Diplomarbeit

# Modeling Agent Interaction Protocols with AUML Diagrams and Petri Nets

by

Lawrence Cabac

Presented to:
Dr. Daniel Moldt
Prof. Dr. Horst Oberquelle

University of Hamburg
Department of Computer Science

Hamburg
December 9, 2003

“The first great consideration is that life goes on in an environment;
not merely *in* it but because of it, through interaction with it.”

John Dewey, *Art as Experience* (1934)

## Acknowledgements

First of all I would like to thank my supervisors, Dr. Daniel Moldt and Prof. Dr. Horst Oberquelle, for their help and support. Especially the help and ideas of Daniel and his confidence in my work were a great support for me. I would also like to thank the staff of the Theoretical Foundations Group of the Computer Science Department, University of Hamburg, for their assistance. Furthermore, I thank all the people that have proofread this work and gave me good advice on my language and the consistency of my argumentation. I am also grateful to them for their help with LaTeX. These persons are (in alphabetical order): Till Dörges, Dagmara Dowbor, Michael Duvigneau, Aleksander Koleski, Julia Lütsch, Henning Mundt, Jan Ortmann, Christine Reese, Frédérique Revuz, Heiko Rölke, Jörn Schumacher, Christina Theilmann. I would also like to thank Dr. Martin Klepper for proofreading the proceedings paper (Cabac et al. 2003). Finally, I would like to thank my family, especially my father George Cabac and my grandmother Ilse Schneider for their support and patience. Especially, I would like to thank my mother Jutta Cabac for her love and trust in my abilities.

**Erklärung**

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

**Notice**

I declare that I have written the submitted work without any help of others and that I did not use any other resources except those cited. Furthermore, I declare that the submitted work has not been put forward in this or a similar form to any board of examiners before. In this work all quotations and all statements, which are adopted from other sources, are marked as such.

Hamburg, December 9, 2003

Lawrence Cabac

## Abstract

Agents are autonomous software components that can communicate with other agents. In the software engineering view agents can be regarded as generalizations of objects. Agents have knowledge of their environment, can act proactively and have a certain goal that they want to reach. Several agents that are communicating with each other to follow an individual or common goal are forming a society of agents called a multi-agent system. As a reference architecture *Mulan*, a Petri net-based agent platform on which multi-agent systems can be build, is introduced.

In this work modeling techniques for agent interaction protocols are introduced. These interactions, also called conversations, are modeled with AUML diagrams and Petri nets. In the agent unified modeling language (AUML) some extensions are proposed for sequence diagrams as defined in the unified modeling language (UML). The agent interaction protocol diagrams that are developed in this work, which are used to model conversations, are a variant of these extended sequence diagrams. The aim in using the agent interaction protocol diagrams, which are restricted to certain elements and constellations of elements, is to achieve a modeling technique for *Mulan*-based agent interaction protocols that is consistent and helps the developers in the development process of *Mulan* protocols. These protocols, which are Petri nets, are implementations of agent interactions within *Mulan*.

The presented work also introduces a straightforward approach for generation of Petri net code structures from AUML agent interaction protocol diagrams. This approach is based on the usage of net components that provide basic tasks and the structure for the Petri nets. Agent interaction protocol diagrams are used to model agent conversations on an abstract level. By mapping elements of the diagrams to net components, it is possible to generate code structures from the drawings. For this approach tool support is provided by combining net components with a tool for drawing agent interaction protocol diagrams. This combined tool is integrated in Renew (**Re**ference **Ne**t **W**orkshop) as a plug-in.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Recurrent manual tasks are inefficient, error-prone and dull. This is the reason why these tasks are increasingly substituted by automatic or semi-automatic procedures in all fields of construction or development. In fact this is the reason why the computer was invented.

Between 1819 and 1822 Charles Babbage constructed the differential engine for the purpose of calculating tables of logarithms. In those days, calculating tables was a very error-prone task and required significant manpower as well as much time. The differential engine was a special purpose machine, but it was soon followed by Babbage's work on the design of the analytical engine. This, although never built by him, is regarded as the first concept of a modern multi-purpose computer.

Although this was almost two hundred years ago similar problems still exist today. Complex tasks have to be accomplished especially in the area of software development.

## 1.1 Motivation and Context

The construction of software requires multiple tasks. These are for example modeling, designing, developing, implementing and debugging of software. In addition, creativity and the ability to communicate with other developers or users are indispensable attributes for a developer. Some tasks are nevertheless frequent, recurrent or dull, and many tasks are error-prone. This view does not focus on errors of design that can be prevented by experience and ability only. Instead, the subject here is errors that are made because of involuntary mistakes. These mistakes are so frequent that debugging has become a normal process in software development. However, developers can be freed of recurrent or frequent tasks by automation, or they can be supported by semi-automatic (computer-aided) construction.

Computer-aided software engineering (CASE) tools are made for developing software systems. These tools integrate support for modeling, construction, refactoring and debugging of applications as well as team development. Their main purpose is to provide an integrated development environment (IDE) for the developers. This means that they combine several tools that are necessary or useful for the development process. Often they also hide these tools behind one central user interface, which is then the only one the developers have to deal with. Furthermore, those tools are designed to support a certain approach or tool dependent policy of the development process. Developers have to conform to standards, constraints and workflows that are given by the IDE, customers or internal regulation of the development company. In addition, project-related guidelines have to be considered.

As a benefit, the tools provide help for developers to facilitate and support their work. Some examples are syntax highlighting, code formatting, context sensitive help, in-line application programming interface (API) documentation, graphical user interface (GUI) development support or code generation from diagrams. All these supporting tools facilitate the development and strip the development process of unnecessary recurrent manual tasks and marginal work. Successful tools for various programming languages exist and are extensively used.

In software development, modeling can be done using the unified modeling language (UML), the de jure standard. Most CASE tools support diagrammatic modeling with at least some of the diagrams defined by the object management group (OMG 2003). These models have their advantages and are broadly applicable, but they are nevertheless only semi-formal techniques. Petri nets give a formal approach to system modeling. Especially in the process of modeling behavioral aspects, Petri nets can be of advantage. In addition, modular structures can be modeled through refinement and coarsening.

While modeling with Petri nets is common, still the idea of programming with Petri nets is a possibility that is not broadly accepted. However, especially when it comes to concurrent and distributed processes, e.g. multi-agent systems, Petri nets have advantages. Building a framework for concurrent and distributed software systems as a multi-agent system on the basis of reference nets (Kummer 2002) - a high-level Petri net formalism - seems obvious and has been done. The framework's reference architecture for the multi-agent system used at the University of Hamburg is called *Mulan* (**Mul**ti-**A**gent **N**ets, see Köhler et al. 2001 and Duvigneau et al. 2003). It is implemented in reference nets and Java and can be executed in *Renew* (The **Re**ference **Ne**t **W**orkshop, Kummer et al. 2001a, Kummer 2002). It is possible to build application software on the basis of this framework. This has been done in several teaching and research projects at the Computer Science Department of the University of Hamburg. Some examples used in this work are taken from these projects. The objectives of the projects were the implementation of the settler game.[1]

In the course of time the reference architecture has been enhanced to satisfy the need of applications and standardizations. It has been augmented by a FIPA-compliant agent management and communication platform CAPA (Concurrent Agent Platform Architecture, see Duvigneau 2002 and Duvigneau et al. 2003). The platform is implemented in Java and as *Mulan* agents. Benefits of CAPA are a standardized communication based

[1] "The Settlers of Catan" is a popular card board game. It has been implemented in a series of teaching and research projects. A description for the first project (Moldt et al. 2001) is available in (Bosch et al. 2002). In this work the game is referred to as the *settler* game and the projects as the *settler* projects (see also Moldt et al. 2002).

on the FIPA standard SL0 (Semantic Language Level 0) and a more sophisticated agent management.

The development processes of application software resemble those of other software systems. Difficulties that occur in any software development also occurred in the development of *Mulan*-based application software. The main challenge is to handle the complexity. This has to be done on various levels of abstraction. Some of these levels are:

**The development process** is so complex that a certain workflow should be established and followed.

**The coordination between groups** of developers has to be handled.

**The readability of *Mulan* protocols** has to be enhanced.

**Debugging** of the application has to be supported.

In conventional programming settings, all these points are elements that are covered by tools offered by integrated development environments. Regarding the challenges of developing *Mulan*-based applications it becomes obvious that integrated development support is needed, this is also recognized by Carl (2003).

The process of implementing application software in *Mulan* requires the construction of *Mulan* protocols that define the behavior of the agents. A *Mulan* protocol is a reference net that describes the communication and the internal behavior of this agent during one conversation. Since the construction of large systems requires the modeling of many *Mulan* protocols, which frequently use similar parts of functionality, the need for software engineering methods for efficient construction of *Mulan* protocols is evident. This includes standardization, conventions and tool support. Theses tools should be integrated in an IDE.

Several methods have been established to handle the complexity of *Mulan* protocols and support their construction. First, net components (Cabac 2002) are used to construct the *Mulan* protocols so that a unified and structured form of the protocols can be achieved. Second, the agent interaction is modeled on an abstract level using agent interaction protocol diagrams (Cabac et al. 2003). These diagrams are defined in the AUML (Agent Unified Modeling Language, Odell et al. 2000, FIPA 2001c) that is standardized by the Foundation of Intelligent Physical Agents (FIPA 2003). The AUML is an extension of the unified modeling language (UML 2003a, Booch et al. 1999, OMG 2003) and some of the proposals of the AUML will be included in the next version of UML, version 2.0 (UML 2003b).

The required effort for development has been successfully reduced by offering the above mentioned tool support for the construction and modeling of *Mulan* protocols. In addition, the form and the structure of *Mulan* protocols have become unified and easily readable, enhancing the quality of code and documentation. Refactoring and debugging are facilitated due to the use of standardized and exchangeable net components. The agents' communication processes are specified and thus documented in the agent interaction protocol diagrams. Moreover, the overview over the developed system during and after modeling and constructing has been enhanced.

This work describes one further step towards an integrated development environment of *Mulan* applications. By combining the two previously mentioned approaches it is possible to generate code structures in the form of net component-based Petri nets from the agent interaction protocol diagrams. In order to do this, both sides have to be considered. The net components have to be adjusted to satisfy the needs and constraints of the diagrams and the generation procedure. It has to be examined whether the diagrams are fit for the generation and how the modeling is to be done to receive a clear description of the procedures, so that code skeletons can be generated.

The introduction of these modeling techniques and a tight integration of them in *Renew* are steps in the direction of a tool-supported development of *Mulan* applications. The vision is that all tools that are available or needed for the development process of an agent-based application, can be combined to form an IDE that includes tools for modeling, development, debugging, documentation, team development support and execution of a-gent-oriented software systems based on Petri nets. Essential for this is a philosophy of interaction-focused development that brakes with the standard architectural or structural approaches in object-oriented software development.

This work focuses on the modeling of agent interaction protocols for *Mulan* agents. The models will be examined regarding their capabilities to extract implementation from them. However, since the implementation of *Mulan* protocols are Petri nets, we regard these as executable models. The question whether a model is executable depends, thus, solely on the level of abstraction. An implementation is only a concrete model. For Petri nets this could be described as implementation through refinement. A coarse Petri net is an abstract model and the refined version of the Petri net is the concrete model or implementation.

## 1.2 About this Work

The goal of this work is to take one step towards an integrated development environment (IDE) for the reference architecture of the multi-agent system *Mulan*. It aims at facilitating the construction of *Mulan* protocols that are the main parts that have to be constructed while implementing applications based on *Mulan*. The results of this work should lead into an agent-based software development process that is stripped of unnecessary redundant or recurrent, error-prone manual tasks. Thereby it contributes to a developer-friendly, ergonomic and faster development of those applications.

Considerable parts of this work are based on preparatory work which was presented in "Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen"[2] (Cabac 2002) and also "A Proposal for Structuring Petri Net-Based Agent Interaction Protocols" (Cabac et al. 2003). Parts of the content of chapters 3, 4, 5 and also pictures in these chapters are taken from those sources without further notice.

[2]Translation of the German title: "Development of geometrically distinguishable components for the unification of *Mulan* protocols."

### 1.2.1 Outline

In chapter 2 the unified modeling language (UML) will be introduced, giving an overview and showing two of the UML diagrams in detail (class and sequence diagram). In addition, the AUML extension for sequence diagrams is presented. The following chapter 3 introduces reference nets and *Renew*. Agents, multi-agent system and the reference architecture for a Petri net-based agent platform *Mulan* are presented in chapter 4.

Chapter 5 introduces net components and uses two examples to show how *Mulan* protocols can be modeled with them. Some general aspects of modeling agent interaction are set up in chapter 6. Furthermore, three graphical approaches of modeling agent interaction with sequence diagrams, Petri nets, and agent interaction protocol diagrams are presented and discussed.

The realization of a plug-in for *Renew* that enables developers to draw agent interaction protocol diagrams within *Renew* and also generate code from these diagrams is prepared in chapter 7. This is followed by the description of the realized tool, its integration in *Renew* and its development process in chapter 8, where also three examples of *Mulan* protocols are presented, comparing generated code structures to custom made *Mulan* protocols.

### 1.2.2 Notation

This text conforms to the convention of text styles shown in table 1.1. This should give the reader some help on where or how he can find additional information about acronyms or proper names. Acronyms are listed in appendix C. However, if code is taken from images of diagrams then it is printed as program code and not as inscription.

| **Category** | **Style** |
|---|---|
| Program code | `Verbatim` |
| Inscriptions from Pictures | *Italic* |
| Persons | Initial Capitals |
| Names of tools | *Slanted* |
| Acronyms | CAPITALS |

Table 1.1: Notations

The full names explaining the acronyms are printed with the referring letters displayed bold only if they differ from the initials. See *Renew* and *Mulan* as examples for that in this introduction.

# Chapter 2

# Unified Modeling Language

The great challenge in software development is the handling of complexity. This refers to the complexity of the systems themselves as well as the complexity of the development processes. These systems and processes have grown so large and complex that a single person cannot completely comprehend their nature. Hence, to be able to construct these systems or follow such construction processes especially with large groups, methodologies have to be used that can handle their complexities.

