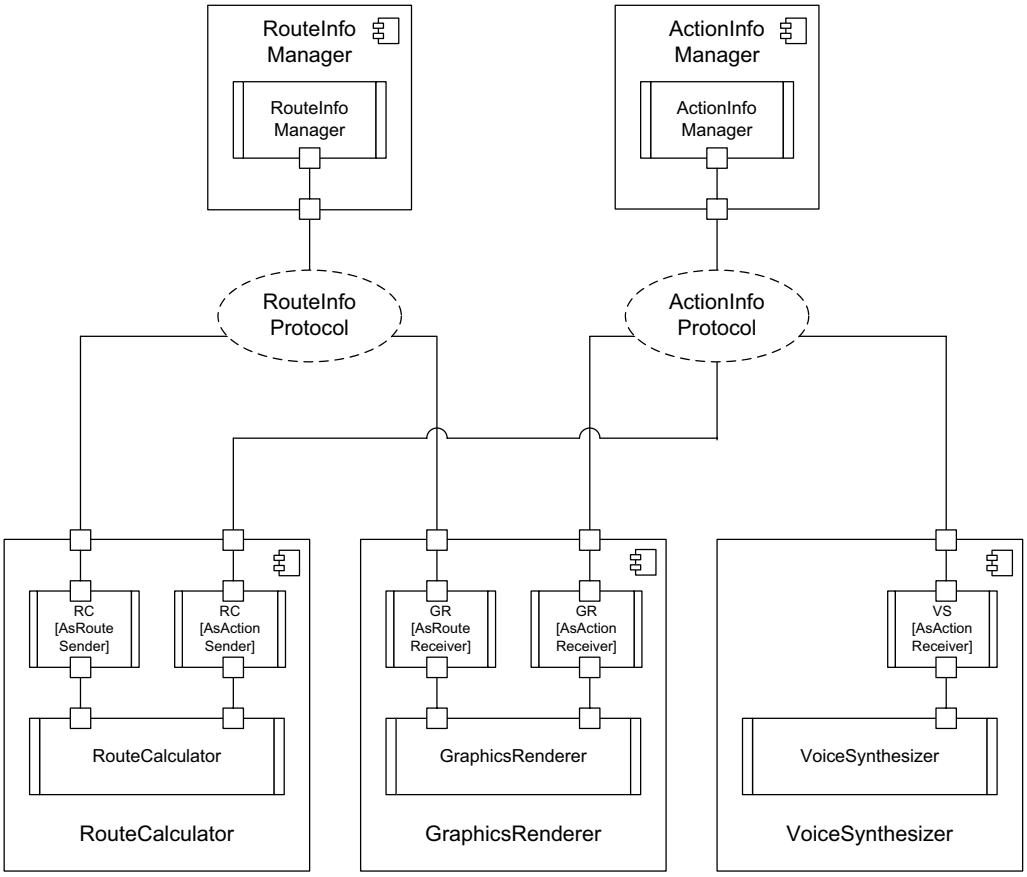


Figure 6.5: Managers and interaction protocols for the communication of route and action info

Manager Processes and Interaction Protocols

All five partitions for coordinating the communication between the three components have been organized in a similar manner, with one subprocess which disallows and another one which enforces the sending or receiving of information (the parts of the PARADIGM model not mentioned in this chapter can be found in Appendix B). Two distinct manager processes and interaction protocols take care of the coordination of these partitions. The *Route Info Manager* coordinates the communication of route information from the Route Calculator to the Graphics Renderer, while the *Action Info Manager* steers the flow of action information from the Route Calculator to both the Graphics Renderer and the Voice Synthesizer. We could specify the managers and interaction protocols in many different ways, e.g. such that they synchronize the global transitions of the sender and receiver(s), or such that they simulate asynchronous fifo channels or a shared data space between communicating peers. In our case, we have chosen to model them as one-place fifo buffers, which allows the three components to behave relatively independently, but ensures that a sending component blocks in case information sent earlier has not yet been received by one or more receiving components.

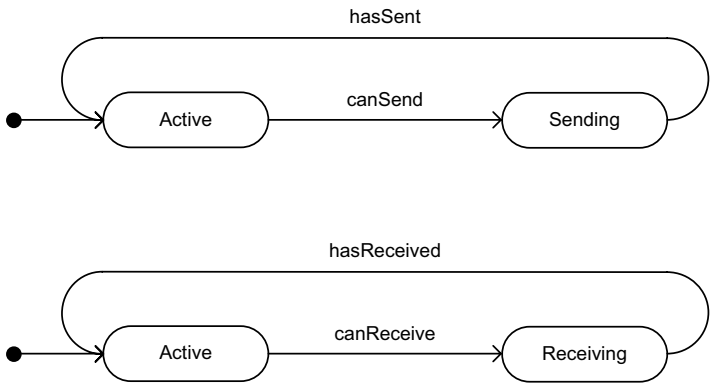


Figure 6.6: Global Processes *Sender* and *Receiver*

Manager processes *RouteInfoManager* and *ActionInfoManager* are depicted in Figure 6.8 and 6.9, respectively. They are modeled as active entities that receive data from a sender and forward this data to one or more receivers. The Route Info Manager simply sequentializes each send and receive, while the Action Info Manager, which has two receivers (Receiver 1 stands for Graphics Renderer, while Receiver 2 stands for Voice Synthesizer), sequentializes a send and two receives. Note that the latter process is a somewhat simplified version of a one-place fifo buffer, since we do not allow the receives to take place in a different order. Both processes have additional transitions by which they can arbitrarily skip forwarding to receivers. We will use these transitions in Section 6.4 to define a partition *AsReceiverSwitch* on top of both processes, which can be used to configure to which of the receivers the information is forwarded. Thereby, we are able to influence the communication flow for different system-level modes of operation.

Finally, we show the set of consistency rules anchored to interaction protocol *RouteInfoProtocol*, as an example of how the manager processes are related to the communicating components (the consistency rules anchored to interaction protocol *ActionInfoProtocol* are specified in a similar manner, only with two receivers instead of one). The consistency rules are shown in Table 6.10. They specify how transitions of the global processes for the *AsRouteSender* and *AsRouteReceiver* partitions are synchronized with manager transitions of process *RouteInfoManager*. Note in particular rule *RI3*, which ensures that the Route Calculator is put back into subprocess *Active* even if the Route Info Manager skips forwarding the data to the Graphics Renderer (in that case, the data in the buffer is assumed to be thrown away and the buffer is considered empty again). However, after the Route Info Manager has taken transition *send data to receiver* and has entered state *Sending data*, it cannot proceed to state *Idle* as long as employee process Graphics Renderer has not entered trap *hasReceived*. We will take this restriction into account in Section 6.4, where we define the coordination for a system-level mode of operation in which the Graphics Renderer will be disabled.

We have now shown the detailed processes for the three car navigation components plus the processes, partitions and interaction protocols we use for coordinating their communication. This combination constitutes our model of the car navigation system, but still *without* any coordination mechanism for the modes of operation. This is the topic of the next section.

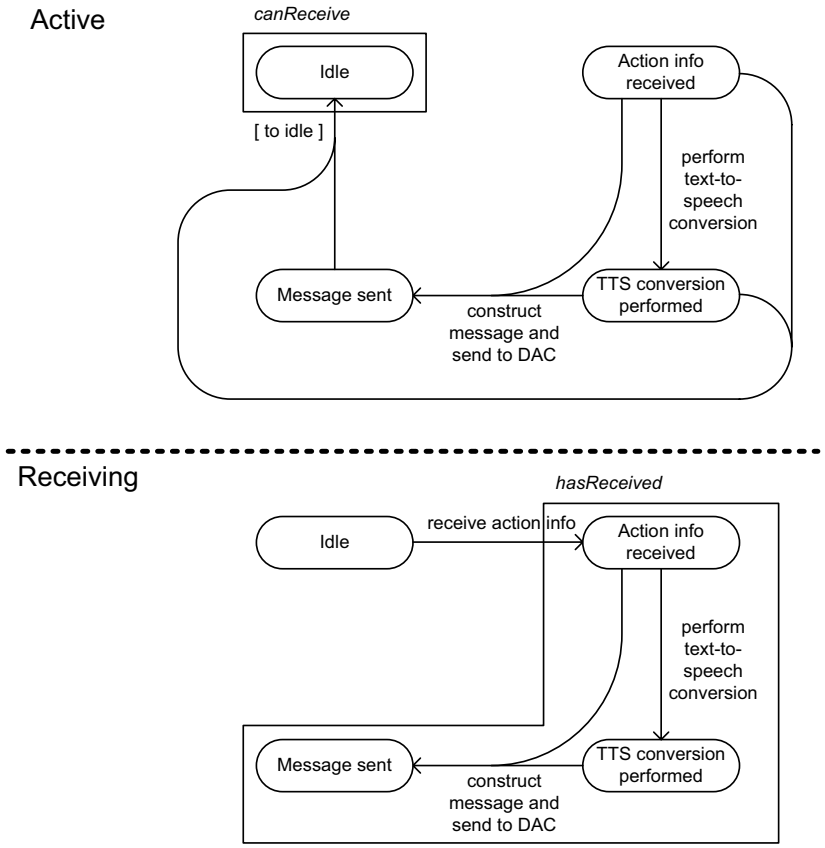


Figure 6.7: Partition *AsActionReceiver* for Process *VoiceSynthesizer*

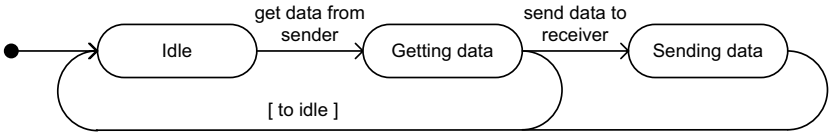


Figure 6.8: Process *RouteInfoManager*

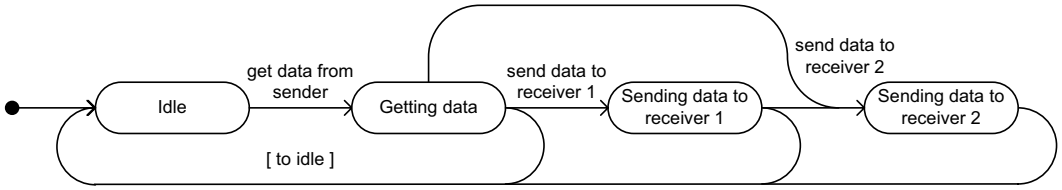


Figure 6.9: Process *ActionInfoManager*

(RI1)		RouteInfoManager :	Idle	—get data from sender→	Getting data
	∗	RC[AsRouteSender] :	Active	—canSend→	Sending
(RI2)		RouteInfoManager :	Getting data	—send data to receiver→	Sending data
	∗	RC[AsRouteSender] :	Sending	—has sent→	Active,
		GR[AsRouteReceiver] :	Active	—canReceive→	Receiving
(RI3)		RouteInfoManager :	Getting data	—[to Idle]→	Idle
	∗	RC[AsRouteSender] :	Sending	—hasSent→	Active
(RI4)		RouteInfoManager :	Sending data	—[to Idle]→	Idle
	∗	GR[AsRouteReceiver] :	Receiving	—hasReceived→	Active

Table 6.10: Consistency Rules anchored to Interaction Protocol *RouteInfoProtocol*

6.4 Modes of Operation

The general way in which we model modes of operation in PARADIGM is as follows. We view a mode of operation as a configuration of an orthogonal set of *switches*. A switch provides a set of *options* for a component, of which one may be selected at a time. For example, the Route Calculator basically provides two switches (the *TTSwitch* and the *CMSwitch*), each with two options, as shown below.

		TTSwitch	
		Travel & Traffic On	*Travel & Traffic Off*
CMSwitch	*Shortest Route*	Mode 1	Mode 2
	Quickest Route	Mode 3	Mode 4

Switch *TTSwitch* provides two options, which respectively enable and disable the usage of travel and traffic information for the calculation of the route. Independently of this, switch *CMSwitch* configures the calculation mode for the component, with two options shortest route or quickest route. In total, the two switches with each two options provide four different modes of operation.

As we indicated in Section 6.2, we distinguish between *component-level* modes of operation, which are configurations of the behavior and functionality of individual components (as in the example in the table above), and *system-level* modes of operation, which are configurations of the functionality of the system as-a-whole. For component-level modes of operation, we model the switches as individual *partitions* with corresponding *global processes* on top of the detailed process of a component. Each global process provides a view on its detailed process which only shows the options and how one can switch between them. Since PARADIGM allows for multiple partitions to be defined on top of a single detailed process, each of the switches can be easily modeled as a separate partition. System-level modes of operation are more complex to model. Since these modes of operation may enable and disable the car navigation components, we model their coordination via an additional component. For this purpose, we define a *Car Navigation System Manager*, *CNS Manager* for short, which coordinates the car navigation components plus the additional components defined in Section 6.3 for managing the communication. The four system-level modes of operation defined earlier in Section 6.2 are modeled using a single partition and global process on top of this CNS Manager, as a system-level switch with four options. A convenient way to model the enabling and disabling of the car navigation components is by creating an additional partition and global process for each of these components, next to the switches for component-level modes of operation. The additional partition, which we call *AsRunnable*, can be used to *run* and *pause* a component.

In the next two subsections, we show modeling examples of the component-level resp. system-level modes of operation on top of the PARADIGM model of Section 6.3.

Component-level Modes of Operation

As an example, consider the Voice Synthesizer component, for which the detailed process was shown earlier in Figure 6.4. This component provides two modes of operation, one in which text-to-speech conversion is applied to the action information, and one in which this conversion is omitted. These two modes can be modeled as a single switch with two options. The partition for this switch is shown Figure 6.11, the corresponding global process in Figure 6.12.

A switch between the two modes of operation can be performed at runtime, and we prefer to be able to switch regardless of the precise state in which the detailed process will be. Therefore, both subprocesses of partition *AsTTSSwitch* and their traps contain all states of the detailed process. In subprocess *TTS ON*, the process is not allowed to skip the text-to-speech conversion: transition *construct message and send to DAC* from state *Action info received* is excluded. In subprocess *TTS OFF*, it is not allowed to perform the text-to-speech conversion, hence transition *perform text-to-speech conversion* is ommitted. Note however, that all remaining transitions are included in the subprocess: should the process enter or be in state *TTS Conversion performed* at the moment a switch to *TTS OFF* has just been made, it is always able to leave this state. Note that, according to the semantics of PARADIGM, if the process is in subprocess *TTS ON* and is changed to subprocess *TTS OFF* while transition *perform text-to-speech conversion* is being taken, the transition will be taken and the process will enter state *TTS conversion performed*. Only thereafter, the process can be said to actually be in subprocess *TTS OFF*.

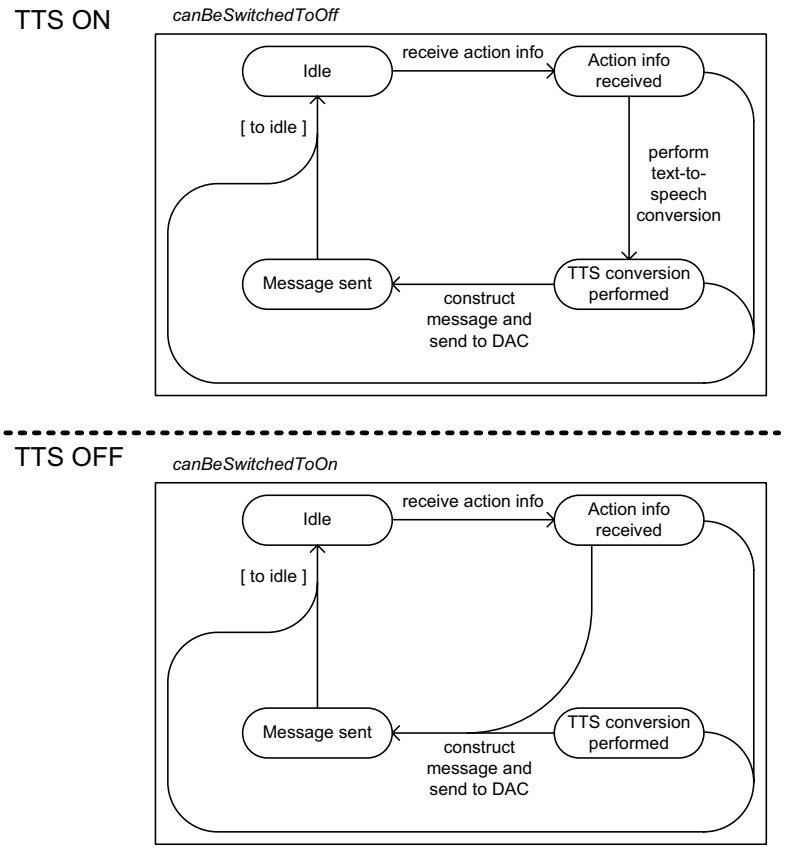


Figure 6.11: Partition *AsTTSSwitch* for Process *VoiceSynthesizer*

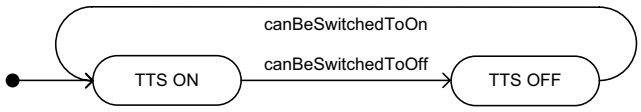


Figure 6.12: Global Process *VS[AsTTSSwitch]*

Since we also require the possibility for the Voice Synthesizer to be enabled and disabled at runtime in order to facilitate the system-level modes of operation, we add another partition *AsRunnable*, as shown in Figure 6.13. Corresponding global process *Runnable* is shown in 6.14. The partition has been defined such that the process can be paused regardless of its current state. In subprocess *Paused*, it will only be allowed to take transitions to state *Idle*. The trap definition for trap *activatable* allows for changing back to subprocess *Running* only if the process has entered state *Idle*, which prohibits the process from starting its execution in some arbitrary state after it has been paused.

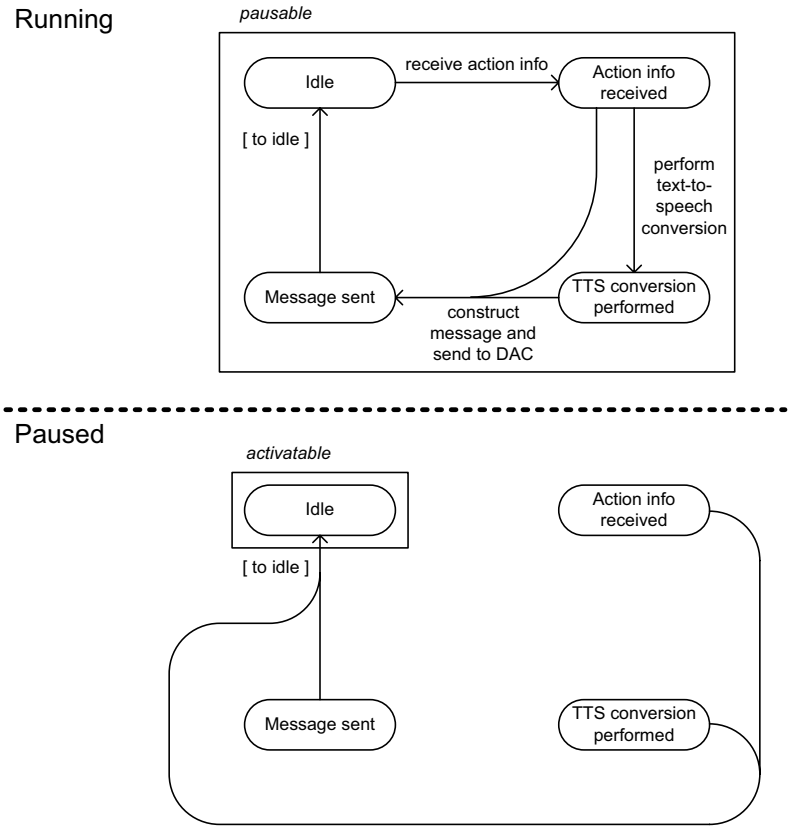


Figure 6.13: Partition *AsRunnable* for Process *VoiceSynthesizer*

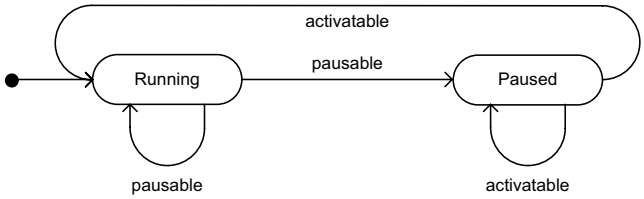


Figure 6.14: Global Process *Runnable*

Note the loops in global process *Runnable*, which allow for checking whether trap *pausable* or *activatable* have been entered without performing a subprocess change. We have applied these kind of loops earlier in the self-managing interaction protocol example of Chapter 4.

In total, we have now defined three different partitions on top of component *Voice Synthesizer*: partition *AsTTSSwitch*, partition *AsRunnable*, and partition *AsActionReceiver* defined earlier in Section 6.3 for coordinating the communication of action information. The resulting component is depicted in Figure 6.15.

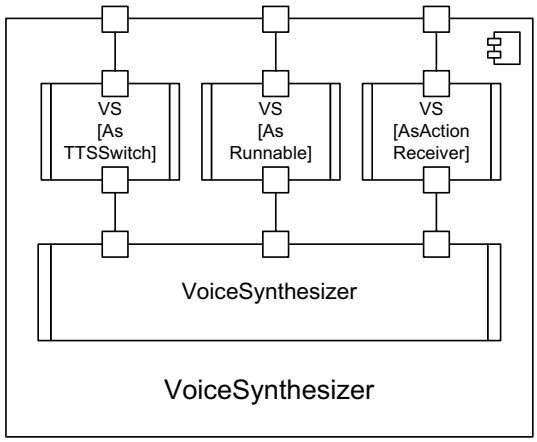


Figure 6.15: Component *Voice Synthesizer*

If we compare the three partitions to each other, the dependencies between them become clear. First, note that subprocess changes for partition *AsTTSSwitch* are always possible, regardless of the subprocesses in which the other two partitions reside. More importantly, there is a dependency between partition *AsRunnable* and partition *AsActionReceiver*: partition *AsRunnable* can be in subprocess *Paused* while partition *AsActionReceiver* is in subprocess *Receiving*. In that case, if the detailed process is in state *Idle*, partition *AsActionReceiver* cannot be changed back to subprocess *Active* until partition *AsRunnable* has been changed to *Running* and the process is able to enter trap *hasReceived*. This dependency relates directly to the discussion in Section 6.3 about interaction protocol *RouteInfoProtocol*. Again, we will take it into account when we define the coordination for the system-level modes of operation, in order to ensure the liveness of the system and its components at runtime.

Partitions and global processes for the Route Calculator and Graphics Renderer component are modeled in a similar manner as those for the Voice Synthesizer – they can be found in Chapter B. For both components, we define a partition *AsRunnable* to run and pause them. For component *Route Calculator*, we define two partitions *AsTTSwitch* and *AsCMSwitch* to switch the usage of travel and traffic info and the calculation mode, respectively. For component *Graphics Renderer*, we add a partition *AsRMSwitch* for the rendering mode (2D or 3D). All these partitions have global processes similar to global process *VS[AsTTSSwitch]*, with two states, one for each configurable option.

As we explained in Section 6.2, the Graphics Renderer includes the possibility to remove route and/or action visualizations by taking a transition to state *Visualizations removed*. This removal has to take place once the Route Calculator component is being paused, since otherwise the Graphics Renderer keeps showing the last route calculated even though the Route Calculator is inactive. A partition *AsRCUsageSwitch* is added to the Graphics Renderer, whose corresponding Global Process has three states: initial state *RC ON*, in which the Route Calculator is assumed to be running, state *Cleanup* in which

the outdated visualizations are removed, and state *RC OFF*, in which the Route Calculator is assumed to be disabled. Although, the partition itself is fairly simple (therefore, we do not show it), it plays an important role in the coordination needed for switching to system-level mode of operation "No Route", as we will show in the next subsection.

An overview of the PARADIGM model including all partitions defined for the three car navigation components is shown in Figure 6.16. For reasons of clarity, the partitions are shown as component ports only (we used a similar visualization in [90] for *CMT2*, a modeling technique for component based software design). Some ports are shown in gray: the partitions and global processes represented by these ports will play no role in the coordination of the system-level modes of operation. Others are shown in black: these ports are used for coordinating the communication between the components. The white ports, which are not yet connected to any interaction protocol, play a role in the next subsection, where we will introduce a new manager process *CNSManager* together with an interaction protocol *CNSProtocol* to coordinate the system-level modes of operation. In addition, we will create partitions on top of processes *RouteInfoManager* and *ActionInfoManager*, which will be used to avoid conflicts between the communication in the system and the coordination of the system-level modes of operation.

Note that we leave unspecified in the model how and when the transitions of global processes *RouteCalculator[AsCMSwitch]*, *RouteCalculator[AsTTSwitch]*, *GraphicsRenderer[AsRMSwitch]* and *VoiceSynthesizer[AsTTSwitch]* are taken. In fact, this makes the PARADIGM model for the car navigation system an *open* model, in contrast with the closed models we have seen so far.

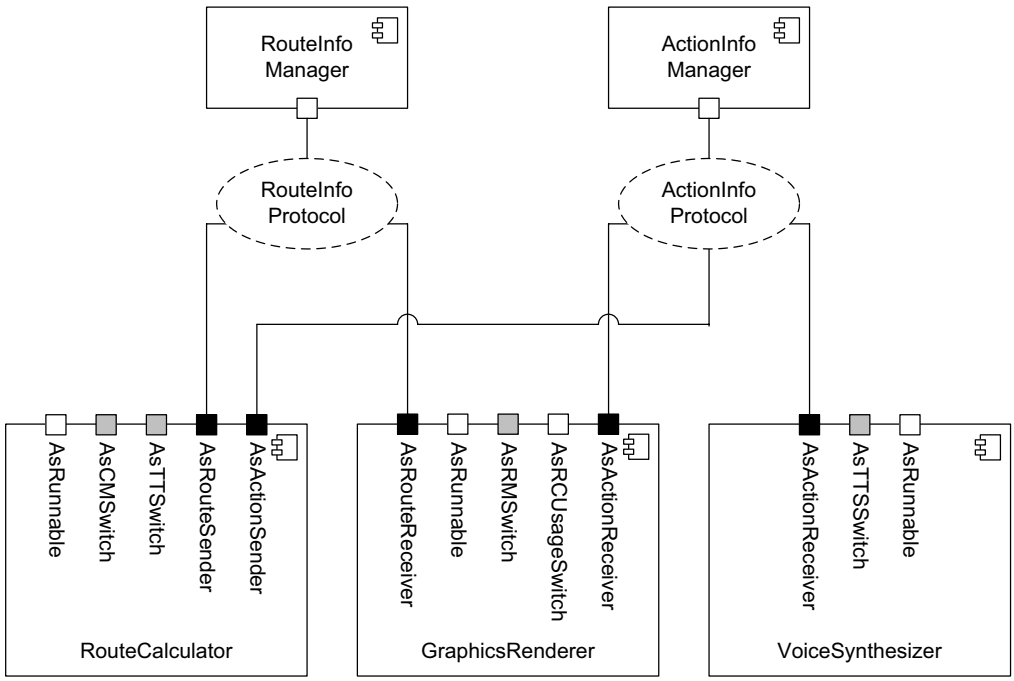


Figure 6.16: Car navigation system with communication and component-level modes of operation

System-level Modes of Operation

Switching between the system-level modes of operation requires careful coordination, especially when components, which continuously communicate with each other, are temporarily disabled (paused). We recall our earlier remarks about the way in which we modeled the communication between the components, and about the dependencies between this communication and the enabling and disabling of components. In order to avoid that processes block undesirably, we do not only need to coordinate the component-level modes of operation and the *AsRunnable* partitions of the components, but also the communication flows between them. To this end, we define a partition *AsReceiverSwitch* on top of the two manager processes for the communication, *ActionInfoManager* and *RouteInfoManager*. For the Route Info Manager, the partition simply defines a choice (two states) between forwarding the route information from the sender to the receiver or not. This partition and the corresponding global process can be found in Appendix B. For the Action Info Manager, which has two receivers, the partition defines to which of both receivers the information must be forwarded to: none, receiver 1 (Graphics Renderer), receiver 2 (Voice Synthesizer), or both. We show partition *AsReceiverSwitch* for process *ActionInfoManager* and the corresponding global process in Figures 6.17 and 6.18. Note again the usage of loops in the global process to check whether trap *changeable* has been entered.

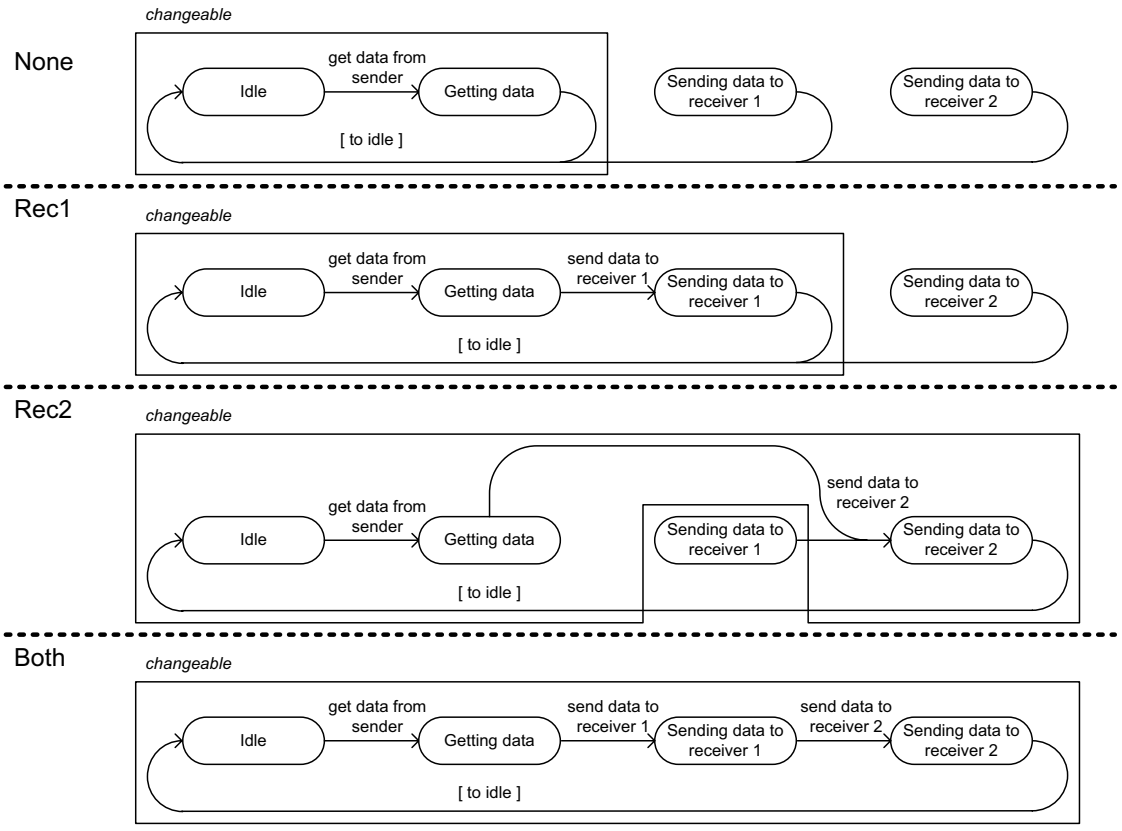


Figure 6.17: Partition *AsReceiverSwitch* for Process *ActionInfoManager*

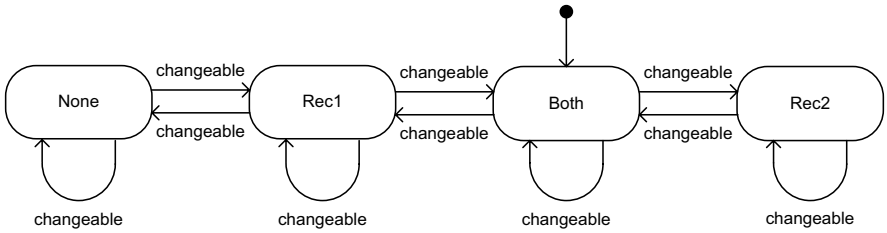


Figure 6.18: Global Process *ActionInfoManager[AsReceiverSwitch]*

We remark that, even if forwarding to both receivers is disabled (subprocess *None*), the process is still able to accept information from the sender. This way, process *RouteCalculator* never blocks upon sending, except when forwarding is enabled and manager process *ActionInfoManager* waits for a receiver to receive the information. Partition *AsReceiverSwitch* for process *RouteInfoManager* has the same property.

The coordination needed for the four system-level modes of operation can now be defined on top of the car navigation components and the components which coordinate the communication between them. As a first step, we define an additional component *CNSManager*. The detailed process for this component is shown in Figure 6.19. The process specifies which actions have to be taken by the system in order to switch from one system-level mode of operation to another. Just as we did for the component-level modes of operation, we define a partition *AsModeSwitch* on top of this detailed process, which represents a switch for four different options (see Figure 6.20) . The global process for this partition (Figure 6.21) has four states which precisely correspond to the four system-level modes of operation. The states of this process are not fully connected to each other, hence, one cannot switch arbitrarily between the system-level modes of operation. This is a modeling choice rather than a technical limitation.

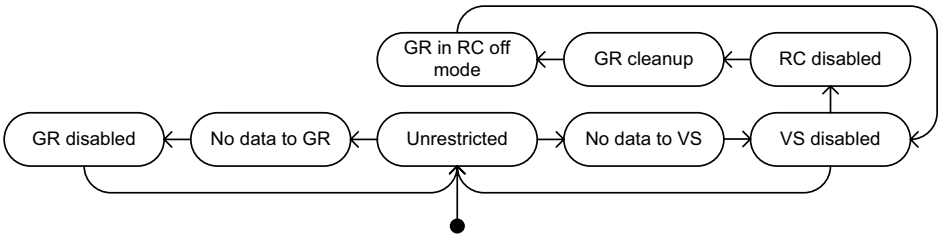


Figure 6.19: Process *CNSManager*

Detailed process *CNSManager* shows which intermediate steps are taken in order to switch from one mode of operation to another. Each time a component is disabled which acts as a receiver of either route or action information, we make sure that we first disable the communication flow to that component prior to disabling the component, in order to prevent the communication manager from blocking upon forwarding the information.

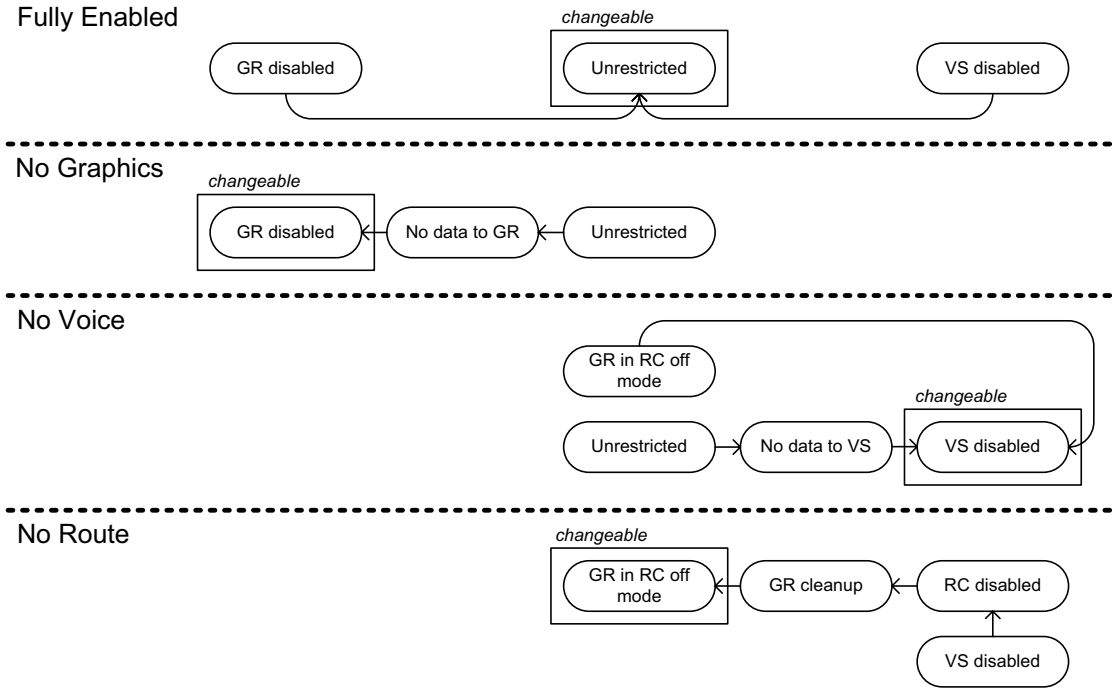


Figure 6.20: Partition *AsModeSwitch* for Process *CNSManager*

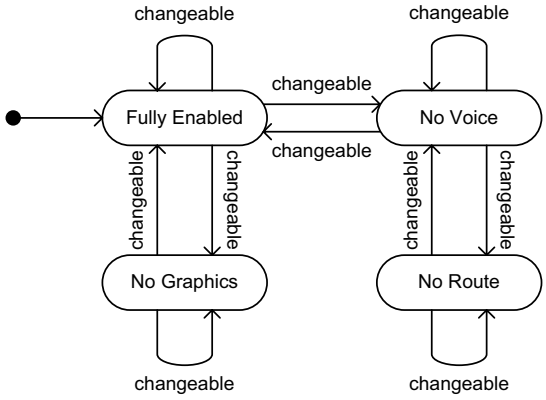


Figure 6.21: Global Process *CNSManager[AsModeSwitch]*

Consider for example the switch from *Fully Enabled* to *No Voice*. We assume that global process *CNSManager[AsModeSwitch]* is in state *Fully Enabled* and process *CNSManager* is in state *Unrestricted*, hence in trap *changeable* of its single partition. In the global process, this enables the transitions from state *Fully Enabled* to either *No Voice* or *No Graphics*. Suppose that global process *CNSManager[AsModeSwitch]* takes a transition to state *No Voice*. Thereby, process *CNSManager* is changed to subprocess *No Voice*, which enables it to enter state *No data to VS*. This transition disables the communication between the Route Calculator and the Voice Synthesizer (interaction protocol *CNSProtocol*, presented later on, will show how). Only after that, the process takes the transition to state *VS disabled*, which disables (pauses) the Voice Synthesizer. Once state *VS disabled* (trap *changeable*) has been entered, the system has been correctly configured for mode of operation *No Voice*.

Interaction protocol *CNSProtocol*, finally, binds the system-level modes of operation to the component-level modes of operation in terms of consistency rules managed by process *CNSManager*. The consistency rules anchored to it are shown in Tables 6.22 and 6.23. In Table 6.22, the consistency rules for enabling and disabling the Graphics Renderer or the Voice Synthesizer are shown. Table 6.23 shows the consistency rules for correctly enabling and disabling the Route Calculator.

We continue the example of the system-level mode change from *Fully Enabled* to *No Voice*. The detailed mode changes can be found in Rules *V1* and *V2*. In consistency rule *V1*, first, the Action Info Manager component is changed from mode *Both* into mode *Rec1* via the *AsReceiverSwitch* partition. This causes the Action Info Manager to stop sending action information to the Voice Synthesizer component. Note that the Route Calculator component is not restricted in any way in its behavior and that the flow of action information from the Route Calculator to the Graphics Renderer component is still enabled. After consistency rule *V1* has been applied, process *CNSManager* is in state *No data to VS*. In order to be sure that the Action Info Manager is in mode *Rec1* before we disable the Voice Synthesizer component, we have added the restriction to the next consistency rule, *V2*, that process *ActionInfoManager* must be in trap *changeable* of subprocess *Rec1*. This ensures that the Action Info Manager has no pending information for the Voice Synthesizer component after the latter one is disabled. After Rule *V2* has been applied, the Action Info Manager has been reconfigured and the Voice Synthesizer has been disabled. During the reconfiguration, the execution of the other components has not been interrupted in any way. The *CNSManager* process is now in state *VS Disabled* and thereby enters trap *changeable*. It is now possible to switch to another system-level mode, e.g. *No Route* or *Fully Enabled*.

The consistency rules for disabling the Graphics Renderer component are similar to those for disabling the Voice Synthesizer. A switch from mode of operation “No Voice” to mode of operation “No Route”, however, requires the disabling of the Route Calculator component, which is a sending instead of a receiving component. The consistency rules involved in this reconfiguration are mentioned in Figure 6.23 (rules *R1*, *R2* and *R3*). Rule *R1* disables all communication between the three components. Next to that, it disables the Route Calculator component. These two actions are not conflicting and can be performed in one rule. In Rule *R2*, we make sure that all communication flows have been disabled by checking whether trap *changeable* has been entered by both the Route Info Manager and the Action Info Manager. In that case, the Graphics Renderer can start cleaning up the temporary reusable visualizations. Once this has been done, rule *R3* makes sure that the Graphics Renderer starts showing a map without a route from the Route Calculator.

(G1)		CNSManager :	Unrestricted	——→	NoDataToGR
	∗	RM[AsReceiverSwitch] :	Enabled	changeable ——→	Disabled,
		AM[AsReceiverSwitch] :	Both	changeable ——→	Rec2
(G2)		CNSManager :	NoDataToGR	——→	GRDisabled
	∗	RM[AsReceiverSwitch] :	Disabled	changeable ——→	Disabled,
		AM[AsReceiverSwitch] :	Rec2	changeable ——→	Rec2,
		GR[AsRunnable] :	Running	pausable ——→	Paused
(G3)		CNSManager :	GRDisabled	——→	Unrestricted
	∗	RM[AsReceiverSwitch] :	Disabled	changeable ——→	Enabled,
		AM[AsReceiverSwitch] :	Rec2	changeable ——→	Both,
		GR[AsRunnable] :	Paused	activatable ——→	Running
(V1)		CNSManager :	Unrestricted	——→	NoDataToVS
	∗	AM[AsReceiverSwitch] :	Both	changeable ——→	Rec1
(V2)		CNSManager :	NoDataToVS	——→	VSDisabled
	∗	AM[AsReceiverSwitch] :	Rec1	changeable ——→	Rec1,
		VS[AsRunnable] :	Running	pausable ——→	Paused
(V3)		CNSManager :	VSDisabled	——→	Unrestricted
	∗	AM[AsReceiverSwitch] :	Rec1	changeable ——→	Both,
		VS[AsRunnable] :	Paused	activatable ——→	Running

Table 6.22: Consistency rules anchored to interaction protocol *CNSProtocol*

Final PARADIGM Model

The final PARADIGM model for the car navigation system with communication, component-level and system-level modes of operation is shown in Figure 6.24. Again, we remark that the model is *open*: the gray ports represent partitions which can be coordinated via interaction protocols external to the model. The figure shows clearly that component communication and coordination of the modes of operation have been modeled separately. This makes it possible to experiment with various communication scenarios and analyze whether interaction protocol *CNSProtocol* is able to ensure continuity of correct service in all system-level modes of operation.