In this chapter the unified modeling language (UML, see OMG 2003) is introduced as a means of modeling. The goal is to handle the complexity of large software systems. This is mainly achieved by using abstractions. Two examples of diagrams used in UML – the class diagram and the sequence diagram – are presented.

In addition, agent interaction protocol diagrams are introduced. They are a variation of extended sequence diagrams proposed in the agent unified modeling language (AUML 2003). These diagrams are used, for example by the Foundation for Intelligent Physical Agents (FIPA 2003), to model agent conversations.

## 2.1 Abstraction

One common approach to handle complexity is to modularize systems or processes, to split a task into smaller tasks and solve them individually (decomposition). Another approach is to model behavior or architecture before the construction. Both approaches are based on abstraction. Besides these, there are also other kinds of abstractions frequently used in software development. Some others are naming, data abstraction or functional abstraction. All these types of abstraction can be applied to systems as well as processes.

Object-orientation (introductions to object-orientation can be found in Sommerville (1996) or Oestereich (2001)) is the result of the software-engineering community's attempt to cope with the growing complexity of programs and their development over the years. In the object-oriented approach numerous ways of abstractions are used. Objects themselves are instances of classes that are typed and named modules. They are (usually) ordered hierarchically in a tree of dependencies called inheritance. Object-oriented programs are composed of objects that use, create or destroy other objects. Objects are modeled in analogy to the (to be modeled) part of the real world or domain.

Surely, not only the software systems but also the software development processes have been subject to progress. Again, the challenge regarding development processes is how to cope with complexity. Several software process models or paradigms have been developed. For object-oriented development there are the Booch method (Booch 1993), the object-oriented modeling technique (OMT, see Rumbaugh et al. 1991) and object-oriented software engineering (OOSE, see Jacobson et al. 1992) that culminated into the unified modeling language (UML, see Booch et al. 1999 and UML Resource Center 2003) that was standardized by the Object Management Group (OMG 2003). The current version of UML is version 1.5 (UML 2003a) but the finalization of the new version 2.0 (UML 2003b) has already started.

### 2.1.1 Implementing and Modeling

The unified modeling language (UML) has established itself as the standard language for software modeling. The great success of UML can be explained by many reasons. These are, for example, simplicity, expressiveness, broad applicability and adaptability. One very practical reason is the existence of a broad variety of tools, supporting the design and construction of models.

One of the most important advantages of computer-aided software development is the fact that the tools can generate code from the models. By this the development process is accelerated significantly, leaving more time for the developer to care about design matters or enhancements instead of manually converting graphical specification into code. This approach tries to benefit directly from the fact that the specification, if modeled in the appropriate way, can be transformed into code. The aim is to generate as much code as possible from graphical specification. One could argue that this weakens the borderline between specification and implementation. At least the border is pushed from the concrete implementation as text, further towards the abstract graphical description, meaning that more implementation work is now already done during modeling. This could be described as implementing by modeling.

The designed model that used to offer the specification now also holds - at least parts of - the implementation. For a language - and for a programming language especially - the syntax and semantics have to be well defined to warrant precision and uniqueness. Only then it is possible to transform the description and receive generated code or code structures that can (after manual augmentation) be compiled into executable code. This holds also for graphic modeling using the unified modeling language.

### 2.1.2 The Purpose of Modeling

Modeling gives the developers the possibility to design the software systems they are about to build before they are actually constructed. This becomes more important as the complexity of software systems increase. A good model represents the system and parts of it at different levels of abstraction. Not only detailed information about parts is revealed but an overview over the whole system is also given. Essentially, modeling is a means of specification, information, documentation, communication and visualization. Especially the visualization of a system is crucial for its construction. If the developers can visualize the system and its structure then they can grasp its complexity. Modeling software systems can be compared to other engineering domains: *"Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large buildings"* (UML Resource Center 2003, web-site). Therefore, the diagrams of UML are a way for the developer to visualize the architecture and behavior of the software system. They can be regarded as the blueprints of software design.

## 2.2 Description of UML

The unified modeling language (UML) has established itself as the standard of software modeling especially for object-oriented software development. It provides a large and extensive set of modeling techniques, with which numerous aspects of software systems can be modeled. Each technique offers a diagram style to model the desired functionality. These diagrams can roughly be divided in two sets, one set describing the static structure and the other describing the (dynamic) behavioral characteristics of the system.

In this section only a small part of UML is described. For an introduction to UML see Sommerville (1996), Oestereich (2001) or Booch et al. (1999). A complete description is available online at the OMG web-site (OMG 2003).

Generally spoken, UML diagrams are used to describe software systems. There are different kinds of diagrams for different purposes and for different stages of modeling and implementation. These diagrams cover various views of the system and are meant for diverse actors (developers and users) in the software development process. Use case diagrams are used to ease the communication between developers and users. Other diagrams are mainly used by the developers.

| **Characteristic** | **Diagram** |
|---|---|
| Static | Class diagram<br>Object diagram<br>Component diagram |
| Dynamic | Use case diagrams<br>State Chart<br>Collaboration diagram<br>Sequence diagram<br>Activity diagram |

Table 2.1: A selection of UML diagram types

In the following sections two of the available UML diagram types (see table 2.1) are described to give a notion of the expressiveness. The class diagram is used to model the architecture of the software showing the static dependencies of the objects, while sequence diagrams describe the behavior of objects in a scenario.

### 2.2.1 Class Diagrams

The static diagrams are used to model the architecture of systems. They allow the developer to model the software architecture at different levels of abstraction and granularity. A class diagram (Booch et al. 1999, chapter 8) displays classes or types and their relationships. It shows the system's structure and the dependencies between the elements. Class diagrams are the frequently used and common diagrams of UML. They are excessively used to generate code.

#### Description

The class diagram is very expressive and can convey inheritance and usage relationships between classes respectively objects. Essentially, the class diagram is a graph with classes and interfaces as nodes. The relationships between the classes, association, generalization and dependency, are displayed as arcs between these nodes. While dependency and generalization are directed, associations can be directed or bidirectional. Associations can exhibit multiplicity and qualifiers and express different kinds of associations such as aggregation or composition, which are indicated by end decorations in the form of diamonds. Classes or interfaces are displayed as boxes with three (interface: two) compartments that are ordered vertically. The top compartment contains the name of the class. The other two compartments are optional and contain fields and methods. The basic elements of a class diagram are listed here:

**Class** is a box containing 3 compartments:

**Name** of the class or interface (in italics or marked ≪interface≫)

**Fields** of the class (optional)

**Methods** of the class or of the interface (optional)

**Associations** are displayed as arcs between classes. They can be directed.

**Aggregations** have hollow diamond ends.

**Compositions** have filled diamond ends.

**Multiplicity** (optional) is a number, a range of numbers or a '*'[1].

**Qualifiers** (optional) are rectangles attached to the end of associations.

**Notes** are boxes that have one folded corner that hold a text.[2]

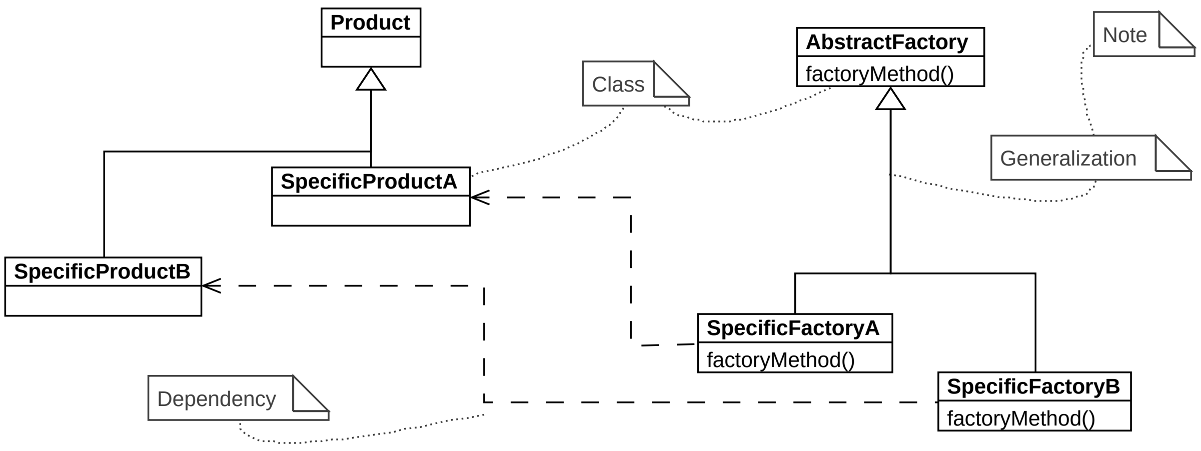


Figure 2.1: The design pattern *factory* as an example for a class diagram.

Figure 2.1 shows an example of a class diagram that describes the *factory* design pattern (Gamma et al. 1995). `SpecificFactoryA` and `SpecificFactoryB` are a subclass from `AbstractFactory`. They implement the abstract method `factoryMethod()`. The diagram is supplemented with note figures that can be used for annotations. In this example they describe the diagram elements.

Dependencies are indicated by dashed lines meaning that `SpecificFactoryA` creates `SpecificProductA` objects in `factoryMethod()` and `SpecificFactoryB` creates objects of the class `SpecificProductB` with its implementation of `factoryMethod()`.

[1]The asterisks ('*'), represents an arbitrary number.

[2]Notes are available in all UML diagram types.

### Generation of Code Structures

It is possible to generate code skeletons from class diagrams. The amount of code that can be generated depends on the level of detail of the diagram. If the diagram is crude the benefit from code generation is small. But the more details of specification are modeled in the diagram, the more useful code can be derived from it. The static structures of generalizations and the realizations - i.e. implementation of interfaces - can be transferred directly into code. This is information that is conveyed through the structure of the diagram. Other information about the implementation can be put into inscriptions like association inscriptions.

Figure 2.2 shows a small example taken from the samples of the *Poseidon for UML Community Edition 2.0.3* (Gentleware 2003).[3] The class diagram describes a class `Circle2` and a class `Point2` that both implement the interface `Fig` that declares three methods `center()`, `isVisible()` and is`Empty()`. `Point2` has a location (`x`, `y`) and offers access methods for the coordinates. `Circle2` has two attributes. One is the `radius` (integer) and the other is the `center` that is a `Point2`. The latter is represented as an association arc pointing at the respective class `Point2`.

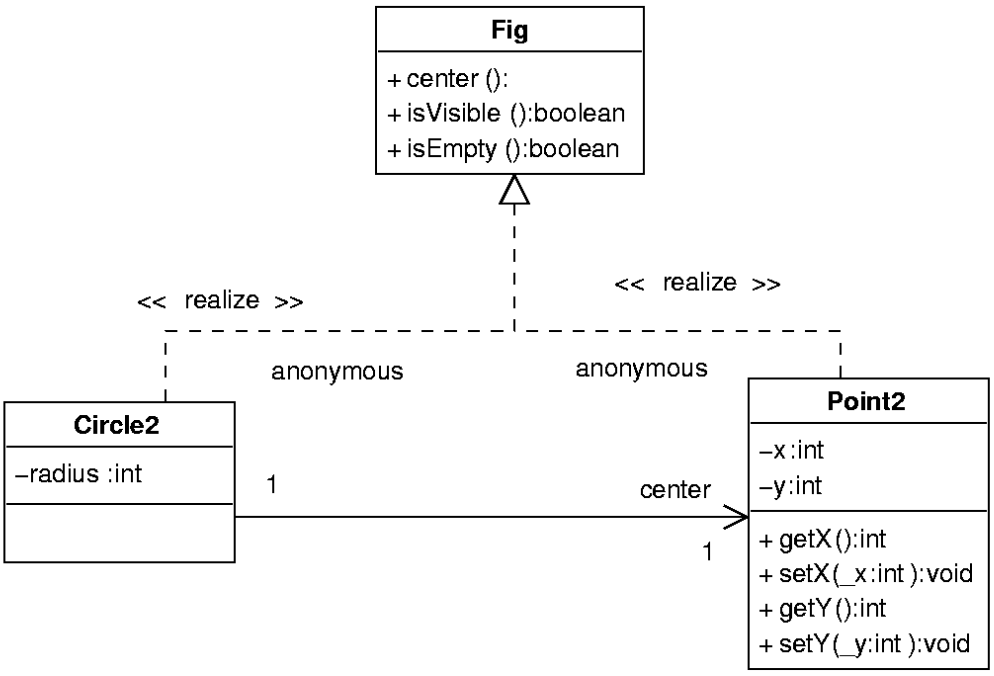


Figure 2.2: Example class diagram.

*Poseidon* can generate code skeletons from this simple example for the interface as well as for the two classes. Here only the part of the generated code of class `Circle2` is

[3]Poseidon for UML is a UML CASE tool.

presented. The code has not been pruned of unnecessary empty comment lines but some empty lines have been erased for compactness.

```
/** Java class "Circle2.java" generated from Poseidon for UML.
 *  Poseidon for UML is developed by <A HREF="http://www.gentleware.com">Gentleware</A>.
 *  Generated with <A HREF="http://jakarta.apache.org/velocity/">velocity</A> template
 * engine.
 */
import java.util.*;

/**
 *
 */
public class Circle2 implements Fig {

  ///////////////////////////////////////
  // attributes
/**
 * Represents ...
 */
    private int radius;

   ///////////////////////////////////////
   // associations
/**
 *
 */
    public Point2 center;

   ///////////////////////////////////////
   // access methods for associations
    public Point2 getCenter() {
        return center;
    }
    public void setCenter(Point2 point2) {
        this.center = point2;
    }
```

Figure 2.3: The generated code of class `Circle2`. Compare with figure 2.1.

The code skeleton of figure 2.3 reflects the realization relationship between `Circle2` and `Fig` by adding the `implements` statement. It also reflects the association between `Circle2` and `Point2` in the attribute `center` that is of type `Point2`. Note that Poseidon, in this example, does not produce any code for the abstract methods that have to be implemented defined in interface `Fig`. Especially the fact that `center()` should return the current `center` of a `Circle2` object is not reflected in the generated code. However, the structural information of the association is reflected in the attribute `center`, for which even access methods are created (optional).

The class diagram gives a good description of the static structure of the described system. However, sometimes it can be very useful to describe scenarios or the sequences of method calls. This can be done with sequence diagrams described in the following section.

### 2.2.2 Sequence Diagrams

Scenarios are descriptions of processes or sequences of action. They can be modeled with interaction diagrams (Booch et al. 1999, chapter 18) that describe the behavior of objects. There are two different kinds of interaction diagrams, the collaboration diagram and the sequence diagram. They are usually used to model the dynamic behavior of objects and the interactions between objects. Especially sequence diagrams give a good notion of the procedures because time is represented directly.

#### Description

In sequence diagrams the order of the displayed elements have a meaning: the vertical dimension represents time and the horizontal represents different instances of objects. Object identifiers that are located on the top of the diagram next to each other, represent the objects that are involved in the described scenario. The vertical dimension represents the temporal progress (downward) in the process. This orders the diagram sequentially. A life line is a vertical line that starts at an object identifier and extends downwards. The existence of an object is indicated by the life line, which runs downward from the object identifier until the object is destroyed. A cross that is located at the end of the life line indicates the destruction of the object. Times of activity of objects are displayed as activations, which run along the life lines. Method calls are displayed as message arcs that are send from one object to another. These messages can be decorated by a solid or a stick arrow tip indicating synchronous or asynchronous messaging. If a message is a return statement the line is dashed. Only an object that is active can send messages, i.e. call methods. Several graphical elements are listed here:

**Object identifiers** describe the objects that are involved in the modeled process.

**Life lines** start at an object identifier and last until the objects are destroyed.

**Activations** mark the active states of the objects. An activated object can act and thus call methods of other objects or of itself.

**Messages** are used to represent the method call.

**Crosses** are used to indicate a destruction of objects.

Figure 2.4 shows an example sequence diagram of two objects `Object1` and `Object2` of the Type `A` and `B`. Object1 creates a new instance of `B` (`Object2`) and calls `aMethod()` of `Object2`. After the method is finished the return statement lets `Object1` continue with its

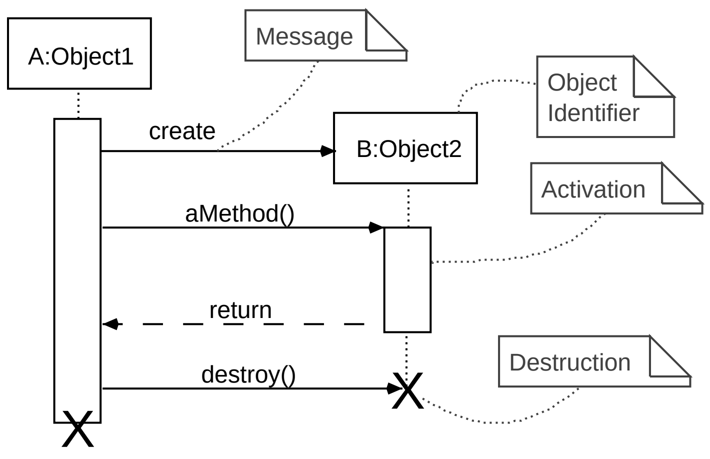


Figure 2.4: Example sequence diagram.

activity, which is the destruction of `Object2`. Finally `Object1` destroys itself. Note that the sequence is clearly recognizable by the top-down ordering of the diagram.

In contrast to the class diagram that reflects the static architecture of a system, the sequence diagram reflects the behavior of it or of parts of it. The sequence diagram does not contain any information of the system's structure. Instead it represents the actions of the objects for a certain scenario. Furthermore, it does not reveal any information of the internal actions of the objects. It focuses thus, on the communication between objects.

## 2.3 Agent Interaction Protocol Diagrams

Sequence diagrams are restricted to show only one scenario each. This is a limitation that can be overcome by introducing control flow elements as supplements in the diagrams. By this, several scenarios can be folded together into one diagram. Proposals have been made by Odell et al. (2000) that cover nested sequence diagrams, combinations of sequence diagrams with other (nested) diagram types and the introduction of new elements providing the control flow elements mentioned above.

This section focuses on the extensions regarding control flow elements. These elements are presented in the next subsection. The diagram types that use these extensions are called extended sequence diagrams, protocol diagrams or agent interaction protocols. To distinguish the variant of these diagrams that is used in this work, they are here, and furthermore in this work, called *agent interaction protocol diagrams* (AIP). Since these diagrams are extended sequence diagrams, it seems appropriate to describe them at this point together with the other UML diagrams, although the term 'agent' has not been defined yet in this work. The latter is done in sections 4.1.3 and 4.1.4. For an understanding of the diagrams the notion of an agent does not have to be further specified. It should only be stated here that an agent could be regarded as a generalization of an object, that agents communicate with other agents by messages and that agents can act proactively.

### 2.3.1 Extending Sequence Diagrams

Agent interaction protocol diagrams are a part of the agent unified modeling language (AUML) (see AUML 2003 and FIPA 2001c) that extends UML with agent related modeling techniques.[4] Extended sequence diagrams and agent interaction protocol diagrams are intended to enhance the modeling capabilities of sequence diagrams to model agent protocols.

### 2.3.2 Additional Elements

In extended sequence diagrams some additional elements are added to the usual elements of sequence diagrams. Those additional elements provide alternative, concurrent and arbitrary splitting in a manner of the three gates AND, XOR and OR.

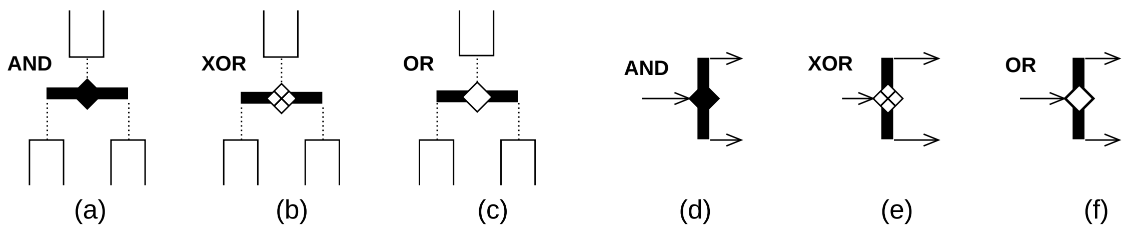


Figure 2.5: New elements of extended sequence diagrams: AND, XOR, OR splits.

Figure 2.5 displays the new elements in a horizontal and a vertical version. The first are used to split the life lines of an agent (a, b and c) and the latter elements are used to split the (vertical) messages (d, e and f).

Odell et al. (2000) propose two different ways of using these new elements: an elaborated version and a short version. Both versions are presented in the FIPA interaction protocol library specification (FIPA 2001c). In the elaborated version the two forms are always used together. This means that a split of messages also enforces a split of life lines. In the short version the life line split can be omitted.

Figure 2.6 shows an example protocol diagram for the contract net protocol as presented in (FIPA 2001a). It shows the short (abbreviated) variant of presenting alternatives with the additional elements.

In addition to the split figures, FIPA describes the complements of the split figures as a variation of their presentation (FIPA 2001c). These are actually join figures – as pointed

[4]The specifications of the agent unified modeling language are defined and maintained by the Foundation for Intelligent Physical Agents (FIPA 2003).

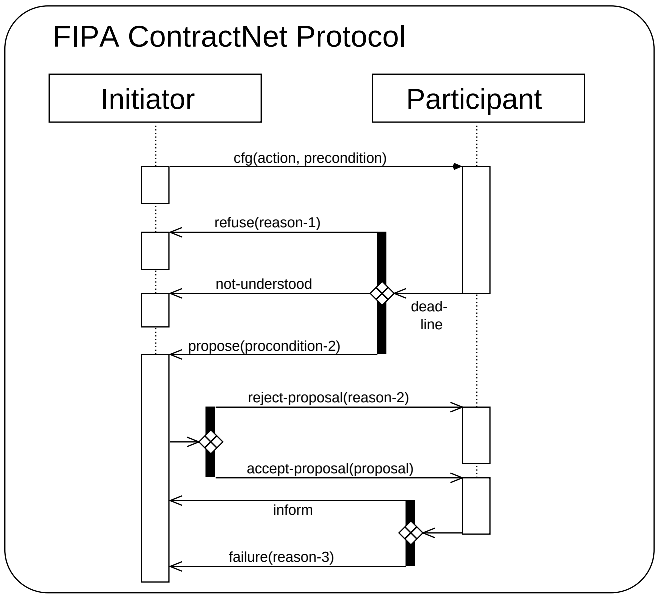


Figure 2.6: The FIPA Contract Net Interaction Protocol diagram.

out in (Cabac et al. 2003) – matching the according split figures as complements. They are displayed in figure 2.7.

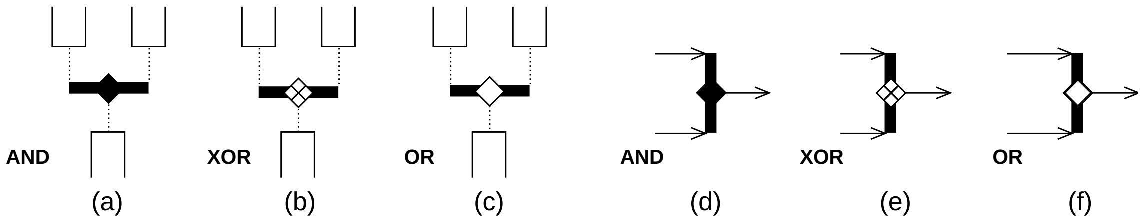


Figure 2.7: New elements of extended sequence diagrams: AND, XOR and OR joins.

A split up life line can be rejoined at some point in the diagram. This reflects a synchronization for the AND split and a merge for a XOR or OR split. Since the FIPA defines them as variation of presentation, they look similar to the split figures. However, the appearances of these figures in diagrams as splits or as joins distinguish the two sorts of figures from each other. In this work and in the implementation of the plug-in (chapter 8) a clear distinction is made between the splits and joins.

In this work, only AND and XOR splits and joins are used. Since the semantics of OR splits and joins has not been defined properly. Furthermore, most of the diagrams are displayed in the elaborated version. The only exceptions to this are the diagrams taken or adapted from other sources. However, the elaborated version used in this work differs slightly from the elaborated version of the original proposal.

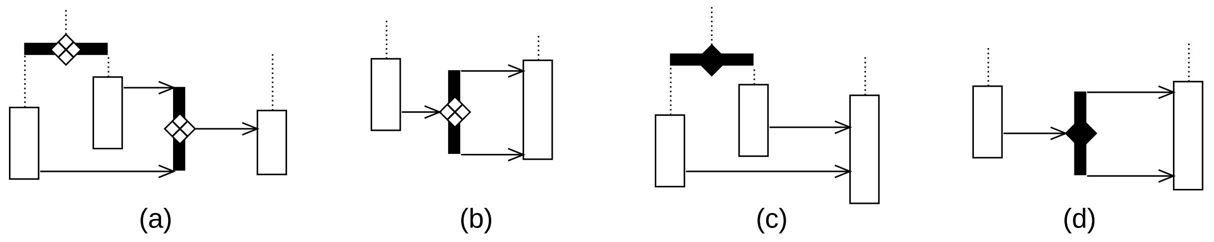


Figure 2.8: Reflecting actual message numbers in diagrams by combining splits and joins. A message sent after a decision results in a single received message (a). Two messages sent concurrently result in two received messages (c); and the short forms (b) and (d).

The difference is, however, only a difference in style or in variation of presentation and thus within the limits of the definition of interaction protocols. The aim in using this style is that especially the number of messages send from an agent or received by an agent is reflected in the number of message arcs drawn in the diagram. So the messages send to one agent after a life line XOR split with more than one outgoing message arc is joined by a message join figure. Figure 2.8 illustrates this. For a detailed discussion see section 6.3. These diagrams, that satisfy the briefly described style, are in this work called agent interaction protocol diagrams (AIP).

## 2.4 Summary

The unified modeling language (UML) is the de jure and well-accepted standard in software modeling. It provides several techniques to support modeling at different stages and for different points of views. A general distinction can be made in architectural and behavioral modeling.

In this chapter two diagram types were presented to show the modeling capabilities of UML. These are the class diagram and the sequence diagram. In addition, agent interaction protocol diagrams, as an extended variant of sequence diagrams, were presented. These diagrams, which are part of the agent unified modeling language (AUML), are meant to overcome the restrictions of sequence diagrams by introducing control flow elements.

# Chapter 3

# Reference Nets and *Renew*

The formalism of Petri nets has the advantage of a dual representation. Petri nets can be formulated as graph or as text. As graph, humans and as text, machines can easily read Petri nets. As a benefit of the textual representation machines can not only read the code but also execute it. The benefit of the graphical representation is that the structure of the net is revealed to humans and that the execution (or simulation) of the nets can be visualized as a token game, which is an animation of the tokens in Petri nets.

## 3.1 P/T-Nets

Jessen and Valk 1987 define nets as tuples of sets. Those sets are the places, the transitions and the flow relation. Together these sets form a directed, bipartite graph. A basic P/T-net (Place/Transition-net) is a net that also includes capacities for places, weights for arcs and an initial marking. While the abstract textual descriptions seem quite unintuitive, the corresponding graphical representation can be very comprehensive. Figure 3.1 shows an example[1] of a Petri net with the basic net elements: Transitions, Places and Arcs. The figure also displays some transition and place inscriptions.