(R1)		CNSManager:	VSDisabled	⟶	RCDisabled
				pausable	
	*	RC[AsRunnable]:	Running	⟶	Paused,
				changeable	
		RM[AsReceiverSwitch]:	Enabled	⟶	Disabled,
				changeable	
		AM[AsReceiverSwitch]:	Rec1	⟶	None
(R2)		CNSManager:	RCDisabled	⟶	GRCleanup
				changeable	
	*	RM[AsReceiverSwitch]:	Disabled	⟶	Disabled,
				changeable	
		AM[AsReceiverSwitch]:	None	⟶	None,
				canBeSwitchedToCleanup	
		GR[AsRCSwitch]:	RCOn	⟶	Cleanup
(R3)		CNSManager:	GRCleanup	⟶	GRinRCOffMode
				canBeSwitchedToOff	
	*	GR[AsRCSwitch]:	Cleanup	⟶	RCOff
(R4)		CNSManager:	GRinRCOffMode	⟶	VSDisabled
				activatable	
	*	RC[AsRunnable]:	Paused	⟶	Running,
				changeable	
		RM[AsReceiverSwitch]:	Disabled	⟶	Enabled,
				changeable	
		AM[AsReceiverSwitch]:	None	⟶	Rec1,
				canBeSwitchedToOn	
		GR[AsRCSwitch]:	RCOff	⟶	RCOn

Table 6.23: More consistency rules anchored to interaction protocol *CNSProtocol*

6.5 Discussion

In the case study presented in this chapter, we have modeled the behavior and interaction of a multi-component car navigation system. We distinguished two different aspects of interaction: the direct communication between components on the one hand, and the more implicit interaction caused by switching system-level modes of operation on the other hand. By the application of partitions, we were able to separately model the role played by an individual component in each of the two aspects of interaction. The interaction itself, modeled in PARADIGM by means of consistency rules, could be conveniently structured into multiple interaction protocols and managed by hierarchically organized manager processes.

In our opinion, this demonstrates that partitions and interaction protocols complement each other in structuring the interaction in PARADIGM models. Partitions provide structure from the perspective of individual *components* and their *operation*, while interaction protocols provide structure from the perspective of the *composition* and the *co-operation* within it. Thereby, we believe to have shown the validity of our choice to extend the PARADIGM language with the interaction protocol concept (see Chapter 4). This extension strengthens the applicability of the language for modeling interaction in general, not only from the perspective of components but also from the perspective of the composition.

We also point out the modeling of interaction in this case study at multiple levels in the manager-employee hierarchy. The *Action Info Manager* and *Route Info Manager* components, together with their

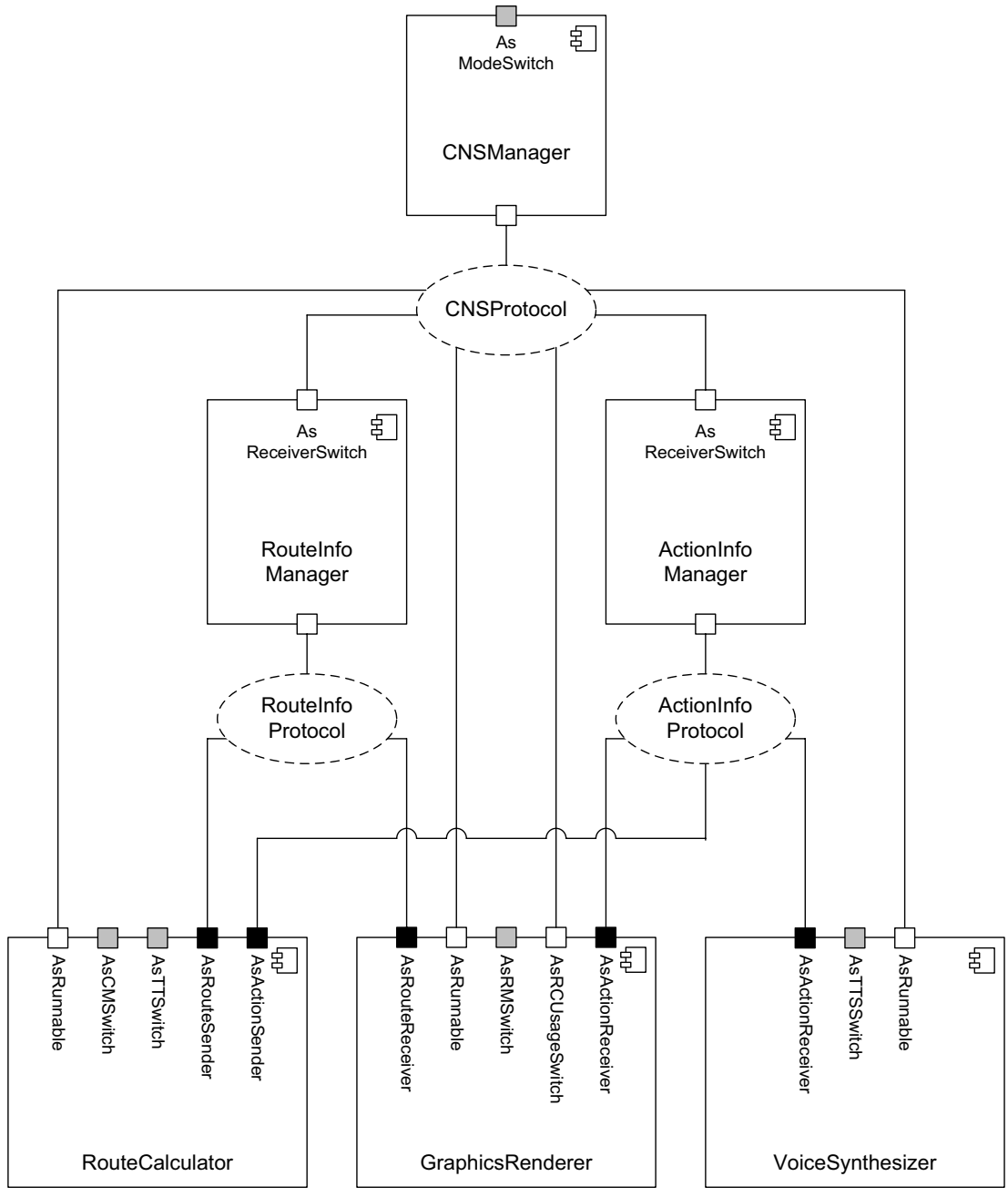


Figure 6.24: Structure of the car navigation system with communication and all modes of operation

(low-level) interaction protocols, abstract from the way in which communication between the three car navigation components takes place. They can be said to "componentize" the communication flows between the components, thereby realizing a convenient interface to the (higher-level) interaction protocol *CNSProtocol*. This way, the latter one can be only concerned with the correct enabling and disabling of components and their communication flows, while manager process *CNSManager* provides the necessary sequentializations for this. Similar to the way in which the car navigation components offer their modes of operation, we have put a partition on top of process *CNSManager* to provide a coherent set of system-level modes of operation to the environment.

Throughout the case study, we have extensively used the PARADE tools to create, execute and validate the PARADIGM model. The full model is relatively large, which makes it hard to manually keep all parts of the model consistent with each other. In this respect, an especially useful feature of the tools is that they enable us to create a PARADIGM model *incrementally*, and to execute individual components and partial models in the PARADE distributed runtime environment. Since all processes are treated in an equal manner in the PARADISE distributed interpreter framework, a global process which is not connected to an interaction protocol can be executed as if it were a detailed process with a simple selector. Thereby, it is possible to execute individual components in isolation and to validate their internal partition constraints. Furthermore, this enables the execution of *open* PARADIGM models, like the model presented in this chapter: global processes *CNSManager[AsModeSwitch]*, *RouteCalculator[AsCMSwitch]*, *RouteCalculator[AsTTSwitch]*, *GraphicsRenderer[AsRMSwitch]* and *VoiceSynthesizer[AsTTSwitch]* can be executed with the tools by using simple selectors in the interpreter for them instead of delegating selectors.

Chapter 7

Branch-and-Bound Algorithms

We use PARADIGM to model a generic branch-and-bound algorithm in three different shapes: sequential, componentized, and parallel. The case study shows two interesting applications of the PARADIGM concepts. Firstly, we apply self-managing interaction protocols to perform the coordination of the componentized and parallel algorithm. We show that the amount of parallel processes in the algorithm can be adapted by applying changes to these self-managing interaction protocols only, i.e. no changes to any of the processes or partitions in the model are required. Secondly, we model the communication between components of the algorithm strictly separate from the algorithm itself, by means of partitions on top of detailed processes. Finally, we illustrate the use of the PARADE tools to combine the different versions of the PARADIGM model for branch-and-bound with the implementation of a branch-and-bound solver. This enables a more elaborate analysis of the relationship between the generic PARADIGM model and the characteristics of a specific branch-and-bound problem.

7.1 Introduction

Branch-and-bound algorithms [96, 98] are a popular variant of backtrack algorithms, often applied to large scale NP-hard combinatorial optimization problems. They exploit knowledge about the feasibility of partial solutions in order to speed up the search for optimal complete solutions. These algorithms are named "branch-and-bound" because of the basic activities within the algorithm. Given a certain problem, the algorithm searches for a solution by dividing the problem into smaller subproblems (branch) and trying to solve these subproblems. While searching for solutions, the algorithm calculates optimality measures (*upper bound* and *lower bound*) for each of the subproblems. These bounds can be used to determine the search order through the search tree, to compare the quality of different solutions, and to eliminate subproblems which are guaranteed not to lead to a better solution. For comparison, a global *least upper bound* is maintained, which indicates the best upper bound found so far.

Branch-and-bound algorithms can be implemented in many ways [13, 19]. Several choices can be made, like which search strategy to apply, how to eliminate subproblems or when to terminate the algorithm. If a *parallel* branch-and-bound algorithm is considered, even more issues play a role [96, 68], e.g. the kind of parallelization that will be used, or the way in which the various parallel processes communicate. Despite these additional concerns, the essential activities of the algorithm are the same.

We base the case study presented in this chapter on earlier work published in [91], in which we implemented a generic branch-and-bound algorithm using the exogenous coordination language MANIFOLD and the IWIM coordination model [3, 4, 11]. In [91], we focused on finding the appropriate characterization of generic components for parallel branch-and-bound, such that we were able to exploit exogenous coordination to make these components unaware of their context. As a direct benefit of this, the global organization of the implementation can be changed without the necessity to change the implementation of the individual components in any way. In this PARADIGM case study, we reuse the insights gained in [91] and use a similar organization of the componentized and parallel version of the PARADIGM model. However, here we focus on the *behavior* of the individual components and the interaction that results from parallelizing the sequential algorithm.

The overall idea of this case study is as follows. We model three versions of the same algorithm: a *sequential*, a *componentized* and a *parallel* version. In the sequential version, we model a branch-and-bound algorithm as a single PARADIGM process. The componentized version builds upon the sequential one: we split the single PARADIGM process into three separate processes: an *Initializer*, which creates the initial problem to be solved, a *Pool Manager*, which maintains a pool of subproblems to be visited, and a *Visitor*, which analyzes subproblems. We coordinate the processes by means of partitions and a *self-managing interaction protocol* (see Chapter 4), in such a way that the processes collaboratively implement a sequential algorithm. In the parallel version, we extend the componentized version by replicating the *Visitor* process, thereby realizing a parallel branch-and-bound algorithm. As in [91], the number of parallel *Visitor* processes in the model can be changed without any change to the processes or partitions (which are considered to be part of the components). The additional effort needed for the parallel version consists of modeling the communication needed for the exchange of the global *least upper bound* (LUB) between the replicated *Visitor* processes. For this, we use an additional process *LUBManager*, and a separate self-managing interaction protocol for its coordination.

In this chapter, we devote a separate section to the usage of the PARADE tools. They play an important role in the case study for two reasons. Firstly, they enable us to model and execute the PARADIGM model and to visualize their execution. Secondly, we use the tools to attach the implementation of a specific branch-and-bound solver to the PARADIGM model. Essentially, this means that the taking of transitions in the PARADIGM model effectuates the execution of operations of implementation objects. If we execute the model combined with the implementation in the PARADE runtime environment, this results in a running branch-and-bound solver. The most important advantage of this extension to the PARADIGM model is that it enables us to take into account problem specific aspects in the analysis and design of the model, like the amount of subproblems created after each visit, the frequency with which solutions are found, or the influence of the subproblem selection strategy on the efficiency of the algorithm. The implementation of the communication required in the componentized and parallel versions of the model can be attached to the additional *global processes* in these versions. This shows that additional concerns introduced by componentizing and parallelizing the model can be addressed separately in both the model and the implementation.

The organization of this chapter is as follows. In Section 7.2, we introduce branch-and-bound algorithms in general by showing the example of a branch-and-bound solver for the *assignment problem*. In Sections 7.3, 7.4 and 7.5, we create the PARADIGM model for a sequential, componentized and parallel version of a generic branch-and-bound algorithm, respectively. For reasons of conciseness, some of the PARADIGM processes and partitions are not included in these sections – they can be found in Appendix B. The use of the PARADE tools in our case study, including the implementation of a concrete branch-and-bound solver, is discussed in Section 7.6. A discussion about the results of the case study can be found in Section 7.7.

7.2 Branch-and-Bound Algorithms

In order to give the reader a brief general picture of how branch-and-bound algorithms work, we present such an algorithm for a simplification of the well-known assignment problem. The problem deals with assigning n tasks to n persons. With each combination of a task and a person, a weight is associated (see Figure 7.1), which can have various meanings, like the time it takes to complete the task or a person's level of familiarity with the task. The problem is to assign each task to exactly one person, such that the sum of weights of these assignments is minimal.

	Task 1	Task 2	Task 3
Anne	9	1	5
Chris	1	3	4
John	7	1	8

Figure 7.1: a branch-and-bound assignment problem

A branch-and-bound algorithm for the assignment problem works as follows. A *subproblem* consists of a list of i task assignments, where $0 \leq i \leq n$. The initial problem is given as the problem where the list of assignments is empty, i.e. $i = 0$. If $i = n$, the subproblem represents a complete solution to the problem: all tasks have been assigned to a person. During a *branch*, new subproblems are generated as follows: one person is chosen as the next candidate person, and a subproblem is created for each assignment of a currently unassigned task to this person. A (simple) *lower bound* to a subproblem consists of the sum of all weights of the tasks currently assigned, added to the sum of the minimal weights of all remaining tasks. As an *upper bound* for a subproblem we can take the sum of all weights of the assigned tasks added to the sum of the maximum weights of all remaining tasks.

Suppose that we are interested in an optimal solution to the assignment problem, and that we apply a best-first strategy for selecting subproblems. The resulting search tree is depicted in Figure 7.2. Initially, the global *least upper bound* is set to ∞. The lower bound for initial problem A is 3, its upper bound is 21. In a first branch from this problem, Anne is chosen as the candidate person. The branch results in three new subproblems, one subproblem for each possible task assignment for Anne. We directly compute the upper and lower bounds for the nodes, and since we apply a best-first selection strategy, we continue with the subproblem with the lowest lower bound, which is subproblem B. We repeat the same procedure two times more and find a first solution D which has lower/upper bound 10. Now, we set the least upper bound to the upper bound of this solution and continue our search.

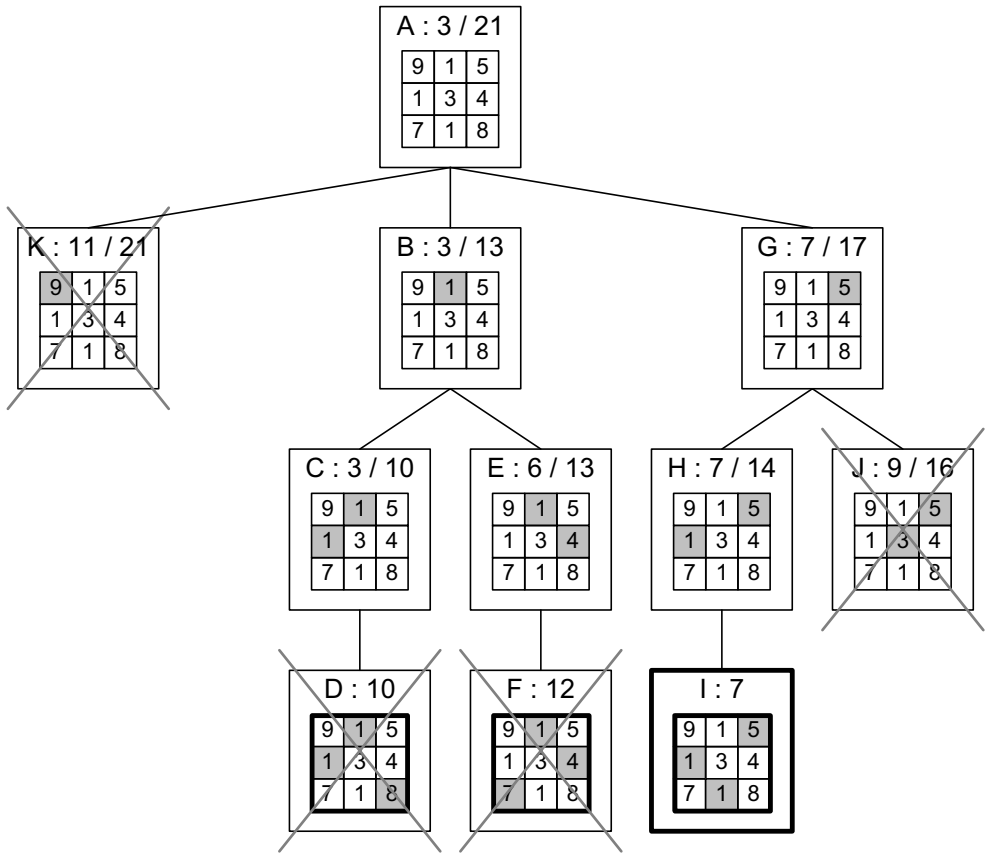


Figure 7.2: a best-first search tree for the assignment problem

The next solution we find, (*F*), has bound 12. We reject this solution, because its bound is higher than the least upper bound found thus far. The third solution we find, (*I*), has 7 as its bound, hence it is better than the solution found thus far. We update the least upper bound and continue the search. However, the lower bounds of the remaining subproblems are all higher than the least upper bound. These subproblems will therefore never lead to a better solution: we can safely eliminate them and terminate the algorithm.

In the illustrated branch-and-bound algorithm, we adopt a so-called *eager* evaluation strategy, that is, the bounds for a subproblem are computed as early as possible. This minimizes the amount of visited subproblems. Another strategy is to postpone the calculation of bounds, which results in a *lazy* evaluation strategy. For some problems, this strategy is particularly efficient in combination with a depth-first search order [19]. As the reader will notice, the PARADIGM model presented in this case study actually models a branch-and-bound algorithm with a *lazy* evaluation strategy. We support selection strategies which are based on the lower bound of nodes, like best-first [19], by giving newly created nodes the lower bound of their parent node, and by initializing the lower bound of the initial node to 0.

7.3 Sequential Branch-and-Bound

Having introduced the general idea of branch-and-bound algorithms, we proceed with creating a PARADIGM model for sequential branch-and-bound. A component diagram of this model is shown in Figure 7.3. The model consists of a single component, which represents the software system that implements the branch-and-bound algorithm. It contains a (detailed) process *Sequential*, which specifies in abstract terms what happens if this system is executed.

We assume the existence of some basic data structures within the component, for which we use a common terminology throughout all versions of the algorithm. Central in this terminology stands the concept of a *node*. A node represents a unit of work and contains a single (sub-)problem. The *initial node* represents the initial problem which has to be solved. A *node pool* facilitates the storage of nodes and the selection of nodes to be visited. The *least upper bound* (LUB for short) is a variable containing the value of the least upper bound found thus far, initially ∞. The initial problem, the node pool and the least upper bound are shown in Figure 7.3 as objects with rounded corners. In Section 7.6, we will implement these data structures in an object-oriented manner as *implementation classes*. For now, we refer to them in the transition labels of PARADIGM processes as abstract entities.

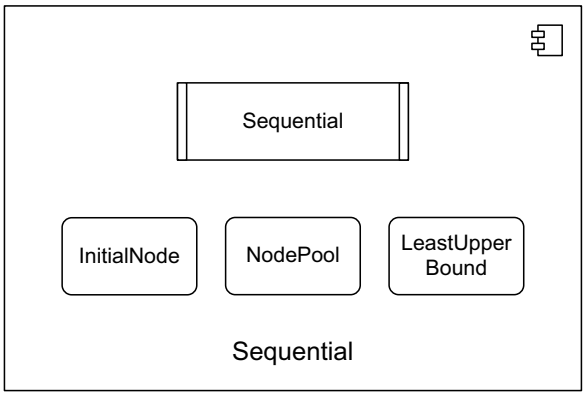


Figure 7.3: Component diagram of the sequential branch-and-bound algorithm

Process *Sequential* is depicted in Figure 7.4. It specifies the steps taken in the execution of the sequential algorithm. The steps are generic with respect to the problem to be solved and the selection strategy to be used. The algorithm adopts a *lazy* evaluation strategy, that is, the bounds are calculated when a node is visited, not when it is created. Note the use of a *choice state* (see Chapter 5) for certain transitions in the model. We use choice states to indicate that a certain non-deterministic choice between transitions will be transformed into a deterministic choice as soon as we attach an implementation to the transition leading to the choice state. Although the *criteria* used for making the choice are outside the scope of the PARADIGM model, the *result* of the choice (in terms of target states) plays a role in it. We present details of this approach in Section 7.6. If we consider the PARADIGM model *without* an implementation, the choice can be regarded as purely non-deterministic, like any other choice in a PARADIGM process.

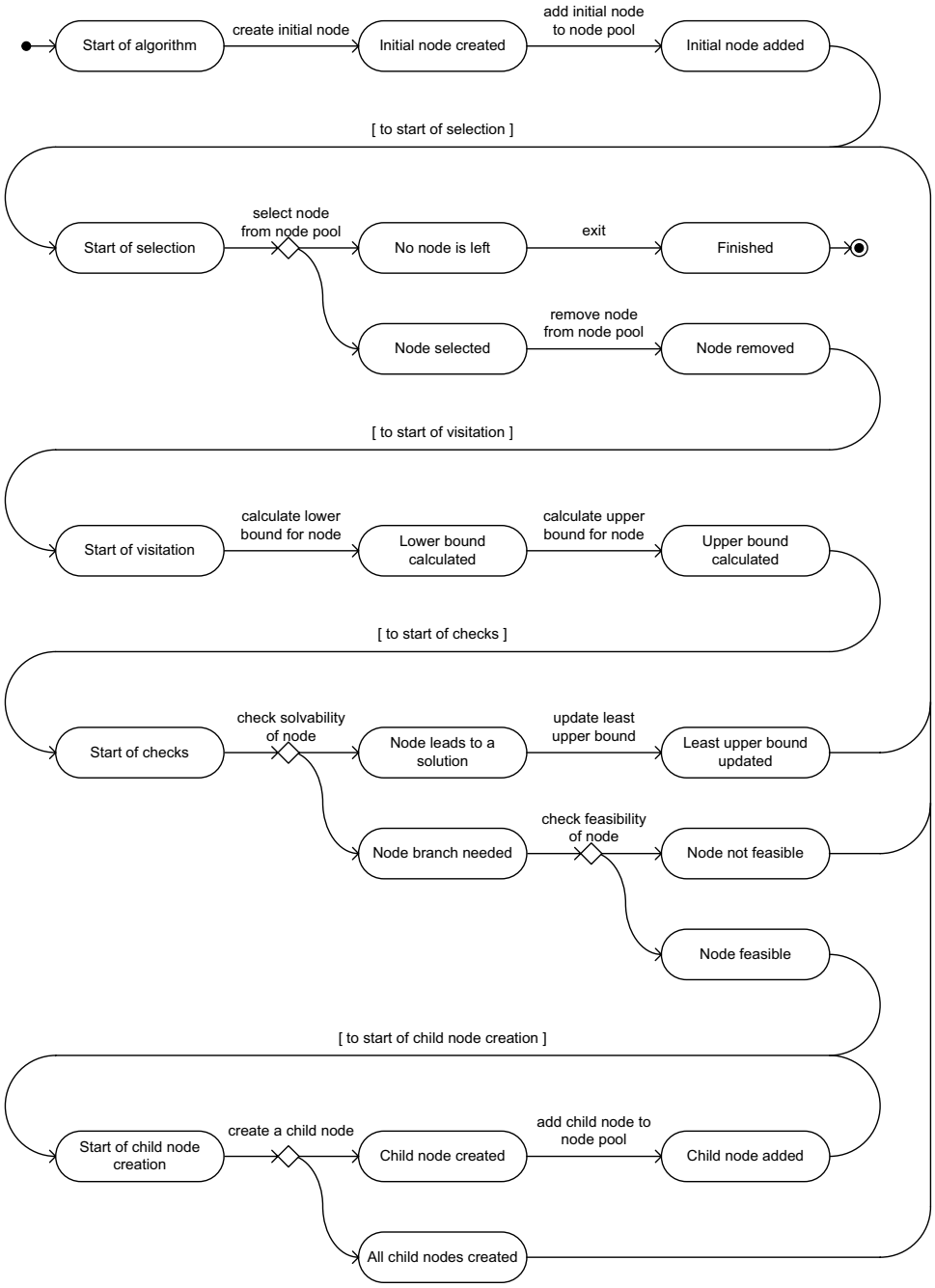


Figure 7.4: Process *Sequential*

The execution of process *Sequential* starts with the creation of an *initial node*. This node contains the problem to be solved. After the initial node has been created and added to the node pool, we continue with the *selection* of one node from the node pool. If the node pool is empty, the process will enter state *No node is left* and exit. Otherwise, it will enter state *Node selected*, the selected node is removed from the node pool and the process continues with *visitation* of the node. The visitation starts with the calculation of lower and upper bounds for the node. After that, a check is performed whether the node directly leads to a solution or not. If the node can be solved directly, we update the value of the least upper bound (if the lower bound of the node is lower than the least upper bound found thus far). Thereafter, we continue with selecting a new node to be visited. If the node cannot be solved directly, we check whether the node is actually feasible, i.e., whether its lower bound does not exceed the least upper bound. If the node is not feasible, we forget it and continue with selection. Otherwise, we perform a branch by creating child nodes and adding them to the node pool. After all child nodes have been created, we continue with selecting the next node to be visited. The algorithm finishes when no nodes are left in the node pool.

Phasing the Sequential Model

In order to create a first idea of how to *componentize* the model, we can divide the sequential process of Figure 7.4 into three different *phases*: a short initialization phase in which the initial node is created (starting at state *Start of algorithm*), a selection phase in which a node is selected from the node pool (starting at state *Start of selection*), and a visitation phase in which activities like bounding and branching are performed (starting at state *Start of visitation*). Especially the visitation phase is relevant for parallelism: in a parallel setting, many parallel running visitation processes could be used to visit nodes from a single node pool. By means of a partition, we are able to formally specify the above phases on top of the detailed process. A global process *Sequential[AsPhased]* at the level of such a partition is given in Figure 7.5. It shows the high-level organization of the detailed process: first initialize, then repeatedly select nodes and visit them, finish in case no nodes are left. Partition *AsPhased* is shown in Figures 7.6 and 7.7. In the next section, we use the three phases *Initializing*, *Selecting* and *Visiting* as a starting point for componentizing the model.

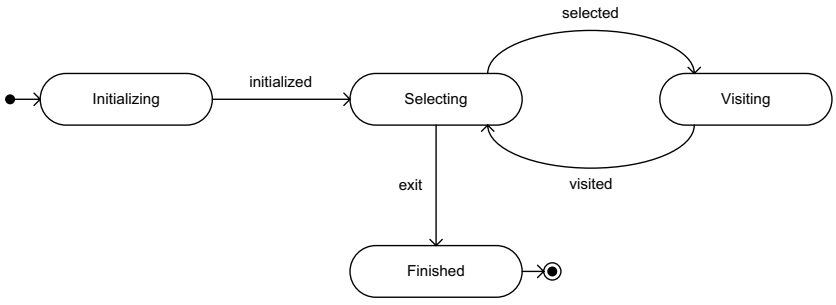


Figure 7.5: Process *Sequential[AsPhased]*

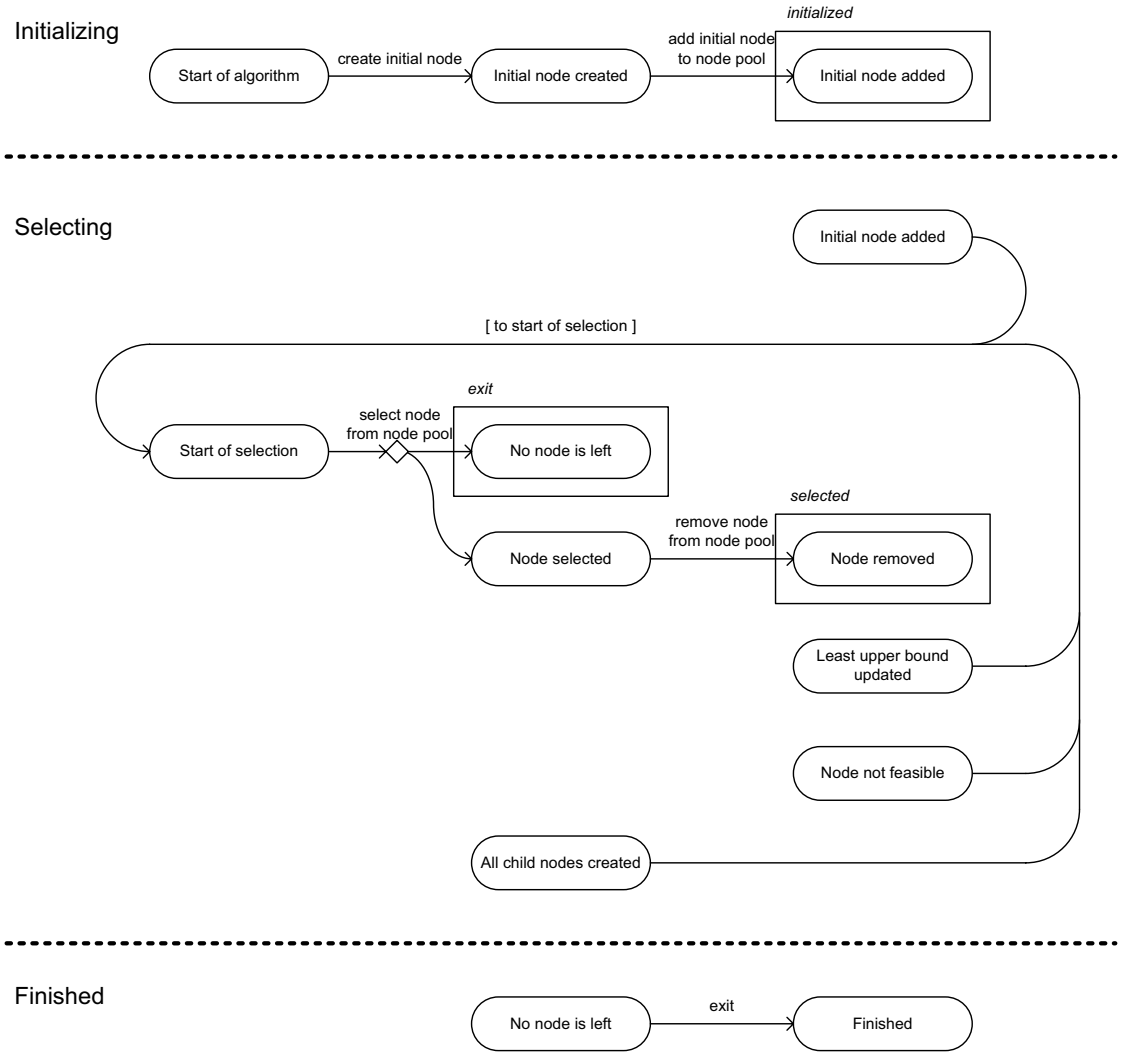


Figure 7.6: Partition *AsPhased* for Process *Sequential* – part 1

7.4 Componentized Branch-and-Bound

As an intermediate step towards a PARADIGM model for a parallel branch-and-bound algorithm, we *componentize* the sequential model by splitting its single component into multiple separate components. Each of the components executes a separate part of the sequential process and encapsulates part of the data structures (i.e., the initial node, the node pool and the least upper bound). We model the composition of the three components by specifying their interaction in terms of partitions and an interaction protocol. The resulting componentized model models the same sequential algorithm, but now as a set of coordinated components.

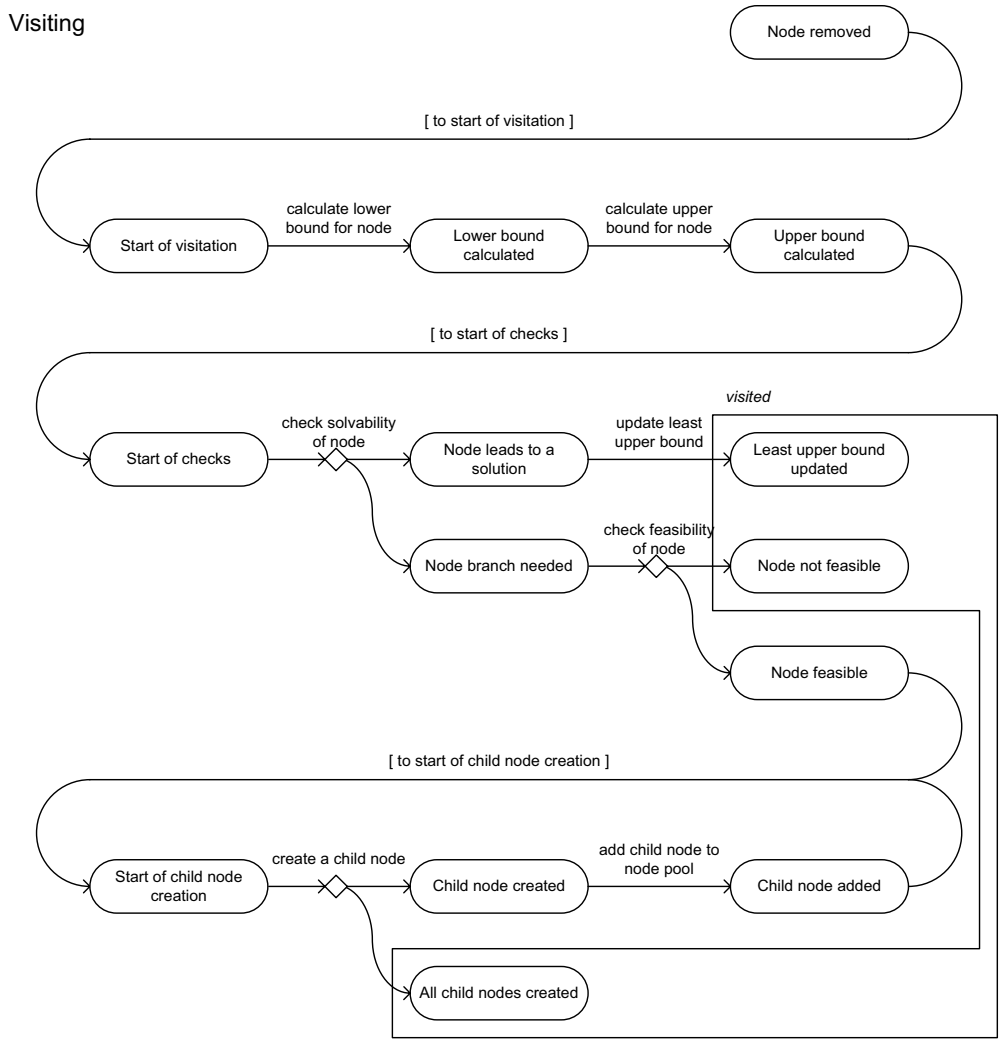


Figure 7.7: Partition *AsPhased* for Process *Sequential* – part 2

A component diagram for the componentized branch-and-bound model is shown in Figure 7.8. We distinguish three components: an *Initializer*, a *Pool Manager* and a *Visitor*. The *Initializer* creates the initial node which contains the subproblem that has to be solved. The *Pool Manager* manages the node pool: selecting and removing nodes from the pool, but also adding nodes to the pool. Finally, the *Visitor* visits selected nodes and creates new child nodes, in the meanwhile maintaining a least upper bound to check the feasibility of nodes. The behavior of the three components is specified in the three respective detailed processes *Initializer*, *PoolManager* and *Visitor*. As we will show, we base these processes on the subprocesses which we distinguished in Section 7.3, but rearrange the transitions slightly as to group into one process the transitions that involve the node pool.

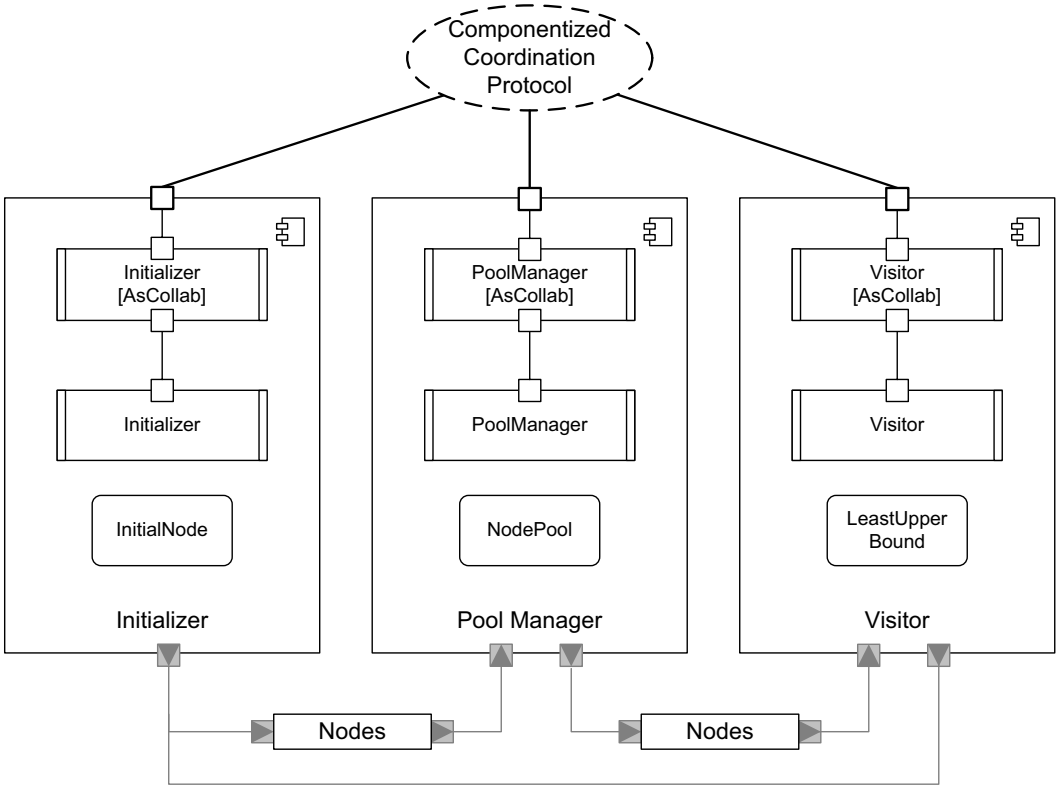


Figure 7.8: Component diagram of the componentized branch-and-bound algorithm

From an implementation perspective, the communication between the three components consists of the exchange of *nodes*: the *Initializer* and *Visitor* components send newly created nodes to the *Pool Manager* component in order for the latter one to add these to the node pool, while the *Pool Manager* component sends selected nodes to the *Visitor* component for visitation. We envision the implementation of the communication as shown below the components in Figure 7.8: the components exchange nodes by means of two unidirectional *channels* (the small rectangles) to which the components are connected by means of *ports* (small squares with an arrow inside indicating the direction of the flow of nodes). This organization is similar to the organization we adopted in [91].

At the modeling level of PARADIGM, however, we abstract from the way in which the components communicate and focus on *interaction*: the way in which the behavior of a component influences the behavior of other components, the causes and the effects of communication. We create partitions *AsCollab* (as collaborator) on top of the detailed processes, which provide a suitable abstraction for modeling this interaction. We coordinate the global processes at the level of partitions *AsCollab* by means of interaction protocol *Componentized Coordination Protocol*. The order in which the consistency rules anchored to this interaction protocol can be applied, is fully determined by the behavior of the three components. Thereby, we avoid an extra manager process to provide additional ordering of these consistency rules, and are able to use an interaction protocol which is *self-managing* (see Chapter 4).

Detailed Processes

We start with creating the detailed processes of the three components, based on the subprocesses of partition *AsPhased* for process *Sequential*. Process *Initializer* depicted in Figure 7.9 is straightforward: it creates the initial node and then exits. The transition to add the initial node to the node pool is not present, although it is part of subprocess *Initializing* in the sequential version of the algorithm. Adding the node to the node pool is now done within process *PoolManager*: this enables us to encapsulate the node pool inside the *Pool Manager* component and keep the remaining components unaware of it.