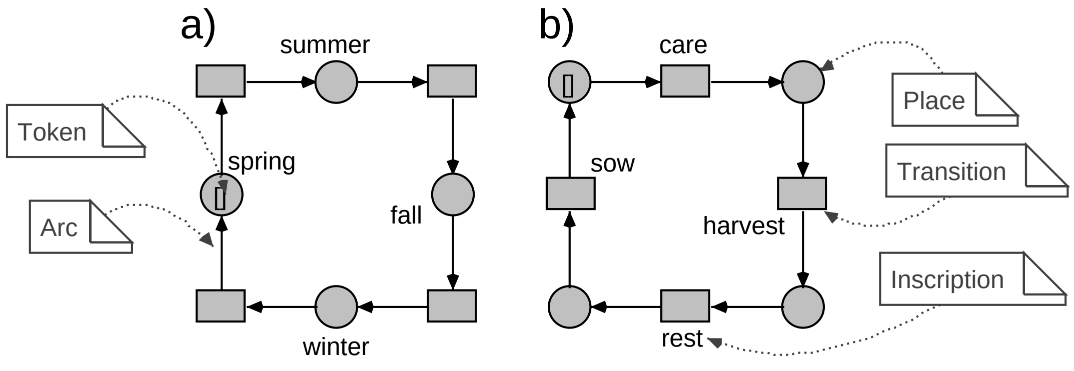


Figure 3.1: The seasons modeled as a Petri net in two different views: The view of the meteorologist (a) that focuses on the states of the seasons and the view of the farmer (b) that focuses on his activities during the seasons.

In the example of figure 3.1 inscriptions of places and transitions are names or labels for the net elements and have no influence on the execution of the net. Transitions are enabled (activated, firable), if the pre-conditions and the post-conditions are satisfied. To satisfy the pre-conditions enough tokens have to be in the input places. To satisfy the post-conditions the capacity of the output places should not be exceeded.

For detailed descriptions of the Petri net formalism see Girault and Valk (2003), Jensen (1996) or Reisig (1982).

## 3.2 Reference Nets

Reference nets (Kummer 2002) are object-oriented high-level Petri nets, in which tokens can be nets again. For these ‘nets within nets’ (Valk 1995), referential semantics is used. Tokens in one net can be references to other nets. In a simple setting of a single nesting of nets, the outer net is called system net while a token in the system net refers to an object net. Nevertheless, object nets themselves can again contain tokens that represent nets,

[1]The example is an adapted version of a net in Petri (2003)

and so a hierarchy of nested nets can be obtained. The benefit of this feature is that the modeled system is modular and extensible. Furthermore, transitions in nets can activate and trigger the firing of transitions in other nets, just like method calls of objects, by using synchronous channels (Kummer 2002) (Christensen and Hansen 1992).

*Renew* (The **Re**ference **Ne**t **W**orkshop (Kummer et al. 2001a), (Renew 2003)) combines the 'nets within nets' paradigm of reference nets with the implementing power of Java. Here tokens can also be Java-objects and nets can be regarded as objects. Objects are instantiations of classes; in a similar way reference nets are instantiated, thus many instances of a net can be produced dynamically while their structure and initial marking are defined in a net template - corresponding to the class defining the structure of an object.

In addition to the net elements of P/T-nets, reference nets offer several additional elements that increase the modeling power as well as the convenience of modeling. These additional elements include some new arc types, virtual places and a declaration. Several inscriptions have been added to the net elements providing functionality for the different net elements. Places can be typed and transitions can be augmented with expressions, actions, guards, synchronous channels and creation inscriptions. All these features are described in the following sections.

### 3.2.1 Types

P/T-nets only deal with black (anonymous) tokens that are indistinguishable from each other. In reference nets tokens can also be of any data type that is available in Java. So tokens can be primitive numerical data types like `int` or `long` as well as any object. Reference nets themselves are objects of the type `de.renew.simulator.NetInstance`. Nevertheless anonymous (black) tokens are also available in *Renew*. The notation for a black token is: '`[]`'.

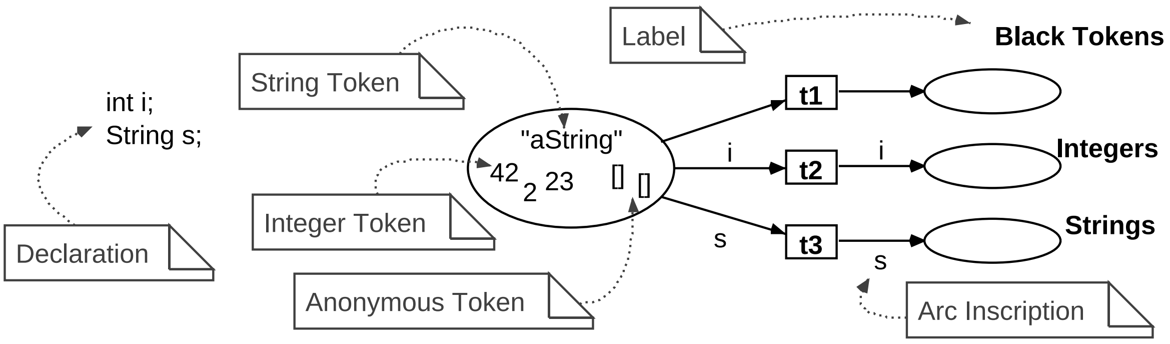


Figure 3.2: Tokens can be objects, primitive data types or anonymous. Objects can only be moved by inscribed arcs, anonymous tokens by arcs without inscription only.

Arcs without inscription can only move black tokens. Variables or expressions in arc inscriptions are required to move other tokens, i.e. tokens that refer to objects, primitive types or net instances. In a simple example an object can be removed from a place and put into another place by adding inscriptions to the two arcs that move the token. This is illustrated in figure 3.2.

If, for instance, a transition moves object tokens from an input place to an output place, the inscriptions of input and output arcs have to be identical. This is the case in the simple net of figure 3.2. The firing of the transitions `t2` or `t3` moves objects but not anonymous tokens, which can only be moved by arcs without inscription (`t1`). The binding of the variable to the object is locally restricted to the neighborhood of the transition.

After the transitions in the example of figure 3.2 have fired six times, all tokens have been removed from their initial place by the three transitions and sorted by their type into the places labeled accordingly (see figure 3.3). The transition t1 can only move anonymous tokens, transition t2 only integers (`int`) and transition t3 only objects of the type `java.lang.String`.

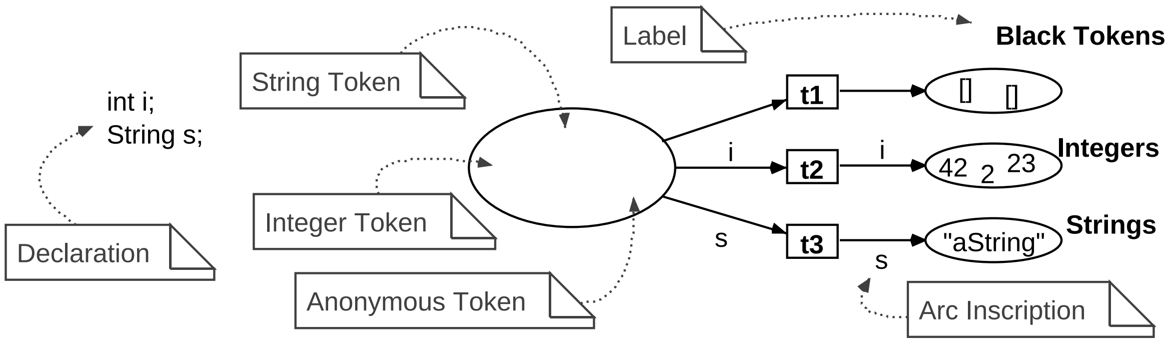


Figure 3.3: After execution of the net the objects are sorted by type.

An inscribed arc can move any type of object. However, if the variable in the inscription is declared in the declaration, then the arc can only move objects of the type of the variable. In the example, the two variables `s` and `i` are declared as `String` and `int`, so `t2` only moves integers and `t3` only strings. By defining the type of the variables the sorting of the tokens is achieved.

## 3.2.2 Inscriptions

Several sorts of inscriptions are available for places, transitions and arcs in reference nets. Place inscriptions do not only define the initial marking of a net, but also the type of a place. A typed place can only hold tokens of the appropriate type.

Arc inscriptions can be constants or variables. Furthermore, expressions are allowed in output arcs inscriptions. Figure 3.4 shows the type inscription of a place and an expression on an output arc.

There is a variety of transition inscriptions: expressions, guards, actions, creation inscriptions and inscriptions for synchronous channels. Expressions are unlabeled, while guards and actions are prefixed with the keywords `guard` or `action`.

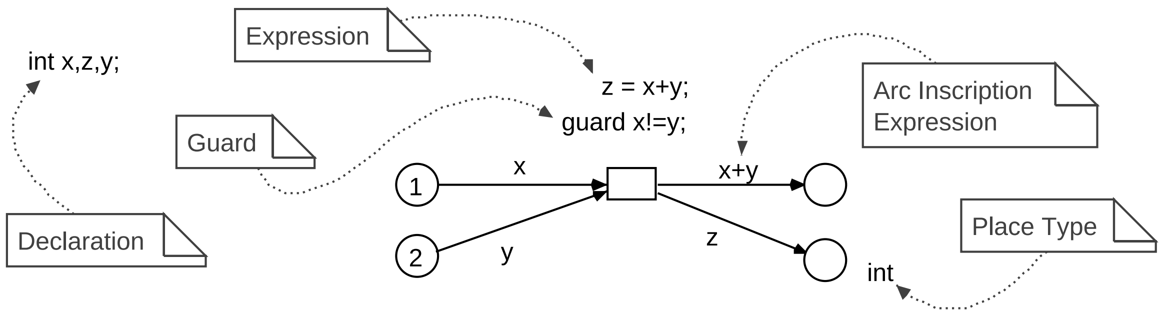


Figure 3.4: Several inscription types.

For the firing of a transition several conditions have to be satisfied. Firstly, the transition has to be activated. This is the case when there are a sufficient number of tokens of the appropriate type – i.e. compatible to the arc inscription – in the input places of that transition. And secondly, the guards have to be satisfied, i.e. the expressions in the guards have to evaluate to `true`.

Expressions on a transition are evaluated while the simulator searches for the binding of the transition. The result of the evaluation of an expression is discarded, if it is not bound to a variable by using the unification operator '`=`'. In contrast to expressions, actions are guaranteed to evaluate only once during firing of the transition. They should be used instead of expressions when these contain Java method calls that have side effects. Expressions on arcs have the same effect as those on transitions, but in contrast to transition expressions, the result is not lost and moved to the output place.

The other two kinds of inscriptions for net creation and synchronous channels are presented in section 3.2.5.

### 3.2.3 Arcs

In addition to the simple unidirectional arc of P/T-nets, reference nets also offer three other kinds of arcs. *Reserve arcs* are equivalent to two arcs connecting the same two nodes in opposite directions. *Test arcs* are very similar to *reserve arcs*, but instead of removing the token at the start of firing and returning it at the end of firing, as it is the case for reserve arcs, they just test whether there exists a token on the connected place without removing it. This means that the main difference between the two is that, while the transition is firing, the token can be used by another test arc concurrently. As shown in figure 3.5, the arcs types are distinguished by the number of their arrows tips. The reserve arc has one arrow tip at each end of the arc while the test arc has no arrow tip at all.

The last arc types, that are special in reference nets, are the *flexible arcs*. For a more detailed description see Reisig (1997) and Kummer et al. (2001b). They are displayed with two arrow tips at one end of the arc. These arcs simulate a flexible number of arcs. The incoming arc withdraws all elements of a `Collection` object that reside on a place

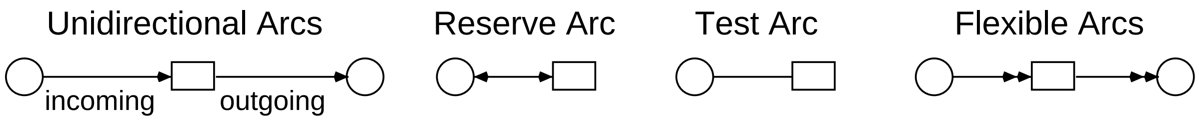


Figure 3.5: Additional arc types in reference nets.

from this place. The transition can only fire, if all elements of the `Collection` are on the place, and only those elements are withdrawn concurrently. Although this only works if the objects in the `Collection` are already known, this mechanism still provides a powerful method for modeling. By using the flexible output arc, all elements of a Java `Collection` object can be put into a place. As an example for the usage of the flexible arc, compare with the net component *NC forall* in section 5.2.

### 3.2.4 Virtual Places

A virtual place is a reference to a place. For any place there can be numerous virtual places that act like place holders for the original place. An arc connected to the virtual place has the same effect as an arc being connected to the original place. So a token that is on the place can be retrieved from any virtual place corresponding to the original place. A token in the net template in an original place or in a virtual place, thus, appears in the net instance in the original place and in every virtual place corresponding to the original. It is, nevertheless, only one token.

Virtual places can only be used within one net, not across different nets or net instances. So they basically are a matter of convenience. Especially, if places have many connecting arcs[2] or if they are connected in different areas of the net the use of virtual places has advantages. In other high-level Petri net formalisms, like coloured Petri nets (Jensen 1996), a similar concept is known as fusion places. Figure 3.6 displays a simple example for virtual places. A token that resides on the original place can be retrieved from either virtual place by firing `t2` or `t3` but not from both.

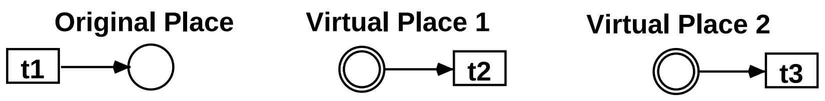


Figure 3.6: Original and virtual places.

Virtual places are identified in *Renew* by a double outline. Additionally, they exhibit the same color as the original place to which they belong. However, for humans it is sometimes difficult to map the virtual places to the originals when more than one original place exist.[3]

[2]Compare with the net for the knowledge base of the *Mulan* agent in figure 4.8

[3]In *Renew* the original place can be found in the net template by double clicking on the virtual place.

### 3.2.5 Net Instances and Synchronous Channels

Reference nets are object-oriented nets. Similar to objects in object-oriented programming languages, where objects are instantiations of classes, net instances are instantiations of net templates. Net templates define the type of nets just like classes define the type of objects. While the net instance has a marking that determines its status, the net template determines only the behavior that is common to all net instances of one type.

The paradigm of 'nets within nets' introduced by Valk (1995), allows tokens to be nets again. In reference nets, tokens can be anonymous, basic data types, Java objects or reference nets. The tokens representing the reference nets in nets are references to net instances. Any net instance can create new net instances similar to an object creating new objects. The new net instance is marked with the initial marking according to the marking the net template. Usually, the new net instance that is created should be bound to a variable of the correct type (`de.renew.simulator.NetInstance`) so that it can be transferred to an output place.

The notation of the creation inscription with the usage of the keyword `new`, to create a new instance, is displayed in figure 3.7. In this example the system net has an initial marking of three integer tokens. Thus the transition can fire three times creating three new net instances.

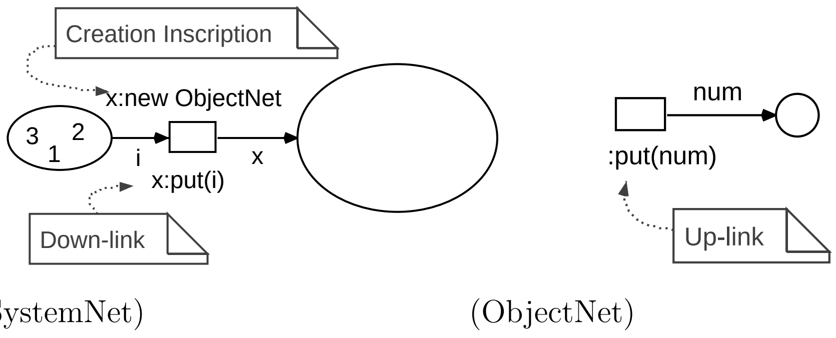


Figure 3.7: Example system net and object net.

The three new net instances are bound to the variable `x` and put into the output place. This is displayed in figure 3.8, in which the net templates (in the back of the image) and the net instances for both nets are displayed. There is one instance of the system net and three instances of the object net (front).[4]

The different net instances are each created during one firing of the transition of the system net that has the creation inscription. The tokens referring to the net instances are

This selects the appropriate original place.

[4]In *Renew* net instances can be identified by the names of the windows and the window background colors. Net templates have a white background color and net instances have an integer number attached to their window title bars that identifies the distinct instances. The identifying numbers are also attached to the tokens.

put into the output place. In the net instance *SystemNet[0]* in figure 3.8 these three tokens are displayed in the output place.

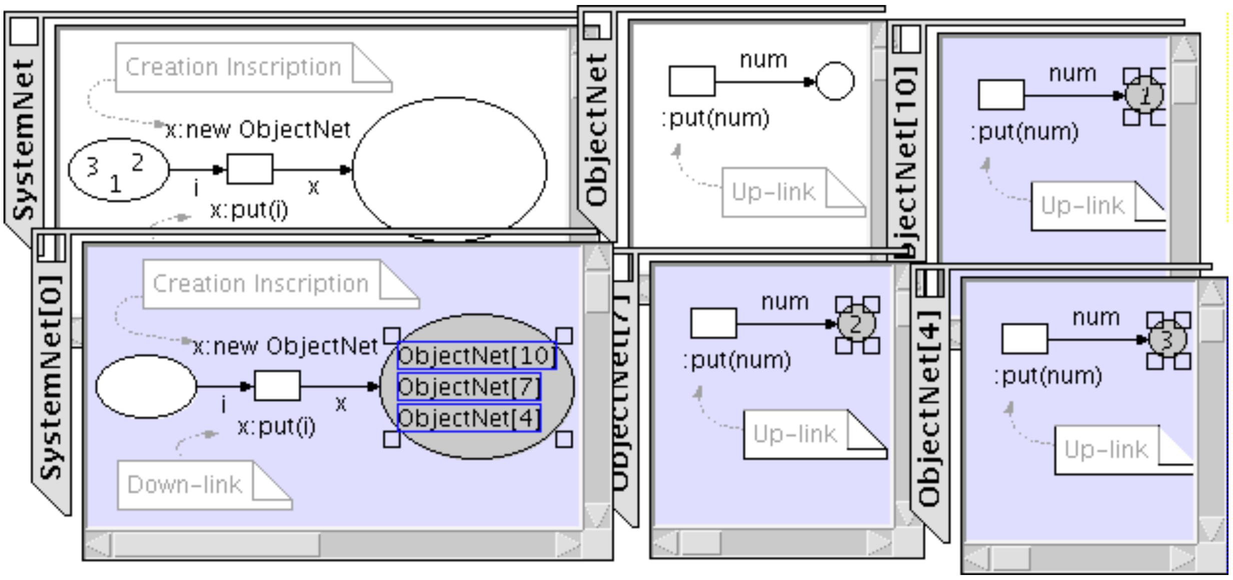


Figure 3.8: A screen shot of a system net and an object net; the templates and several net instances.

For the communication between net instances, synchronous channels are used. A synchronous channel consists of two (or more) inscribed transitions. There are two types of transition inscriptions: *down-links* and *up-links*. Two transitions that form a synchronous channel can only fire simultaneously and only if both transitions are activated. *Down-link* and *up-link* belong to a single net or to different nets. In both cases any object can be transferred from either transition to the other. If two different net instances are involved, it is thus possible to synchronize these two nets and to transfer objects in either direction through the synchronous channel. For this the system nets must hold the reference to the object nets as tokens.

The simple example of figures 3.7 and 3.8 does not only show the creation of net instances, but also the application of synchronous channels. A synchronous channel `put(.)` connects the two transitions of system net and object net. The system net holds the reference to the object net instance `x` that is created during the firing of the transitions. The *down-link* `x:put(i)` calls the *up-link* in the object net `:put(num)`. The integers are taken from the input place of the system net and bound to the variable `i` used as an argument in the channel inscription. Both transitions fire simultaneously and the two variables `i` and `num` are unified. Thus `num` is bound to the same integer as `i`, which finally is put into the output place of the object net. So the different numbers in the output places can distinguish the different net instances of the object net.

## 3.3 *Renew*

With *Renew* it is possible to draw and simulate Petri nets and reference nets. The simulation engine can execute a net that is loaded in the editor. For this the simulator creates an instance of the net. Any simulated net can instantiate other nets. Hence it is possible to produce many instances of different nets. The relationship between net template, also simply called net, and net instance can be compared to the relationship of class and object (see section 3.2).

### 3.3.1 Editor

Figure 3.9 shows the graphical user interface (GUI) of *Renew*, a simple Petri net in the back and a net instance.

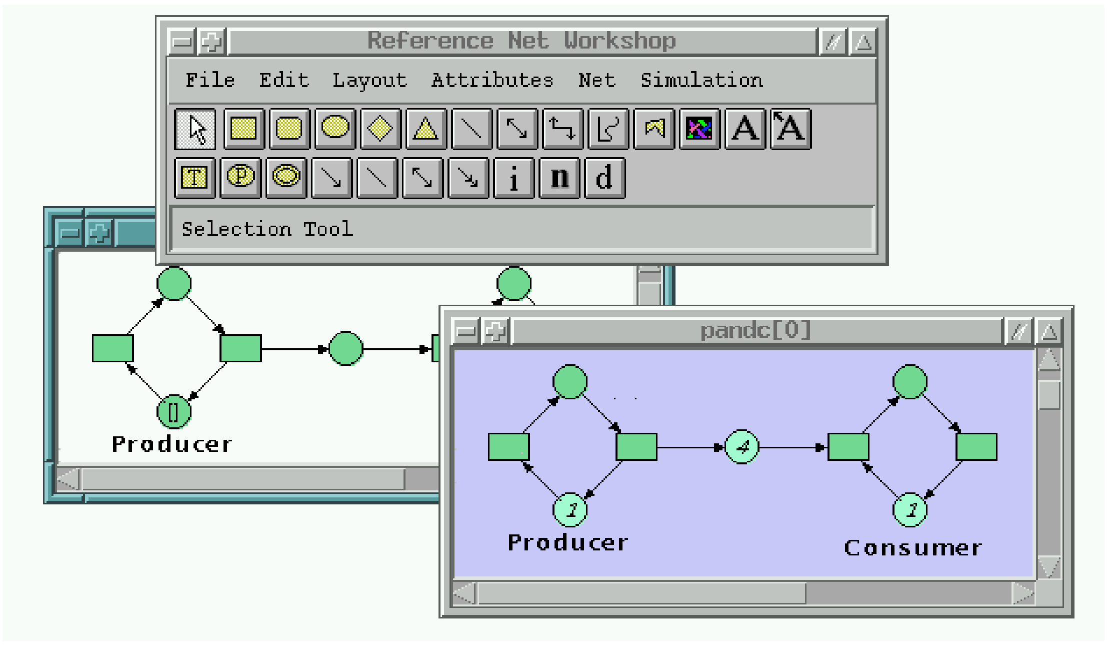


Figure 3.9: *Renew* GUI, Petri net and net instance (producer-consumer example).

The user interface consists of the menu bar, two palettes and a status line. The menu bar offers menus for general operations, attribute manipulations, layout adjustment and Petri net-specific operations. It also provides the possibility to control the simulation. Of the two palettes the first one consists of usual drawing tools while the second one holds the Petri net drawing tools. The latter palette provides the tools for creating transitions, places, virtual places, arcs, test arcs, reserve arcs, inscriptions, names and declarations. In addition to these tools, the editor reacts in a context-sensitive manner to facilitate the drawing of nets. One example is the dropping of arcs on the background that creates a new place if the arc starts at a transition and vice versa. Another example is the right click on inscribable elements that produces an inscription for this element with a context sensitive default value.

### 3.3.2 Simulator

Net templates hold the initial marking while net instances hold the current marking. In figure 3.9 the producer-consumer example has been started. In the net template (background) one of two black tokens (‘`[]`’) of the initial marking can be seen in the place labeled *Producer*. While the net instance by default only shows the number of tokens in a place it is also possible to show the contents of the places by clicking on the numbers (compare with figure 3.8).

*Renew* can operate in different modes. This can be achieved by exchanging the net compiler. Modes are available for P/T-nets, reference nets, feature structure nets (Wienberg 2001), timed Petri nets, boolean Petri nets and workflow nets (Jacob 2002) (Kummer et al. 2001b). In the Java mode, which is the basic mode for reference nets, it is possible to use any kind of Java objects as tokens. In fact, transition inscriptions can also hold method calls of objects.

### 3.3.3 Plug-ins

*Renew* is implemented in Java and, since version 1.7, extensible through a plug-in mechanism that is presented in (Schumacher 2003). The plug-in mechanism allows extending the functionality of *Renew* in a way that special requirements can be satisfied without the need to change the application system itself.

Plug-ins can be included in *Renew* by providing the classes; i.e. placing the plug-in's Java archive (*jar*-file) in the ‘plugins’ folder. The plug-in has to be present at starting time. A dynamic approach of being able to include plug-ins dynamically would be preferable because of flexibility reasons, but this is not supported yet.

Schumacher (2003, p. 34) gives a general definition of a plug-in, which is the basis for the notion of a *Renew* plug-in. In his view a system is composed of components and plug-ins are special components that extend the behavior of the system.

**Definition 3.1** ***Plug-in*** *Plug-ins are components that change the behavior of one or more other components in the system. This is done by using the provided interface of the components.*[5]

In this work two plug-ins for *Renew* are presented. The first is the net component plug-in, described in detail in (Cabac 2002), presented in section 5.2.2. This plug-in provides the functionality of including subnets – net components (see section 5.1) – into a Petri net

[5]Translated from the German original: “Plugins sind Komponenten, die das Verhalten einer oder mehrerer anderer Komponenten im System verändern. Dies geschieht über von diesen zur Verfügung gestellte Schnittstellen.”

by offering the net components as new drawing elements to the developer. The fact that net components are held in adaptable repositories adds to the flexibility of this approach.

The second plug-in provides the possibility of drawing agent interaction protocol diagrams in *Renew*. This plug-in is presented in (Cabac et al. 2003) and described in chapter 7. One objective of this work is to extend the diagram plug-in with code generating functionality, which is achieved by combining the two plug-ins. This is described in section 7.2.2.

## 3.4 Summary

Reference nets are high-level Petri nets that are extensions of P/T-nets. They offer the possibility to nest net instances as token references in hierarchies of nets, applying the 'nets within nets' paradigm. Tokens can be anonymous, basic data types, Java objects and instances of nets. New net instances can be produced during execution in a manner comparable to objects. The relationship of net template and net instance can be compared to that of class and object. Reference nets offer synchronous channels to provide communication and synchronization between different net instances (or within one net).

*Renew* is an editor and simulator for reference nets and other (Petri net) formalisms. It is implemented in Java and has an inscription language that is Java oriented. Nets can be drawn comfortably in *Renew* using the graphical user interface and loaded nets can be executed directly in *Renew*.

# Chapter 4

# Agents and *Mulan*

This chapter gives an introduction to agents, agent-oriented software engineering and the Petri net-based reference architecture of a multi-agent system *Mulan* (**Mul**ti-**A**gent **N**ets, (Köhler et al. 2001)). First, the basic concepts of software agents, agent-oriented software development and multi-agent systems are introduced. After this, *Mulan* is presented. It is implemented with reference nets and runs within the Petri net editor and simulator *Renew* (Reference Net Workshop, (Kummer et al. 2001a) see the previous chapter). *Mulan* is a reference architecture for a multi-agent system that complies with the FIPA specification for multi-agent systems. CAPA (Concurrent Agent Platform Architecture, (Duvigneau et al. 2003)), an extension to *Mulan*, provides FIPA-compliant communication and agent management. More detailed information about agents and multi-agent system are given by Rölke (2003).

## 4.1 Software Agents

While the object-oriented paradigm is still state of the art, some limitations of the object-oriented view lead developers and researchers towards new technologies. One – still very new – approach is the agent-oriented view. It can be understood as a natural extension to object-orientation.