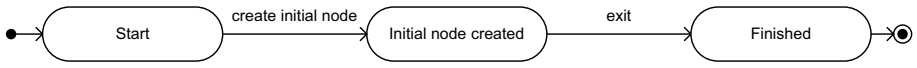


Figure 7.9: Process *Initializer*

Process *PoolManager*, shown in Figure 7.10, is an extended version of subprocess *Selecting* in the sequential model. Since this part of the original sequential process is repetitive, we have added a transition back to state *Start*. Also, a new initial state *Idle* has been added. A new transition *add node to node pool* is present, which replaces the original transitions *add initial node to node pool* and *add child node to node pool*. Intuitively, the process repeatedly performs either the selection of a node from the node pool, or the addition of a node to the node pool. Note the added transition from state *No node is left* to state *Start*. We will use this transition in Section 7.5 in the parallel version of the algorithm to resolve the situation in which the node pool is only temporarily empty.

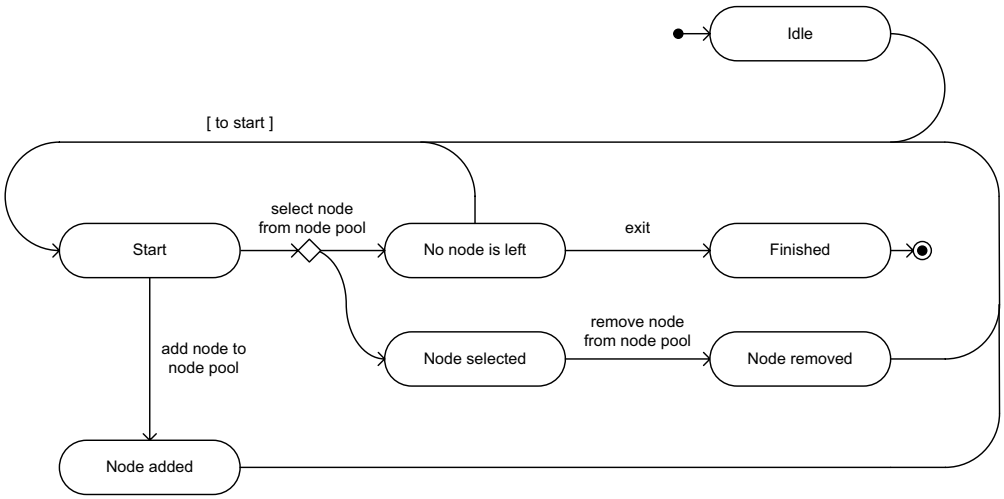


Figure 7.10: Process *PoolManager*

Finally, process *Visitor* is shown in Figure 7.11. It is an adaptation of subprocess *Visiting* in the sequential algorithm: an initial state is specified, the process has a termination state, and backward transitions to state *Start of visitation* have been added, one from each state inside trap *visited* of the original subprocess. Similar to process *Initializer*, the transition to add a node to the node pool (in this case, a child node) has been removed: actions in which the node pool is involved, are performed by process *PoolManager* only.

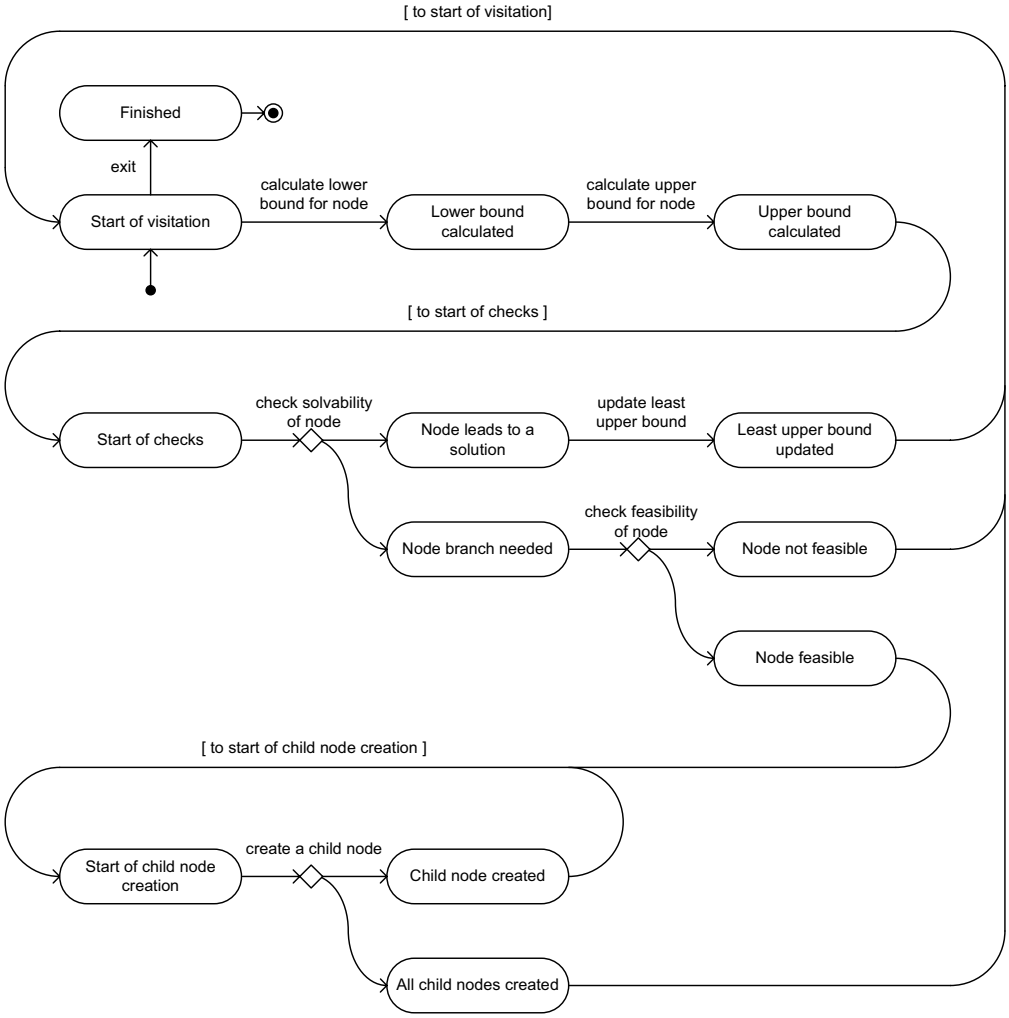


Figure 7.11: Process *Visitor*

Partitions

On top of the three detailed processes, we create *AsCollab* partitions and corresponding global processes which abstract from the details irrelevant to the interaction between the processes. They are shown in Figures 7.12, 7.13 and 7.14. Global Process *Initializer[AsCollab]* is only presented here for reasons of completeness. Global Process *PoolManager[AsCollab]* nicely shows the intuitive idea of the pool manager: starting in subprocess *Idle*, the pool manager either selects a node from the node pool (subprocess *Selecting*) or adds a node to it (subprocess *Adding*). In case the node pool is empty, the process either exits (subprocess *Finished*) or adds a node to the empty pool manager (subprocess *Adding*). The latter subprocess change becomes relevant in the context of multiple parallel visitors, as we will show in Section 7.5.

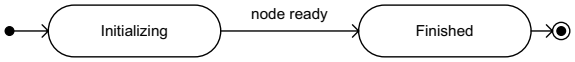


Figure 7.12: Process *Initializer[AsCollab]*

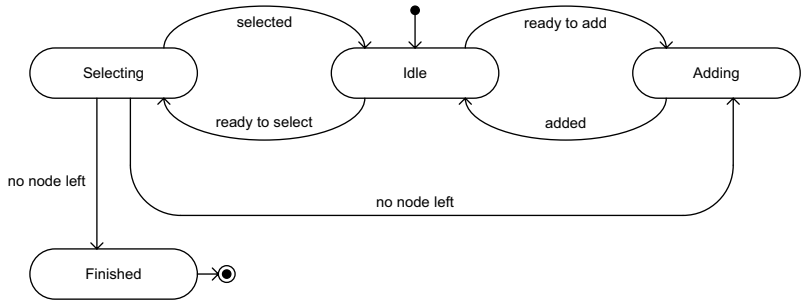


Figure 7.13: Process *PoolManager[AsCollab]*

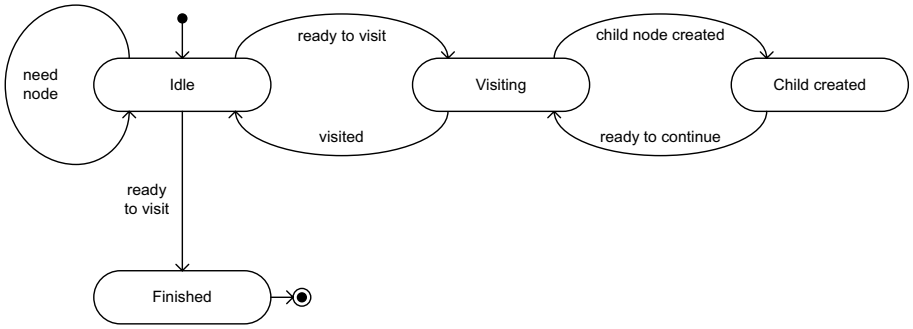


Figure 7.14: Process *Visitor[AsCollab]*

A similar intuitive idea can be captured from Global Process *Visitor[AsCollab]*: in subprocess *Idle* it is able to repeatedly indicate that it needs a node. If a node is available, it will eventually go to subprocess *Visiting*. If a branch takes place, at every creation of a child node the process changes to subprocess *Child created* and back to *Visiting*. Finally, if the node has been visited, the process changes back to subprocess *Idle*. If all nodes have been visited, the process can be switched to subprocess *Finished*.

As an illustration of how the global processes relate to the detailed processes, we show partition *AsCollab* for process *PoolManager* in Figure 7.15. Note in particular the nested traps *ready to add* and *ready to select* in subprocess *Idle*. Because trap *ready to add* is the only trap containing state *Idle*, the first subprocess change from subprocess *Idle* must be a change to subprocess *Adding*. This means that before any selection takes place, at least one node (the initial node) must be added to the node pool. For the *Visitor* and *Initializer* processes, the definition of the subprocesses and traps for their *AsCollab* partitions is relatively straightforward – they can be found in Appendix B.

Interaction Protocols

The self-managing interaction protocol for the coordination of the processes can be found in Table 7.16. It contains eight rules, of which the first five are of interest. Rule *R1* is applied whenever process *Initializer* wants to send a node to process *PoolManager*. Given the definition of process *Initializer*, this rule is applied only once, after which process *PoolManager* adds the node to the node pool. Rule *R2* models process *Visitor* requesting a node from process *PoolManager*. If the request can be fulfilled, rule *R3* is applied: process *PoolManager* sends a (selected) node to process *Visitor*. If the node pool is empty, rule *R4* is applied: the algorithm finishes. Whenever process *Visitor* wants to send a (child) node to process *PoolManager*, rule *R5* ensures that this node is added to the pool. The remaining rules *R6*, *R7* and *R8* allow single subprocess changes if a certain trap has been entered.

It may seem surprising that the processes of the componentized version do not contain any transitions which explicitly address the communication of nodes, e.g. in terms of actions like *send node* or *receive node*. For understanding this communication, the synchronizations specified in the interaction protocol are in fact sufficient. Communication from *Initializer* to *PoolManager*, from *PoolManager* to *Visitor* and from *Visitor* back to *PoolManager* can be assumed to take place at the application of the Rules *R1*, *R3* and *R5*, respectively. As we will discuss in Section 7.6, we realize the communication in the concrete implementation by attaching code for sending and receiving nodes to the transitions of the global processes at the level of partitions *AsCollab*. At each synchronization specified in the interaction protocol, *send* and *receive* actions are executed by the global processes on taking the respective transitions of the applied consistency rule. Hence, understanding the communication in the implementation is equal to understanding the interaction protocol in the model.

In the next section, we will *parallelize* the algorithm by replicating the *Visitor* component. On the one hand, the replication can be performed relatively easily: it only involves changes to the interaction protocol, given certain assumptions about the communication infrastructure. On the other hand, a new issue pops up: the replication of the *least upper bound*, which is encapsulated in the *Visitor* component. We will define an additional component, and additional coordination models, in order to keep a *global* least upper bound in the system.

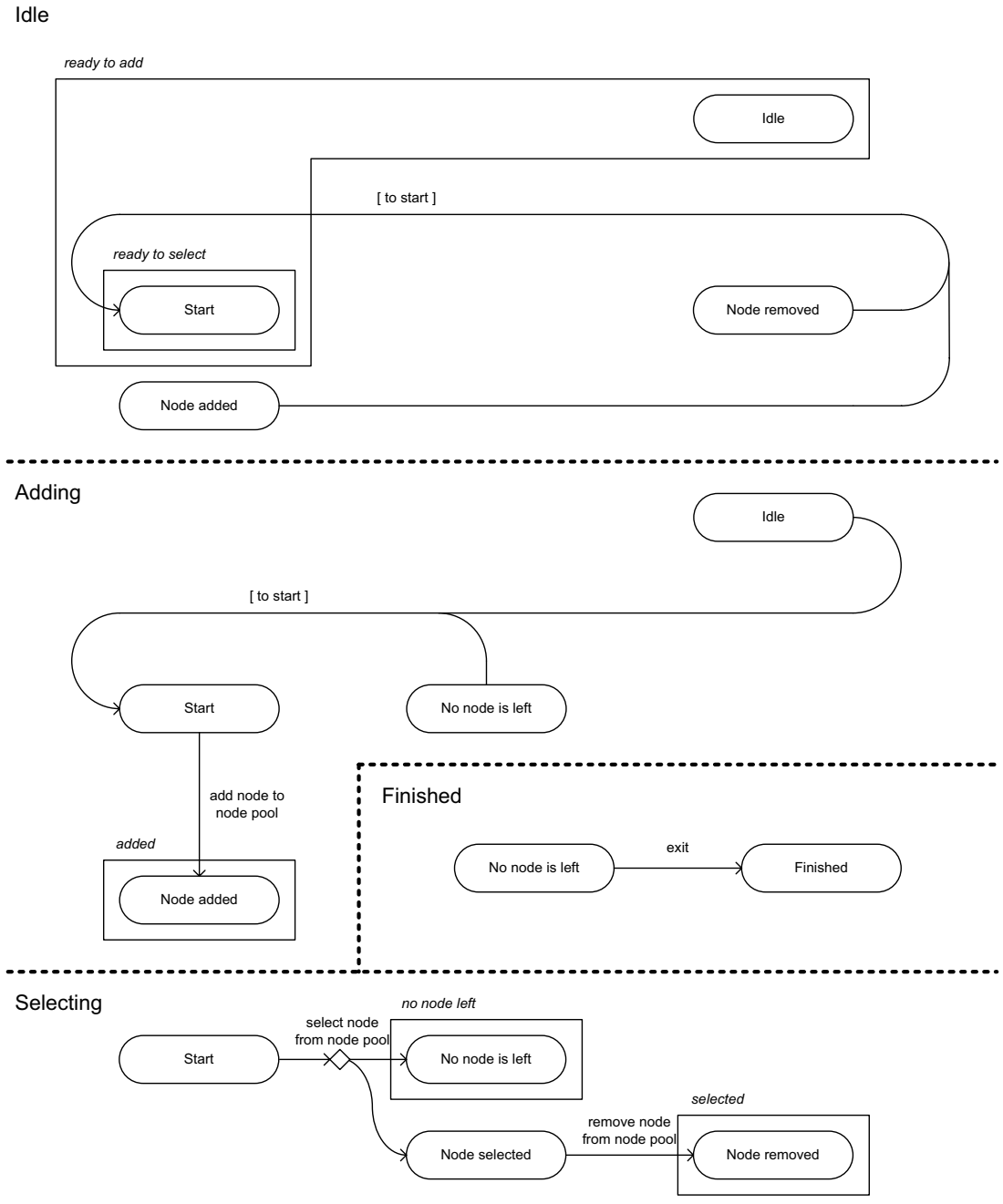


Figure 7.15: Partition *AsCollab* for Process *PoolManager*

(R1)	∗	Initializer[AsCollab]:	Initializing	—nodeReady→	Finished,
		PoolManager[AsCollab]:	Idle	—readyToAdd→	Adding
(R2)	∗	Visitor[AsCollab]:	Idle	—needNode→	Idle,
		PoolManager[AsCollab]:	Idle	—readyToSelect→	Selecting
(R3)	∗	PoolManager[AsCollab]:	Selecting	—selected→	Idle,
		Visitor[AsCollab]:	Idle	—readyToVisit→	Visiting
(R4)	∗	PoolManager[AsCollab]:	Selecting	—noNodeLeft→	Finished,
		Visitor[AsCollab]:	Idle	—readyToVisit→	Finished
(R5)	∗	Visitor[AsCollab]:	Visiting	—childNodeCreated→	ChildCreated,
		PoolManager[AsCollab]:	Idle	—readyToAdd→	Adding
(R6)	∗	PoolManager[AsCollab]:	Adding	—added→	Idle
(R7)	∗	Visitor[AsCollab]:	ChildCreated	—readyToContinue→	Visiting
(R8)	∗	Visitor[AsCollab]:	Visiting	—visited→	Idle

Table 7.16: Interaction Protocol *Componentized Coordination Protocol*

7.5 Parallel Branch-and-Bound

Based on the componentized branch-and-bound algorithm of the previous section, the step towards a parallel branch-and-bound algorithm is relatively small. Now we have three components: *Initializer*, *Pool Manager* and *Visitor*. Our idea is to replicate the *Visitor* component, in order to have multiple "workers" which visit several nodes in the search tree of the branch-and-bound problem in parallel.

The replication of the *Visitor* component introduces two issues. Firstly, the *existing* interaction between the replicated components and the remaining components must be adapted properly: ideally, only the coordination protocol needs to be adapted and no changes to the components themselves are necessary. Secondly, because the replicated components keep state over their repetitive behavior (in our case, they keep track of the least upper bound), *new* interaction is needed between the replicated components to facilitate the communication of this state. We will address both issues in the next subsections, but first we show an overview of the PARADIGM model for the parallel branch-and-bound algorithm in Figure 7.17.

The overview contains a component diagram of a particular instance of the model, which contains three visitors. Two different self-managing protocols coordinate the communication. On top, the existing interaction with the *Pool Manager* and *Initializer* components is shown, coordinated by the *Node Coordination Protocol*, which is an extension of the *Componentized Coordination Protocol* from the previous section. The communication infrastructure has not changed: we still assume the existence of two channels to communicate nodes. At the bottom, a new component *LUB Manager* is added. This component maintains the global least upper bound, and communicates with the *Visitor* components to keep the least upper bound up-to-date in all components. The communication is coordinated by a *LUB Coordination Protocol*.

With regard to the implementation, we assume a similar communication infrastructure for the communication of bounds as we assume for the nodes, be it that the *Initializer* component plays no role in this communication. The source resp. sink port of the communication channels for both nodes and bounds are used by all instances of the replicated visitor component.

Node Coordination

In the parallel version of the model, we can reuse all processes and partitions of the componentized model without changes. Only the interaction protocol needs to be extended and slightly adapted to incorporate the coordination of multiple visitors. An extended interaction protocol *Node Coordination Protocol*, which supports an arbitrary number of *Visitor* components, is shown in Table 7.18. Rules *P1* to *P3* and *P5* to *P8* are equal to rules *R1* to *R3* and *R5* to *R8* in the componentized version of the interaction protocol, except that the rules containing a transition of process *Visitor[AsCollab]* are replicated for each instance i of this process ($0 \leq i < n$).

Rule *P4* is adapted and rules *P9(i)* are added in order to cope with correct termination of the algorithm. In the componentized version with one visitor, termination of the algorithm can be done as soon as both the node pool is empty and the visitor is idle (ready to visit a new node). In the parallel version, this constraint on termination is not sufficient: other visitors could be busy with nodes from which branching is feasible but not yet performed. In that case, the node pool could be only temporarily empty. The extended rule *P4* makes sure that the algorithm terminates only if the node pool is empty and *all* visitors are idle. In case a visitor needs a node but the node pool is temporarily empty, additional rule *P9* allows a new child node from a different visitor to be added to the node pool. To this end, the consistency rule prescribes global process *PoolManager[AsCollab]* to take transition *noNodeLeft* from subprocess *Selecting* to subprocess *Adding*.

An interesting property of any interaction protocol is that it is *stateless*: it applies any consistency rule whenever possible. In the case of a self-managing interaction protocol, there is even no manager process which keeps state. Sometimes, this leads to interesting behavior. Take for example rules *P2(i)* and *P3(i)*, which coordinate a visitor's request for a node and the pool manager's reply with a selected node. Neither the interaction protocol nor the pool manager keeps track of whether the node is accepted by the same visitor as the one that originally placed the request for it. Indeed, this correspondence is in fact not needed for the coordination to be performed consistently: only the *number* of requests is of importance.

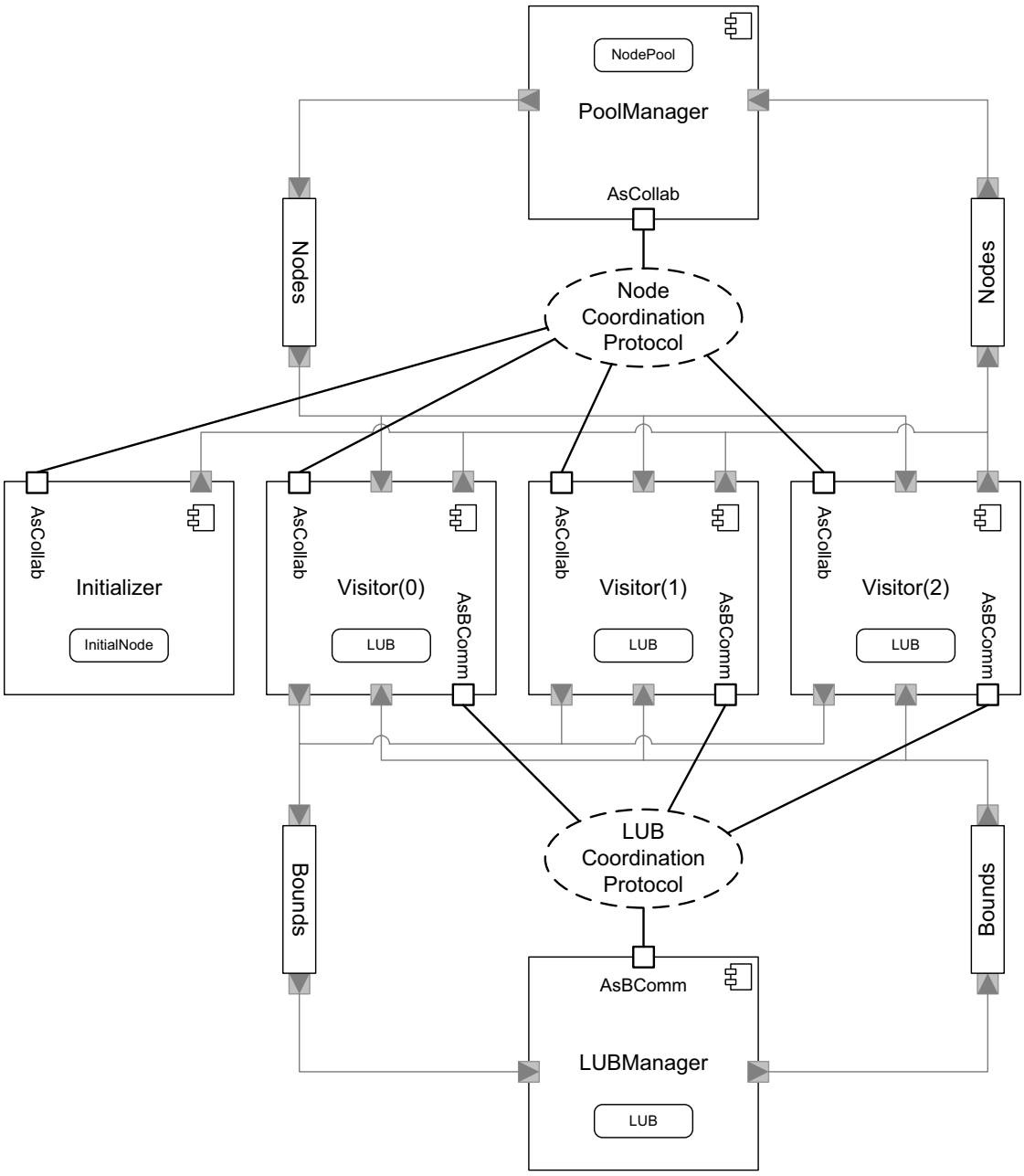


Figure 7.17: Component diagram of the parallel branch-and-bound algorithm

(P1)	∗	Initializer[AsCollab]:	Initializing	—nodeReady→	Finished,
		PoolManager[AsCollab]:	Idle	—readyToAdd→	Adding
(P2(i))	∗	Visitor(i)[AsCollab]:	Idle	—needNode→	Idle,
		PoolManager[AsCollab]:	Idle	—readyToSelect→	Selecting
(P3(i))	∗	PoolManager[AsCollab]:	Selecting	—selected→	Idle,
		Visitor(i)[AsCollab]:	Idle	—readyToVisit→	Visiting
(P4)	∗	PoolManager[AsCollab]:	Selecting	—noNodeLeft→	Finished,
		Visitor(0)[AsCollab]:	Idle	—readyToVisit→	Finished,
		...,			
		Visitor(n-1)[AsCollab]:	Idle	—readyToVisit→	Finished
(P5(i))	∗	Visitor(i)[AsCollab]:	Visiting	—childNodeCreated→	ChildCreated,
		PoolManager[AsCollab]:	Idle	—readyToAdd→	Adding
(P6)	∗	PoolManager[AsCollab]:	Adding	—added→	Idle
(P7(i))	∗	Visitor(i)[AsCollab]:	ChildCreated	—readyToContinue→	Visiting
(P8(i))	∗	Visitor(i)[AsCollab]:	Visiting	—visited→	Idle
(P9(i))	∗	PoolManager[AsCollab]:	Selecting	—noNodeLeft→	Adding,
		Visitor(i)[AsCollab]:	Visiting	—childNodeCreated→	ChildCreated

Table 7.18: Interaction Protocol *Node Coordination Protocol* ($0 \leq i < n$)

Bound Coordination

In many cases, the performance of branch-and-bound algorithms considerably depends on finding a first solution as early as possible. This way, we are able to reject nodes early in the search process due to infeasibility. Therefore, in parallel versions of these algorithms it is important that once a solution is found, an updated least upper bound is communicated to all visitors as soon as possible, in order to allow them to reject nodes in an early stage.

For this reason, we extend our parallel branch-and-bound PARADIGM model with a component and the necessary coordination models to facilitate the communication of least upper bounds between visitors. Intuitively, the idea is as follows. An additional component, *LUB Manager*, keeps track of a *global* least upper bound. Once a *Visitor(i)* component finds a solution, it sends the upper bound of this solution to the *LUB Manager* component. After that, the *Visitor(i)* component updates its local least upper bound if applicable and continues. Meanwhile, the *LUB Manager* component checks whether the upper bound sent to it is an improvement of the global least upper bound, updates it if this is the case, and sends the updated global least upper bound to components *Visitor(i+1)* to *Visitor(i+n-1)* (indices modulo *n*), which update their local least upper bound directly.

We omit the details of the *LUBManager* process (which can be found in Appendix B) and focus on the communication of bounds, which is coordinated by means of partitions *AsBComm* (as bound communicator) and corresponding global processes on top of both process *LUBManager* and all processes *Visitor(i)*, in combination with a dedicated interaction protocol *LUB Coordination Protocol*. The interaction protocol is of particular interest and can be best understood in coherence with global processes *LUBManager[AsBComm]* and *Visitor(i)[AsBComm]*. The global processes are shown in Figures 7.19 and 7.20, the consistency rules anchored to the interaction protocol are listed in Table 7.21.

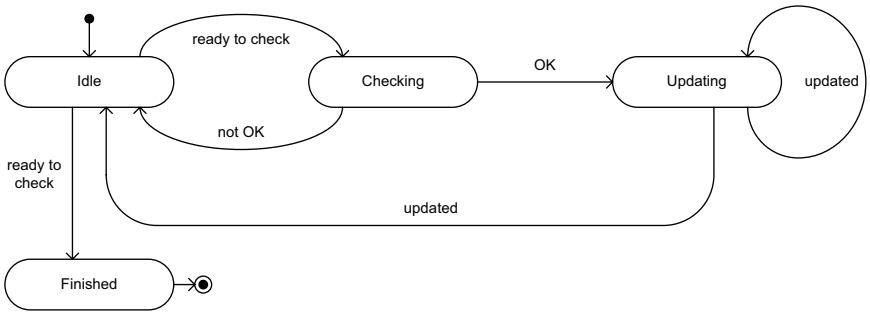


Figure 7.19: Process *LUBManager[AsBComm]*

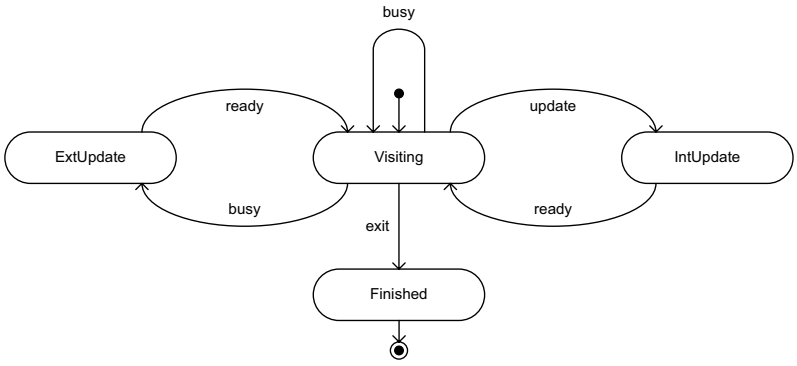


Figure 7.20: Process *Visitor(i)[AsBComm]*

(L1(i))	∗	Visitor(i)[AsBComm]:	Visiting	—update→	IntUpdate,
		LUBManager[AsBComm]:	Idle	—readyToCheck→	Checking
(L2(i))	∗	LUBManager[AsBComm]:	Checking	—not OK→	Idle,
		Visitor(i)[AsBComm]:	IntUpdate	—ready→	Visiting
(L3(i))	∗	LUBManager[AsBComm]:	Checking	—OK→	Updating,
		Visitor(i)[AsBComm]:	IntUpdate	—ready→	Visiting,
		Visitor(i+1)[AsBComm]:	Visiting	—busy→	ExtUpdate,
		...,			
		Visitor(i+n-1)[AsBComm]:	Visiting	—busy→	ExtUpdate
(L4(i))	∗	LUBManager[AsBComm]:	Updating	—updated→	Updating,
		Visitor(i)[AsBComm]:	ExtUpdate	—ready→	Visiting
(L5)	∗	LUBManager[AsBComm]:	Updating	—updated→	Idle,
		Visitor(0)[AsBComm]:	Visiting	—busy→	Visiting,
		...,			
		Visitor(n-1)[AsBComm]:	Visiting	—busy→	Visiting
(L6)	∗	LUBManager[AsBComm]:	Idle	—readyToCheck→	Finished,
		Visitor(0)[AsBComm]:	Visiting	—exit→	Finished,
		...,			
		Visitor(n-1)[AsBComm]:	Visiting	—exit→	Finished

Table 7.21: Interaction Protocol *LUB Coordination Protocol* ($0 \leq i < n$, all indices modulo n)

Interaction protocol *LUB Coordination Protocol* has six consistency rules anchored to it, which intuitively serve the following purpose:

- *L1(i)*: process *Visitor(i)* sends a potential LUB update to process *LUBManager*.
- *L2(i)*: the potential LUB update turns out not to be an improvement of the current LUB – both process *Visitor(i)* and process *LUBManager* ignore the value and continue.

- *L3(i)*: the potential LUB update improves the current LUB – process *Visitor(i)* updates and continues its work, process *LUBManager* prepares for sending updates, all other *Visitor* processes prepare for receiving an update.
- *L4(i)*: each time this rule is applied, process *LUBManager* sends the global LUB to one of the prepared *Visitor* processes. Due to the fact that this rule changes the subprocess of the *Visitor* process involved, the rule can only be applied once per *Visitor* process.
- *L5*: Only if all *Visitor* processes are busy again, process *LUBManager* continues waiting for a next potential LUB update.
- *L6*: If all *Visitor* processes have finished, process *LUBManager* also finishes.

The *LUB Coordination Protocol* is a typical example of a self-managing protocol: the state information needed to steer the application of consistency rules is completely deferred to the coordinated processes. At the same time, none of the processes is aware of the global organization of the system. Even process *LUBManager* does not "know" the amount of *Visitor* processes to which it sends LUB updates.

7.6 Implementing Branch-and-Bound

The various versions of the PARADIGM model for the branch-and-bound algorithm can be executed and visualized using the PARADE distributed runtime environment and runtime viewer (see Chapter 5). This way, the behavior and interaction of the processes and the application of consistency rules can be inspected and analyzed. As an illustration, a snapshot of the runtime environment running the PARADIGM model for the componentized branch-and-bound algorithm from Section 7.4 is shown in Figure 7.22. It shows process *PoolManager* and process *Visitor*, together with the consistency rules of interaction protocol *Componentized Coordination Protocol*. At the moment of the snapshot, process *PoolManager* has just finished selecting a node from the node pool.

In addition to the visualization of the execution of a PARADIGM model, the PARADE distributed runtime environment enables the modeler to extend a PARADIGM model with an *implementation model*. Essentially, this means that Java code is attached to the transitions of the PARADIGM processes, which is executed once a transition is taken by the PARADISE interpreter. In those cases where a PARADIGM model is used to analyze the interaction within a software system, such an extension is valuable for at least four reasons. Firstly, the modeler gains more confidence in whether his PARADIGM model is a valid abstraction from a real implementation. If it turns out to be impossible or very hard to extend the model with an implementation, it is likely that the model can be improved. Secondly, the modeler can consider various implementation-level aspects at the modeling level, like the frequency of interaction and potential bottlenecks in the communication. Thirdly, if the extended PARADIGM model can indeed be used as a concrete software system, it is possible to use the model as a starting point for an efficient distributed implementation. Finally, the implementation attached to the PARADIGM model can serve completely *different* purposes. For example, it could simulate timings on the transitions, collect data about the amount of times transitions are taken, or transform a non-deterministic choice into a stochastic one.

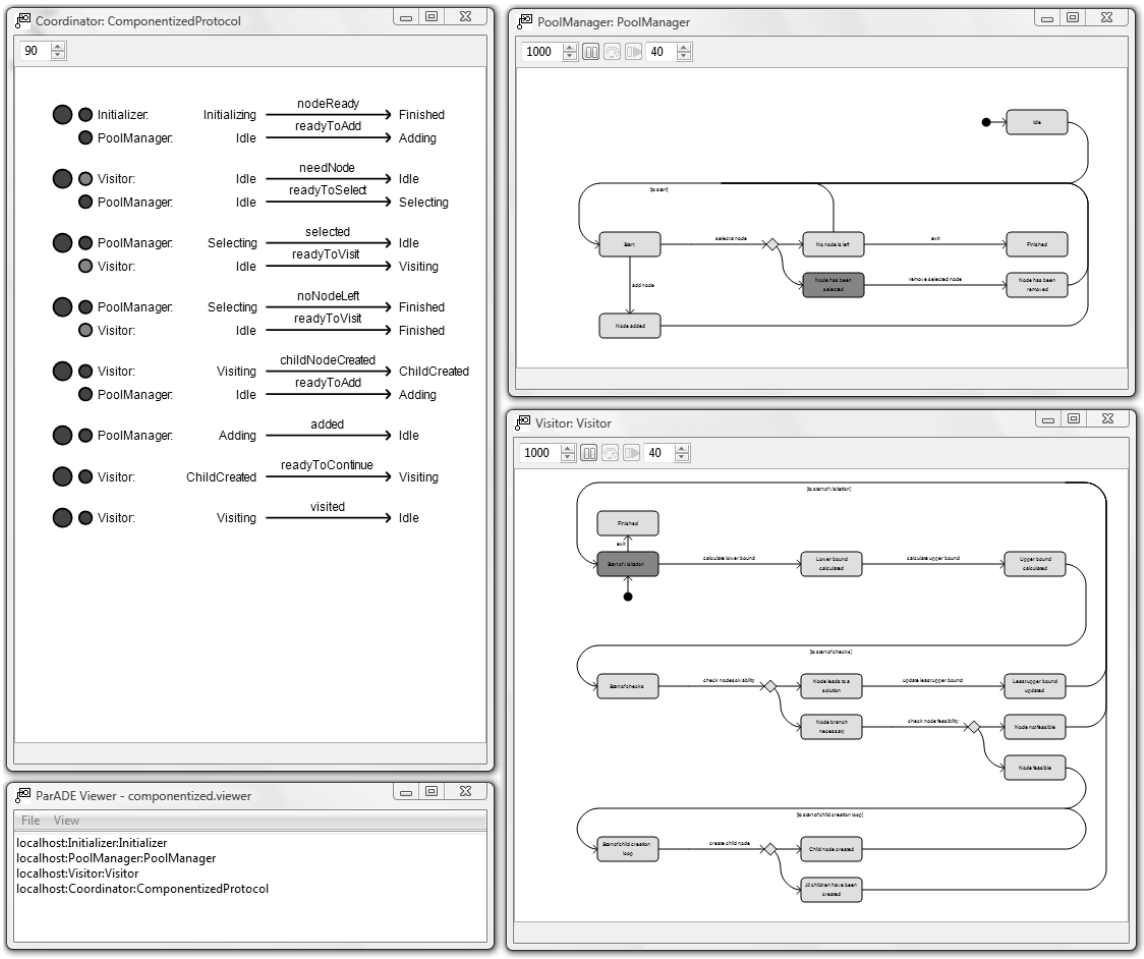


Figure 7.22: The PARADE runtime viewer showing the componentized branch-and-bound model

In the remainder of this section, we illustrate how the software functionality extension of PARADE can be applied to the three versions of the PARADIGM model for the branch-and-bound algorithm. We extend the different versions with code to solve the *assignment problem*, which we introduced in Section 7.2. We realize the extension in three steps. Firstly, we implement the lower-level objects, like nodes, a node pool, and the least upper bound, by creating *implementation classes*. Secondly, we implement *action classes*, which act as a bridge between the PARADIGM processes and the implementation classes. Finally, we create an *implementation model*, in which we specify how the implementation classes and action classes are attached to the processes of the PARADIGM model.

Implementation Classes

In our approach, the *implementation classes* provide the basic functionality of the system. These can be e.g. abstract data types for data used by the system, or classes that implement communication primitives

used to communicate the data. In the case of branch-and-bound algorithms, we consider three abstract data types: *Node* (meant as equivalent to (sub)problem), *Node Pool* and *Bound*. In the case of *distributed* branch-and-bound algorithms, we consider two additional classes which encapsulate communication primitives: *InputPort* and *OutputPort*, which can be used to receive resp. send data from one component to another. The class specifications of these implementation classes are shown in Figure 7.23, together with some examples of specializations of these classes. Instances of implementation classes are meant to be *passive* objects: they do not own a thread of control. Their operations are either called by other implementation classes, or by *action classes*.

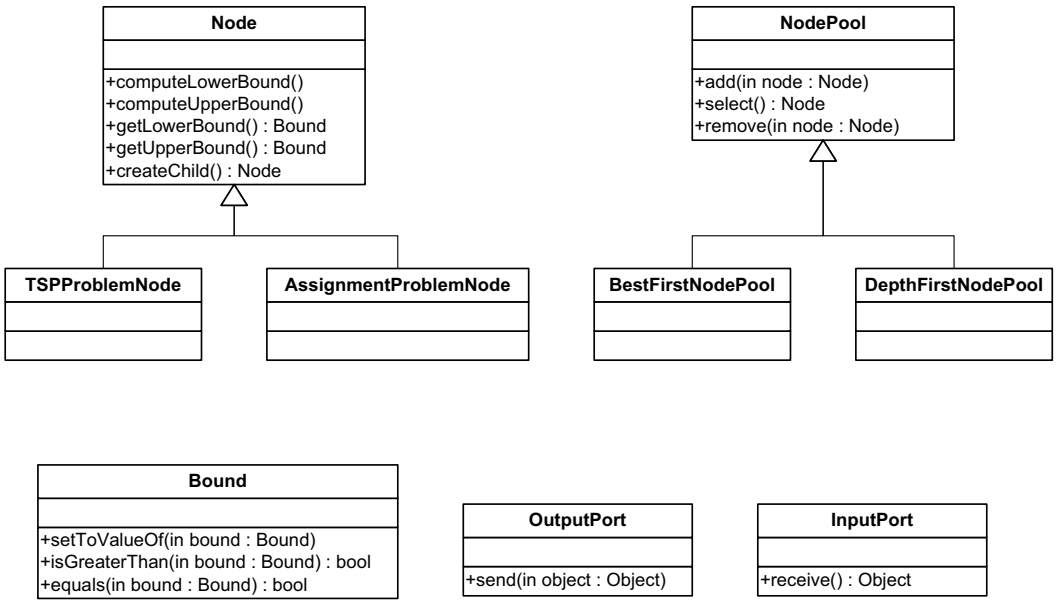


Figure 7.23: Implementation classes for branch-and-bound algorithms

Action Classes

Action classes act as the bridge between PARADIGM processes and implementation classes. Conceptually, the behavior of an action class is an operationalization of the semantics of a transition label in a PARADIGM process. The operationalization is specified in terms of operations invoked on a set of instances of implementation classes. The other way around, one can view a transition label in a PARADIGM process as an abstract term which denotes an activity composed out of operation invocations on instances of implementation classes. An action class in PARADE always provides a single operation, *run()*, which on invocation intuitively results in "performing the activity". Action classes are meant to be both stateless and passive.

A simple example of an action class coded in Java is given in the code snippet below. It performs the activity of adding a node to the node pool. Firstly, two parameters *NodePool* and *Node* are fetched. After that, the *add* operation of the *nodes* class is invoked with the node object as a parameter.

```
public class AddNodeToNodePoolAction extends Action {
    @Override
    public void run() {
        NodePool nodes = (NodePool) this.getParameter("NodePool");
        Node node = (Node) this.getParameter("Node");
        nodes.add(node);
    }
}
```

Within action classes, it is possible to influence the behavior of the PARADIGM process through the use of a *result label*. The result label can be mapped onto the names of the target states of the transition to which the action is attached. An example of an action class which uses a result label is shown below. It implements the activity of selecting a node from the node pool. The action class is attached to transition *select node from node pool* of process *PoolManager*. In the execution of this process, the result labels "selected" and "not selected" are used to determine whether a transition is taken to state *Node selected* or *No node is left*, respectively. Also, note that the node selected from the node pool is stored in a parameter with name *Node*. Thereby, the selected node can be used within subsequent transitions.

```
public class SelectNodeFromNodePoolAction extends Action {
    @Override
    public void run() {
        NodePool nodes = (NodePool) this.getParameter("Nodes");
        Node node = nodes.getSelection();
        if (node != null) {
            this.setResult("selected");
            this.setParameter("Node", node);
        } else
            this.setResult("not selected");
    }
}
```

Finally, the attachment of action classes to PARADIGM transitions is done in one or more *implementation models*.