### 4.1.1 Object-Orientation

In object-orientation, an object is an encapsulated entity that has a clearly defined interface. It represents a concrete or abstract object of the real world or the modeled world. The interface of the object defines methods that can be accessed by other objects. Internal representation and data structures are usually not revealed to the outside world to hide the implementation details. The static structure of objects is defined in classes that can be seen as templates for a type of objects. This is why classes determine the type of objects. In the object-oriented view, a relation is defined on the objects, which is called generalization or inheritance. Objects are in some way specializations of other objects and thus categorized. In this relation objects form a hierarchy of inheritance. For example, in Java the most general object is `Object` as the root of the hierarchy.

There are numerous reasons why the object-oriented approach had so much success in recent years. For example, due to inheritance, it is possible to build new objects by extending existing ones. This saves much development time and effort. All kinds of functionality are implemented in frameworks and toolkits that are used to build application software on their basis, without reinventing or reimplementing the same functionality over again. Re-use driven development is the key to efficient programming that has become possible because it is based on object-oriented programming languages like *C++* or Java. However, some limitations and shortcomings exist that demand a more sophisticated view. Designing software using a metaphor of active entities is one of them.

### 4.1.2 Concept Formation

Concept formation is the building of categories of terms or ideas in language acquisition. All terms or concepts are ordered in a hierarchy that is already known to the individual. Actually, new terms are acquired or concepts are formed by generalization and discrimination (Edelmann 1994). This means that new concepts are learned by augmenting the hierarchy of concepts at the appropriate position – under the more general super-concept. Furthermore, to distinguish this new concept from other concepts that are also generalized by the same super-concept, the differences between those have to be stated. Pettijohn (1998) describes concept formation like this.

> Psychologists use the term concept formation, or concept learning, to refer to the development of the ability to respond to common features of categories of objects or events. Concepts are mental categories for objects, events, or ideas

> that have a common set of features. Concepts allow us to classify objects and events. In learning a concept, you must focus on the relevant features and ignore those that are irrelevant ...

This description is easily adaptable to the object-oriented view. Oestereich (2001, p. 45) writes about object-orientation:

> Classes can constitute specializations of other classes, this means classes can be arranged hierarchically. They receive (inherit) the features of the superclasses and can - according to the requirements - specialize (override) them. However, classes cannot eliminate inherited features.[1]

Comparing these two statements, it can be said that object-orientation is closely related to concept formation. The hierarchy of concepts is very similar to the class hierarchy that is built during the conceptualization of object-oriented development. Classes can be regarded as concepts of objects. This similarity holds, probably, a big part of the expressive power of object-orientation. The focus of concept formation lies on the static structure of the relations between terms. This is also true for the object-oriented view. The inheritance hierarchy describes the static relations of generalization or specialization.

In learning theories another form of acquiring knowledge is known. This approach to learning focuses on *activity*. It is described as success-driven acting (Edelmann 1994, pp. 295-320). The concepts that are thus formed are not any more just passive entities but categories of behaviors. The concepts are no longer those of static entities but activities. Edelmann states that these concepts of activities can again be categorized in more general behavioral patterns that are structured in a hierarchical-sequential order.

Similar statements can be made for software development. Modeled entities can be more than just **re**acting, passive objects. Hence, the focus of agent-orientation lies on **pro**active entities. This orientation towards active entities leads to a view on software development that focuses on the action of entities or the action as entity. Agents are in this view active, acting entities.[2] It also clearly distinguishes agents from objects that are passive.

### 4.1.3 Informal Approach to Agents

As described in this chapter, agent-oriented software engineering is influenced by the areas of software engineering and artificial intelligence. There are also influences of areas like distributed systems and social science. All together this leads to a view on systems that emerge from (self) organizing structures.

An agent can be seen as an even more abstract version of an encapsulated entity than an object. While objects export methods as interface to offer functionality to other objects,

[1]Translated.

[2]This can also be seen in the etymology of the word *agent* that comes from *agere* (latin): to drive, do or act (Costello 1996).

agents only communicate via messages. These messages are basically strings that have a certain form, i.e. a language. Instead of being used by another object the agent receives messages and 'decides' what to do with or in response to this message.

The agent, in opposition to the object, can decide to react and may reply to that message or just ignore it. This introduces a notion of autonomy. Another ability, besides the ability to decide, is connected to making decisions. To be able to decide the agent has to have some criteria on which it can base its decisions. From this follows that an agent has to possess some knowledge. This knowledge includes facts and beliefs about itself, its environment – the world it is situated in – and also other agents. In the agent-oriented view this knowledge can only be partial.

Agents can be physical or virtual. The difference is that physical agents have representations in the real world, e.g. robots. The physical agent can thus interact directly with its environment by using its effectors and sensors. Virtual or software agents have no manifestation in the real world and are often compared to software objects or components.[3]

Figure 4.1 visualizes the difference between the physical and the virtual agent in their environments. In addition to its communication channels, the physical agent also has sensors that serve cognition and effectors that can manipulate the environment. The only connection of software agents with the environment, and thus with other agents, lies in their communication channels. Ferber (1999) distinguished between the purely situated and the purely communicating (software) agents. On the following pages, we always refer to software agents.

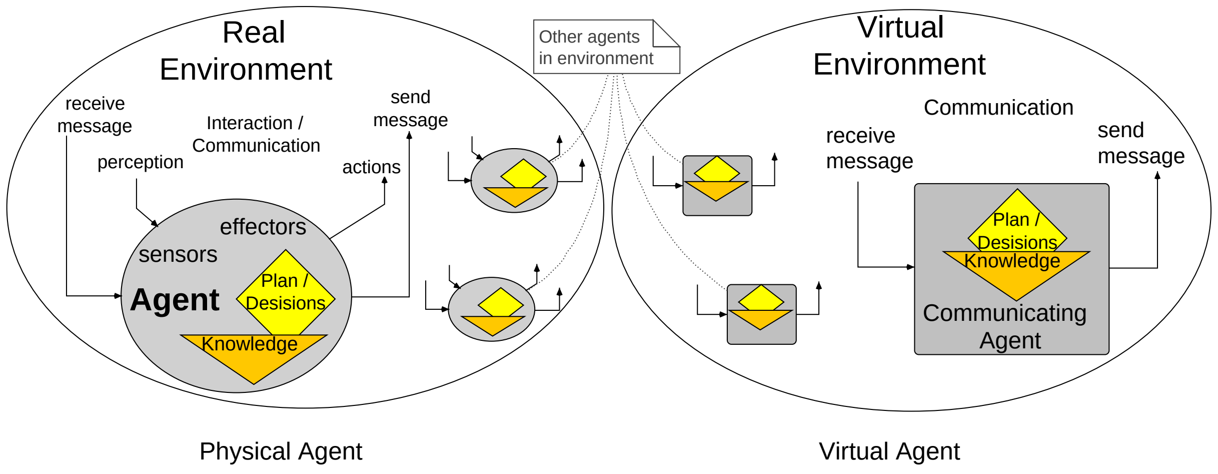


Figure 4.1: Abstract model of an agent and its environment.

[3]The possibility of simulating a physical agent through a virtual one exists and so the difference is of theoretical nature.

### 4.1.4 Agent Definitions

Agents are, so far, independent software components that can decide their actions on the basis of their knowledge, and they communicate with other agents. However, to be able to achieve some results they also have to have some goal or interest. Since they 'live' in an environment that is populated with other agents, naturally, the agents' goals coincide or differ from each other. Therefore the agents have to coordinate, cooperate or compete with other agents. This means they have to act within a social system.

In a social system, in which agents exist, adaptability seems to be essential. They have to adapt to conventions, protocols and exceptional events. This means that agents have to possess a certain kind of intelligence (Russell and Norvig 1995). So while objects just react to their input (always in a predictable way), agents can act and even decide on their actions.

To sum this up: A software agent is an adaptive, intelligent and independent software component that has a certain goal or interest, has some knowledge about itself and its surroundings and can communicate with other agents.

Here are some definitions of various authors. Surely, the aspects that cover their topic of interest bias their view. So the researchers of artificial intelligence focus on intelligence and social behavior of the agents while the software engineers discuss agents as an extension to object-orientation. Ferber (1999) provides a general definition, can be broadly applied and lists the characteristics of agents.

**Definition 4.1 (Agent)** *"An agent is a physical or virtual entity*

*(a) which is capable of acting in an environment,*
*(b) which can communicate with other agents,*
*(c) which is driven by a set of tendencies . . . ,*
*(d) which possesses resources of its own,*
*(e) which is capable of perceiving its environment . . . ,*
*(f) which has only partial representation of its environment . . . ,*
*(g) which possesses skills and can offer services,*
*(h) which may be able to reproduce itself,*
*(i) whose behavior tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representations and the communications it receives." (Ferber 1999, p. 9).*

Bergenti, Shehory, and Sturm (2003) stresse the differences of the two views – artificial intelligence and software engineering – and offer two different definitions.

**Definition 4.2 (Agent)** *"An Agent must be proactive, intelligent, and it must conversate* [sic] *instead of doing client-server computing. . . "*

*"An agent is a software component with internal (either reactive or proactive) threads of execution, . . . that can be engaged in complex and stateful interactions* [sic] *protocols." (Bergenti et al. 2003, p. 127, slide 14).*

Russell and Norvig (1995) offer a very short definition of agents, which can be viewed as a first notion.

**Definition 4.3 (Agent)** *"An agent is just something that perceives and acts." (Russell and Norvig 1995, p. 7).*

An often quoted definition of agents is given by Jennings and Wooldridge (1998). It is a pragmatic view on agents in the agent-oriented software development and varies only slightly from the one given by Ferber (definition 4.1).

**Definition 4.4 (Agent)** *"An Agent is an encapsulated computer system, situated in some environment, and capable of flexible autonomous action in that environment in order to meet its design objectives." (Jennings and Wooldridge 1998, p. 5).*

These definitions, as already mentioned above, are biased by the author' perspective. Furthermore, many of the words used in these definitions require further explanation. But the main point here is to present the reader a general notion of the concepts and not a full discussion of all terms. The latter is not intended within this work and would also exceed its extent. Since the authors in their publications go into further detail, we leave the definitions as they are without further examination.

Generally, it can be said that the definitions vary but do not contradict each other. They merely show different viewpoints. It should just be stated that the agent concept in software engineering is more general than that of the artificial intelligence. In software engineering, objects can be considered very simple reactive agents, whereas artificial intelligence demands from an agent methods that are intelligent.

In this work we will assume the software engineering view, which is related to the definitions 4.1 by Ferber and 4.4 by Jennings and Wooldridge.

## 4.2 Multi-Agent Systems

Agents do not only act, but also interact with each other. On the one hand, they communicate with other agents by receiving or giving information, they can send orders, requests or demands to other agents. On the other hand, they have to follow their own interests or goals that can be in opposition to the goals of other agents. Nevertheless, the goals of two agents can also coincide, giving the two agents a chance for cooperation. It is also possible that agents who have different goals can support each other for the benefit of both agents. Precisely this fact can possibly enable the agents to reach their goals in cooperation, when they would have failed on their own. Thus agents have to have mechanisms of coordination, competition and cooperation.

### 4.2.1 Informal Approach to Multi-Agent Systems

Social behavior is somehow necessary to achieve a consensus between different parties. This means that agents not only have to coordinate their actions but also negotiate with other agents about their actions. It leads the agents into a somehow organized environment. Sometimes such an organization of agents is also called group, society or population.

The term multi-agent system is strongly connected to the term agent as it could be seen in Ferber's definition 4.1, in which he uses the term multi-agent system to define the term agent. Just as with the term agent the multi-agent system is used in many different ways according to the point of view.

Reese (2003) describes three views on multi-agent systems, which are all valid but differently motivated. The first point of view focuses on the discrimination of the agent-oriented view from other technologies like object-orientation or distributed systems. The second point of view focuses on the infrastructure for the agents. Here a multi-agent system is the technical basis on which the implementation is build. This could be regarded as the middleware and the motivation for this lies in software engineering (SWE). The third point of view focuses on the aspect that a network of agents together with the infrastructure forms a system of loosely connected agents that could possibly interact. This satisfies the notion of agents in artificial intelligence (AI). Reese pleads for the following distinction of these views.

- An **agent network** connects agent platforms to a system on which distributed agents can form multi-agent systems (AI).
- An **agent platform** is the technical realization of infrastructure for the agents (SWE).
- A **multi-agent system** is oriented on the application. A coordinating system of agents with a common purpose.

Figure 4.2 shows this distinction again. Agents reside on agent platforms while some agents across platforms can form a multi-agent system. The agent network consists of three platforms. It is also possible to build other multi-agent systems within this network by connecting other agents. So multiple multi-agent systems can coexist on the same agent network.

### 4.2.2 Definitions of Multi-Agent Systems

In this section as well as in the sections about definitions for the term *agent*, the intention is to show the approaches to multi-agent system. However, a full discussion of the term is not provided. A good introduction to multi-agent system is offered by Ferber (1999). It is a broad definition that covers all aspects of a multi-agent system and tries to be also very general.

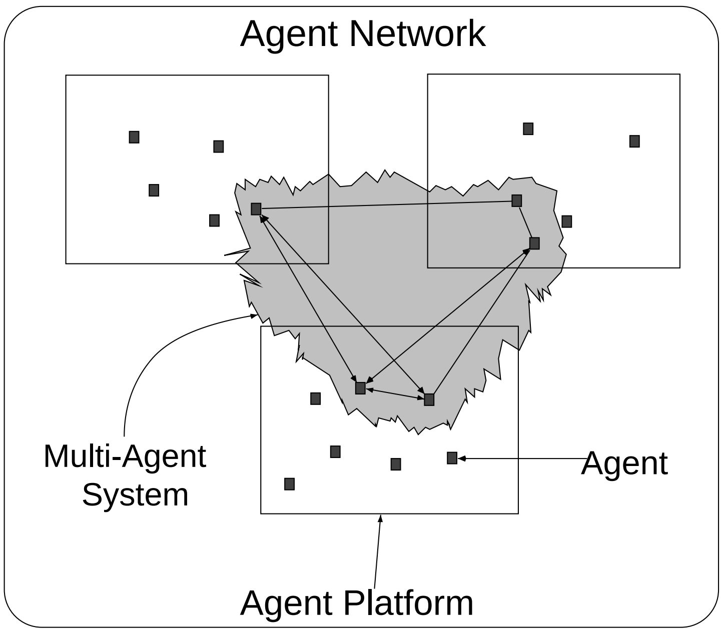


Figure 4.2: Agent network versus multi-agent system.