Implementation Models

In PARADE, an implementation model specifies the relation between PARADIGM process transitions and action classes. Each implementation model is specific for a single process type (see also Chapter 5). A visualization of the contents of an implementation model is given in Figure 7.24. It shows which action classes are attached to the transitions of process *PoolManager*. It is not obligatory to attach an action class to every transition in a process. If no action class is attached to a transition, this simply means that the taking of that transition does not result in the execution of implementation code.

The attachment of action classes to transitions is not restricted to detailed PARADIGM processes only. On the contrary, in the PARADIGM model versions of the componentized and distributed branch-and-bound algorithm, we exploit the possibility to attach code to *global processes*, which are treated in the PARADISE interpreter framework (see Chapter 3) as ordinary processes. As Figure 7.25 shows, we attach the action classes which realize the communication of nodes to process *PoolManager[AsCollab]*, a global

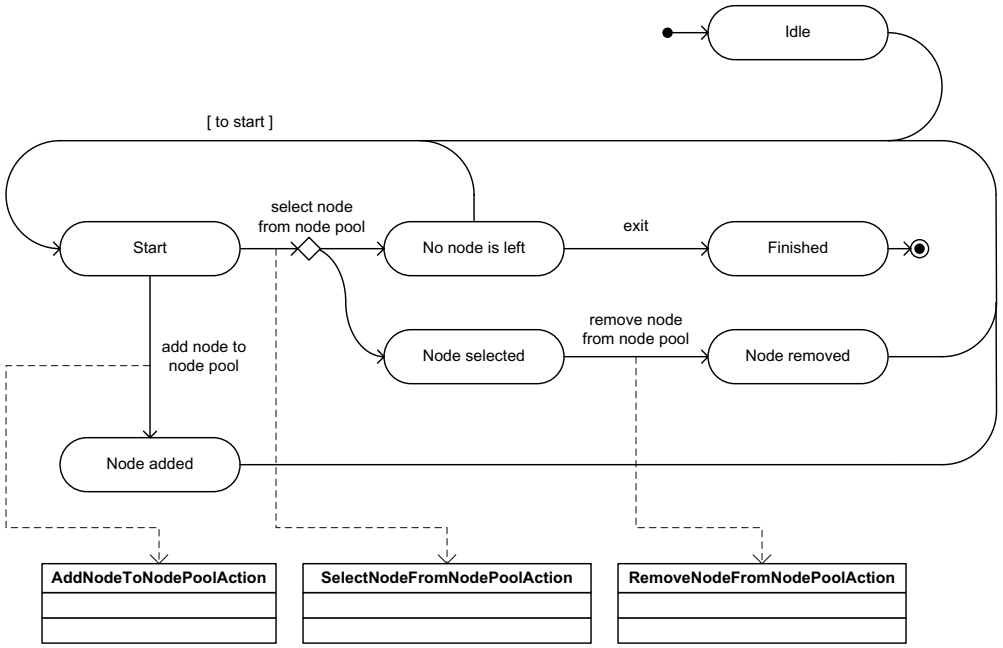


Figure 7.24: Action classes attached to transitions of Process *PoolManager*

process defined on top of detailed process *PoolManager*. Thereby, we nicely separate the actions implementing the essential activities of the algorithm from the actions that implement additional concerns in the context of componentization or parallelization.

Both processes *PoolManager* and *PoolManager[AsCollab]* are part of a single component and share the instances of implementation classes defined in this component. Their transitions are potentially taken *concurrently*, or at least in an *interleaved* manner, depending on the subprocess definitions of the partition at whose level the global process is defined. This should be taken into account in the specification of the object accesses in the action classes.

If we execute the extended PARADIGM model in the PARADE runtime environment, the running model appears to the user as an inefficient Java implementation of a lazy branch-and-bound algorithm. The implementation could produce output on a console or in a window, like the amount of nodes in the node pool, the number of times a solution is found, or the depth of the search tree. Such information can be valuable in improving the PARADIGM model. The information could also be used to *evolve* the PARADIGM model *on-the-fly*, for example by adapting the amount of visitors or the number of node pools in the model. This latter possibility, the evolution of a PARADIGM model on-the-fly, is the subject of the next and final case study.

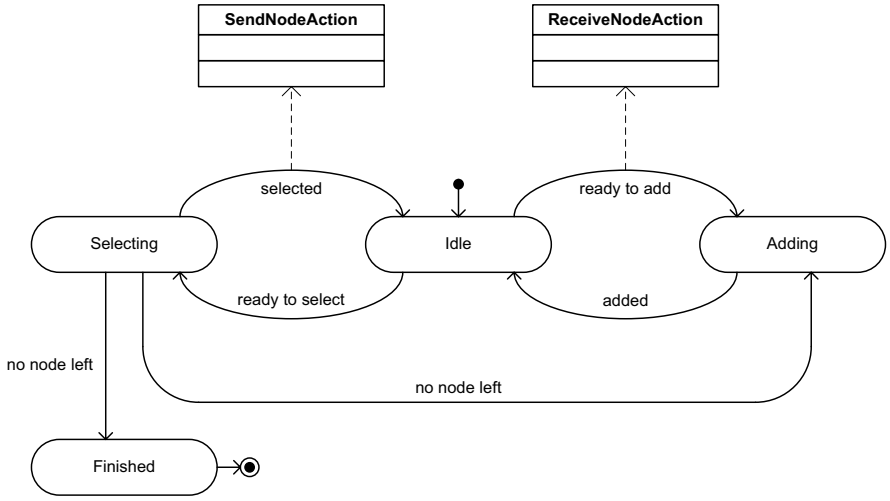


Figure 7.25: Action classes attached to transitions of Process *PoolManager[AsCollab]*

7.7 Discussion

In this case study, we applied the PARADIGM language to model the componentization and parallelization of a branch-and-bound solver. As we have shown, componentization in a PARADIGM model is a matter of splitting processes into logical parts. We have applied the partition concept in order to formally represent the phases in which a process resides. Based on such phases, new processes can be created, which requires the addition of initial and final states, and the addition of loops if applicable. The newly created processes are coordinated by applying partitions and interaction protocols in the usual manner.

We exploited the notion of *self-managing interaction protocol* to extend the componentized model with multiple *Visitor* processes, without changing any of the processes or partitions in that model. The absence of an additional manager process, as an ordering mechanism for the application of consistency rules, actually forces the modeler to (more) carefully think about the design of the global processes at the level of partitions.

We were able to model the additional aspects introduced with componentization and parallelization completely separately from the existing model. Throughout all versions, the *core algorithm* is captured in the *detailed processes*, while *additional concerns* (interaction, the communication of nodes and bounds) are captured in *partitions*, *global processes* and *interaction protocols*.

Especially relevant in the context of modeling *software* is the insight that, in order to achieve an adequate componentization, the modeler must have an abstract but clearly defined idea of how to organize the underlying implementation. The (often informal) semantics of the *transition labels* used in the PARADIGM processes must be adequately understood. In this respect, the modeling of branch-and-bound algorithms is a safe choice, since this domain can be considered well studied and understood.

As the careful reader will have noticed, we have used the same transition labels for all transitions of detailed processes throughout all three versions of the model. We stress that the same property holds for the units of activity to which these transition labels refer: we were able to use the same implementation throughout all versions of the model. Moreover, the separation between the core algorithm and the additional concerns in the *model* is also reflected in the *implementation*: the action classes attached to the detailed processes realize the core activities of the algorithm, while the action classes attached to the global processes realize the sends and receives of nodes and bounds.

Finally, we point out how we perceive the relationship between a PARADIGM model for a software system and the implementation of that system. In the PARADE implementation extension, we use *action classes* as behavioral glue between the data objects: they call operations of implementation objects and pass (parts of) these objects to other implementation objects as operation parameters. Action classes can be seen as providing a *behavioral pattern*, which steers a thread of control through various implementation objects. The PARADIGM processes in the model sequentialize a set of actions: they represent a higher-level behavioral glue, abstracting from implementation details, but giving insight into how the system, seen as a process, behaves. They have state, but at a higher level of abstraction, not necessarily mapping hierarchically onto states at the implementation level. In our opinion, this way of abstracting from the implementation is a key factor in achieving an implementation-independent behavioral model.

Chapter 8

Evolution On-the-Fly

In this case study, we use PARADIGM for the modeling of *evolution on-the-fly*: the application of changes to a PARADIGM model while it is being executed. In our approach, the coordination of such evolution is performed by a process *McPal*, which is added to the PARADIGM model to be evolved. We introduce a general technique, *scaffolding*, to temporarily observe and steer the behavior of running processes while changes are applied. By exploiting the functionality of the PARADE tools, we are able to implement the application of changes to PARADIGM models being executed in the PARADE distributed runtime environment. Our approach is illustrated with two different changes to a running scheduler/worker model.

8.1 Introduction

In this chapter, we present the third and final case study, which focuses on the modeling of *evolution on-the-fly* – changing a PARADIGM model (its processes, partitions and interaction protocols) while that model is being executed. We use the extended version of the PARADIGM language (Chapter 4) for modeling the various evolutionary phases of the model as well as the temporary phases required to evolve the model in a consistent manner. In addition, we use the features of the PARADE runtime environment (Chapter 5) to apply the changes to the components of a model being executed in the PARADE distributed runtime environment.

The research done within this case study is closely related to work of Groenewegen and De Vink presented in [46, 40]. Similar to their work, we explicitly model the evolutionary steps by means of a special process called *McPal* (Managing changing Processes ad libitum). This process manages the evolution of the PARADIGM model, including itself, by taking transitions whose transition labels denote steps of a certain change to the model. Process *McPal* is generic with respect to the model being changed and the actual changes being applied. Because *McPal* applies changes to a model on-the-fly, care must be taken that these changes do not cause undesired behavior – where the meaning of "undesired" depends on the context in which the changes are applied. The PARADIGM language provides a direct means to avoid undesired behavior by constraining the behavior of processes through the application of *partitions*. This is exploited in [46] and [40] at the level of detailed processes: a special partition *Evol* is used for this purpose.

In this case study, we *generalize* the technique of temporarily constraining the behavior of processes in two respects. Firstly, in line with the ideas presented in Chapter 3, we consider detailed and global processes both as processes. Therefore, as we will explain, we treat the application of changes to global processes in the same way as that of detailed processes. Thereby, we have created a more general technique for modeling evolution on-the-fly, which can be applied in the same manner regardless of the type of entities involved in the changes. Secondly, in our generalization, the partitions and interaction protocols needed to coordinate the evolution of the model are added prior to the evolution itself, and removed afterwards. We call this generalized technique *scaffolding* – the activity of temporarily extending a running model with partitions and interaction protocols in order to constrain the behavior of processes in that model for the sake of evolution. Because the extensions are temporary, this leads to a cleaner model before and after the changes. In addition, we are better able to adapt the temporary extensions to the needs of the changes.

In addition to the above contribution, we provide an implementation of PARADIGM models which evolve on-the-fly, by means of the PARADE tools. We directly make use of the functionality of these tools to execute both the model and its evolution. We use the flexibility of the PARADISE framework (Chapter 3) for updating the distributed interpreter which runs the model along with updating the model artifacts themselves. To the transition labels in process *McPal*, we attach a concrete implementation of the steps needed for changing the model. The individual steps update parts of the model via the system channels of components running in the PARADE distributed runtime environment (Chapter 5).

We apply evolution on-the-fly to a PARADIGM model of a scheduler and three workers. Initially, the workers (concurrent processes) in the model perform their work almost in sequence, due to the way in which they are coordinated by the scheduler. In addition, the scheduler performs the scheduling of the workers in a non-deterministic manner. We will show how to evolve this model into one in which the amount of concurrency between the workers is considerably increased and the scheduler schedules round-robin. The changes are applied in two steps: firstly, we change the way in which the workers are coordinated, and secondly, we change the scheduling strategy of the scheduler. For each of these changes, as we will show, we put a particular scaffold in place before the changes are applied.

The case study points out the suitability of PARADIGM for the modeling of evolution on-the-fly. In addition, it shows how the scaffolding technique leads to cleaner PARADIGM models, a more uniform manner in which coordination for the purpose of migration takes place, and a better focus on the specific coordination needs for a certain migration in the model.

This chapter is structured as follows. In Section 8.2 we provide an introduction to the proposed technique for modeling evolution on-the-fly, including the McPal process and the notion of scaffolding. In Section 8.3, we introduce the initial scheduler/worker model, which is the model we will evolve. In Sections 8.4 and 8.5, we model the two changes mentioned above. Section 8.6 is devoted to the implementation of evolution on-the-fly in the PARADE distributed runtime environment. We discuss the insights of the case study in Sections 8.7.

8.2 Evolution On-the-Fly

As a starting point for considering the modeling of evolution on-the-fly, we take the extended PARADIGM language presented in Chapter 4. That is, we consider three language concepts as the entities amenable to evolution: processes, partitions and interaction protocols. Each of these entities can be either created, updated, or deleted. The creation, update and deletion of individual consistency rules is performed by updating the interaction protocol to which these consistency rules are (to be) anchored.

Just as in [46] and [40], it is the role of an additional process *McPal* to manage the evolution of the model, i.e. to execute the actions to create, update and delete modeling entities. For all migrations considered in this case study, we use the same process *McPal*, which is shown in Figure 8.1. Process *McPal* can be split up into two different phases, *McPalBeforeEvol* and *McPalDuringEvol*, as defined in partition *AsEvol* and global process *McPal[AsEvol]* in Figure 8.2.

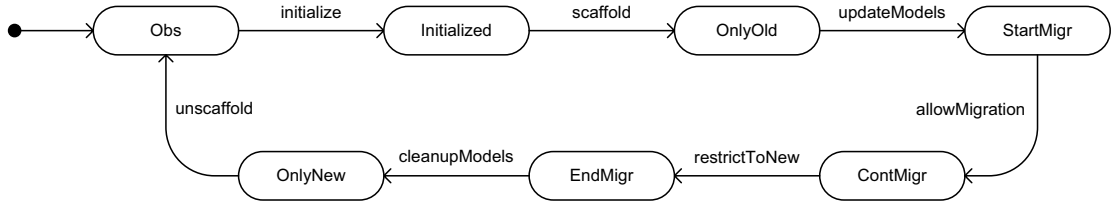


Figure 8.1: Process *McPal*

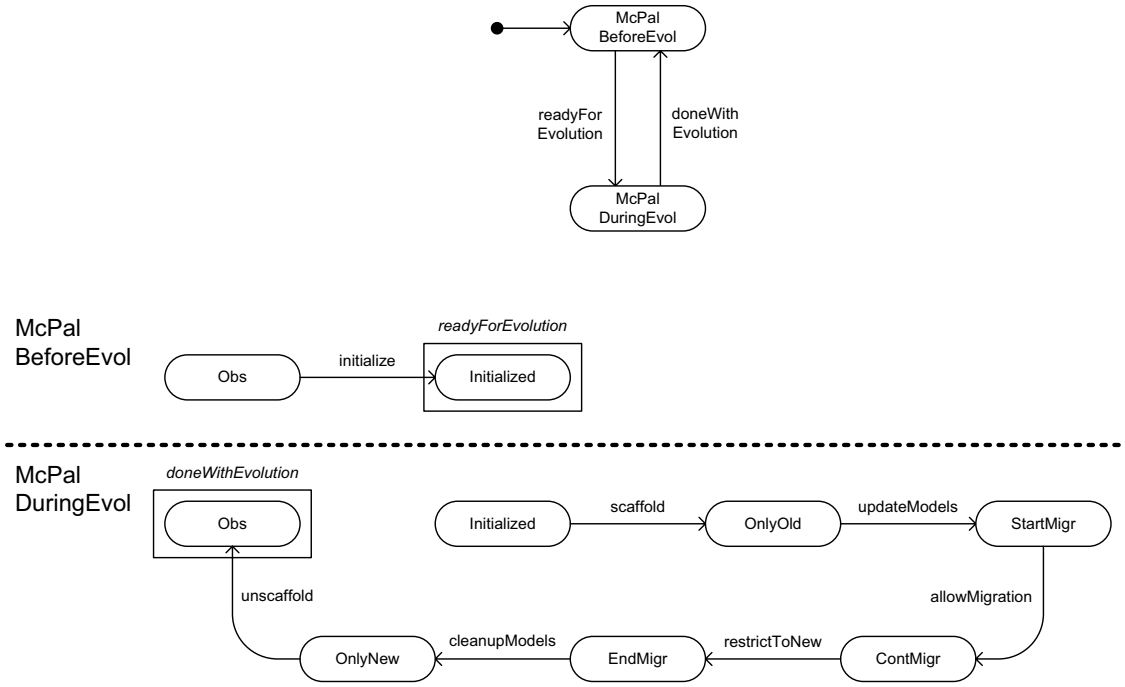


Figure 8.2: Process *McPal[AsEvol]* and Partition *AsEvol* for Process *McPal*

Intuitively, in phase *McPalBeforeEvol*, the process preprares itself for a single *migration*, i.e., a single run through all its process steps. It does this by defining the semantics of the transitions belonging to its phase *McPalDuringEvol*. In other words, after process *McPal* has taken transition *initialize*, the actions for a specific migration have been associated with the subsequent transitions of the process. Now, in the second phase, *McPalDuringEvol*, process *McPal realizes* the migration of the model by taking the transitions whose semantics have been defined in the first phase. At the modeling level, we consider the execution of the creations, updates and deletions associated with a single transition as a single, non-divisible, *atomic* action. After having completed all migration steps, *McPal* returns to phase *McPalBeforeEvol*, in which it prepares itself for the next migration by defining new semantics to the transitions of phase *McPalDuringEvol*. Note that we do not further specify how global process *McPal[AsEvol]* is coordinated – the process is primarily added to the model for illustration purposes. It can be easily executed in the PARADE distributed runtime environment by treating it as a detailed process, like we mentioned earlier in the discussion of Chapter 6 about some of the global processes in the PARADIGM model for the car navigation system.

Before we discuss process *McPal* in detail, we first introduce the notion of *scaffolding*. Because creations, updates and deletions are applied to a model *while it is being executed*, process *McPal* needs to take care that these changes do not lead to undesired behavior, like technical inconsistencies in the behavior of the model, or behavior which does not adhere to the modeler's requirements (e.g. deadlock or starvation). In our approach, *McPal* uses *temporary partitions and interaction protocols* which constrain the behavior of the running processes in the model for the sake of correctly applying the changes. These temporary entities together are called a *scaffold*. *McPal* builds a scaffold around the model before changing that model, and removes the scaffold once the changes have been applied. A scaffold can be very simple or relatively elaborate, depending on the size and scope of the changes.

A scaffold always consists one interaction protocol, with process *McPal* as a manager, and several partitions and global processes on top of processes in the model to be evolved, which are coordinated by the interaction protocol of the scaffold. The example of Figure 8.3 shows a scaffold (the part with bold lines) with one covering interaction protocol, which poses dynamic constraints on two detailed processes (one employee, one manager) and one global process. Note in particular that we apply a partition on top of a *global* process. Although this way of working is not common in PARADIGM, we indicated this possibility earlier in Chapter 3. The existing interaction protocol of the model is only implicitly constrained, via the processes involved in its consistency rules. The purpose of the scaffold is to constrain the behavior of the processes on top of which its partitions are created. The scaffold can therefore be limited to those processes which either are changed themselves, or interact with entities to be changed in such a way that this interaction needs to be constrained.

We now have a closer look at process *McPal*. Each realization of a migration consists of the same sequence of transitions, by means of which the application of changes to the model is coordinated. Process *McPal* starts the migration by taking transition *scaffold*. With the taking of this transition, the original model M_o is extended with a scaffold. This results in a "scaffolded" model $[M_o]$, with which we denote that the scaffold [ ] restricts the execution of the model to *at most* all possible behavior defined in M_o already. In the next transition, *updateModels*, the model-to-be-evolved is altered by creating and updating processes, partitions and protocols. In this step, we only consider *additions* to the model – we do not (yet) remove any entities. By this, we achieve an updated scaffolded model $[M_o] \cup M_m \cup M_n$, where M_n stands for the *next* model and M_m represents the *migration* model – the intermediate situation which bridges M_o and M_n. Naturally, $M_o \subseteq M_m \supseteq M_n$. While we do this, the scaffold ensures that the execution of the model is constrained to *at most* the behavior which was possible in the original model M_o: the models have been updated, but the scaffolds do not allow the newly defined behavior yet.

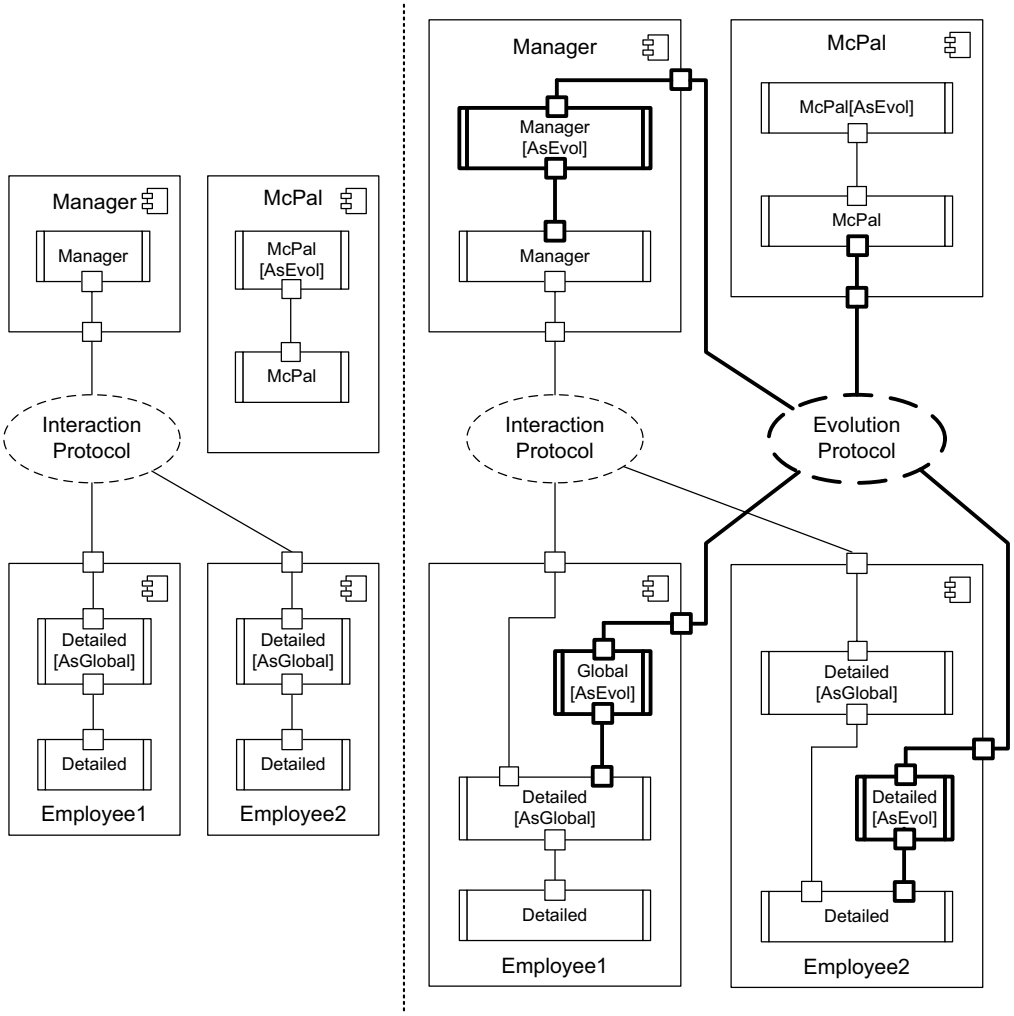


Figure 8.3: A PARADIGM model without (left) and with a scaffold (in bold on the right)

In the next step, we use the scaffolds to allow the new behavior to be taken into account in the execution of the model. That is, we turn $[M_o] \cup M_m \cup M_n$ into $[M_o \cup M_m \cup M_n]$. After this step, the system could continue behaving according to the original model, but it could also (deliberately or enforced) start behaving according to newly defined behavior. As soon as the model behaves according to the desired new behavior (this should be observed by means of the scaffold), *McPal* restricts the behavior of the system to the new model by taking transition *restrictToNew*, thus achieving model $M_o \cup M_m \cup [M_n]$. After that, it removes the parts of the original model which are no longer relevant in the the new model with transition *cleanupModels* – here, the deletion of entities or the removal of parts of entities takes place. Thus, we yield $[M_n]$. Finally, the scaffolds are removed with transition *unscaffold*, which concludes the migration to model M_n.

This same sequence of steps is applied repetitively, once for each change to a model. At each change, the model itself, i.e. the specification of processes, partitions and interation protocols, moves through phases of *expansion* and *contraction*, growing and shrinking. The scaffold, in turn, ensures that this expansion and contraction takes place at the appropriate moments in time, taking into account the runtime state of the processes in the model as it is being executed. Note that a scaffold is a *behavioral* construct, directly related to the behavior of process *McPal*. It enables a modeler to apply constraints to the behavior of processes, and to alter these constraints during a migration. In addition, it enables a modeler to observe the state of processes at an appropriate level of abstraction, as a means to decide when a next step in a migration can be taken. The scaffold can be said to realize the semantics of transitions *allowMigration* and *restrictToNew*. In contrast, the semantics of transitions *updateModels* and *cleanupModels* are given by means of a specification, for each of these transitions, of the precise creations, updates and deletions of entities in the model.

In this case study, we apply the above technique for modeling evolution on-the-fly to a small model which represents a system consisting of a scheduler with three workers. We have reused this example from [46] in order to compare our approach to the modeling of evolution on-the-fly with the approaches presented in [46] and [40]. We start with presenting the initial model of this scheduler-worker system.

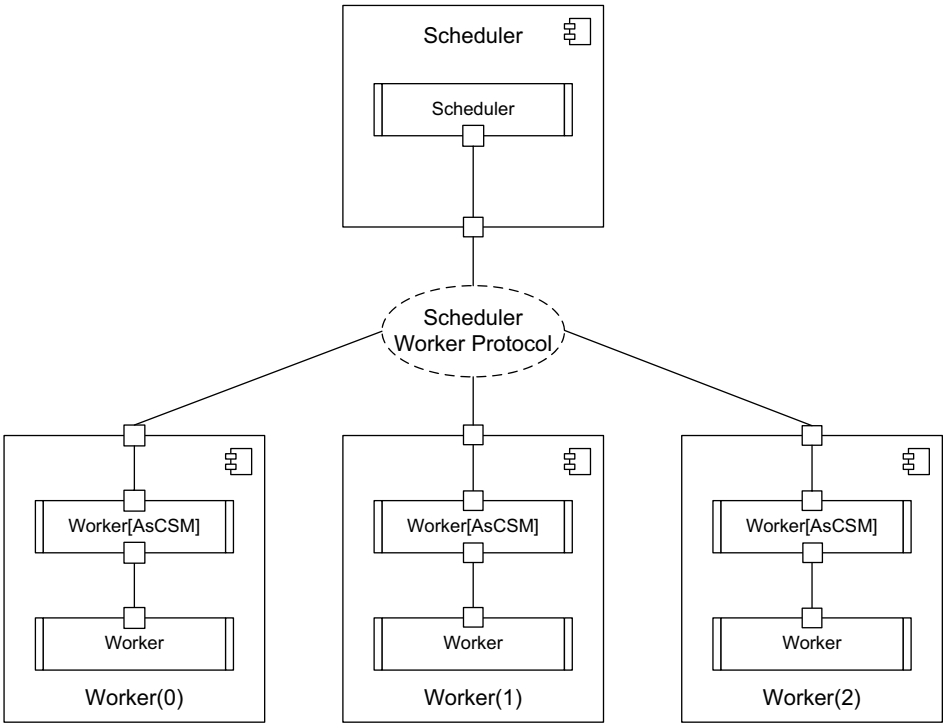


Figure 8.4: Component diagram for a Scheduler and three Workers

8.3 The Scheduler-Worker System

The architecture of the model for the scheduler-worker system is shown in Figure 8.4. All worker components have the same behavior, but run concurrently. Their detailed process *Worker(i)* is shown in Figure 8.5. They each have a critical section (state *Crit*) in which only one of them may be active at a time. This is to be coordinated via a partition *AsCSM* (critical section manipulator) and corresponding global process *Worker(i)[AsCSM]*, shown in Figure 8.6. The partition creates a view of the detailed process, by means of which the entrance and exit of the critical section, represented by subprocess *Busy*, can be controlled.

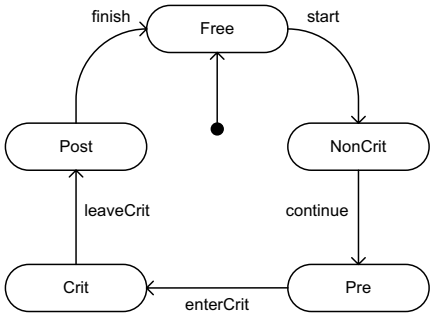


Figure 8.5: Process *Worker*

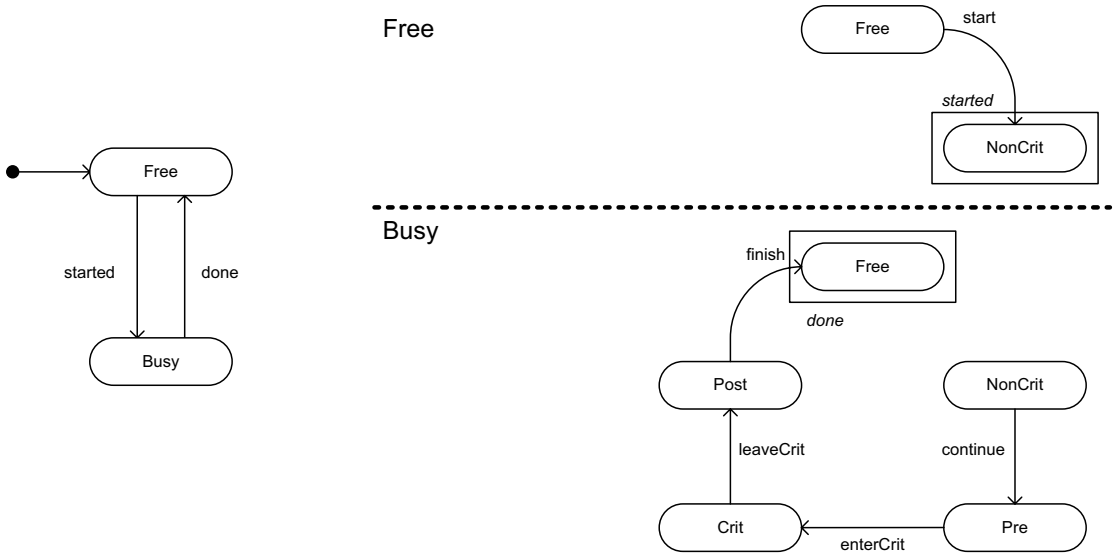


Figure 8.6: Global process *Worker[AsCSM]* and Partition *AsCSM* for Process *Worker*

It is up to the scheduler component to coordinate the critical sections of the workers. Process *Scheduler* is depicted in Figure 8.7. The process manages the three *Worker[AsCSM]* processes by means of interaction protocol *SchedulerWorkerProtocol*, whose six rules (two for each worker) are given in Table 8.8. Initially, process *Scheduler* schedules the workers in a non-deterministic order.

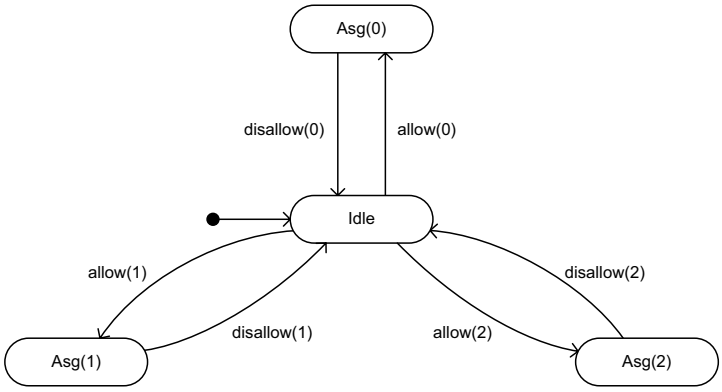


Figure 8.7: Process *Scheduler*

(R1,2,3)		Scheduler:	Idle	—allow(i)→	Asg(i)
	∗	Worker(i)[AsCSM]:	Free	—started→	Busy
(R4,5,6)		Scheduler:	Asg(i)	—disallow(i)→	Idle
	∗	Worker(i)[AsCSM]:	Busy	—done→	Free

Table 8.8: Interaction Protocol *SchedulerWorkerProtocol*

In the initial model, subprocess *Busy* of partition *AsCSM* is particularly large, while the overlap between subprocesses *Busy* and *Free* is minimal. The result of this is that the coordination of the critical section results in an almost sequential execution of the three workers. Our first change to the model is to improve this coordination in such a way that more concurrency between the workers is achieved.

8.4 First Migration

The first migration involves an increment in concurrency. To this end, we define new subprocesses with different traps for partition *AsCSM*. As a consequence, we also need to change global processes *Worker[AsCSM]* and interaction protocol *SchedulerWorkerProtocol*. The new subprocesses are intended to enable the scheduler to schedule the next worker directly after a worker has left its critical section, while it carries out some post activities.

Target and Migration Models

The intended *target models* for partitions *AsCSM* and processes *Worker[AsCSM]* after the first migration are shown in Figure 8.9, the consistency rules for the interaction protocol in Table 8.10. In the partition definition, subprocesses *Free* and *Busy* have been replaced by new subprocesses *OutCS* and *InCS*, which have considerably more overlap. Also, the traps have been enlarged to allow for more concurrency in the execution of the workers. The consistency rules of the target have a similar structure as the initial consistency rules, but deal with different global states and transitions.

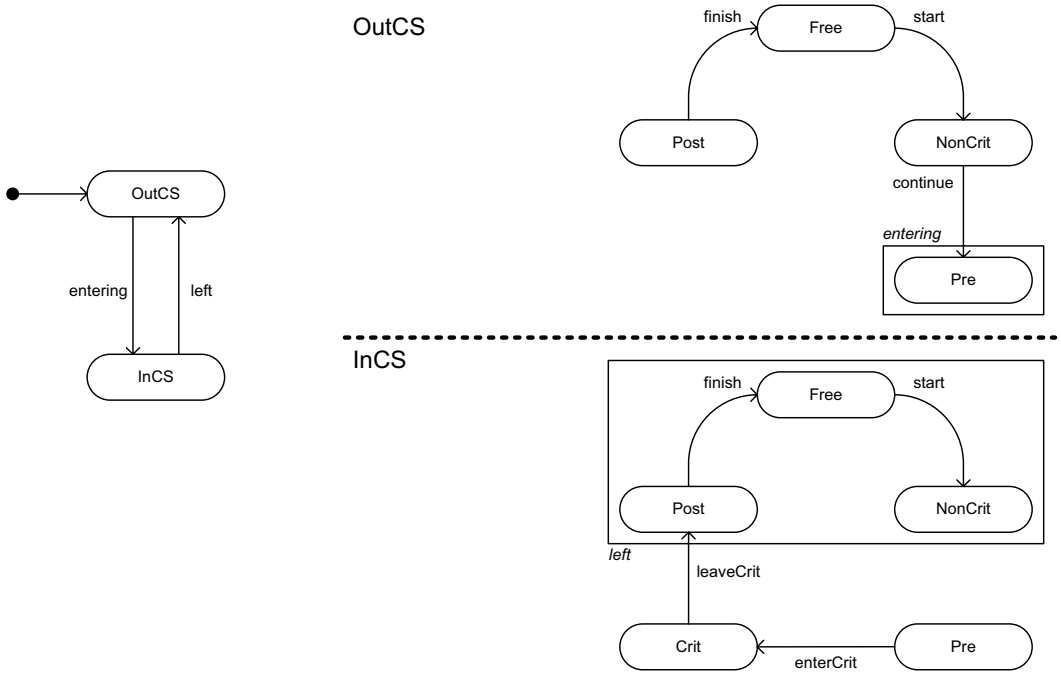


Figure 8.9: Process *Worker[AsCSM]*.EVOL1.*next* and Partition *AsCSM*.EVOL1.*next* for Process *Worker*

(N1,2,3)		Scheduler:	Idle	—allow(i)→	Asg(i)
	∗	Worker(i)[AsCSM]:	OutCS	—entering→	InCS
(N4,5,6)		Scheduler:	Asg(i)	—disallow(i)→	Idle
	∗	Worker(i)[AsCSM]:	InCS	—left→	OutCS

Table 8.10: Interaction Protocol *SchedulerWorkerProtocol*.EVOL1.*next*

Next to the target models, we explicitly define a bridge between the initial models and the target models, for both the *AsCSM* partitions and the *SchedulerWorkerProtocol* interaction protocol. We do this in terms of a *migration model*, which combines the initial model and the target model, and adds additional constructs to allow migration from the former to the latter. In Figure 8.11, we show global process *Worker[AsCSM].*EVOL1.*mig* at the level of partition *AsCSM.*EVOL1.*mig*. This partition contains the two subprocesses *Free* and *Busy* from the initial model plus the two subprocesses *OutCS* and *InCS* from the target model. The global process defines the transitions between the initial and the target subprocesses. In Table 8.12, the migration model for the interaction protocol is shown. Next to the initial and target consistency rules, additional rules $M1 - M3$ are added, which enable controlled migration from subprocesses *Free* and *Busy* to subprocesses *OutCS* and *InCS*.

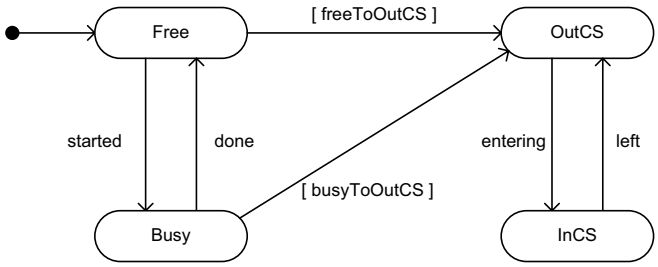


Figure 8.11: Process *Worker[AsCSM].*EVOL1.*mig*

Migration Scenario

In order to fully realize the migration from the initial to the target models, we define the temporary semantics for the transitions of process *McPal* contained in its subprocess *McPalDuringEvol*. We distinguish between three different types of transitions:

- Transitions *updateModels* and *cleanupModels* are meant for realizing changes to the model. For each of these two transitions, we specify which model creations, updates and deletions are performed.
- Transitions *scaffold* and *unscaffold* are meant for adding and removing the scaffold to and from the model, respectively. For these two transitions, we specify which temporary partitions and interaction protocols for the scaffold need to be added resp. removed.
- Transitions *allowMigration* and *restrictToNew* manage the runtime constraints involved in realizing the changes. We specify their semantics in terms of consistency rules which are anchored to the interaction protocol(s) of the scaffold.

(R1,2,3)		Scheduler :	Idle	—allow(i)→	Asg(i)
	∗	Worker(i)[AsCSM] :	Free	—started→	Busy
(R4,5,6)		Scheduler :	Asg(i)	—disallow(i)→	Idle
	∗	Worker(i)[AsCSM] :	Busy	—done→	Free
(N1,2,3)		Scheduler :	Idle	—allow(i)→	Asg(i)
	∗	Worker(i)[AsCSM] :	OutCS	—entering→	InCS
(N4,5,6)		Scheduler :	Asg(i)	—disallow(i)→	Idle
	∗	Worker(i)[AsCSM] :	InCS	—left→	OutCS
(M1,2,3)		Scheduler :	Asg(i)	—disallow(i)→	Idle
	∗	Worker(i)[AsCSM] :	Busy	—[busyToOutCS]→	OutCS,
		Worker(i+1)[AsCSM] :	Free	—[freeToOutCS]→	OutCS,
		Worker(i+2)[AsCSM] :	Free	—[freeToOutCS]→	OutCS

Table 8.12: Interaction Protocol *SchedulerWorkerProtocol*.EVOL1.*mig* (indices modulo 3)

For this migration scenario, the semantics of the transitions *updateModels* and *cleanupModels* are straightforward, because we only consider updates and no creations or deletions. With transition *updateModels*, we update the initial model with the migration model, as follows: Initial process *Worker(i)[AsCSM]* for each worker i is replaced by migration process *Worker(i)[AsCSM]*.EVOL1.*mig*, and the consistency rules of interaction protocol *SchedulerWorkerProtocol* are replaced by the consistency rules of interaction protocol *SchedulerWorkerProtocol*.EVOL1.*mig*. With transition *cleanupModels*, we perform a similar update: the migration model is updated to the target model.

The crucial concern in performing the above model updates *on-the-fly* is that we must be sure not to update the migration model to the target model too early. We should restrict the behavior of the processes *Worker[AsCSM]*.EVOL1.*mig* to the target model *Worker[AsCSM]*.EVOL1.*next* only after these processes actually have taken a transition from either state *Free* or *Busy* to state *OutCS*. We therefore organize the scaffold such that we can observe this. The scaffold consists of a partition on top of each of the global processes *Worker[AsCSM]*, and a single interaction protocol, managed by process *McPal*. Both are shown in Figure 8.13 and Table 8.14, respectively.