**Definition 4.5 (Multi-Agent System)** *"The term 'multi-agent system' (or MAS) is applied to a system comprising the following elements:*

*(1) An environment, E, that is, a space which generally has a volume.*

*(2) A set of objects, O. These objects are situated, that is to say, it is possible at a given moment to associate any object with a position in E. These objects are passive, that is, they can be perceived, created, destroyed and modified by the agents.*

*(3) An assembly of agents, A, which are specific objects ($A \subseteq O$), representing the active entities of the system.*

*(4) An assembly of relations, R, which link objects (and thus agents) to each other.*

*(5) An assembly of operations, O, making it possible for the agents of A to perceive, produce, consume, transform and manipulate objects from O.*

*(6) Operators with the task of representing the application of these operations and the reaction of the world to this attempt at modification, which we shall call the laws of the universe (...)" (Ferber 1999, p. 11).*

In contrast to this formal approach, Bergenti, Shehory, and Sturm (2003) offer two views on multi-agent systems that are influenced by their different perspectives. Again these views are the one of artificial intelligence and that of software engineering.

**Definition 4.6 (Multi-Agent System)** *"A multiagent system is a society of individual (AI software agents) that interact by exchanging knowledge and by negotiating with each other to achieve either their own interest or some global goal..."*

*"A multiagent system is a software systems* [sic] *made up of multiple independent and encapsulated loci of control (i.e. the agents) interacting with each other in the context of a specific application viewpoint." (Bergenti et al. 2003, p. 127, slide 15).*

The different definitions show that different positions exist about the notion of a multi-agent system. The artificial intelligence focuses on the social activities of agents and intelligent solutions, while software engineering focuses on the technical solutions. Ferber manages to give a broadly applicable definition.

We will focus on two aspects regarding multi-agent systems. In the next section the agent platform *Mulan* is described as a realization of an operational Petri net-based multi-agent system. After that the following chapters are concerned with the implementation of multi-agent applications based on *Mulan*. The focus is laid on the agents' interactions, which are modeled with Petri nets and AUML diagrams. Since this is all connected to the realization of the multi-agent application, the point of view is that of software engineering.

## 4.3 *Mulan*

*Mulan* (**Mul**ti-**A**gent **N**ets, Köhler et al. 2001) is a reference architecture of a multi-agent system that is based on *Renew*. It is implemented with Petri nets as a system of reference nets and it complies with the open specifications of the Foundation for Intelligent Physical Agents (FIPA 2003) for multi-agent systems. A more detailed description of *Mulan* is given by Rölke (2003, chapter 5).

### 4.3.1 *Mulan* Architecture

*Mulan* is a multi-agent system that is modeled and implemented in reference nets. One advantage of Petri nets is the fact that the coarse model of a system can be refined to a detailed model that is executable. This can be called implementing through refinement. The great advantage of this approach is that the gap between modeling and implementation is eliminated. Nevertheless, from now on the coarse model is referred to as just the model of the system whereas the refined model is called implementation. The system consists of numerous nets. In the model (figure 4.3), the first net is the infrastructure that describes an abstract communication system consisting of locations and communication paths.

The locations can be seen as real world locations, i.e. they can be visualized as different computers. The locations are modeled as places on which the platforms reside as reference nets. These nets provide the communication channels for the agents for internal and external communication. Communication is internal when the two communicating parties are resident on the same platform and external if not. In the case of external communication, the platform provides the message transport service (MTS) that transfers the messages across locations. In addition to this, platforms offer the possibility to agents to enter or leave the platform.

*Mulan* agents reside on a platform. One place of a platform holds all agents that are present on this particular platform. *Mulan* agents are reference nets again. These agents

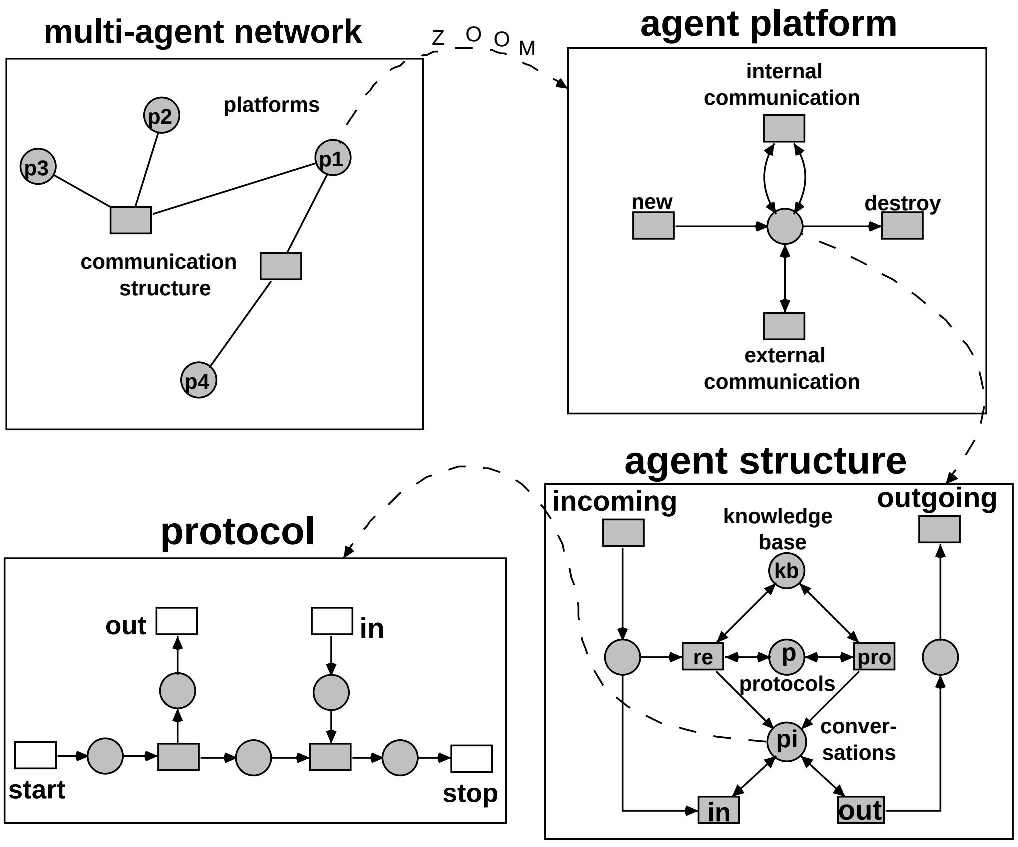


Figure 4.3: The structure of *Mulan*

have incoming and outgoing communication channels that enable the agent to communicate over the platform with other agents and possibly to services offered by the platform. In the same manner as the platform holds the agent nets in one place, the agents hold their protocol nets for their conversations in one place.

Similar to the platform providing communication channels to the agents, the *Mulan* agents provide channels for the initialization, destruction and communication between agent and protocol.

All these channels mentioned above are modeled and implemented as synchronous channels. Thus by synchronizing the different nets over platforms or over different layers of the nested nets, the communication between these nets is provided. This is illustrated in figure 4.3. The figure shows the net within a net hierarchy of the system. Agents are nets that exist on platforms. The platforms are also nets. There can be many platforms and the agents can communicate with each other within and across platforms. Protocols are nets within the agents and control their behavior. The figure is taken from (Köhler et al. 2001).

## 4.3.2 *Mulan* Agents

Agents are complex entities. They have to provide functionality for communication, interaction, knowledge acquisition, storage and retrieval, decision-making and conversation handling. All this is realized in *Mulan* agents by a modular, hierarchical design. All components of the *Mulan* agents are realized as reference nets. This means that all modules are actually nets and the modularity is provided by the 'nets within nets' paradigm (see section 3.2).

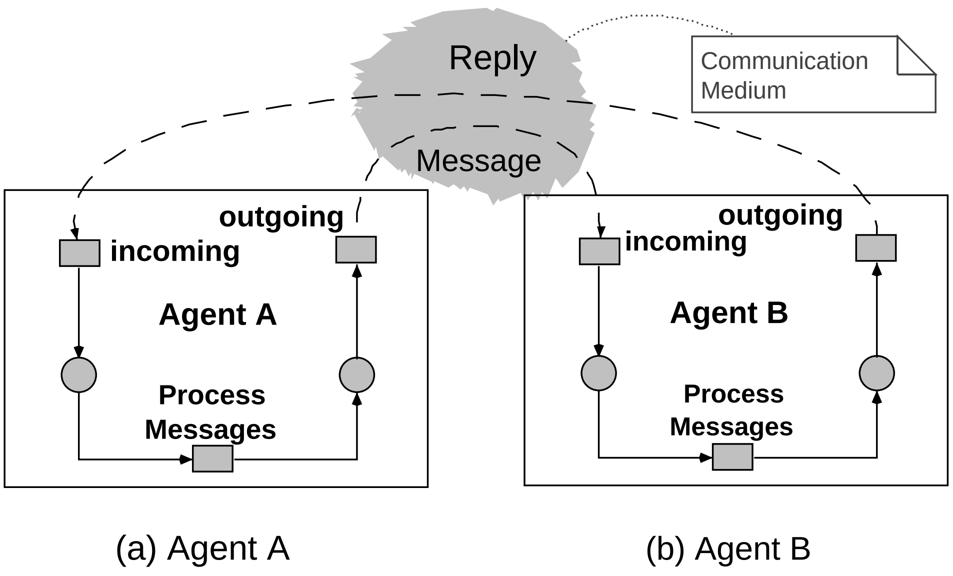
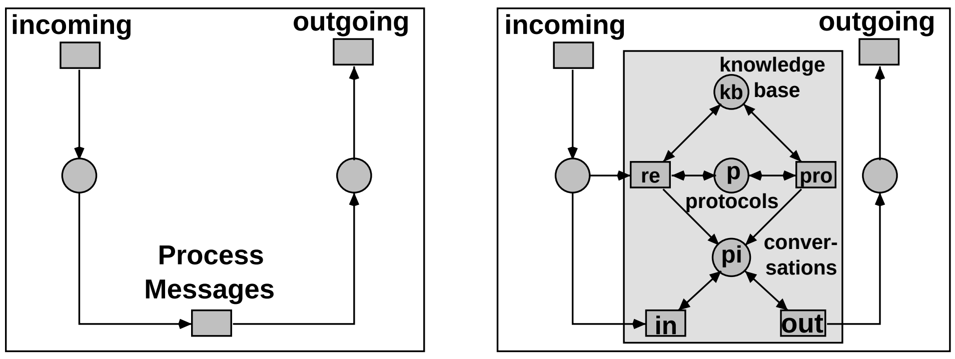


Figure 4.4: A model of two Agents communicating with each other.

From an outside viewpoint the agents can be seen as black boxes that communicate with each other. They have two channels to send and receive messages. In this very simple model, as shown in figure 4.4, the internal processes of the agents are hidden. Of course, the messages have to be transported by a medium that is provided by the *Mulan* Platform. To achieve a more detailed model the Petri nets have to be refined. This leads to the model of an agent, as shown in the architecture overview in figure 4.3.

Figure 4.5 shows the refinement of the model as a step to reveal more details of the agent.

Besides the main part of the *Mulan* agent, which from now on is simply referred to as the agent, the complete agent consists also of the knowledge base and the protocol factory. Its behavior is defined in the *Mulan* protocols. This is already indicated in figure 4.3 (agent structure) where the knowledge base `kb` and the protocol factory `p` are inserted as tokens on similarly labeled places. Active *Mulan* protocols `pi` are held in the conversation place. Protocols can be initialized, i.e. conversations can be started proactively as well as reactively and messages leave and enter the agent over the synchronous channels *incoming* and *outgoing*.



(a) Coarse Model (b) Refined Model

Figure 4.5: Abstract model of the *Mulan* agent (b as a refinement of a).

All the elements shown in this first refinement of the model are also present in the second refinement in figure 4.6 and in the implementation in figure 4.7. Actually, the implementation is a refinement of the model that includes some additions. These additions (white net elements) are the initialization of the agent with the generation of the initial knowledge and some semaphores to check whether the agent is idle or busy.

The communication between the different nets is realized through synchronous channels marked in the net as *access* for the access to the knowledge base, *proactive* and *reactive* for initializing the *Mulan* protocols and finally *Start*, *Stop*, *In* and *Out* for the interaction of the agent with the protocols. Regarding the communication channels, only the distinction between new and running conversations have to be made, deciding the conflict of the two transitions *reactive* and *In*. Furthermore, the name of the agent is automatically added to a message before sending.

Software agents communicate by exchanging messages. In fact, this is their only way of interaction with their environment or other agents. *Mulan* agents accomplish this by synchronization with the platform net over synchronous channels (see section 3.2.5). These communication channels are labeled *send* and *receive*.

Agents have to store their knowledge or belief in order to be able to make decisions and to reach their goals, and they also have to act in certain adaptable ways. In *Mulan*, the first is realized through the implementation of the knowledge base as a reference net. The protocol factory and the protocols accomplish the latter. The following section describes the knowledge base and the protocol factory. After that, a detailed description of the *Mulan* protocols is given in section 4.3.4.