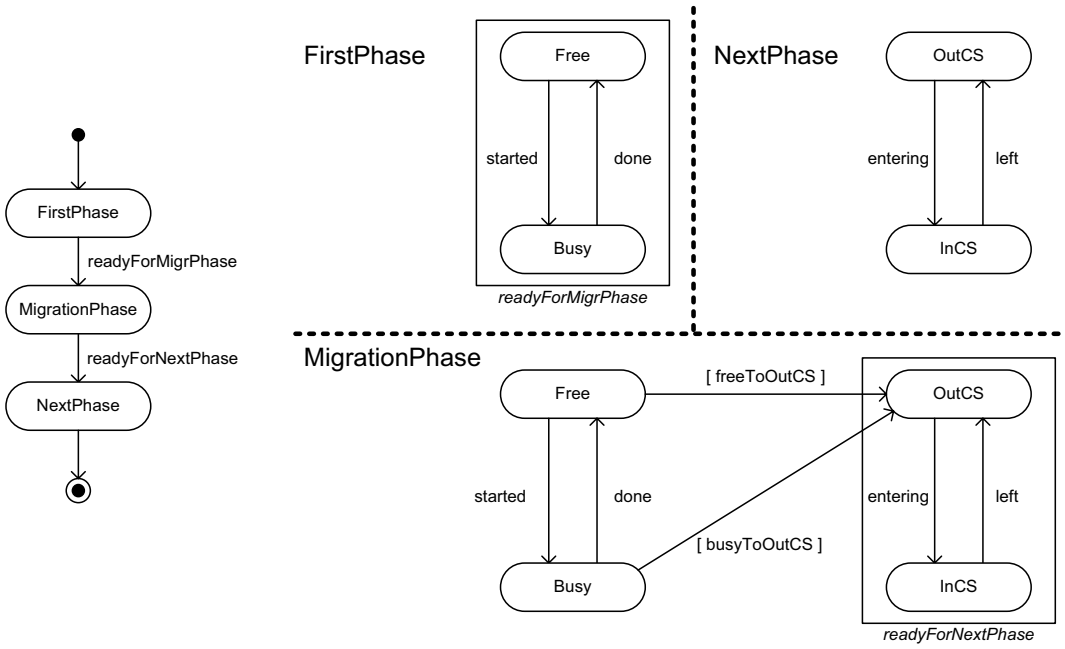


Figure 8.13: Process *CSM[AsEvol]* and Partition *AsEvol*

(R1)		McPal:	StartMigr	—allowMigration→	ContMigr
	∗	CSM(0)[AsEvol]:	FirstPhase	—readyForMigrPhase→	MigrationPhase,
		CSM(1)[AsEvol]:	FirstPhase	—readyForMigrPhase→	MigrationPhase,
		CSM(2)[AsEvol]:	FirstPhase	—readyForMigrPhase→	MigrationPhase
(R2)		McPal:	ContMigr	—restrictToNew→	EndMigr
	∗	CSM(0)[AsEvol]:	MigrationPhase	—readyForNextPhase→	NextPhase,
		CSM(1)[AsEvol]:	MigrationPhase	—readyForNextPhase→	NextPhase,
		CSM(2)[AsEvol]:	MigrationPhase	—readyForNextPhase→	NextPhase

Table 8.14: Protocol *EvolutionProtocol*

In subprocess *MigrationPhase* of partition *AsEvol*, we use a trap, *readyForNextPhase*, as an indicator for the fact that a process *Worker(i)[AsCSM]* has taken a transition to state *OutCS*. As the consistency rules anchored to interaction protocol *EvolutionProtocol* show, only if all processes *Worker(i)[AsCSM]* are inside this trap, process *McPal* is able to take transition *restrictToNew*. Note furthermore that subprocess *MigrationPhase* of partition *AsEvol* allows the taking of a transition to *OutCS*, but it does not *enforce* it, because the transitions between states *Free* and *Busy* are allowed as well.

The entire migration from the initial model to the target model can now be performed by *McPal* as follows. In its phase *McPalBeforeEvol*, process *McPal* defines the semantics for the transitions in phase *McPalDuringEvol*. Thereafter, *McPal* enters phase *McPalDuringEvol* and the actual migration starts. First, the scaffold is installed, which yields an extended model as shown in Figure 8.15 (scaffold elements are shown with bold lines). After that, by taking transition *updateModels*, *McPal* applies changes to partitions *AsCSM*, global processes *Worker(i)[AsCSM]* and interaction protocol *SchedulerWorkerProtocol* in order for the initial model to be updated to the migration model. Since the scaffold is present, the execution of the global processes *Worker(i)[AsCSM]* is restricted to the subprocesses *Free* and *Busy* (state *FirstPhase* of global processes *CSM(i)[AsEvol]*.

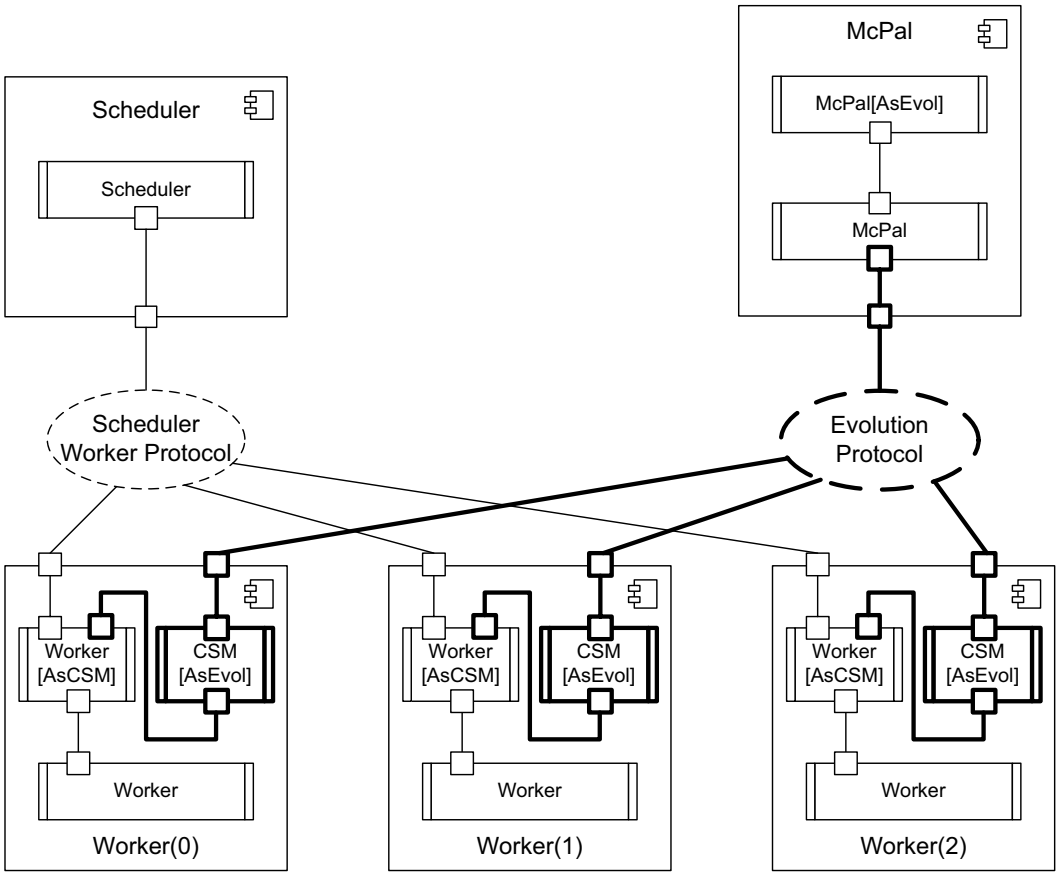


Figure 8.15: The Scheduler/Worker model being scaffolded for the first migration

By taking transition *allowMigration*, global processes *CSM(i)[AsEvol]* enter state *MigrationPhase*. Now, at least one of the transitions *[freeToOutCS]* and *[busyToOutCS]* is enabled for each of the processes *Worker(i)[AsCSM]*. It is up to the (non-deterministic) choice of interaction protocol *Scheduler-WorkerProtocol* when (if at all) migration takes place. However, as soon as one of the migration rules is applied, the migration is non-reversible. All *Worker(i)[AsCSM]* processes then change to state *OutCS*, by which they enter trap *readyForNextPhase* of partition *AsEvol* and *McPal*'s transition *restrictToNew* is enabled. After process *McPal* has taken this transition, the scaffold restricts the execution of global processes *Worker(i)[AsCSM]* to the subprocesses *InCS* and *OutCS*. As soon as process *McPal* takes transition *cleanupModels*, the migration model is updated to the target model . Finally, *McPal* removes the scaffold and returns to phase *McPalBeforeEvol*, ready for a new migration.

8.5 Second Migration

In the second migration, we change the non-deterministic scheduler into a round-robin scheduler. The idea is as follows. The scheduler, at each round-robin cycle, checks for each worker whether it wants to access its critical section. If this is the case, it allows the worker to enter the critical section, if not, it continues with checking the next worker. Our starting point for the second migration is the target model of the first migration – please note that, in the context of this second migration, we refer to this model as the *initial model*.

Target and Migration Models

The intended migration requires three different changes. Firstly, process *Scheduler* needs to be adapted to a round-robin scheduler. We create a target model for this process as shown in Figure 8.16. For each worker, the scheduler checks if it wants to enter its critical section, and decides upon this check whether or not to allow this. Secondly, we extend processes *Worker(i)[AsCSM]* with an extra subprocess, which enables us to distinguish whether a worker wants to access its critical section or not (yet).

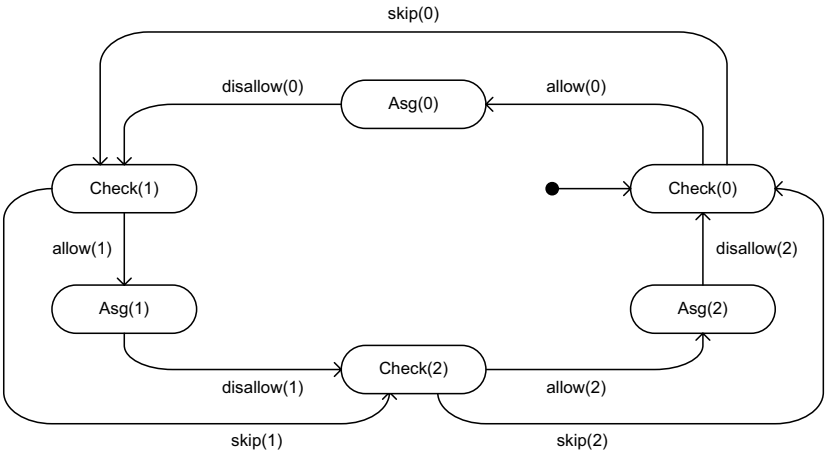


Figure 8.16: Process *Scheduler*.EVOL2.*next*

Partition *AsCSM.*EVOL2.*next* and global process *Worker(i)[AsCSM].*EVOL2.*next* are shown in Figure 8.17. The extra subprocess *OutCSBlock* contains two traps, *entering* and *notYetEntering*, on which the decision to grant access to the critical section can be taken (note also the absence of a direct transition from state *OutCS* to *InCS* in the global process, which is present in the initial model).

The third and last change to the model involves the interaction protocol *SchedulerWorkerProtocol*, in which the changes to process *Scheduler* and processes *Worker(i)[AsCSM]* need to be reflected. The consistency rules anchored to the interaction protocol in the target model are shown in Table 8.18.

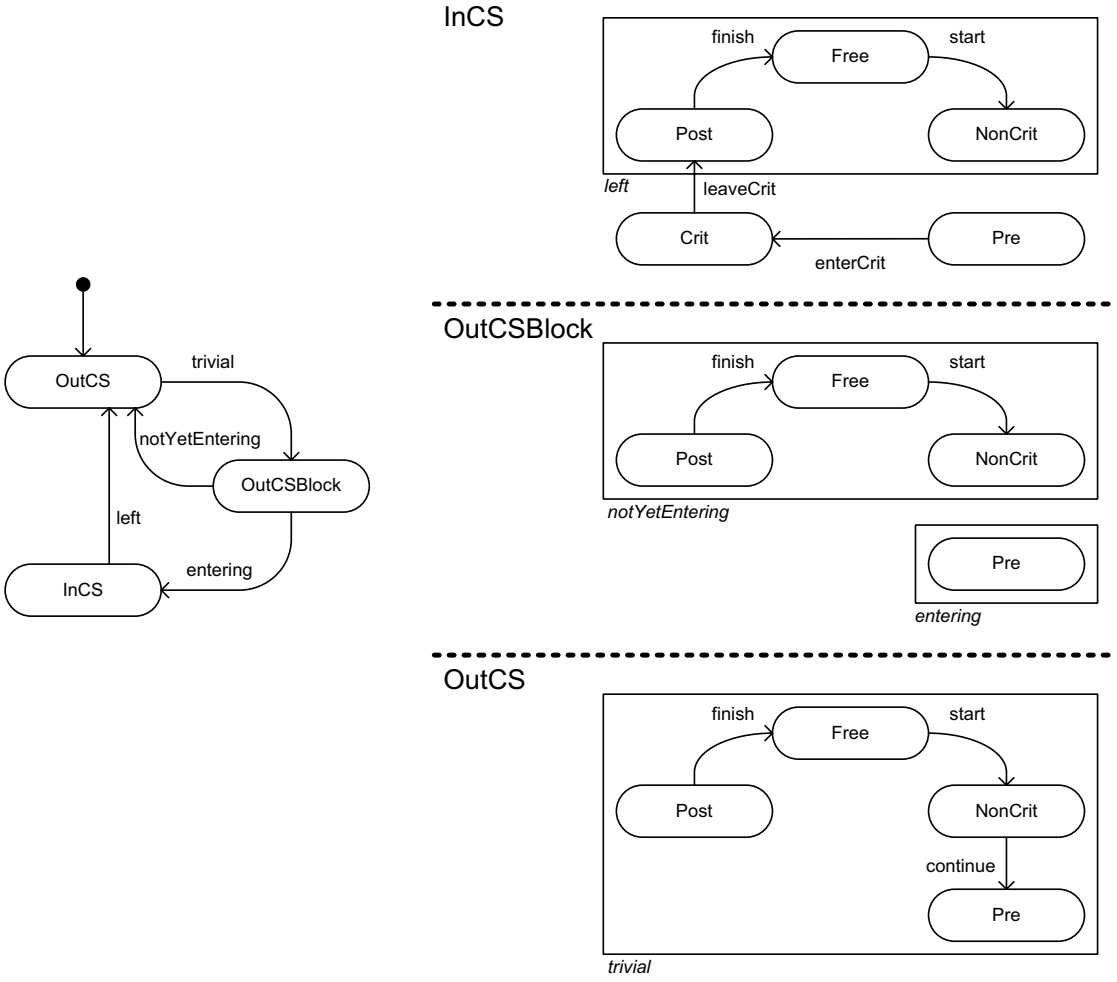


Figure 8.17: Process *Worker[AsCSM].*EVOL2.*next* and Partition *AsCSM.*EVOL2.*next*

(N'1,2,3)		Scheduler:	Check(i)	—allow(i)→	Asg(i)
	*	Worker(i)[AsCSM]:	OutCSBlock	—entering→	InCS
(N'4,5,6)		Scheduler:	Asg(i)	—disallow(i)→	Check(i+1)
	*	Worker(i)[AsCSM]:	InCS	—left→	OutCS,
		Worker(i+1)[AsCSM]:	OutCS	—trivial→	OutCSBlock
(N'7,8,9)		Scheduler:	Check(i)	—skip(i)→	Check(i+1)
	*	Worker(i)[AsCSM]:	OutCSBlock	—notYetEntering→	OutCS,
		Worker(i+1)[AsCSM]:	OutCS	—trivial→	OutCSBlock

Table 8.18: Interaction Protocol *SchedulerWorkerProtocol.*EVOL2.*next* (indices modulo 3)

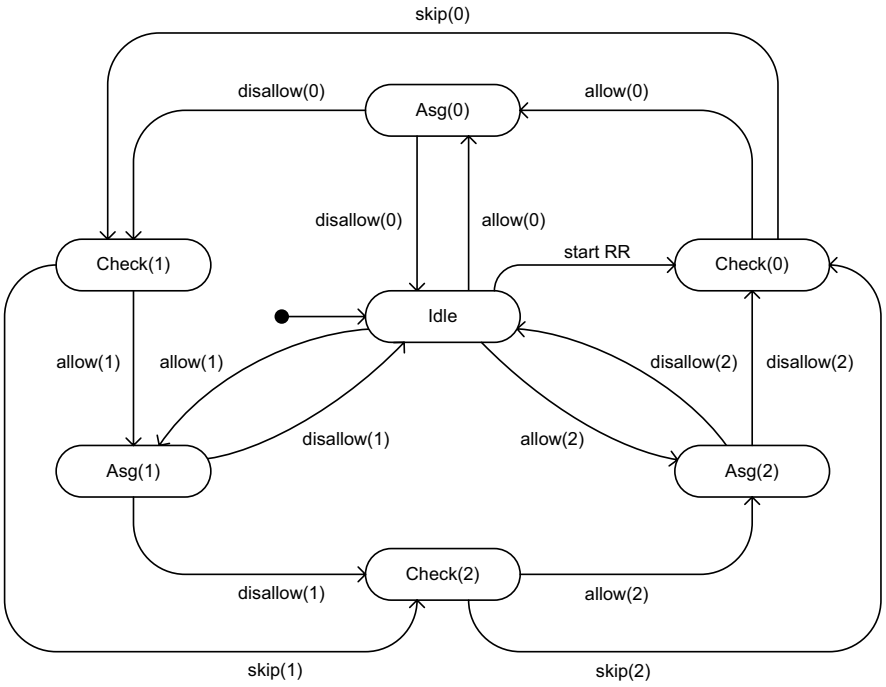


Figure 8.19: Process *Scheduler.*EVOL2.*mig*

The migration model, i.e. the intermediate model which bridges the initial and the target models, can be easily created by taking the union of the initial and the target model. We add two extra elements to ensure that the scheduler can be switched from non-deterministic into round-robin, regardless of the state in which it resides. To the union of the initial and target model for process *Scheduler*, we add a transition from state *Idle* to state *Check(0)*, as shown in Figure 8.19. To the union of the initial and target model for interaction protocol *SchedulerWorkerProtocol*, we add the corresponding consistency rule, which is shown in Table 8.20. This consistency rule bootstraps the round-robin cycle of process *Scheduler* if the current state at the moment of migration happens to be state *Idle*.

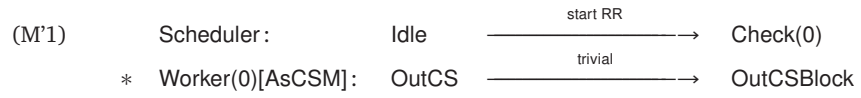


Table 8.20: Extra consistency rule for interaction protocol *SchedulerWorkerProtocol*.EVOL2.*mig*

Migration Scenario

We define the semantics of the transitions contained in subprocess *McPalDuringEvol* of process *McPal* in a similar manner as we did for the first migration. Again, transition *updateModels* updates the initial model to the migration model, and transition *cleanupModels* updates the migration model to the target model. In order to perform the changes on-the-fly, the only thing we need to ensure in this case is that we do not cleanup the migration model before process *Scheduler* has started with its round-robin cycle. The states of the workers play no role in this case, since they depend on the management performed by the scheduler. Therefore, we create a small scaffold, which consists of a partition on top of process *Scheduler*, coordinated by an interaction protocol which is managed by *McPal*. We show the subprocesses of partition *AsEvol* for process *Scheduler* in Figure 8.21. The global process at the level of this partition is equal to global process *CSM[AsEvol]* of the first migration. Trap *readyForNextPhase* indicates that process *Scheduler* has started with the round-robin cycle. The consistency rules for the interaction protocol *EvolutionProtocol* are similar to those of the interaction protocol for the scaffold in the first migration.

The migration scenario executed by process *McPal* differs in one respect from the first migration. In the first migration, we allowed the model to continue with the behavior specified in its initial model, and waited until we observed that all processes in the model were ready for migration to the target model. In the second migration, however, subprocess *MigrationPhase* actually *enforces* the scheduler to start its round-robin cycle: it cannot continue with its old non-deterministic behavior.

Now that we have illustrated the technique for *modeling* evolution on-the-fly, we briefly give the reader an idea of how this technique can be easily *implemented* with the PARADE tools.

8.6 Implementing Evolution On-the-Fly

When modeling the evolution of a model on-the-fly in PARADIGM, we basically assume that creations, updates and deletions of entities in the model "take place": at the taking of transitions *updateModels* and *cleanupModels* in process *McPal*, the model-to-be-evolved is updated according to the semantics defined for these transitions. The same holds for the addition and removal of the scaffold, done at

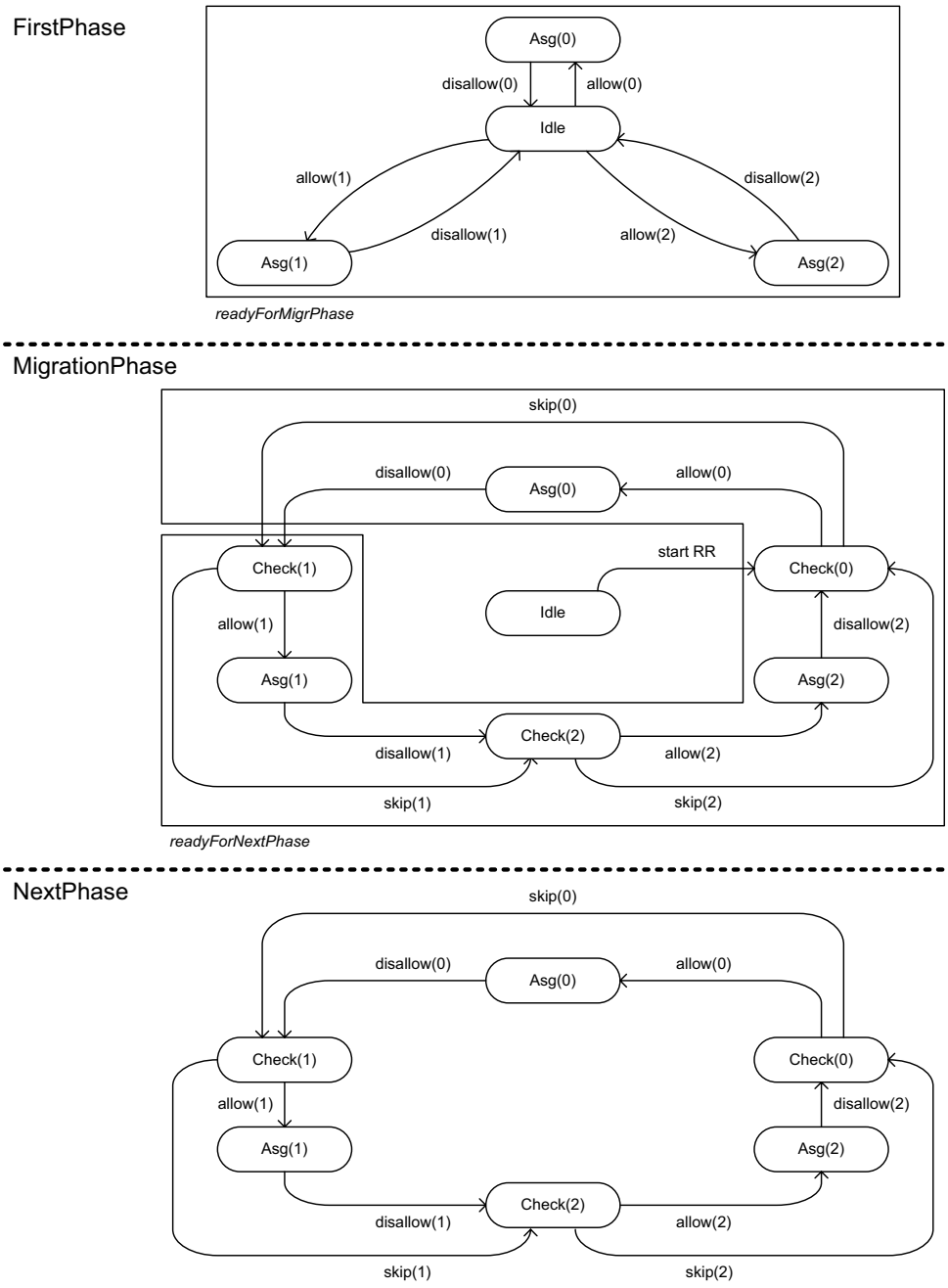


Figure 8.21: Partition *AsEvol* for Process *Scheduler*

transitions *scaffold* and *unscaffold*. In an implementation, changes to the model need to be concretely performed. The PARADE distributed runtime environment (DRE), which is based upon the PARADISE interpreter framework for PARADIGM, provides support to realize such changes to the model during its execution. We refer to Chapter 5 for a more detailed presentation of the terminology used in PARADE.

In the implementation of evolution on-the-fly with the PARADE tools, we start with the execution of a model which only contains one component, *McPal*. This component does not only manage the *evolution* of a model, but it also performs the *creation* of this model in the PARADE DRE. Hence, the first migration is a migration from an *empty* model to an *initial* model. In subsequent migrations, the actual *evolution* of the initial model is performed.

The semantics of transitions *initialize*, *scaffold* and *unscaffold*, and *updateModels* and *cleanupModels* are defined in an *implementation model*, one of the extensions to PARADIGM models provided by PARADE. The implementation model specifies for each of these transitions an *action class* which performs the activities associated with the transition at hand. The two code snippets below show the Java code of two action classes for transition *initialize* and *scaffold*, respectively. As can be seen, a *System Handler* is used to perform updates to the model. For transition *initialize*, an update of component *McPal* itself is performed, as specified in XML-document “mcpalduringevol.component". Transition *scaffold* involves the creation, update or deletion of elements outside component *McPal*, the so-called *context* of the component. XML-document “schedulerworker.evol1.context" specifies the exact update. The updates are sent to the appropriate hosts and components in the PARADE DRE via the same system channels used for the creation and start-up of the initial model.

```
public class InitializeAction extends Action {
    @Override
    public void run() {
        this.getSystemHandler().updateComponentLayout("mcpalduringevol.component");
    }
}
```

```
public class ScaffoldAction extends Action {
    @Override
    public void run() {
        this.getSystemHandler().updateContextLayout("schedulerworker.evol1.context");
    }
}
```

The operations *updateComponentLayout* and *updateContextLayout* of the PARADE system handler perform the creations, updates and deletions *as if* they are performed as a single atomic action. At this level of modeling, where we consider the changes involved in the taking of a single transition, these operations free the modeler from thinking about the order in which changes are applied. In addition, in case of newly created processes and interaction protocols, these operations take care of their startup. Likewise, in case of processes and interaction protocols which must be deleted, a shutdown is performed prior to deletion. With these features, the PARADE system handler provides a simple and powerful interface for a modeler to evolve its PARADIGM model on-the-fly, while it is being executed in the PARADE distributed runtime environment.

8.7 Discussion

In the case study about *evolution on-the-fly* presented in this chapter, we have shown that the PARADIGM language is suitable for creating models which coordinate their own evolution. We reused the idea of having a separate component *McPal* for coordinating the evolution, as presented in [46] and [40]. The contribution of this case study to the existing approach is twofold.

Firstly, we have established a clear general approach to the coordination of evolution on-the-fly, called *scaffolding*. The approach is general in the sense that we structure the scaffold for a model in the same way, regardless of the precise changes which must be applied. Furthermore, the scaffold is perceived as a *temporary* construct, which leads to a cleaner model-to-be-evolved and a better support for adapting the scaffold to the needs of a certain set of changes. We made extensive use of *interaction protocols* as introduced in Chapter 4, to separate the consistency rules of the model-to-be-evolved from those that play a role in coordinating the evolution on-the-fly.

Secondly, have illustrated the *implementation* of evolution on-the-fly for PARADIGM models by means of the PARADE tools (Chapter 5). These tools are able to create, update and delete entities within running PARADE components, as well as components themselves. The evolution of the Scheduler-Worker model in this case study only demonstrated the update of processes, partitions and interaction protocols, and not their creation or deletion. Further study is needed to show that creations and deletions can be performed in a similar way, with similar scaffolds to coordinate the changes. Nevertheless, we have used creations and deletions for constructing and removing the scaffolds themselves.

Furthermore, we point out the clarity of the structure of process *McPal* in our approach. The six transitions contained in subprocess *McPalDuringEvol* are actually three nested groups, each consisting of two complementary transitions. The first and last transitions, *scaffold* and *unscaffold*, construct and destruct the *coordination structure* needed for the migration to take place: a temporary growing and shrinking of the model. The nested transitions *updateModels* and *cleanupModels* realize changes to the *model-to-be-evolved*: again, growing and shrinking, expansion and contraction, be it at the level of the persistent model. Finally, the middle transitions use the temporary coordinative structure, the scaffold, to evolve the model in its *execution*: once more, growing and shrinking, but now in terms of the possible runtime states of the model. Consequently, in our approach, evolution on-the-fly can be perceived as a matter of expansion and contraction at three different levels: the coordination needed for on-the-fly evolution itself, the static model-to-be-evolved, and the runtime state of the model being executed.

Chapter 9

Conclusions and Future Work

For both the *Foundations* and the *Case Studies* part of the thesis, we summarize and draw conclusions. In addition, we present ideas for future work in three different directions: theory, tool development and applications.

9.1 Conclusions for the Foundations (Chapters 2 to 5)

Summary

We started our research with developing a *software implementation* of the language (Chapter 2), in terms of PARADISE, a distributed interpreter framework for PARADIGM models (Chapter 3). Thereby, we focused on adopting a key feature of the language as a major design principle for the implementation: as much concurrency between processes as possible. Based on the insights gained by this effort, we *extended* the modeling language (Chapter 4) in order to incorporate *interaction protocols*, a concept for structuring consistency rules and for ensuring their conflict-free application. This latter aspect allowed us to generalize the shape of consistency rules in order to model interaction between peers, without an additional manager process. We implemented the extended PARADIGM language in the PARADE *software tools* (Chapter 5), with which PARADIGM models can be edited, executed and visualized. Here, the focus was on a distributed runtime environment which supports evolution on-the-fly.

Conclusions

From both a practical and a theoretical perspective, the interaction protocol concept is a valuable addition to the PARADIGM *language.* The implementation of PARADIGM in a fully distributed manner has encouraged us to reconsider the application of consistency rules in the language. This has led to the introduction of the interaction protocol concept. According to our findings, this concept has proven to be a valuable addition to the PARADIGM language for the following reasons:

- It introduces separation of concerns at the level of consistency rules, offering a new structuring dimension which differs from a split-up along the manager processes in a model.
- Since it has its own behavior, we can easily introduce consistency rules with empty manager parts, which are applied without explicit coordination by means of a manager process. This allows for modeling the interaction between peers.

- It provides a notion of concurrency in the application of consistency rules which was not explicitly present in the original language. If two consistency rules are anchored to two different interaction protocols, they can be applied concurrently.
- It ensures that consistency rules are applied atomically and in a conflict-free manner, provided the definition of soundness in Chapter 4 is taken into account. This definition is based on the light-weight statically verifiable condition of potential conflict.

It is possible to implement PARADIGM *in a fully distributed manner, whereby each process runs on a separate processor and communication between processors is done solely via asynchronous unbounded FIFO channels.* In the PARADISE distributed interpreter framework, each process in a PARADIGM model runs on a different *virtual node*. The implementation of partitions requires a minimum of communication between the detailed and global process involved. The implementation of consistency rules is best served by using at least one additional virtual node to keep them separate from the nodes on which the processes run. Relative efficient application of consistency rules is possible by exploiting the semantic properties of the trap concept.

The usage of views on views can be achieved in PARADIGM. Our effort in tool development has shown that the application of partitions on top of global processes (views on views), like we used in Chapter 8, is a straightforward extension. We believe that an operational semantics for this notion can be established without considerable effort. At the level of our implementation in PARADISE, it puts an additional requirement on the application of consistency rules: in order to apply a consistency rule, a two-phase commit is needed over its manager process and all global processes mentioned in its employee part.

Evolution on-the-fly with PARADIGM *can be implemented by building a framework of interpreters for individual language concepts which can be dynamically composed by a runtime environment into an interpreter for a full model.* The PARADE distributed runtime environment (DRE) provides support for executing models which evolve on-the-fly. This support is based on the flexibility of our PARADISE distributed interpreter framework. The PARADE DRE *dynamically composes* a distributed PARADISE interpreter for a PARADIGM model based on a model specification in XML. Hence, in PARADE there is no intrinsic difference between starting a system and evolving it.

9.2 Conclusions for the Case Studies (Chapters 6 to 8)

Summary

For purposes of validation, we successfully applied the extended PARADIGM language and the PARADE tools in three different case studies, focusing on three related but different applications of the language and the tools. In Chapter 6, we focused on the modeling of non-trivial interaction in the context of reconfigurable software. In Chapter 7, we modeled the interaction of three different shapes of generic branch-and-bound algorithms. We used the software extensions of the PARADE tools to attach a running branch-and-bound solver to the model, which provided insights in the applicability of the notions in PARADIGM for software design. Finally, in Chapter 8, we applied PARADIGM and the PARADE tools in combination for the creation and execution of a model evolving on-the-fly. We introduced a generic technique, *scaffolding*, for the enforcement of temporary constraints on the behavior of PARADIGM processes.

Conclusions for Chapter 6

Covering interaction protocols can be applied to address different aspects of interaction separately. Manager processes can be applied hierarchically to coordinate lower level interactions in terms of higher level interactions. In the case study of Chapter 6, we used covering interaction protocols to distinguish between different domains of communication at a lower level. We also used a higher-level interaction protocol on top of manager processes which coordinate the lower-level interaction protocols. This way of working shows that manager processes can be applied as a "componentization" of an interaction protocol to gain control over lower-level interaction at a higher level of abstraction.

The PARADE *tools can be used for the modeling and execution of open* PARADIGM *models*. As we showed in Chapter 6, a PARADIGM model does not need to model an entire closed system in order to be executed in the PARADE distributed runtime environment. Since all processes are treated in an equal manner in the PARADISE distributed interpreter framework, a global process which is not connected to an interaction protocol can be executed as if it were a detailed process with a simple selector. This strengthens the applicability of PARADIGM and the PARADE tools for modeling open systems, e.g. in the context of distributed services.

Conclusions for Chapter 7

Self-managing interaction protocols can be used as an inter-component coordination mechanism for the control flow in models of reconfigurable systems. In Chapter 7, we successfully applied self-managing interaction protocols for the coordination of an arbitrarily sized pool of independent visitor processes. At the modeling level, the addition or removal of visitor processes only requires changes to the self-managing interaction protocol(s). Hence, no changes to any processes in the model are necessary to accommodate such reconfigurations.

The application of PARADE*'s software extensions in the case study of Chapter 7 shows the value of the* PARADIGM *world view for purposes of software design.* We developed software for a specific branch-and-bound solver in terms of a small set of Java classes. By using *implementation models* with *choice states* as introduced in Chapter 5, and combining these with intermediate *action classes*, we created a bridge between the implemented branch-and-bound solver and a PARADIGM model, using the latter one for its coordination. Although the direct application of PARADIGM models for the purpose of software development requires considerable performance improvements and probably also code generation facilities, the way of working adopted in Chapter 7 shows that a PARADIGM model can be used as a high-level coordination mechanism for a lower-level object-oriented software implementation. In this case study, this exercise did not only yield a very precise understanding of how the intended software functionality must be implemented in object-oriented terms, but it also led to a better understanding of the intended meaning of transition labels in the model.

Another interesting application of the world view of PARADIGM in the design of software can be found in the fact that we successfully separated the core activities of the algorithm from the side issues of communication as introduced in the componentized and parallel versions of the algorithm. The core activities were captured in detailed processes, while the side issues were taken up in global processes. In other words, we used the *multiple views* principle of PARADIGM at the *modeling level* as a means to add *software aspects* at the *implementation level*.

Conclusions for Chapter 8

PARADIGM *can be effectively applied as a language to create models of systems which evolve on-the-fly. The* PARADE *tools support the execution and visualization of such models.* In Chapter 8, we showed a PARADIGM model which coordinates its own evolution while it is being executed. An additional process *McPal* takes care of the necessary coordination. The PARADE distributed runtime environment supports creations, updates and deletions of modeling artifacts while they are interpreted by a dynamically constructed PARADISE distributed interpreter. The creations, updates and deletions automatically result in the appropriate construction, update and destruction of concept interpreters in the interpreter.

The scaffolding technique introduced in Chapter 8 enables modelers to coordinate the evolution of a PARADIGM *model on-the-fly via temporary coordination constructs with a uniform architecture. This leads to cleaner* PARADIGM *models and a better focus on locality of change.* Scaffolding means the temporary addition of an interaction protocol and a set of global (employee) processes on top of existing processes in a model. These temporary constructs coordinate the expansion and contraction of the behavior of a specific set of processes during evolution on-the-fly, while leaving remaining constructs in the model untouched. We strongly believe that the technique is generally applicable for any change to a PARADIGM model. The technique is fully supported by our tools.

9.3 Future Work

Future Work in the Direction of Theory

In our implementation of PARADIGM, we decided to treat detailed and global processes in the same way. This yields a new perspective on partitions, consistency rules and interaction protocols in the language. In fact, we can regard these as different types of *constraints* on the behavior of processes. It would be interesting to see whether these constraints could be formalized in terms of communicating *automata*, like e.g. the constraint automata [6, 88] used as an operational semantic model for the exogenous coordination language Reo [5]. For example, the behavior of two processes related by a partition constraint could perhaps be translated into a set of two or three automata, for which a product automaton can be taken to yield the behavior of the two constrained processes in combination.

A more formal approach to the scaffolding technique presented in Chapter 8 can further underpin the strengths of this idea. Especially the three-level view on expansion and contraction (coordination level, modeling level and runtime behavioral level) is useful as a basis for more formal approaches to evolution on-the-fly in PARADIGM.

Finally, an interesting exercise would be to transform the informal operational semantics of our implementation of PARADIGM, now presented as pseudo code in Appendix A, into a specification in a suitable high-level modeling language. In particular, we consider the object-oriented modeling language *Creol* [56, 57], which is executable on the rewriting logic platform Maude [20]. The Creol language is being applied as a high-level modeling language in the European CREDO Project [94], and also for the purpose of testing multi-threaded asynchronous systems [1]. It adopts asynchronous message passing between concurrent objects running on individual processors, which highly corresponds with the notion of asynchronous channel communication between processes running on virtual nodes as used in the PARADISE interpreter framework.

Future Work in the Direction of Tool Development

As we concluded in Chapter 5, the tools developed in the course of this thesis are a useful basis for the further development of an integrated tool suite for PARADIGM. We envision the development of a graphical editor for PARADIGM models which includes support for modeling evolution on-the-fly. Secondly, a range of tools for the *analysis* of PARADIGM models can be developed. The theoretical directions for future work pointed out above can be used as a starting point for this.

Another useful direction for tool development is to establish efficient *code generators* for PARADIGM models, in order to build skeletons for distributed software systems. It is also possible to improve upon performance in the existing distributed interpreter for PARADIGM through certain optimizations. Typical examples are the removal of the necessity to implement global processes with a dedicated thread of control, or the establishment of techniques to compile the control flow within PARADIGM models into more efficient means of communication at the implementation level. This way, performance could be improved, possibly losing some adaptivity.

Future Work in the Direction of Applications

The results of this thesis are envisaged to be especially useful in the context of the modeling and analysis of dynamic orchestrations or choreographies of *distributed evolving services*. A precise understanding of the non-trivial and evolving interaction between these services is a necessity. The PARADIGM language, especially in its extended shape, could be valuable in this field for the study of multiple related concerns in interaction, like we showed in Chapter 6, or for the study of evolution on-the-fly, as presented in Chapter 8. More applications of the language are needed in this direction. In this thesis, we did not validate the concept of *non-covering* interaction protocols. We foresee applications for this concept in the direction of e.g. delegation of managership, as we mentioned in Chapter 4.

The methodological aspects of our approach towards combining models and software in Chapter 7, especially its reflection on the design of object-oriented sofware, should be studied further. This approach could yield valuable insights for the community which studies model driven software development.

Bibliography

[1] B. Aichernig, A. Griesmayer, R. Schlatte, and A.W. Stam. Modeling and Testing Multi-Threaded Asynchronous Systems with Creol. In *Proc. TTSS 2008*, volume 243 of *ENTCS*, pages 3–14. Elsevier, 2009.

[2] J.P. Almeida, M. van Sinderen, D.A.C. Quartel, and L.F. Pires. Designing Interaction Systems for Distributed Applications. *IEEE Distributed Systems Online*, 6(3), 2005.

[3] F. Arbab. MANIFOLD version 2: Language Reference Manual. Technical report, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, 1995.

[4] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In *Proc. COORDINATION 1996*, volume 1061 of *LNCS*, pages 34–56. Springer, 1996.

[5] F. Arbab. Reo: a Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[6] F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani. Modeling Component Connectors in Reo by Constraint Automata. *Electronic Notes in Theoretical Computer Science*, 97:25–41, 2004.

[7] F. Arbab, F.S. de Boer, J.V. Guillen Scholten, and M.M. Bonsangue. MoCha: A Middleware Based on Mobile Channels. In *Proc. COMPSAC 2002*, pages 667–673. IEEE Computer Society, 2002.

[8] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[9] E.M.M. Babu, A. Malinowski, and J. Suzuki. Matilda: A Distributed UML Virtual Machine for Model–Driven Software Development. In *Proc. WMSCI 2005, Orlando, Florida, USA*, 2005.

[10] J.A. Bergstra and P. Klint. The ToolBus Coordination Architecture. In *Proc. COORDINATION 1996*, volume 1061 of *LNCS*, pages 75–88. Springer, 1996.

[11] M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutellà, and G. Zavattaro. A Transition System Semantics for the Control-Driven Coordination Language MANIFOLD. *Theoretical Computer Science*, 240(1):3–47, 2000.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison–Wesley, 1999.

[13] A. de Bruin, G.A.P. Kindervater, and H.W.J.M. Trienekens. Towards an Abstract Parallel Branch and Bound Machine. Technical Report 96, Erasmus University Rotterdam, Faculty of Economics, 1995.

[14] L. Cabac and D. Moldt. Formal Semantics for AUML Agent Interaction Protocol Diagrams. In *AOSE 2004, Revised Selected Papers*, volume 3382 of *LNCS*, pages 47–61. Springer, 2005.

[15] E. Cariou and A. Beugnard. The Specification of UML Collaborations as Interaction Components. In *Proc. UML 2002 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 3–23. Springer, 2002.

[16] E. Cariou, A. Beugnard, and J.M. Jézéquel. An Architecture and a Process for Implementing Distributed Collaborations. In *Proc. EDOC 2002*, pages 132–143. IEEE Computer Society, 2002.

[17] H.N. Castejón. Synthesizing State-Machine Behaviour from UML Collaborations and Use Case Maps. In *Proc. SDL 2005*, volume 3530 of *LNCS*, pages 339–359. Springer, 2005.

[18] H.N. Castejón and R. Bræk. A Collaboration-Based Approach to Service Specification and Detection of Implied Scenarios. In *Proc. SCESM 2006*, pages 37–43. ACM, 2006.

[19] J. Clausen and M. Perregaard. On the best Search Strategy in Parallel Branch-and-Bound: Best–First Search versus Lazy Depth–First Search. *Annals of Operations Research*, 90:1–17, 1999.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proc. RTA 2003*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

[21] L. de Alfaro and T.A. Henzinger. Interface Automata. In *Proc. ESEC/FSE 2001*, pages 109–120. ACM, 2001.

[22] L. de Alfaro and T.A. Henzinger. Interface Theories for Component-Based Design. In *Proc. EMSOFT 2001*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[23] F.S. de Boer, M.M. Bonsangue, L.P.J. Groenewegen, A.W. Stam, S. Stevens, and L.W.N. van der Torre. Change Impact Analysis of Enterprise Architectures. In *Proc. IRI 2005*, pages 177–181. IEEE Systems, Man and Cybernetics Society, 2005.

[24] F.S. de Boer, M.M. Bonsangue, J. Jacob, A.W. Stam, and L.W.N. van der Torre. A Logical Viewpoint on Architectures. In *Proc. EDOC 2004*, pages 73–83. IEEE Computer Society, 2004.

[25] F.S. de Boer, M.M. Bonsangue, J. Jacob, A.W. Stam, and L.W.N. van der Torre. Enterprise Architecture Analysis with XML. In *Proc. HICSS 2005*. IEEE Computer Society, 2005.

[26] G. Decker and A. Barros. Interaction Modeling using BPMN. In *BPM 2007 Workshops, Revised Selected Papers*, volume 4928 of *LNCS*, pages 208–219. Springer, 2008.

[27] G. Decker and F. Puhlmann. Extending BPMN for Modeling Complex Choreographies. In *Proc. CoopIS, DOA, ODBASE, GADA, and IS 2007, Part I*, volume 4803 of *LNCS*, pages 24–40. Springer, 2007.

[28] B. Dobing and J. Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, 2006.

[29] B.M. Duc. *Real-Time Object Uniform Design Methodology with UML*. Springer, 2007.

[30] The Eclipse Development Platform. Website, 2008. `http://www.eclipse.org/`.

[31] The Eclipse Modeling Framework (EMF) Project. Website, 2008. `http://www.eclipse.org/modeling/emf/`.

[32] G. Engels and L.P.J. Groenewegen. Specification of Coordinated Behaviour in the Software Development Process. In *Proc. EWSPT 1992*, volume 635 of *LNCS*, pages 58–60. Springer, 1992.

[33] G. Engels and L.P.J. Groenewegen. SOCCA: Specifications of Coordinated and Cooperative Activities. *Software Process Modelling and Technology*, pages 71–102, 1994.

[34] G. Engels and L.P.J. Groenewegen. Object-Oriented Modeling: a Roadmap. In *Proc. ICSE 2000 – Future of SE Track*, pages 103–116. ACM, 2000.

[35] V. Ermagan and I.H. Krüger. A UML2 Profile for Service Modeling. In *Proc. MoDELS 2007*, volume 4735 of *LNCS*, pages 360–374. Springer, 2007.

[36] Foundation for Intelligent Physical Agents. FIPA Interaction Protocol Library Specification, August 2001.

[37] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (third edition)*. Addison–Wesley, 2004.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification, Third Edition*. Addison–Wesley, 2005.

[40] L.P.J. Groenewegen and E.P. de Vink. Dynamic System Adaptation by Constraint Orchestration. *Computing Research Repository*, abs/0811.3492, 2008.

[41] L.P.J. Groenewegen and G. Engels. Coordination by Behavioural Views and Communication Patterns. In *Proc. EWSPT 1995*, volume 913 of *LNCS*, pages 189–192. Springer, 1995.

[42] L.P.J. Groenewegen, N. van Kampenhout, and E.P. de Vink. Delegation Modeling with Paradigm. In *Proc. COORDINATION 2005*, volume 3453 of *LNCS*, pages 94–108. Springer, 2005.

[43] L.P.J. Groenewegen, A.W. Stam, P.J. Toussaint, and E.P. de Vink. Paradigm as Organization-Oriented Coordination Language. *Electronic Notes in Theoretical Computer Science*, 150(3):93–113, 2006.

[44] L.P.J. Groenewegen, M.R. van Steen, and G. Oosting. Modelling Parallel Phenomena. In *Proc. ESC 1986, Antwerp, Belgium*, pages 45–51, September 1986.

[45] L.P.J. Groenewegen and E.P. de Vink. Operational Semantics for Coordination in Paradigm. In *Proc. COORDINATION 2002*, volume 2315 of *LNCS*, pages 191–206. Springer, 2002.

[46] L.P.J. Groenewegen and E.P. de Vink. Evolution-On-The-Fly with Paradigm. In *Proc. COORDINATION 2006*, volume 4038 of *LNCS*, pages 97–112. Springer, 2006.

[47] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., 2001.

[48] J.V. Guillen Scholten. *MoCha, easyMoCha, chocoMoCha Electronic Manual, version 0.96b*. Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, 2005.

[49] J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. PhD thesis, Leiden University, The Netherlands, IPA Dissertation Series 2006-21, 2007.

[50] J.V. Guillen Scholten and F. Arbab. Coordinated Anonymous Peer-to-Peer Connections with MoCha. In *FIDJI 2004, Revised Selected Papers*, volume 3409 of *LNCS*, pages 68–77. Springer, 2004.

[51] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[52] D. Harel. Statecharts in the Making: a Personal Account. In *Proc. HOPL III (2007)*, pages 5–1–5–43. ACM, 2007.

[53] Information Technology for European Advancement 2. Website, 2008. `http://www.itea2.org/`.

[54] International Telecommunication Union (ITU). ITU-T Recommendation Z.100 (11/99) – Specification and Description Language (SDL), 1999.

[55] J. Jacob. A Rule Markup Language and its application to UML. In *Proc. ISoLA 2004*, volume 4313 of *LNCS*, pages 26–41. Springer, 2004.

[56] E.B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. In *Proc. SEFM 2004*, pages 188–197. IEEE Computer Society, 2004.

[57] E.B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, 2007.

[58] H. de Jong and P. Klint. ToolBus: The Next Generation. In *Proc. FMCO 2002*, volume 2852 of *LNCS*, pages 220–241. Springer, 2003.

[59] H. Jonkers, M.M. Lankhorst, R. van Buuren, S.J.B.A. Hoppenbrouwers, M.M. Bonsangue, and L.W.N. van der Torre. Concepts for Modelling Enterprise Architectures. *International Journal of Cooperative Information Systems, special issue on Architecture in IT*, 13(3):257–287, 2004.

[60] H. Jonkers, R. van Buuren, F. Arbab, F.S. de Boer, M.M. Bonsangue, H. Bosma, H.W.L. ter Doest, L.P.J. Groenewegen, J.V. Guillen Scholten, S.J.B.A. Hoppenbrouwers, M.-E. Iacob, W. Janssen, M.M. Lankhorst, D. van Leeuwen, H.A. Proper, A.W. Stam, L.W.N. van der Torre, and G. Veldhuijzen van Zanten. Towards a Language for Coherent Enterprise Architecture Descriptions. In *Proc. EDOC 2003*, pages 28–39. IEEE Computer Society, 2003.

[61] P. Klint. *The ToolBus Programming Guide*. Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, 2002. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ToolBus`.

[62] C. Kobryn. Modeling Components and Frameworks with UML. *Communications of the ACM*, 43(10):31–38, 2000.

[63] J.-L. Koning, M.-P. Huget, J. Wei, and X. Wang. Extended Modeling Languages for Interaction Protocol Design. In *AOSE 2001, Revised Papers and Invited Contributions*, volume 2222 of *LNCS*, pages 68–83. Springer, 2002.

[64] F.A. Kraemer and P. Herrmann. Service Specification by Composition of Collaborations – An Example. In *Proc. WI-IATW 2006*, pages 129–133. IEEE Computer Society, 2006.

[65] F.A. Kraemer, P. Herrmann, and R. Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In *Proc. CoopIS, DOA, GADA, and ODBASE 2006, Part II*, volume 4276 of *LNCS*, pages 1613–1632. Springer, 2006.

[66] M. Lankhorst. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Springer, 2005.

[67] G. Lenzini, A. Tokmakoff, and J. Muskens. Managing Trustworthiness in Component-based Embedded Systems. *Electronic Notes in Theoretical Computer Science*, 179:143–155, 2007.

[68] B. di Martino, N. Mazzocca, and S. Russo. Paradigms for the Parallelization of Branch and Bound Algorithms. In *Proc. PARA 1995*, volume 1041 of *LNCS*, pages 141–150. Springer, 1996.

[69] S. Matougui and A. Beugnard. Two Ways of Implementing Software Connections Among Distributed Components. In *Proc. CoopIS, DOA, and ODBASE 2005, Part II*, volume 3761 of *LNCS*, pages 997–1014. Springer, 2005.

[70] A. McNeile and N. Simons. Protocol Modelling: A Modelling Approach that supports Reusable Behavioural Abstractions. *Software and Systems Modeling*, 5(1):91–107, 2006.

[71] P.J.A. Morssink and L.P.J. Groenewegen. Paradigm and Logic Programming. In *Proc. 1990 ESS, San Diego, CA, USA*, pages 26–30, 1990.

[72] Object Management Group. OMG Unified Modeling Language (OMG UML) Specification, v1.3, March 2000.

[73] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, v2.1.2, November 2007.

[74] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2, November 2007.

[75] J.J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing Agent Interaction Protocols in UML. In *Proc. AOSE 2000*, volume 1957 of *LNCS*, pages 121–140. Springer, 2001.

[76] G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language. In *Proc. UML 1999*, volume 1723 of *LNCS*, pages 99–115. Springer, 1999.

[77] J. Paoli, C. M. Sperberg-McQueen, T. Bray, F. Yergeau, and E. Maler. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, W3C, 2006. `http://www.w3.org/TR/2006/REC-xml-20060816`.

[78] The ParADE Distributed Runtime Environment, Related Tools and Documentation. Website, 2009. `http://paradigm.liacs.nl/parade/`.

[79] J.G. Quenum, S. Aknine, J.-P. Briot, and S. Honiden. A Modeling Framework for Generic Agent Interaction Protocols. In *DALT 2006, Selected, Revised and Invited Papers*, volume 4327 of *LNCS*, pages 207–224. Springer, 2006.

[80] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The Architecture of a UML Virtual Machine. In *Proc. OOPSLA 2001*, pages 327–341. ACM, 2001.

[81] The ITEA Robocop Research Project. Website, 2001. `http://www.hitech-projects.com/euprojects/robocop/`.

[82] F. Rösler, B. Geppert, and R. Gotzhein. CoSDL – An Experimental Language for Collaboration Specification. In *SAM 2002, Revised Papers*, volume 2599 of *LNCS*, pages 1–20. Springer, 2003.

[83] F. Rössler. *Collaboration-Based Design of Communicating Systems with SDL*. PhD thesis, University of Kaiserslautern, Germany, 2002.

[84] F. Rössler, B. Geppert, and R. Gotzhein. Collaboration-Based Design of SDL Systems. In *Proc. SDL 2001: Meeting UML*, volume 2078 of *LNCS*, pages 72–89. Springer, 2001.

[85] T. Schattkowsky and W. Mueller. A UML Virtual Machine for Embedded Systems. In *Proc. ISNG 2005, Las Vegas, Nevada, USA*, 2005.

[86] B. Selic. Tutorial: an Overview of UML 2. In *Proc. ICSE 2006*, pages 1069–1070. ACM, 2006.

[87] B. Selic. UML 2: a Model-driven Development Tool. *IBM Systems Journal*, 45(3):607–620, 2006.

[88] M. Sirjani, M.M. Jaghoori, C. Baier, and F. Arbab. Compositional Semantics of an Actor-Based Language Using Constraint Automata. In *Proc. COORDINATION 2006*, volume 4038 of *LNCS*, pages 281–297. Springer, 2006.

[89] The ITEA Space4U Research Project. Website, 2003. `http://www.hitech-projects.com/euprojects/space4u/`.

[90] A.W. Stam. CMT2 – Towards Modeling Techniques for Component Based Software Design, Master's Thesis. Technical report, LIACS, Leiden University, 2000.

[91] A.W. Stam. A Framework for Coordinating Parallel Branch and Bound Algorithms. In *Proc. COORDINATION 2002*, volume 2315 of *LNCS*, pages 332–339. Springer, 2002.

[92] A.W. Stam, L.P.J. Groenewegen, and E.P. de Vink. Restructuring Paradigm Models for the ToolBus Architecture: A Case Study. *Electronic Notes in Theoretical Computer Science*, 150(1):127–142, 2006.

[93] A.W. Stam, J. Jacob, F.S. de Boer, M.M. Bonsangue, and L.W. N. van der Torre. Using XML Transformations for Enterprise Architectures. In *ISoLA 2004, Revised Selected Papers*, volume 4313 of *LNCS*, pages 42–56. Springer, 2004.

[94] A.W. Stam and A.H. Salden. Towards Composition of Distributed Evolving Services: the CREDO Approach. In *Proc. INSERTech 2008*. ICST, 2008.

[95] M.R. van Steen. *Modelling Dynamic Systems by Parallel Decision Processes*. PhD thesis, Leiden University, The Netherlands, 1988.

[96] H.W.J.M. Trienekens and A. de Bruin. Towards a Taxonomy of Parallel Branch and Bound Algorithms. Technical Report EUR-CS-92-01, Erasmus University Rotterdam, Dept. of Computer Science, 1992.

[97] The ITEA Trust4all Research Project. Website, 2005. `http://www.hitech-projects.com/euprojects/trust4all/`.

[98] M. Turkensteen. *Advanced Analysis of Branch-and-Bound Algorithms*. PhD thesis, Rijksuniversiteit Groningen, The Netherlands, 2007.

[99] H. Wada, E.M.M. Babu, A. Malinowski, J. Suzuki, and K. Oba. Design and Implementation of the Matilda Distributed UML Virtual Machine. In *Proc. SEA 2006, Dallas, Texas, USA*, 2006.

[100] J.M. Zaha, A.P. Barros, M. Dumas, and A.H.M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *Proc. CoopIS, DOA, GADA, and ODBASE 2006, Part I*, volume 4275 of *LNCS*, pages 145–162. Springer, 2006.

Appendix A

PARADISE Pseudo Code

This appendix contains pseudo code specifications for each of the PARADISE framework elements. Each of the elements in the framework is specified as a class. For each class, we provide the following definitions:

- **require** definitions: *arguments* which are required in order to create an instance of the class;
- **declare** definitions: *variables* local to instances of the class;
- **method** definitions: *operations* which the class implements, possibly with a return value.

For the specification of the method bodies, we use a combination of assignment statements, common control structures, synchronous method calls, set manipulations, function applications and communication operations on channels. We include a short informal explanation of the pseudo code with each PARADISE element.

A.1 Processes

Process Handler

A *Process Handler* object executes a PARADIGM process. It requires a definition of a PARADIGM process, a set of role handlers and a selector. The *run* method of the process handler executes the process, meanwhile taking into account constraints of both partitions (via the provided detailed and/or global role handlers) and interaction protocols (via the provided selector). This process handler halts the execution of its process as soon as a final state is entered.

require: a function T from transition names t to transition definitions t
  each t with:
    source state name $t.s_{source}$
    target state name $t.s_{target}$
require: an initial state name $s_{initial}$
require: a set of final state names S_{final}
require: a set of role handlers H_{role}
require: a selector s

declare: a current state name $s_{current}$, initially unknown
declare: a set of enabled transition names $T_{enabled}$, initially $\emptyset$
declare: a selected transition name $t_{selected}$, initially unknown

method run()
$s_{current} := s_{initial}$
for each $h \in H_{role}$ **do**
$h.notify(s_{current})$
end for
while $(s_{current} \notin S_{final})$ **do**
repeat
$T_{enabled} := \{t \mid T(t).s_{source} = s_{current}\}$
for each $h \in H_{role}$ **do**
$T_{enabled} := h.restrict(T_{enabled})$
end for
$s.setEnabledTransitions(s_{current}, T_{enabled})$
until $s.selectionMade()$
$t_{selected} := s.getSelectedTransition()$
$s_{current} := T(t_{selected}).s_{target}$
for each $h \in H_{role}$ **do**
$h.notify(s_{current})$
end for
end while
end method

A.2 Partitions

Detailed Role Handler

A *Detailed Role Handler* object manages a partition at the side of a detailed process. It requires a definition of the subprocesses of a partition, including the traps, and a partition channel to communicate with a global role handler. The *notify* method is called by a process handler after every state change, in order for the detailed role handler to check whether new traps have been entered. The *restrict* method is used by a process handler to determine the set of enabled transitions, according to the current subprocess.

require: a function Σ from subprocess names σ to subprocess definitions ς
each ς with:
a set of transition names $\varsigma.T$
a function $\varsigma.\Theta$ from trap names θ to sets of state names S
require: a partition channel c

declare: a current state name $s_{current}$, initially unknown
declare: a current subprocess name $\sigma_{current}$, initially unknown
declare: a set of current trap names $Tr_{current}$, initially $\emptyset$

```
method notify(s)
  s_current := s
  if (σ_current ≠ unknown) ∧ (Tr_current ⊂ {θ | s_current ∈ Σ(σ_current).Θ(θ)}) then
    Tr_current := {θ | s_current ∈ Σ(σ_current).Θ(θ)}
    c.send(⟨σ_current, Tr_current⟩)
  end if
end method

method restrict(T)
  while ¬(c.isEmpty()) do
    σ_current := c.receive()
    Tr_current := {θ | s_current ∈ Σ(σ_current).Θ(θ)}
    if Tr_current ≠ ∅ then
      c.send(⟨σ_current, Tr_current⟩)
    end if
  end while
  if σ_current ≠ unknown then
    T_result := T ∩ (Σ(σ_current).T)
  end if
  return T_result
end method
```

Global Role Handler

A *Global Role Handler* object manages a partition at the side of a global process. It only requires a partition channel in order to communicate with a detailed role handler. The *notify* method is called by a process handler after every state change, in order for the global role handler to send the new state (the newly prescribed subprocess) to the detailed role handler. The *restrict* method is used by a process handler to determine the set of enabled transitions, according to the current set of traps entered.

require: a partition channel c

declare: a current state name $s_{current}$, initially unknown
declare: a set of enabled transition names $T_{enabled}$, initially $\emptyset$

```
method notify(s)
  if s_current ≠ s then
    s_current := s
    c.send(s_current)
    T_enabled := ∅
  end if
end method
```

```
method restrict(T)
  while ¬(c.isEmpty()) do
    ⟨σ, Tr⟩ := c.receive()
    if s_current = σ then
      T_enabled := Tr
    end if
  end while
  T_result := T ∩ T_enabled
  return T_result
end method
```

A.3 Selectors

Simple Selector

A *Simple Selector* object non-deterministically selects a transition out of a set of enabled transitions. Method *setEnabledTransitions* is used by a process handler to provide the set of transitions from which one must be selected. Method *selectionmade* is used by a process handler to test whether a selection has already been made. Method *getSelectedTransition* returns the selected transition. The split-up between testing if a selection has been made and retrieving the actual selection is only relevant for the *Delegating* and *Managing Selector*, but has been used in this selector as well for reasons of uniformity. All *Selector* classes require that their methods are called in a certain order: first provide the set of transitions, then repetitively test if a selection has been made, finally get the selection as soon as the test returns true.

```
declare: a set of enabled transition names T_enabled, initially ∅
declare: a selected transition name t_selected, initially unknown

  method setEnabledTransitions(s, T)
    T_enabled := T
  end method

  method selectionMade()
    if T_enabled ≠ ∅ then
      t_selected := t | t ∈ T_enabled
      return true
    else
      return false
    end if
  end method

  method getSelectedTransition()
    return t_selected
  end method
```

Delegating Selector

A *Delegating Selector* object delegates the selection of a transition out of a set of enabled transitions to an interaction protocol / a manager process. Further differences with a *Simple Selector* object are twofold. Firstly, the *Delegating Selector* requires a set of employee channels in order to connect to employee proxies at different interaction protocols. Secondly, in method *setEnabledTransitions*, the set of transitions is sent to one of employee proxies in order for the latter one to return a selected transition in method *selectionMade*.

require: a function C from state names s to employee channels c

declare: a current state name $s_{current}$, initially unknown
declare: a set of enabled transition names $T_{enabled}$, initially $\emptyset$
declare: a selected transition name $t_{selected}$, initially unknown

method setEnabledTransitions(s, T**)**
  if $(s_{current} \neq s) \vee (T_{enabled} \neq T)$ **then**
    $s_{current} := s$
    $T_{enabled} := T$
    $C(s_{current})$.send($T_{enabled}$)
  end if
end method

method selectionMade()
  if $\neg(C(s_{current})$.isEmpty()) **then**
    $t_{selected} := C(s_{current})$.receive()
    $C(s_{current})$.send(sync)
    $T_{enabled} := \emptyset$
    return true
  else
    return false
  end if
end method

method getSelectedTransition()
  return $t_{selected}$
end method

Managing Selector

A *Managing Selector* object selects a transition out of a set of enabled transitions together with a consistency rule to be applied. It requires mappings from consistency rules to transitions of the manager process, and from consistency rules to manager channels (in order to communicate with manager proxies at different interaction protocols). Method *selectionMade* implements a two-phase commit, in order to ensure that a selected consistency rule is indeed enabled.

require: a function $R_{manager}$ from consistency rule names r to transition names t
require: a function C from consistency rule names r to manager channels c

declare: a set of enabled transition names $T_{enabled}$, initially $\emptyset$
declare: a set of enabled consistency rule names $R_{enabled}$, initially $\emptyset$
declare: a selected transition name $t_{selected}$, initially unknown

method setEnabledTransitions(s, T**)**
  $T_{enabled} := T$
end method

method selectionMade()
  for each $c \in codom(C)$ **do**
    while $\neg$(c.isEmpty()) **do**
      $R_{enabled} := (R_{enabled} \setminus \{r \mid C(r) = c\}) \cup$ c.receive()
    end while
  end for
  if $\{r \mid r \in R_{enabled}, R_{manager}(r) \in T_{enabled}\} \neq \emptyset$ **then**
    $r_{selected} := r \mid r \in R_{enabled}, R_{manager}(r) \in T_{enabled}$
    $C(r_{selected})$.send($r_{selected}$)
    repeat
      $M := C(r_{selected})$.receive()
      if $M \neq$ sync **then**
        $R_{enabled} := (R_{enabled} \setminus \{r \mid C(r) = C(r_{selected})\}) \cup M$
      end if
    until $M =$ sync
    if $r_{selected} \in R_{enabled}$ **then**
      $C(r_{selected})$.send(commit)
      $t_{selected} := R_{manager}(r_{selected})$
      return true
    else
      $C(r_{selected})$.send(release)
    end if
  end if
  return false
end method

method getSelectedTransition()
  return $t_{selected}$
end method

A.4 Proxies and Self-Manager

Employee Proxy

An *Employee Proxy* object communicates with the delegating selector of a single employee process on behalf of an interaction protocol. It requires a unique role, an employee channel and a set of rule handlers which handle consistency rules in which the employee process related to this proxy plays a role. Method *check* tests for the availability of newly enabled transitions from the employee process, and notifies the rule handlers accordingly. Method *take* is used by a rule handler to take one of the transitions in the employee part of its consistency rule.

require: a role name ρ
require: an employee channel c
require: a set of rule handlers $\mathtt{H}_{\mathtt{rule}}$

declare: a set of enabled transition names $\mathsf{T}_{\mathsf{enabled}}$, initially $\emptyset$

```
method check()
  while ¬(c.isEmpty()) do
    T_enabled := c.receive()
  end while
  for each h ∈ H_rule do
    h.notify(⟨ρ, T_enabled⟩)
  end for
end method

method take(t)
  c.send(t)
  repeat
    M := c.receive()
  until M = sync
  T_enabled := ∅
  for each h ∈ H_rule do
    h.notify(⟨ρ, T_enabled⟩)
  end for
end method
```

Manager Proxy

A *Manager Proxy* object communicates with the managing selector of a manager process on behalf of an interaction protocol. It requires a manager channel and a mapping from consistency rules to rule handlers which handle these rules. Method *check* tests for the availability of a consistency rule to be applied, and applies the rule if applicable. Method *notify* is used by a rule handler to inform the manager proxy about the enabling or disabling of a consistency rule.

require: a manager channel c
require: a function $\mathtt{H}_{\mathtt{rule}}$ from consistency rule names r to rule handlers $\mathtt{h}_{\mathtt{rule}}$

declare: a set of enabled consistency rule names $\mathsf{R}_{\mathsf{enabled}}$, initially $\emptyset$

```
method check()
  if ¬(c.isEmpty()) then
    r_applied := c.receive()
    c.send(sync)
    M := c.receive()
    if M = commit then
      H_rule(r_applied).apply()
    end if
  end if
end method
```

```
method notify(⟨r, i⟩)
  if i = true then
    if r ∉ R_enabled then
      R_enabled := R_enabled ∪ {r}
      c.send(R_enabled)
    end if
  else
    if r ∈ R_enabled then
      R_enabled := R_enabled \ {r}
      c.send(R_enabled)
    end if
  end if
end method
```

Self-Manager

A *Self-Manager* object selects and applies consistency rules with empty manager parts. It requires a mapping from consistency rules to their corresponding rule handlers. Method *check* tests whether a consistency rule with an empty manager part is enabled and applies one if possible. Method *notify* is used by a rule handler to inform the self-manager about the enabling or disabling of a consistency rule.

require: a function H_{rule} from consistency rule names r to rule handlers h_{rule}

declare: a set of enabled consistency rule names $R_{enabled}$, initially $\emptyset$

```
method check()
  if R_enabled ≠ ∅ then
    r_applied := r | r ∈ R_enabled
    H_rule(r_applied).apply()
  end if
end method

method notify(⟨r, i⟩)
  if i = true then
    R_enabled := R_enabled ∪ {r}
  else
    R_enabled := R_enabled \ {r}
  end if
end method
```

A.5 Rule and Ruleset Handler

Rule Handler

A *Rule Handler* object manages a single consistency rule – its enabling and its application. It requires a consistency rule definition, a mapping from roles to employee proxies and a manager proxy, which

can be a self-manager. Method *notify* is used by an employee proxy to notify a change in the set of enabled consistency rules. If applicable, the manager proxy is informed about the enabling or disabling of the consistency rule. Method *apply* is used by a manager proxy to apply a consistency rule.

require: a function $r_{employee}$ from role names ρ to transition names t
require: a rule name r
require: a function $\mathsf{P}_{\mathtt{employee}}$ from role names ρ to employee proxies $\mathtt{p}_{\mathtt{employee}}$
require: a manager proxy $\mathtt{p}_{\mathtt{manager}}$

declare: a set of enabled employee transition names $\mathsf{T}_{\mathsf{enabled}}$, initially $\emptyset$

method notify($\langle \rho, \mathsf{T} \rangle$**)**
  if $r_{employee}(\rho) \in \mathsf{T}$ **then**
    $\mathsf{T}_{\mathsf{enabled}} := \mathsf{T}_{\mathsf{enabled}} \cup \{r_{employee}(\rho)\}$
    if $\mathsf{T}_{\mathsf{enabled}} = codom(r_{employee})$ **then**
      $\mathtt{p}_{\mathtt{manager}}$.notify($\langle$r, true$\rangle$)
    end if
  else
    if $\mathsf{T}_{\mathsf{enabled}} = codom(r_{employee})$ **then**
      $\mathtt{p}_{\mathtt{manager}}$.notify($\langle$r, false$\rangle$)
    end if
    $\mathsf{T}_{\mathsf{enabled}} := \mathsf{T}_{\mathsf{enabled}} \setminus \{r_{employee}(\rho)\}$
  end if
end method

method apply()
  for each $\rho \in dom(r_{employee})$ **do**
    $\mathsf{P}_{\mathtt{employee}}(\rho)$.take($r_{employee}(\rho)$)
  end for
end method

Ruleset Handler

A *Ruleset Handler* object manages an interaction protocol. It requires a set of manager and/or employee proxies, which may include a self-manager. Its *run* method continuously loops over the *check* method of each of the proxies.

require: a set of proxies P

method run()
  while true **do**
    for each p $\in$ P **do**
      p.check()
    end for
  end while
end method

A.6 Views on Views Support

The following specifications for some of the PARADISE elements are applicable to the situation in which views on views are used (see e.g. Chapter 8). They only differ from the previous definitions in that they implement a full two-phase commit for the application of consistency rules. This involves changes in five classes: *Delegating Selector*, *Employee Proxy*, *Rule Handler*, *Manager Proxy* and *Self-Manager*.

Delegating Selector

For the *Delegating Selector* class, method *selectionMade* has been adapted in order to check for a commit after a transition has been selected.

require: a function C from state names s to employee channels c

declare: a current state name $s_{current}$, initially unknown
declare: a set of enabled transition names $T_{enabled}$, initially $\emptyset$
declare: a selected transition name $t_{selected}$, initially unknown

```
method setEnabledTransitions(s, T)
  if (s_current ≠ s) ∨ (T_enabled ≠ T) then
    s_current := s
    T_enabled := T
    C(s_current).send(T_enabled)
  end if
end method

method selectionMade()
  if ¬(C(s_current).isEmpty()) then
    t_selected := C(s_current).receive()
    C(s_current).send(sync)
    M := C(s_current).receive()
    if M = commit then
      T_enabled := ∅
      return true
    else
      return false
    end if
  else
    return false
  end if
end method

method getSelectedTransition()
  return t_selected
end method
```

Employee Proxy

The *Employee Proxy* class has been extended with two methods *commit* and *release*, which are used by a rule handler to either agree upon the taking of a transition or to abort it.

require: a role name ρ
require: a selector channel c
require: a set of rule handlers $\mathtt{H}_{\mathtt{rule}}$

declare: a set of enabled transition names $\mathsf{T}_{\mathsf{enabled}}$, initially $\emptyset$

```
method check()
  while ¬(c.isEmpty()) do
    T_enabled := c.receive()
  end while
  for each h ∈ H_rule do
    h.notify(⟨ρ, T_enabled⟩)
  end for
end method

method take(t)
  c.send(t)
  repeat
    M := c.receive()
    if M ≠ sync then
      T_enabled := M
    end if
  until M = sync
  for each h ∈ H_rule do
    h.notify(⟨ρ, T_enabled⟩)
  end for
end method

method commit()
  c.send(commit)
  T_enabled := ∅
  for each h ∈ H_rule do
    h.notify(⟨ρ, T_enabled⟩)
  end for
end method

method release()
  c.send(release)
end method
```

Rule Handler

The *Rule Handler* class has been extended with two methods *commit* and *release*. These methods are used by a manager proxy or a self-manager, to either agree upon the application of a consistency rule or to abort this application.

require: a function $r_{employee}$ from role names ρ to transition names t
require: a rule name r
require: a function $\mathsf{P}_{\texttt{employee}}$ from role names ρ to employee proxies $\mathsf{p}_{\texttt{employee}}$
require: a manager proxy $\mathsf{p}_{\texttt{manager}}$

declare: a set of enabled employee transition names $\mathsf{T}_{\mathsf{enabled}}$, initially $\emptyset$

method notify($\langle \rho, \mathsf{T} \rangle$)
  if $r_{employee}(\rho) \in \mathsf{T}$ **then**
    $\mathsf{T}_{\mathsf{enabled}} := \mathsf{T}_{\mathsf{enabled}} \cup \{r_{employee}(\rho)\}$
    if $\mathsf{T}_{\mathsf{enabled}} = codom(r_{employee})$ **then**
      $\mathsf{p}_{\texttt{manager}}$.notify($\langle$r, true$\rangle$)
    end if
  else
    if $\mathsf{T}_{\mathsf{enabled}} = codom(r_{employee})$ **then**
      $\mathsf{p}_{\texttt{manager}}$.notify($\langle$r, false$\rangle$)
    end if
    $\mathsf{T}_{\mathsf{enabled}} := \mathsf{T}_{\mathsf{enabled}} \setminus \{r_{employee}(\rho)\}$
  end if
end method

method apply()
  for each $\rho \in dom(r_{employee})$ **do**
    $\mathsf{P}_{\texttt{employee}}(\rho)$.take($r_{employee}(\rho)$)
  end for
end method

method commit()
  for each $\rho \in dom(r_{employee})$ **do**
    $\mathsf{P}_{\texttt{employee}}(\rho)$.commit()
  end for
end method

method release()
  for each $\rho \in dom(r_{employee})$ **do**
    $\mathsf{P}_{\texttt{employee}}(\rho)$.release()
  end for
end method

Manager Proxy

The *check* method of the *Manager Proxy* class has been adapted such that it forwards the commit/release from a managing selector to the appropriate rule handler.

require: a manager channel c
require: a function H_{rule} from consistency rule names r to rule handlers h_{rule}

declare: a set of enabled consistency rule names $R_{enabled}$, initially $\emptyset$

```
method check()
  if ¬(c.isEmpty()) then
    r_applied := c.receive()
    H_rule(r_applied).apply()
    c.send(sync)
    M := c.receive()
    if M = commit then
      H_rule(r_applied).commit()
    else
      H_rule(r_applied).release()
    end if
  end if
end method

method notify(⟨r, i⟩)
  if i = true then
    if r ∉ R_enabled then
      R_enabled := R_enabled ∪ {r}
      c.send(R_enabled)
    end if
  else
    if r ∈ R_enabled then
      R_enabled := R_enabled \ {r}
      c.send(R_enabled)
    end if
  end if
end method
```

Self-Manager

The *check* method of the *Self-Manager* class has been adapted such that it sends a commit/release to the appropriate rule handler in order to confirm or abort the application of a consistency rule.

require: a function H_{rule} from consistency rule names r to rule handlers h_{rule}

declare: a set of enabled consistency rule names $R_{enabled}$, initially $\emptyset$

```
method check()
  if R_enabled ≠ ∅ then
    r_applied := r | r ∈ R_enabled
    H_rule(r_applied).apply()
    if r_applied ∈ R_enabled then
      H_rule(r_applied).commit()
    else
      H_rule(r_applied).release()
    end if
  end if
end method

method notify(⟨r, i⟩)
  if i = true then
    R_enabled := R_enabled ∪ {r}
  else
    R_enabled := R_enabled \ {r}
  end if
end method
```

Appendix B

Additional PARADIGM Models

This appendix contains all PARADIGM processes, partitions and interaction protocols which are part of the case study models, but which have not been included in earlier chapters. Combined with the models shown in the individual case study chapters, the models included in this appendix form complete PARADIGM models. This appendix only contains additional models for Chapters 6 and 7, since the model of Chapter 8 has been shown in that chapter in its entirety. We present the additional PARADIGM models without further explanation.

B.1 Additional Models for Chapter 6

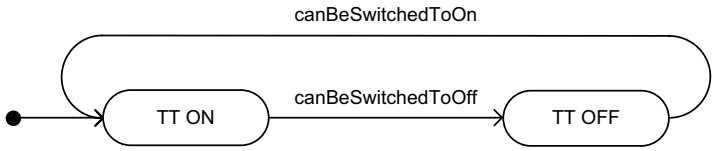


Figure B.1: Process *RouteCalculator[AsTTSwitch]*

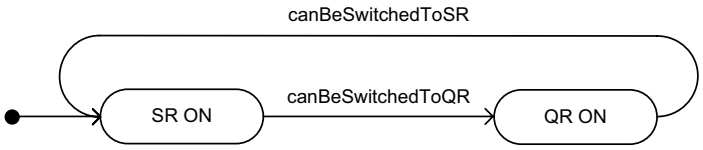


Figure B.2: Process *RouteCalculator[AsCMSwitch]*

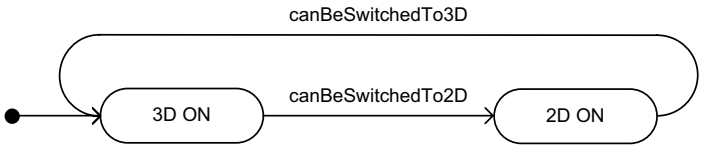


Figure B.3: Process *GraphicsRenderer[AsRMSwitch]*

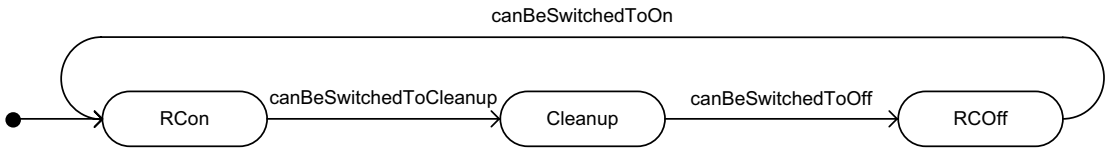


Figure B.4: Process *GraphicsRenderer[AsRCUsageSwitch]*

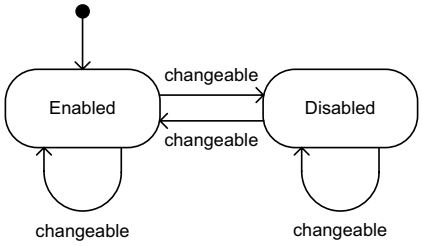


Figure B.5: Process *RouteInfoManager[AsReceiverSwitch]*

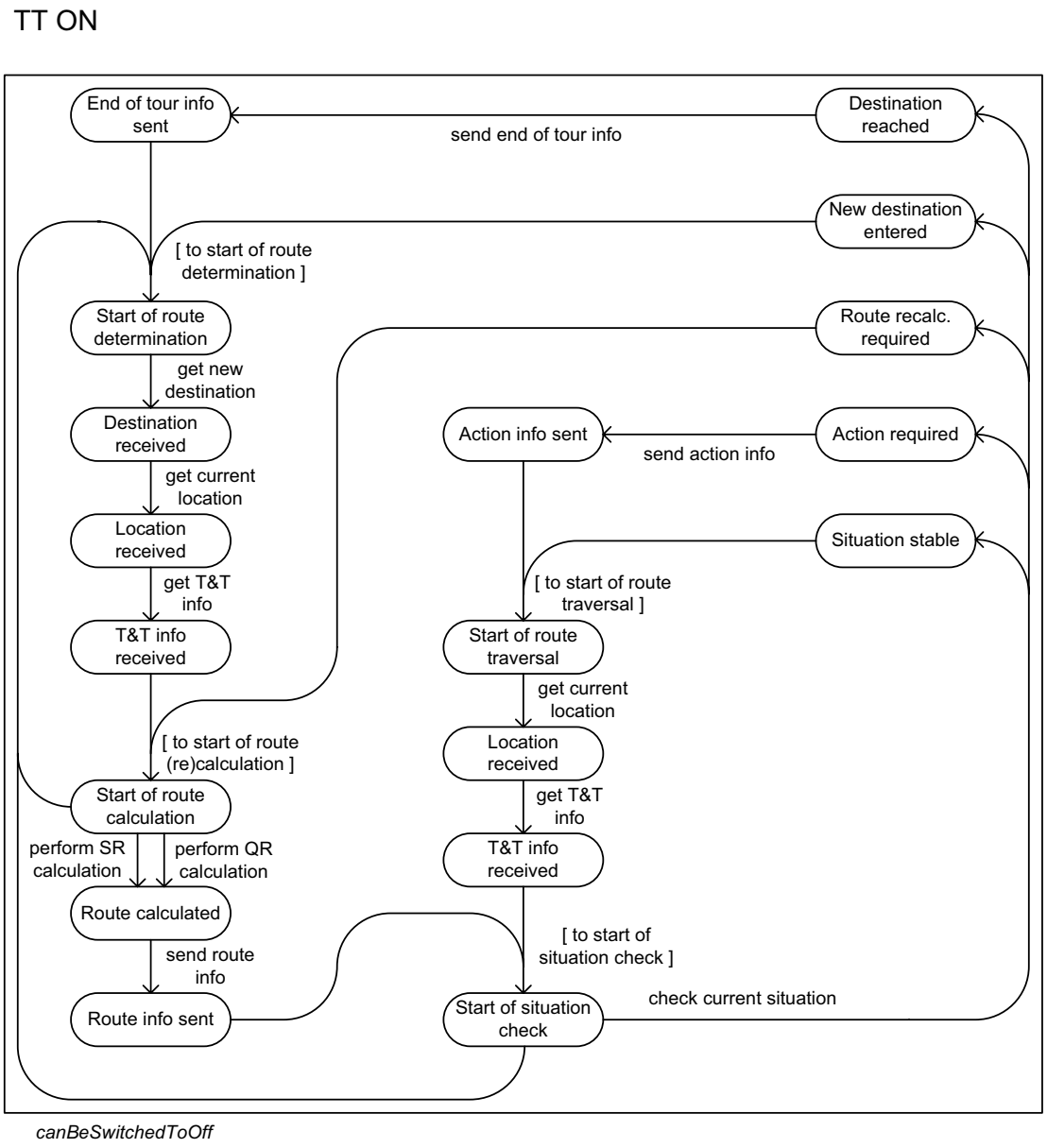


Figure B.6: Partition *AsTTSwitch* for Process *RouteCalculator*: Subprocess *TT ON*

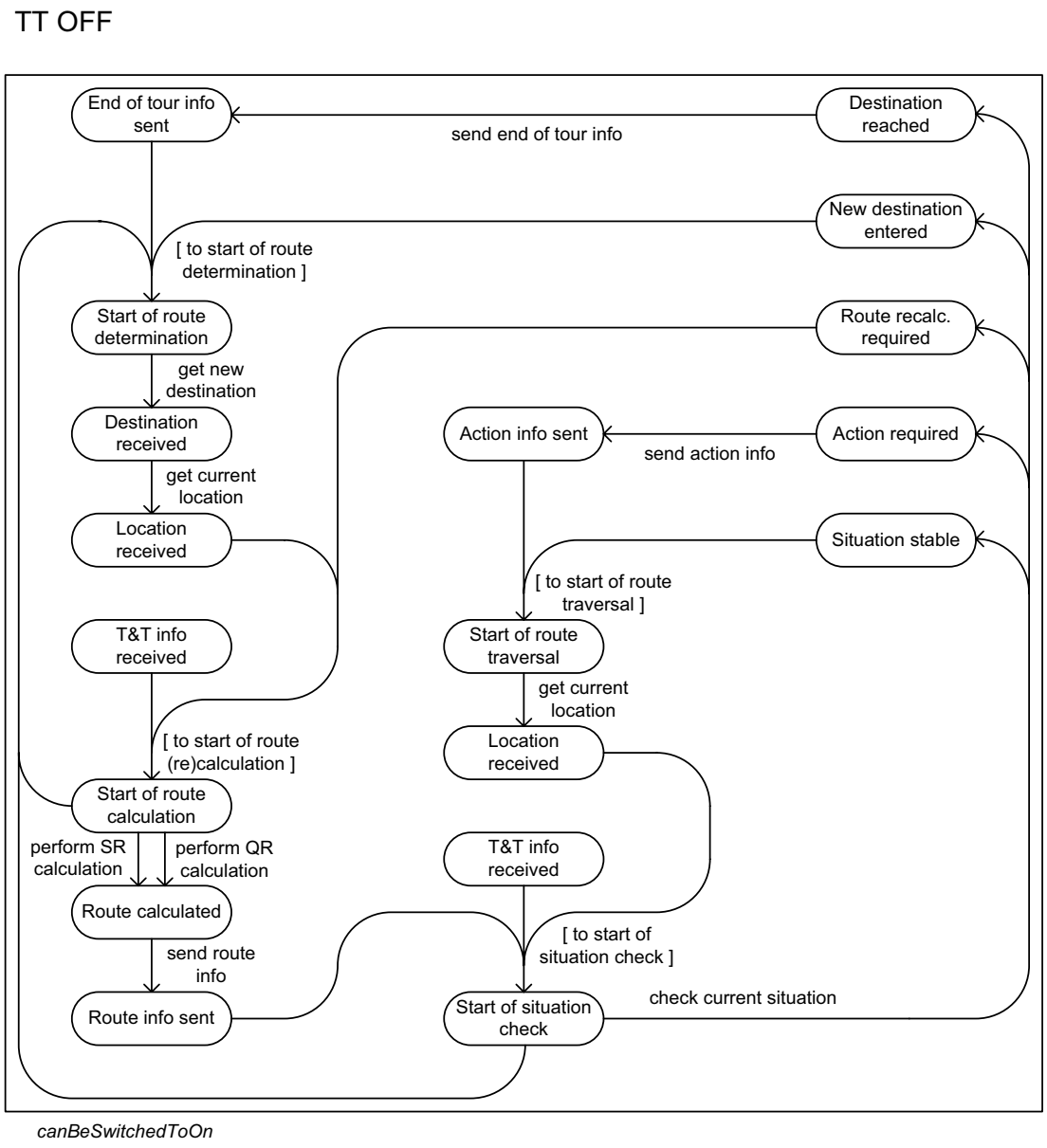


Figure B.7: Partition *AsTTSwitch* for Process *RouteCalculator*: Subprocess *TT OFF*

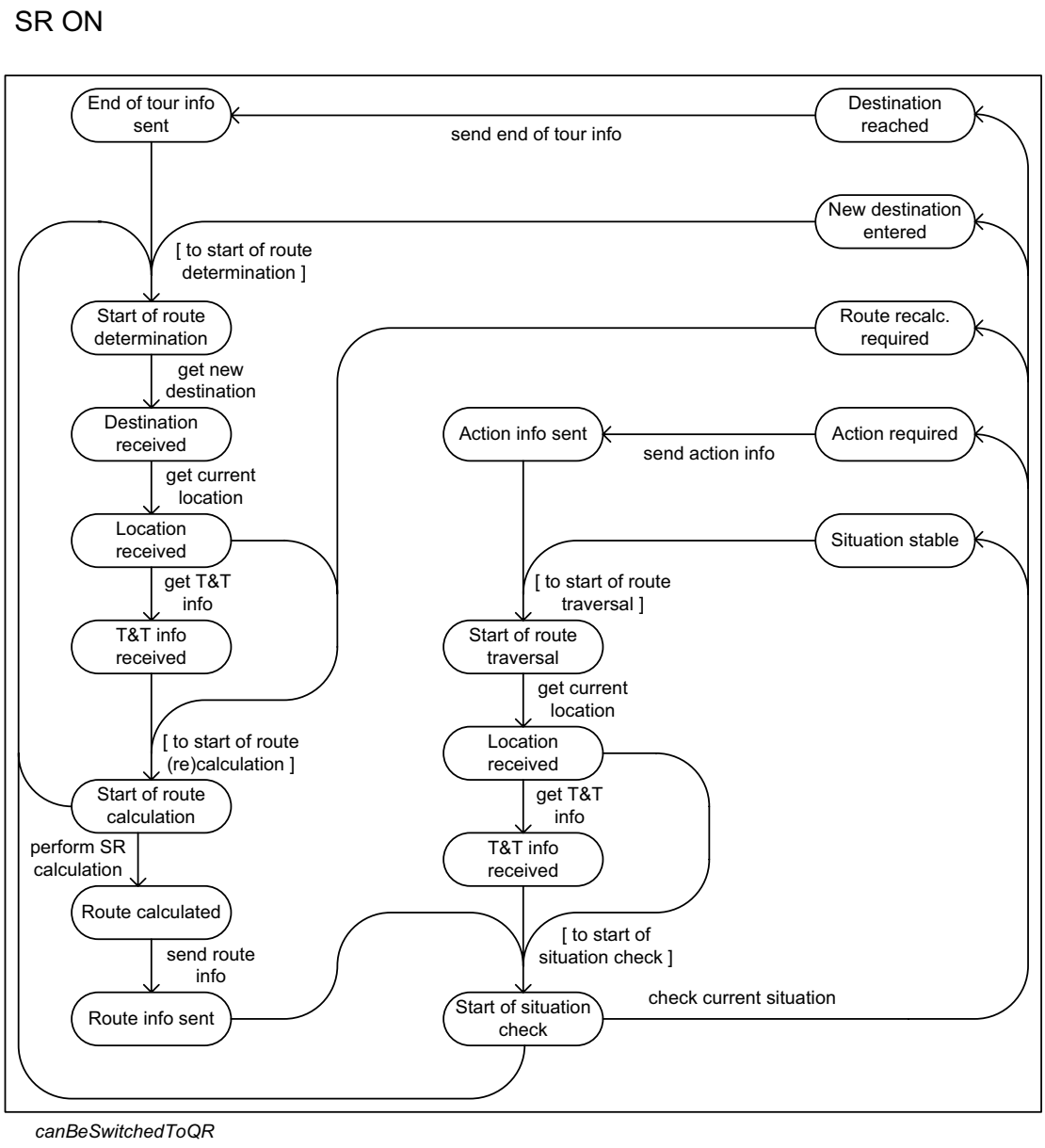


Figure B.8: Partition *AsCMSwitch* for Process *RouteCalculator*: Subprocess *SR ON*

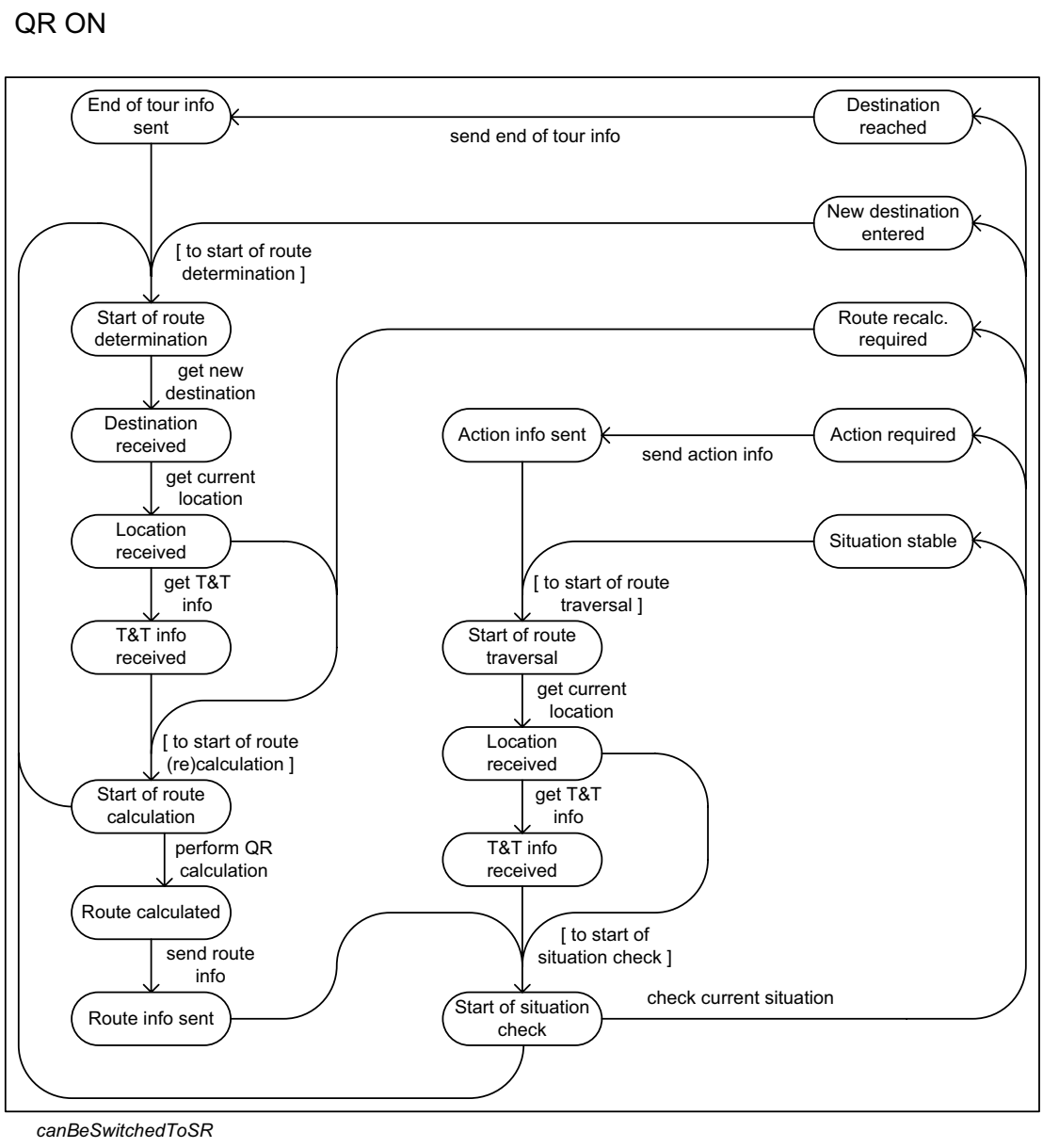


Figure B.9: Partition *AsCMSwitch* for Process *RouteCalculator*: Subprocess *QR ON*

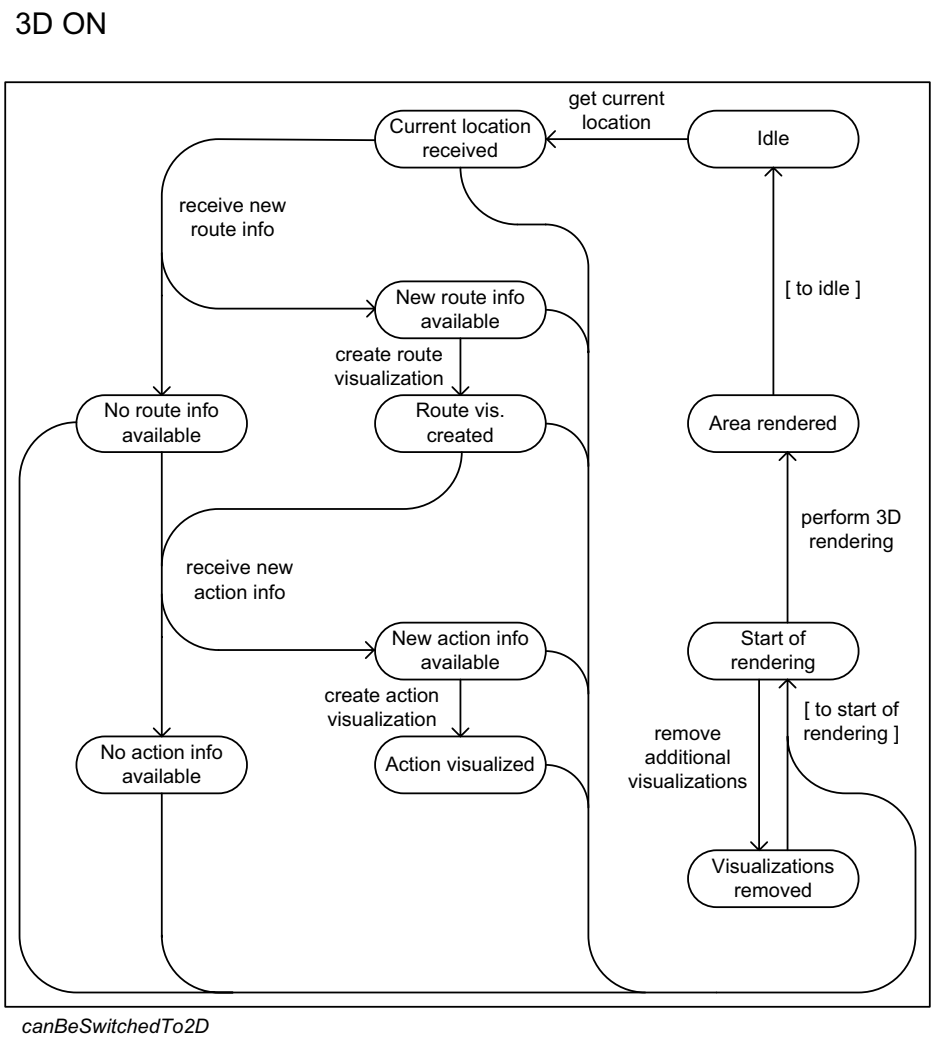


Figure B.10: Partition *AsRMSwitch* for Process *GraphicsRenderer*: Subprocess *3D ON*

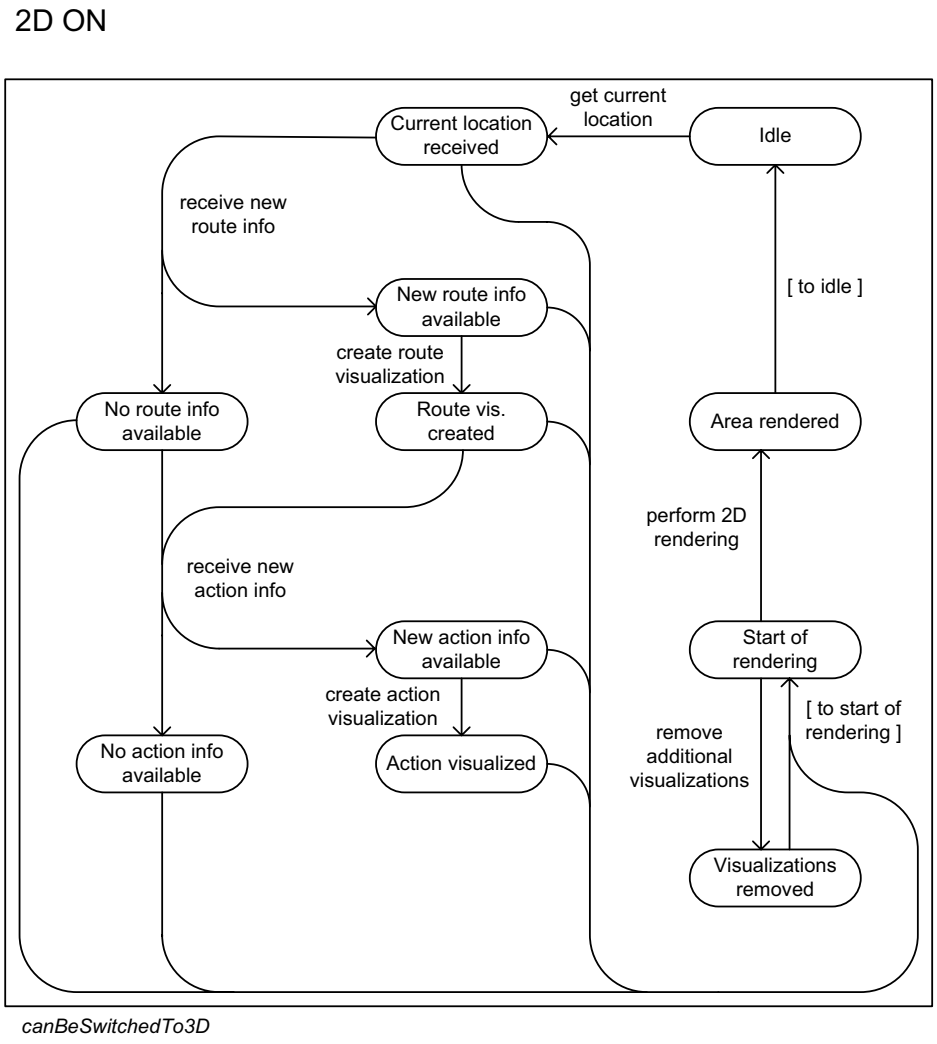


Figure B.11: Partition *AsRMSwitch* for Process *GraphicsRenderer*: Subprocess *2D ON*

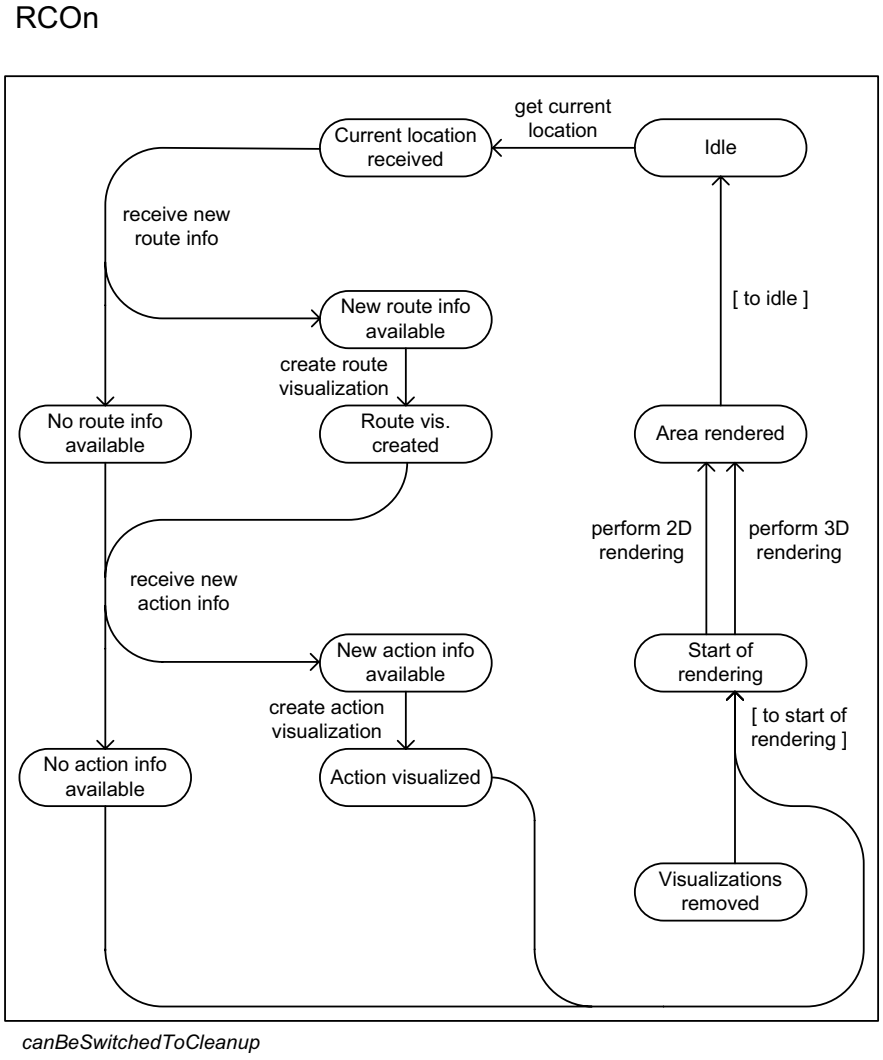


Figure B.12: Partition *AsRCUsageSwitch* for Process *GraphicsRenderer*: Subprocess *RC ON*

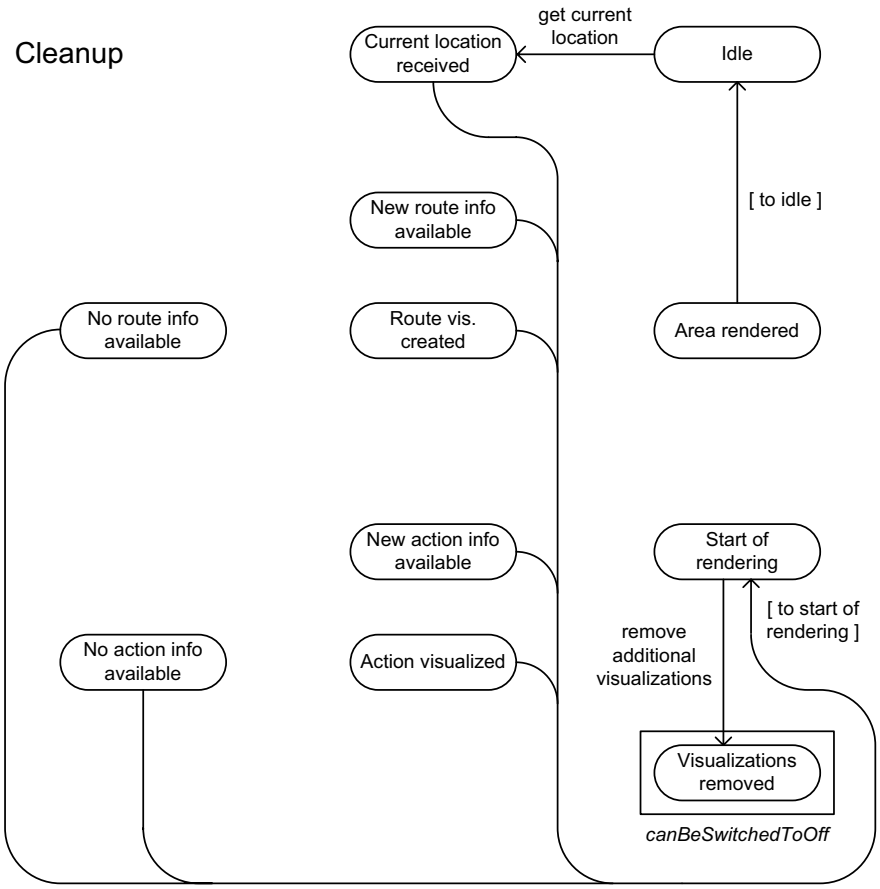


Figure B.13: Partition *AsRCUsageSwitch* for Process *GraphicsRenderer*: Subprocess *Cleanup*

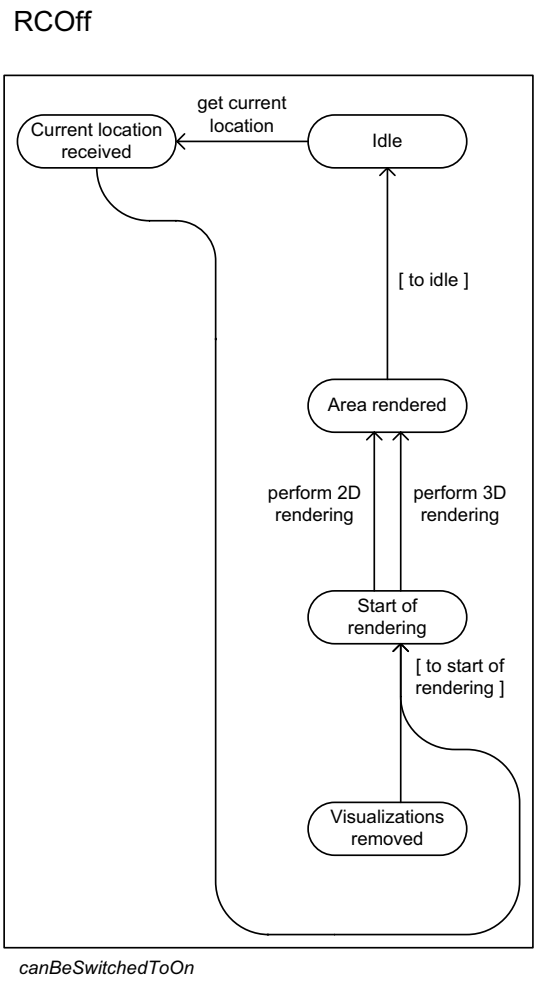


Figure B.14: Partition *AsRCUsageSwitch* for Process *GraphicsRenderer*: Subprocess *RC OFF*

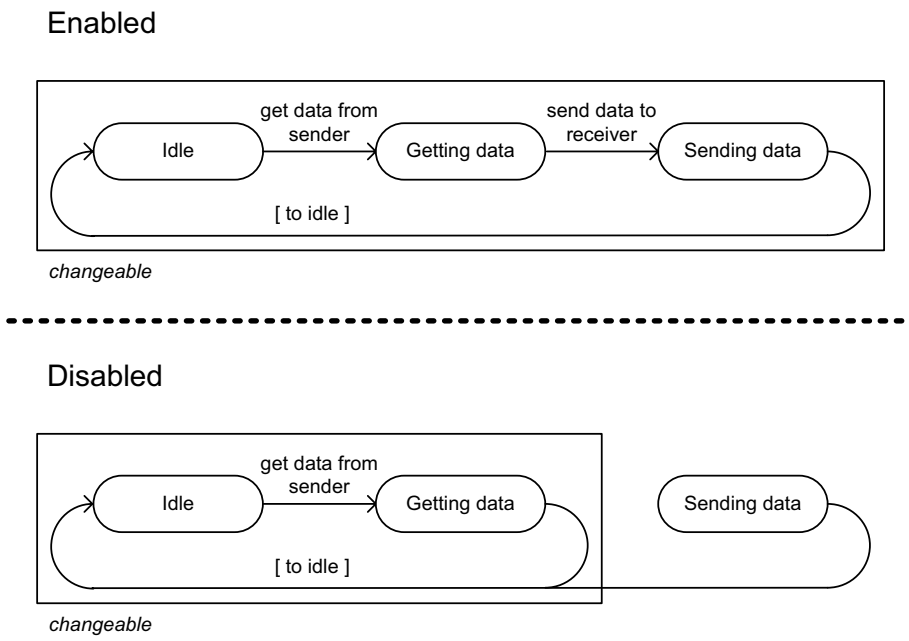


Figure B.15: Partition *AsReceiverSwitch* for Process *RouteInfoManager*

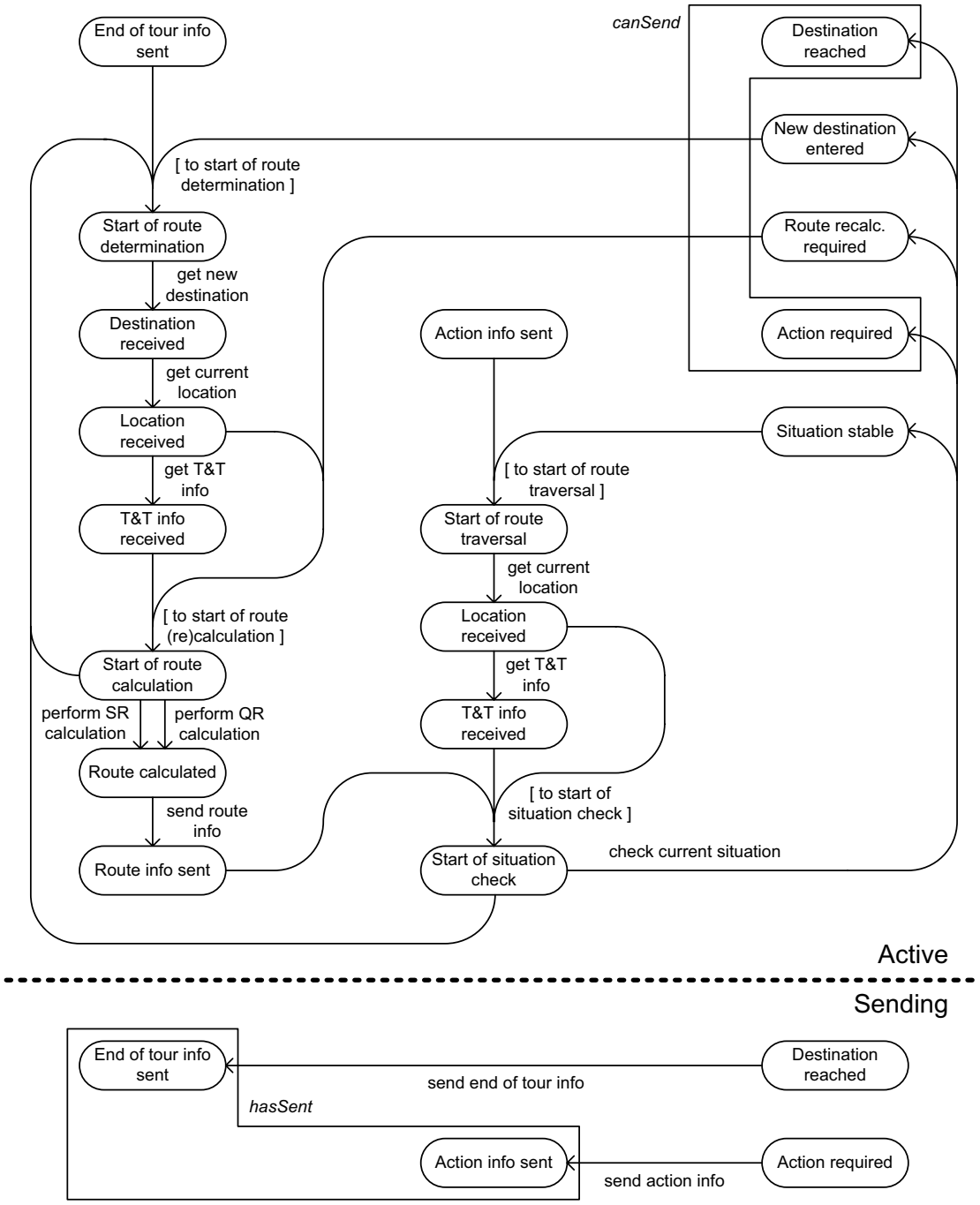


Figure B.16: Partition *AsActionSender* for Process *RouteCalculator*

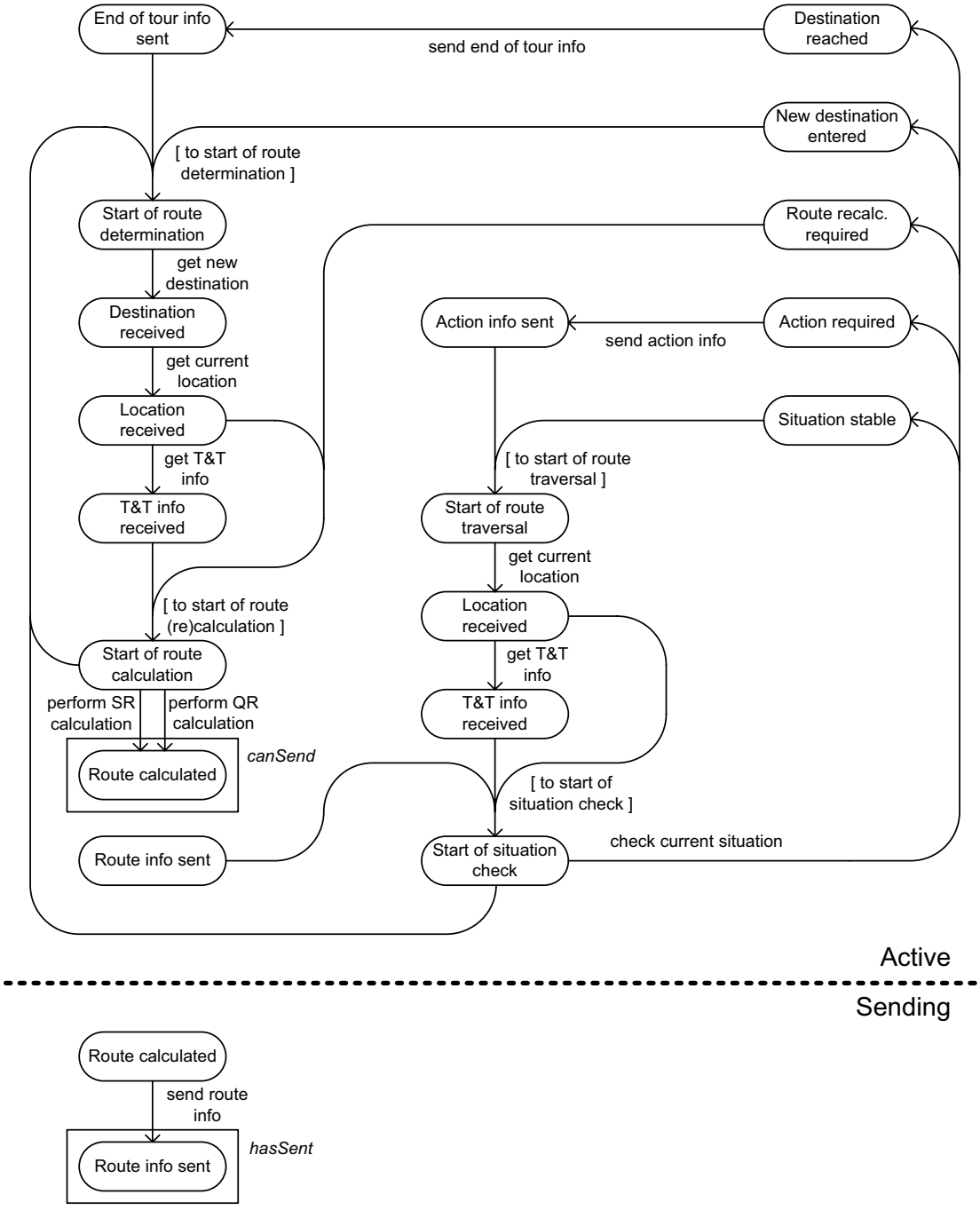


Figure B.17: Partition *AsRouteSender* for Process *RouteCalculator*

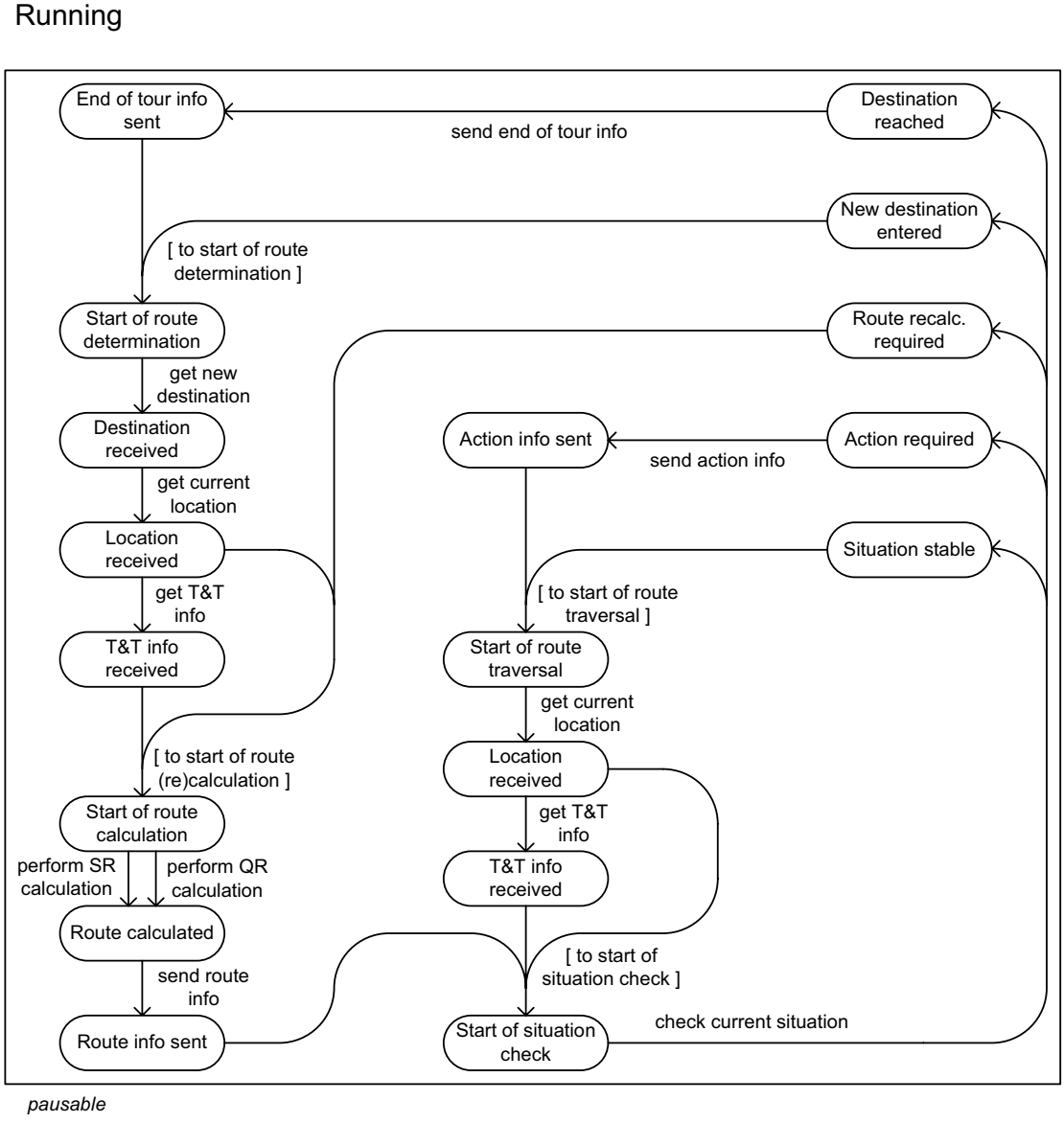


Figure B.18: Partition *AsRunnable* for Process *RouteCalculator*: Subprocess *Running*

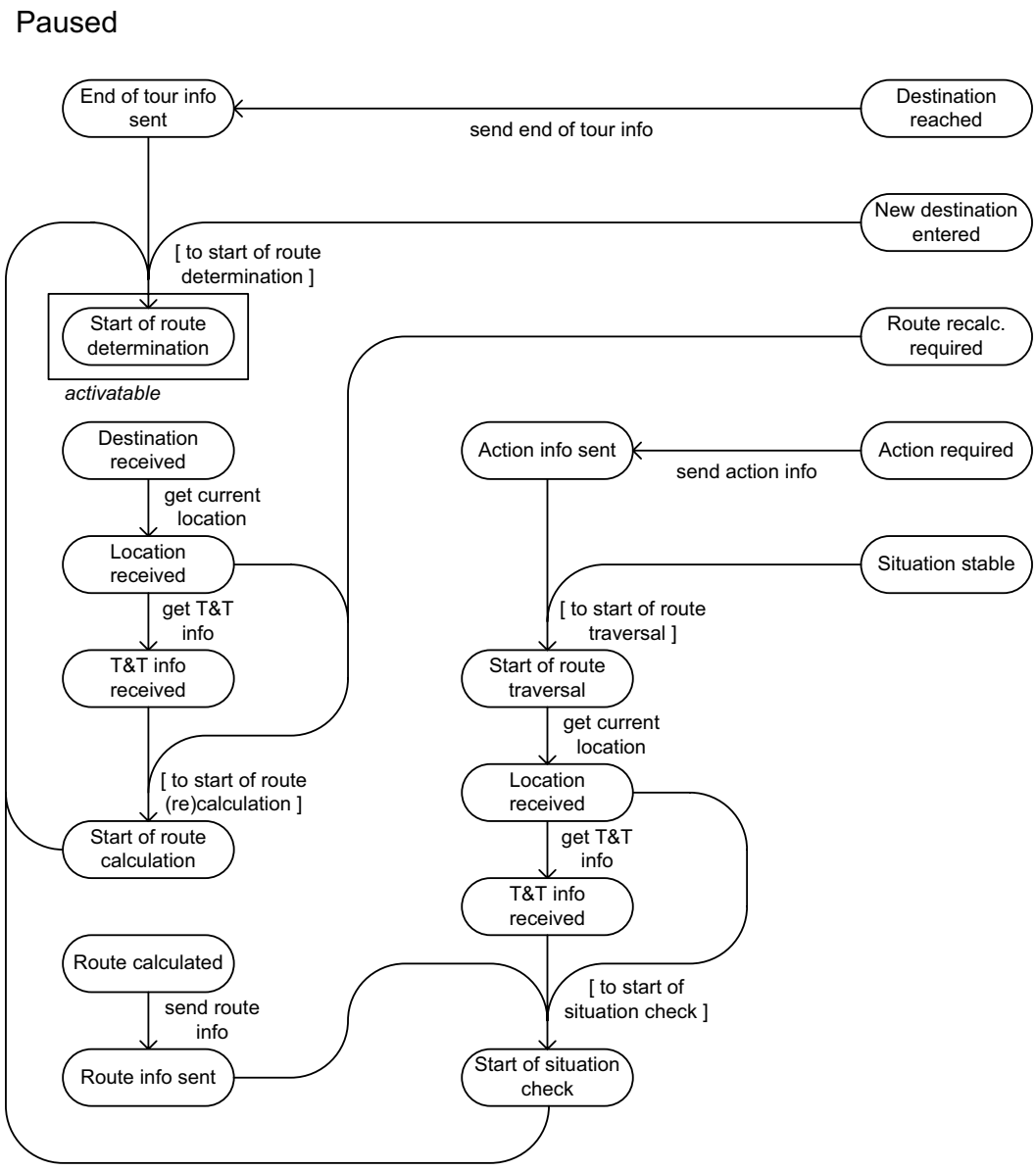


Figure B.19: Partition *AsRunnable* for Process *RouteCalculator*: Subprocess *Paused*

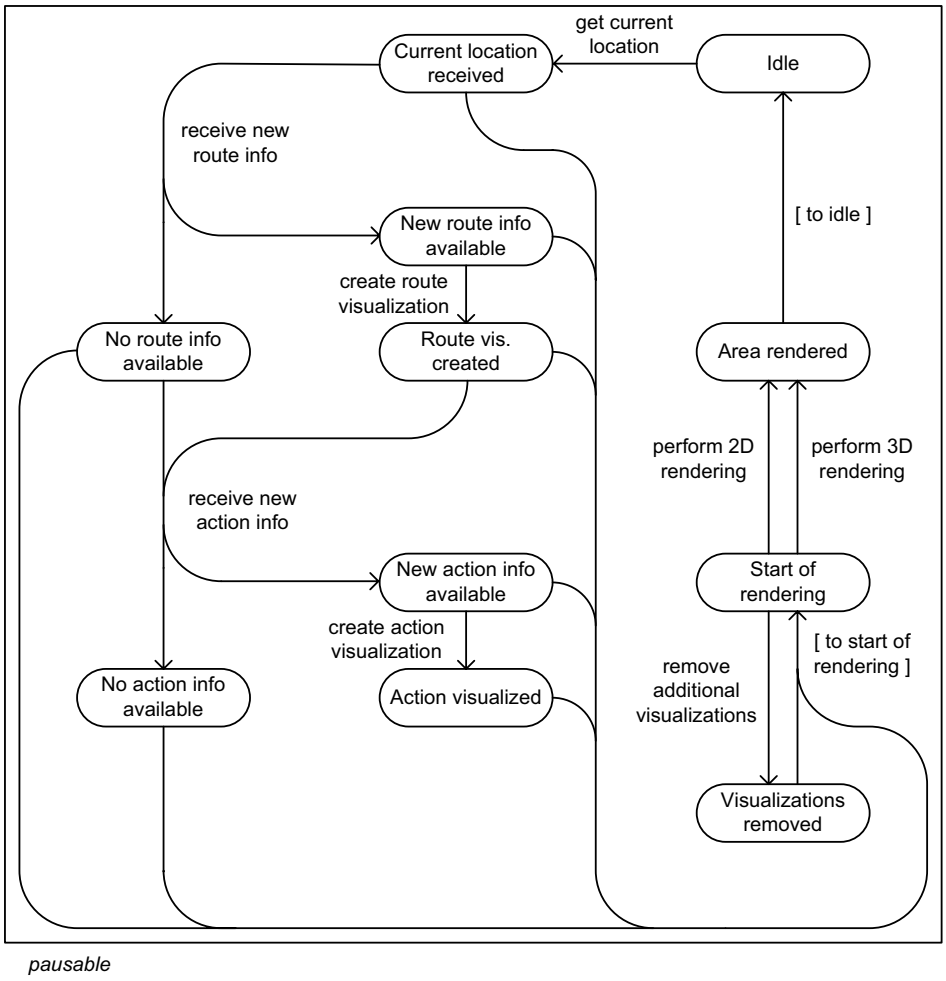


Figure B.20: Partition *AsRunnable* for Process *GraphicsRenderer*: Subprocess *Running*

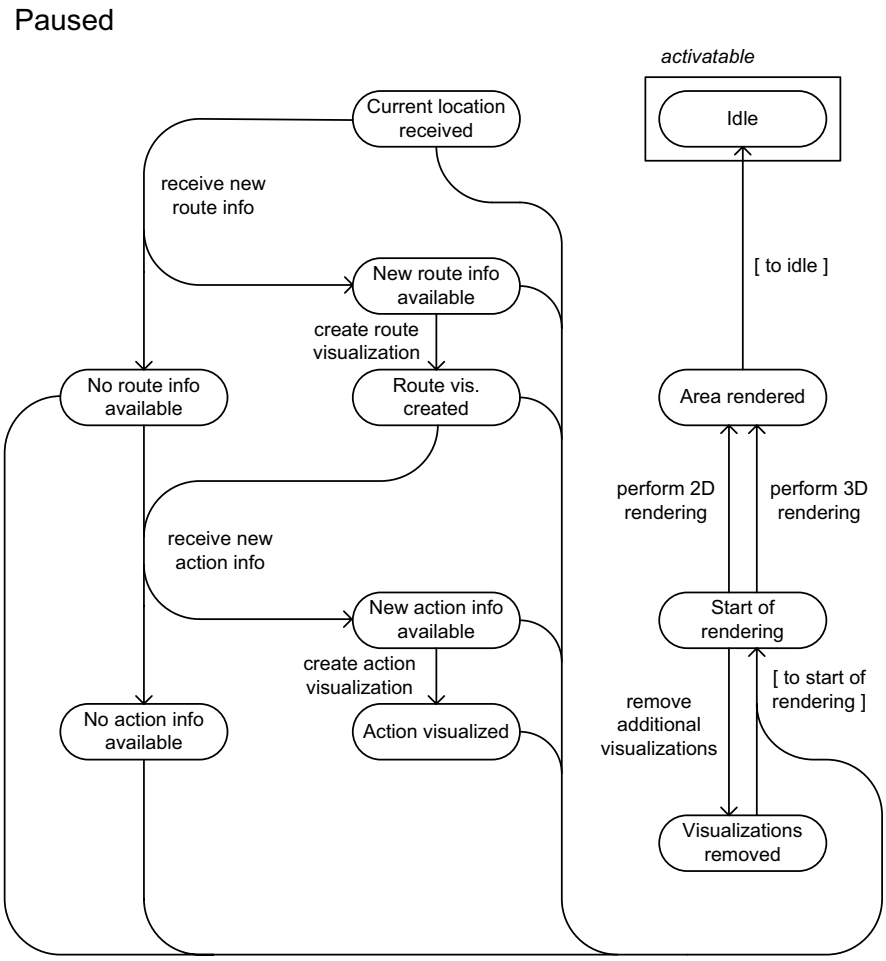


Figure B.21: Partition *AsRunnable* for Process *GraphicsRenderer*: Subprocess *Paused*

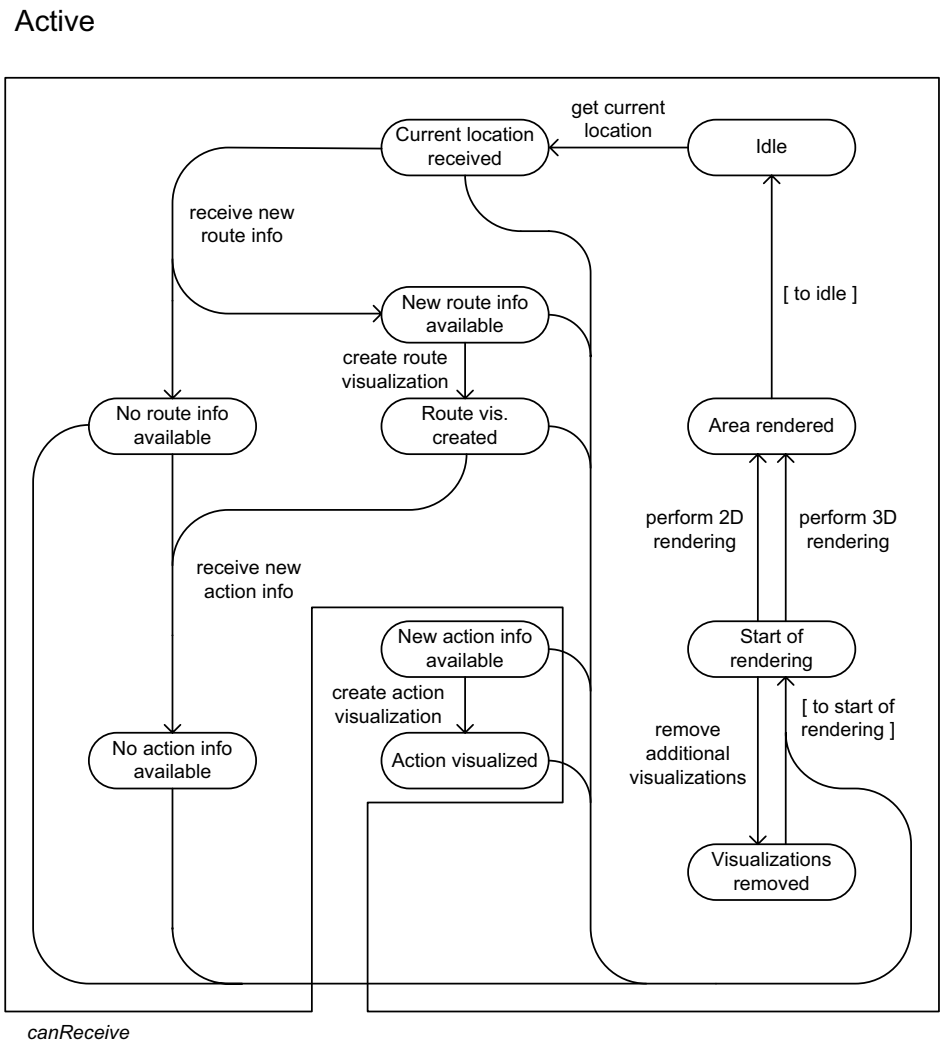


Figure B.22: Partition *AsActionReceiver* for Process *GraphicsRenderer*: Subprocess *Active*

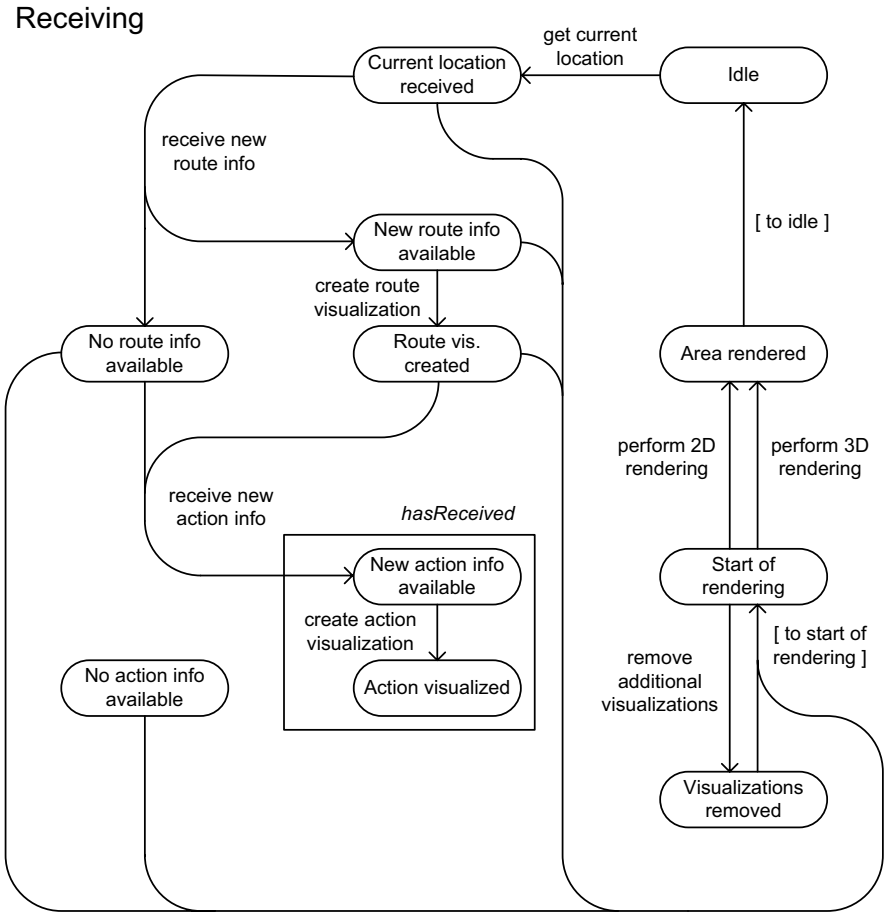


Figure B.23: Partition *AsActionReceiver* for Process *GraphicsRenderer*: Subprocess *Receiving*

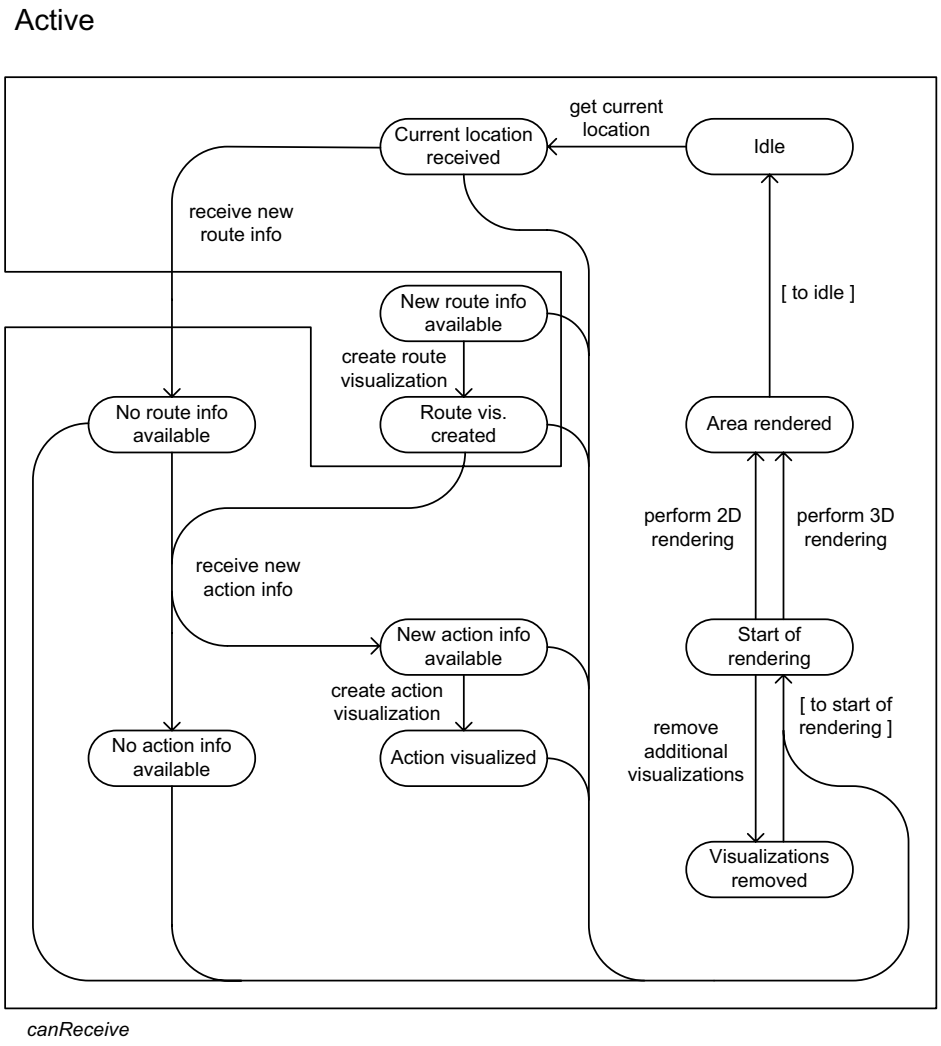


Figure B.24: Partition *AsRouteReceiver* for Process *GraphicsRenderer*: Subprocess *Active*

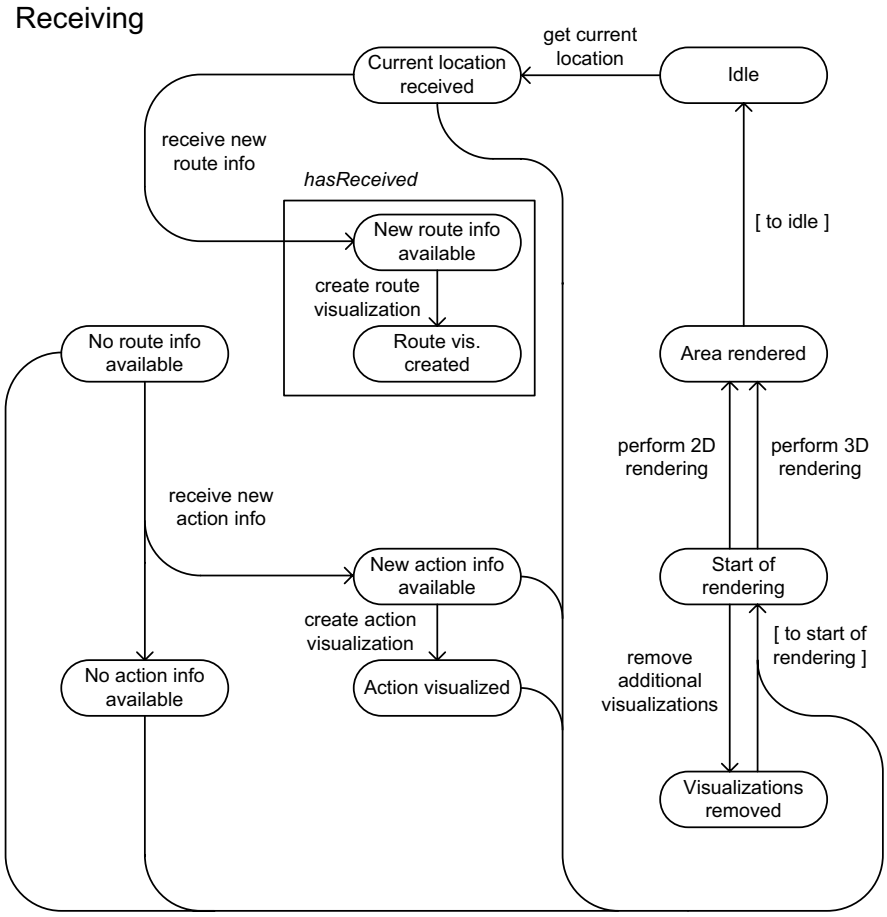


Figure B.25: Partition *AsRouteReceiver* for Process *GraphicsRenderer*: Subprocess *Receiving*

(AI1)		ActionInfoManager :	Idle	—get data from S→	Getting data
	∗	RC[AsActionSender] :	Active	—canSend→	Sending
(AI2)		ActionInfoManager :	Getting data	—send data to R1→	Sending data to R1
	∗	RC[AsActionSender] :	Sending	—has sent→	Active,
		GR[AsActionReceiver] :	Active	—canReceive→	Receiving
(AI3)		ActionInfoManager :	Sending data to R1	—send data to R2→	Sending data to R2
	∗	GR[AsActionReceiver] :	Receiving	—hasReceived→	Active,
		VS[AsActionReceiver] :	Active	—canReceive→	Receiving
(AI4)		ActionInfoManager :	Getting data	—send data to R2→	Sending data to R2
	∗	RC[AsActionSender] :	Sending	—has sent→	Active,
		VS[AsActionReceiver] :	Active	—canReceive→	Receiving
(AI5)		ActionInfoManager :	Getting data	—[to Idle]→	Idle
	∗	RC[AsActionSender] :	Sending	—hasSent→	Active
(AI6)		ActionInfoManager :	Sending data to R1	—[to Idle]→	Idle
	∗	GR[AsActionReceiver] :	Receiving	—hasReceived→	Active
(AI7)		ActionInfoManager :	Sending data to R2	—[to Idle]→	Idle
	∗	VS[AsActionReceiver] :	Receiving	—hasReceived→	Active

Table B.26: Consistency Rules anchored to Interaction Protocol *ActionInfoProtocol*

B.2 Additional Models for Chapter 7

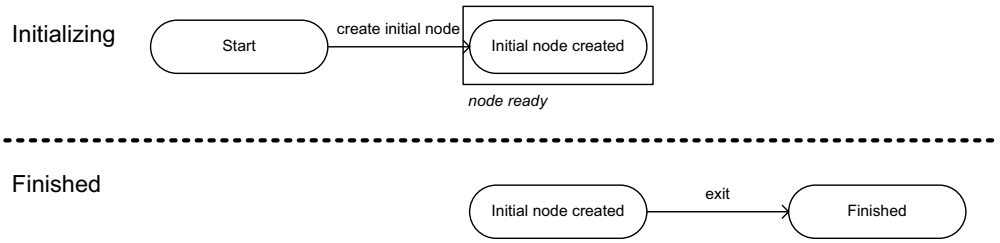


Figure B.27: Partition *AsCollab* for Process *Initializer*

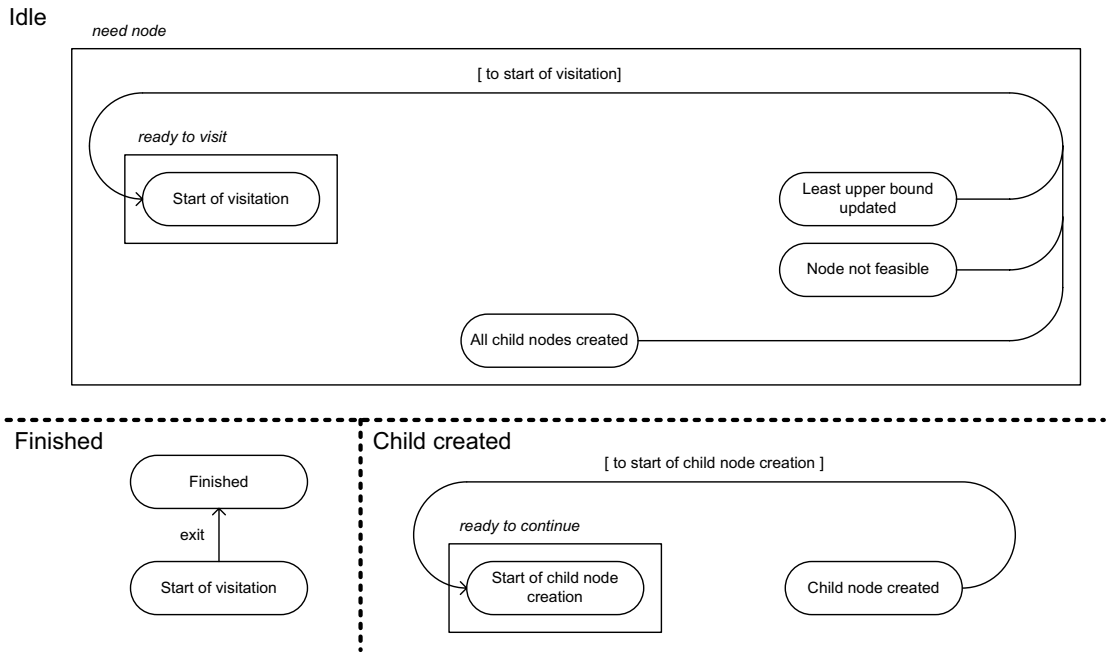


Figure B.28: Partition *AsCollab* for Process *Visitor*: Subprocesses *Idle*, *Finished* and *Child created*

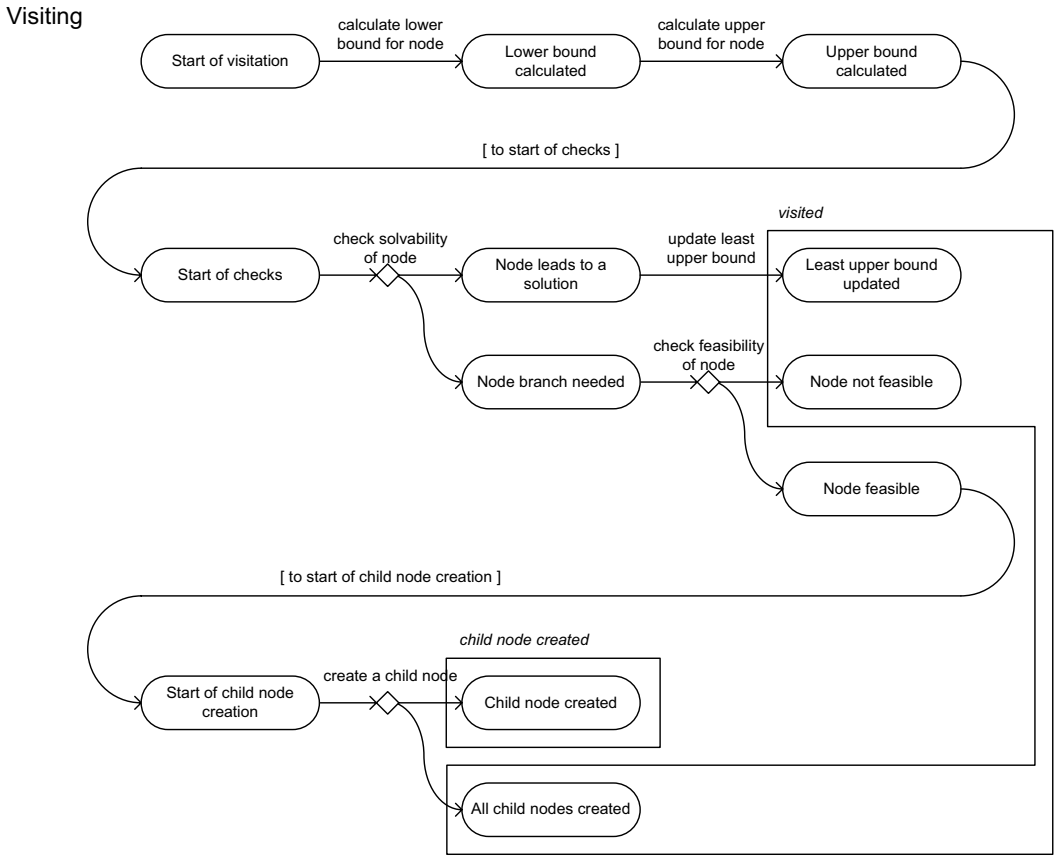


Figure B.29: Partition *AsCollab* for Process *Visitor*: Subprocess *Visiting*

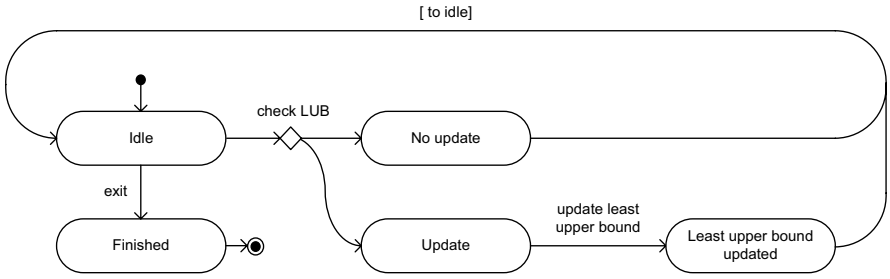


Figure B.30: Process *LUBManager*

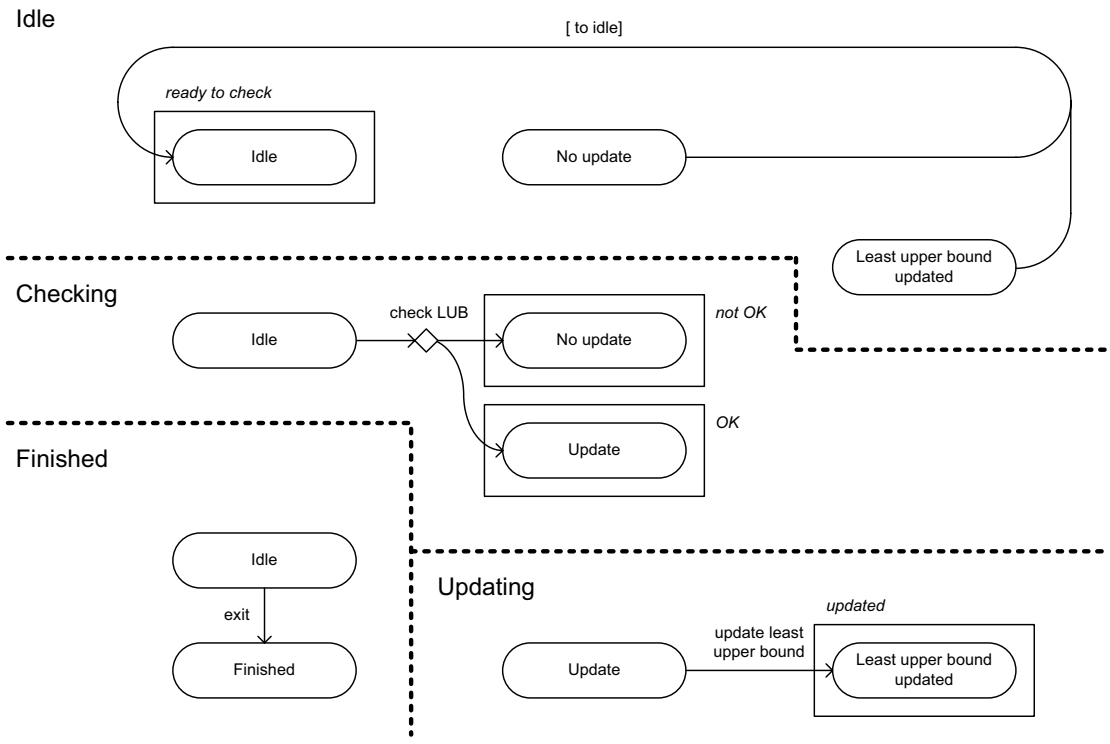


Figure B.31: Partition *AsBComm* for Process *LUBManager*

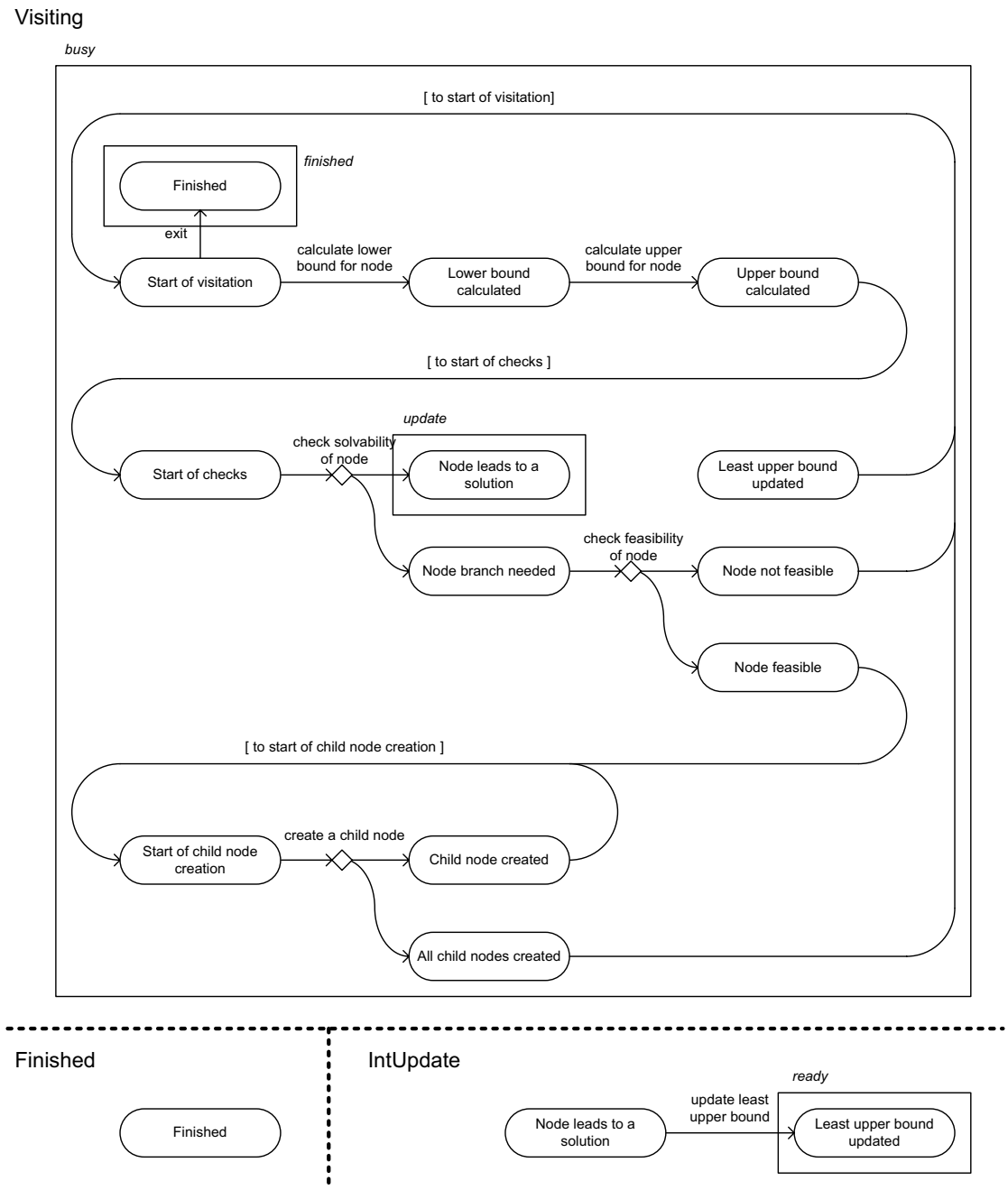


Figure B.32: Partition *AsBComm* for Process *Visitor*: Subprocesses *Visiting*, *Finished* and *IntUpdate*

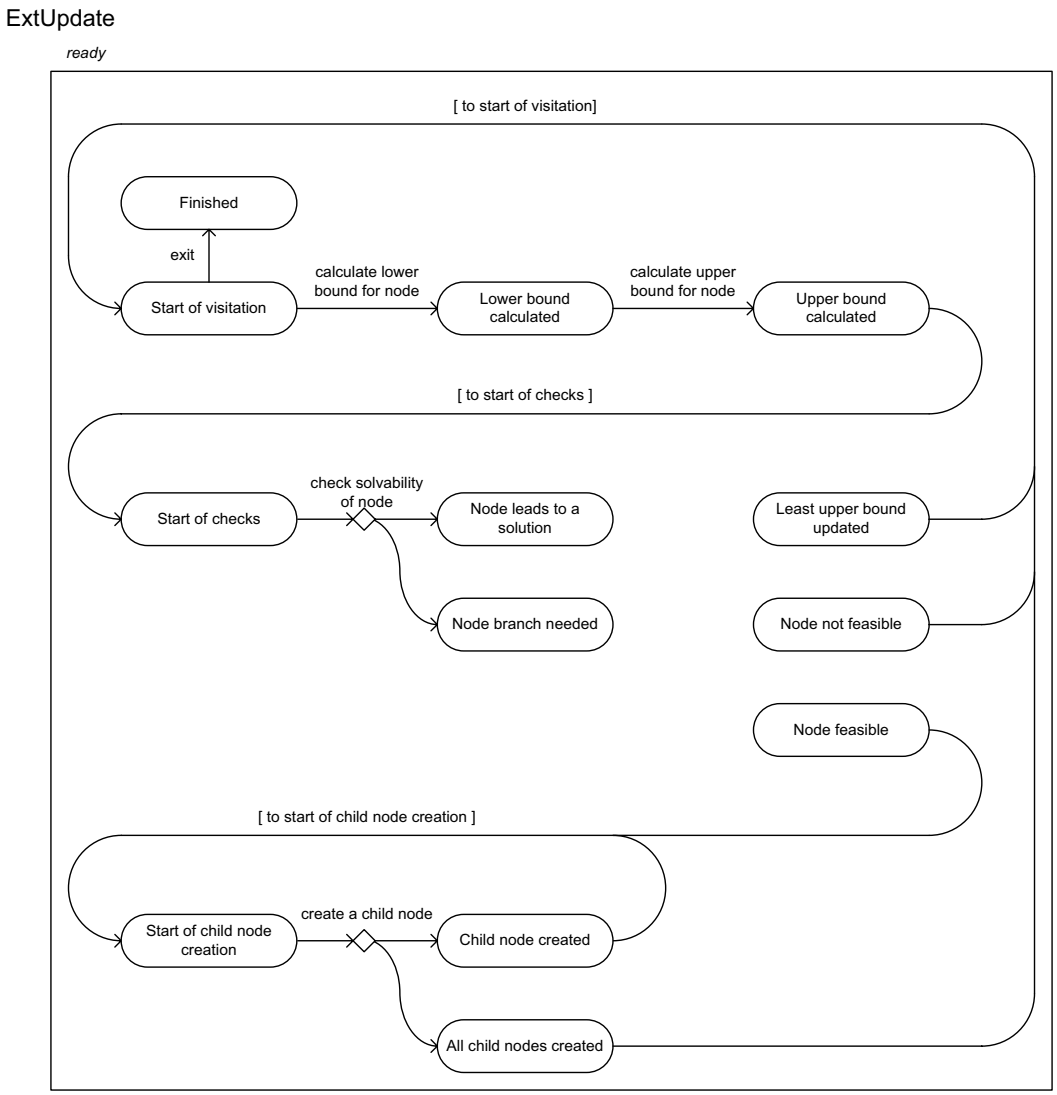


Figure B.33: Partition *AsBComm* for Process *Visitor*: Subprocess *ExtUpdate*

Samenvatting

Centraal in dit proefschrift staat het begrip *interactie* – wederzijds beïnvloedend gedrag. In hedendaagse computersystemen speelt interactie een steeds belangrijker rol. Software wordt meer en meer ontwikkeld als een verzameling zelfstandig werkende componenten die services aanbieden aan andere componenten. Iedere component kent een eigen "levenscyclus" – hij wordt ontwikkeld, doorontwikkeld, uitgebreid, gesplitst, samengevoegd, en weer weggegooid. Concrete toepassingen ontstaan doordat componenten worden samengesteld tot een groter geheel. Zulke toepassingen zijn alleen bruikbaar als er sprake is van zinvolle interactie tussen de componenten.

De modelleertaal PARADIGM, ontwikkeld aan het Leiden Institute of Advanced Computer Science (LIACS), is een taal die speciaal geschikt is voor het modelleren en analyseren van interactie. Een PARADIGM-model beschrijft op een abstract niveau het gedrag van individuele componenten, de rollen die de componenten kunnen spelen in interactie met andere componenten, en de manier waarop die interactie tussen de rollen plaatsvindt. De taal heeft een aantal interessante eigenschappen, niet in de laatste plaats het feit dat het een *executeerbare* taal is: PARADIGM-modellen kunnen door een computer worden uitgevoerd. Als bijzondere bijkomstigheid kunnen PARADIGM-modellen zo worden ingericht dat ze zichzelf *veranderen* terwijl ze worden uitgevoerd – zo kunnen we ook de levenscyclus van componenten modelleren.

In dit proefschrift hebben we ons gericht op de vraag welke mogelijke nuttige rollen PARADIGM kan spelen in het ontwikkelen van software. Om deze vraag te beantwoorden, hebben we drie complementaire activiteiten uitgevoerd. Ten eerste hebben we gereedschappen (software) ontwikkeld om PARADIGM-modellen op een computer te kunnen uitvoeren en te visualiseren. Ten tweede hebben we de taal uitgebreid met nieuwe concepten, die ons beter in staat stellen om structuur aan te brengen in de interactie tussen componenten. Dat vereenvoudigt niet alleen het analyseren van de modellen, maar is tevens van belang voor een efficiëntere executie van de taal door een computer. Tenslotte hebben we drie case studies uitgevoerd om de gereedschappen en de nieuwe concepten te kunnen beoordelen. De case studies tonen aan dat de uitgebreide versie van PARADIGM goed kan worden ingezet voor het modelleren en analyseren van non-triviale interactie in softwaresystemen. In combinatie met de ontwikkelde gereedschappen kunnen PARADIGM-modellen rechtstreeks worden gebruikt om concrete softwaresystemen mee te realiseren. Tenslotte leent de uitgebreide versie van PARADIGM zich voor het gestructureerd modelleren van de coördinatie van de evolutie van interagerende softwarecomponenten.

Curriculum Vitae

Andries Stam werd geboren op 14 oktober 1975 te Vlaardingen. Hij behaalde het VWO-diploma aan het Stedelijk Gymnasium te Schiedam in 1994, waarna hij startte met de studie Informatica aan de Universiteit Leiden. Na een stage bij het consultancybedrijf ID Research te Gouda sloot hij deze studie af in 2000 met het judicium “bene meritum”. Vervolgens werkte hij vijf jaar lang als consultant bij ID Research, Ordina Institute for Research and Innovation te Gouda, Ordina Public Consulting te Rosmalen en Ordina System Integration and Development Technology Consulting te Amersfoort. In deze periode voerde hij verscheidene opdrachten uit bij onder andere de Belastingdienst te Apeldoorn, De Telegraaf te Amsterdam, ING Nederland, het Korps landelijke politiediensten te Zoetermeer en ICTU te Scheveningen. Van 2002 tot 2004 werkte hij vanuit Ordina als onderzoeker aan het Nederlandse onderzoeksproject ArchiMate.

In 2005 startte Andries als onderzoeker aan het LIACS. Tot eind 2007 schreef hij het grootste deel van dit proefschrift en werkte hij aan het Europese onderzoeksproject Trust4all. Ook doceerde hij in die periode enkele maanden per jaar Enterprise Architecture aan de Master-opleiding ICT in Business. Sinds 2008 werkt hij als onderzoeker bij Almende te Rotterdam, een bedrijf dat zich richt op de toepassing van principes van zelforganisatie in hybride mens-agent systemen. Vanuit Almende is hij thans verantwoordelijk voor het Europese onderzoeksproject CREDO.

Andries combineert zijn werk in de informatica met verscheidene aanstellingen als kerkmusicus in Vlaardingen, Schiedam en Rotterdam. Als organist en begeleider van koren en solisten gaf hij in de afgelopen vijftien jaar meer dan driehonderd concerten in Nederland, Duitsland, Frankrijk, Engeland, Tsjechië en Hongarije. In oktober 2008 trouwde Andries met Wilma Stolk, in juli 2009 werd hij vader van zoon Wouter.