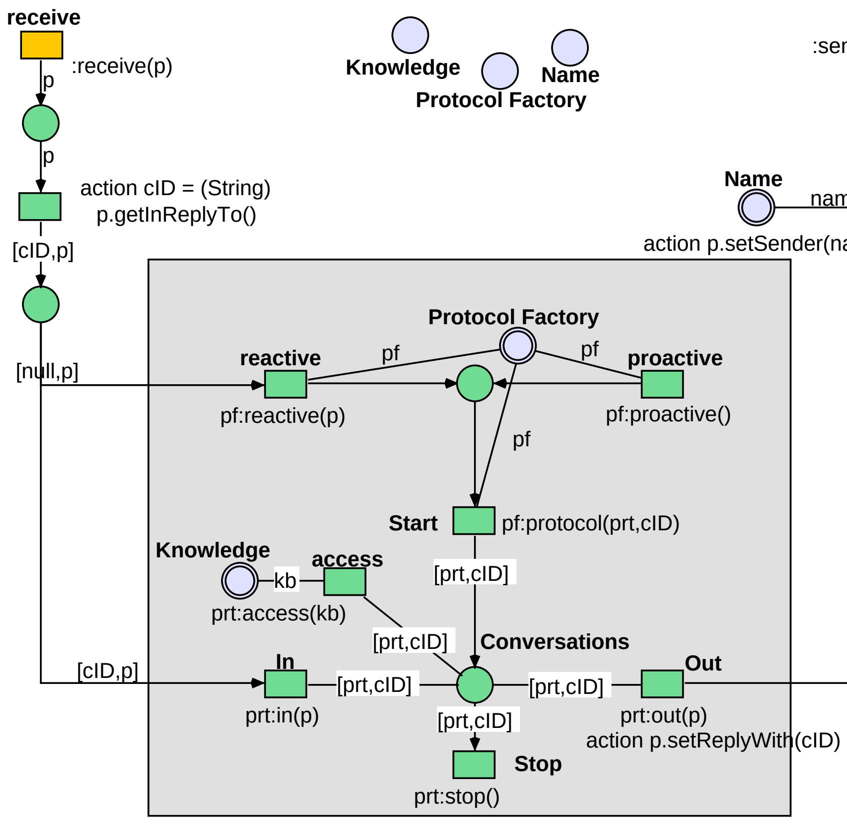


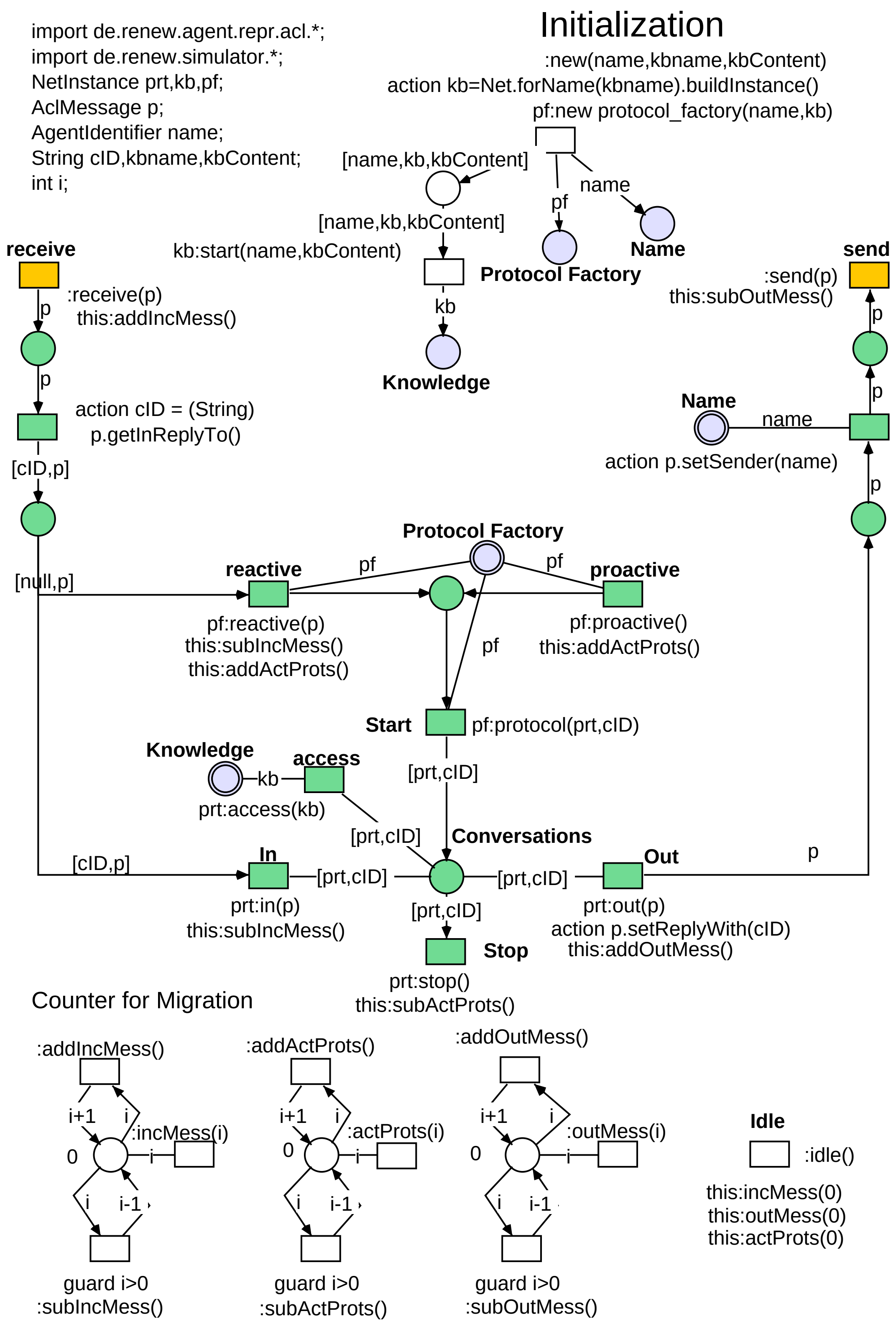
Figure 4.6: The *Mulan* agent, striped of some administrative elements.

## 4.3.3 Knowledge Base and Protocol Factory

The knowledge base is an important part of a *Mulan* agent. Nevertheless, it is quite a simple component. Realized as a reference net, it provides the possibility of storing, retrieving and modifying information as key-value tuples. Basic functionality is provided for the creation of new entities and checking for existence of a key. Some additional functionality regarding protocols is provided for reasons of convenience.

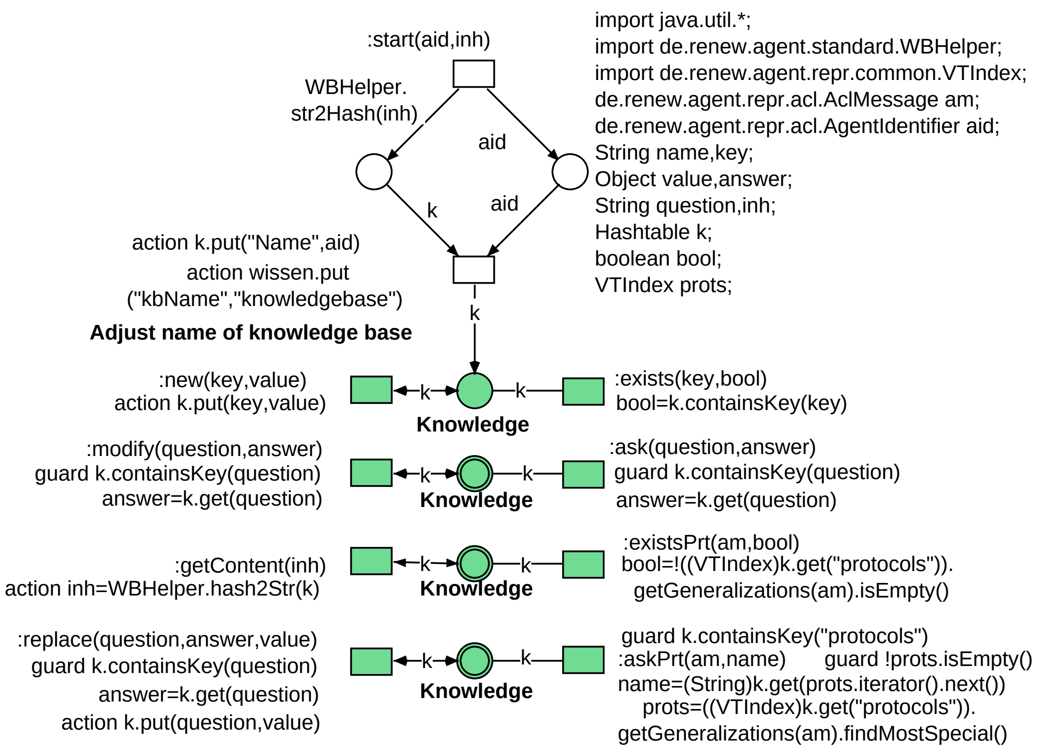
The functionality is signified by the names of the synchronous channels. For instance: '`:exists(.,.)`' checks whether a key exists in the knowledge base. The knowledge `k` is accessed through a a test arc because other actions can be performed concurrently. In contrast, the creation of a new entry (`:new(.,.)`) should be be done synchronized so the access is done by using a reserve arc.

Figure 4.8 shows a simplified version of the knowledge base displaying all important and basic parts. Except for the basic functionality to handle information, the initialization is also displayed (white net elements). Before the knowledge base can be used, the initial knowledge is read from a file and put into the knowledge base. The knowledge itself is described as key-value tuples and stored in a hash table of which a reference is held on



```
import de.renew.agent.repr.acl.*;
import de.renew.simulator.*;
NetInstance prt,kb,pf;
AclMessage p;
AgentIdentifier name;
String cID,kbname,kbContent;
int i;
```

Figure 4.7: The *Mulan* agent.


:start(aid,inh)
WBHelper.
str2Hash(inh)
aid
aid
k
action k.put("Name",aid)
action wissen.put
("kbName","knowledgebase")
Adjust name of knowledge base
k
:new(key,value)
action k.put(key,value)
Knowledge
:exists(key,bool)
bool=k.containsKey(key)
:modify(question,answer)
guard k.containsKey(question)
answer=k.get(question)
Knowledge
:ask(question,answer)
guard k.containsKey(question)
answer=k.get(question)
:getContent(inh)
action inh=WBHelper.hash2Str(k)
Knowledge
:existsPrt(am,bool)
bool=!((VTIndex)k.get("protocols")).
getGeneralizations(am).isEmpty()
:replace(question,answer,value)
guard k.containsKey(question)
answer=k.get(question)
action k.put(question,value)
Knowledge
guard k.containsKey("protocols")
:askPrt(am,name) guard !prots.isEmpty()
name=(String)k.get(prots.iterator().next())
prots=((VTIndex)k.get("protocols")).
getGeneralizations(am).findMostSpecial()

```
import java.util.*;
import de.renew.agent.standard.WBHelper;
import de.renew.agent.repr.common.VTIndex;
de.renew.agent.repr.acl.AclMessage am;
de.renew.agent.repr.acl.AgentIdentifier aid;
String name,key;
Object value,answer;
String question,inh;
Hashtable k;
boolean bool;
VTIndex prots;
```

Figure 4.8: The knowledge base of the *Mulan* agent.

the central place *Knowledge*. Three virtual places exist of this place for the reason of convenience and net layout.

The protocol factory is responsible for the initialization and the start of the *Mulan* protocols. This can be done reactively or proactively. If an agent receives a message for which no conversation is already active, an appropriate response has to be found. By consulting the knowledge base, a protocol is chosen that can handle the processing of the message as well as the possible following conversation. Then the protocol factory instantiates and starts the protocol. Furthermore, the protocol factory produces a conversation identifier `cID` (also in figure 4.7) to map the conversation to the protocol.

Finally, the *Mulan* protocol net instance is put into the conversations place. Alternatively, the protocol factory can proactively start a *Mulan* protocol in order to initiate a conversation or to initiate an internal process. Figure 4.9 shows the protocol factory with the reactive part at the left side, the proactive part on the right side, the initialization of the protocol factory and identifier counter in the middle (white net elements) and the protocol initialization, start and release at the bottom. With the last transition a tuple of the protocol and the conversation identifier is put into the conversations place of the agent (figure 4.7).

The actual activities of the agents are handled by the *Mulan* protocols. These protocols

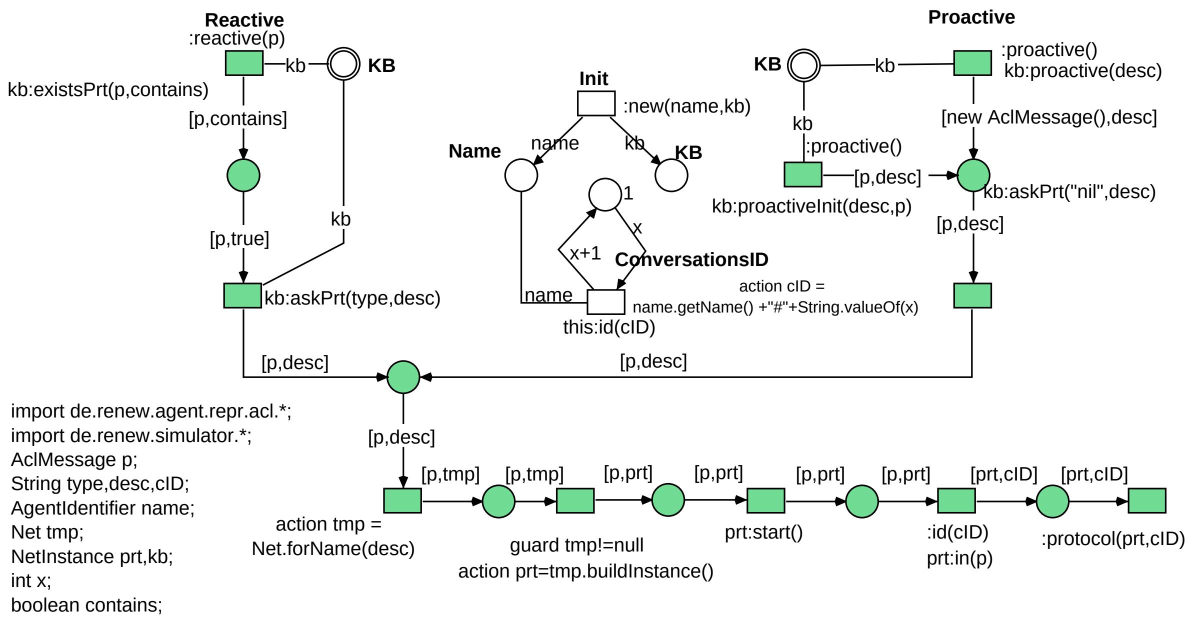


Figure 4.9: The protocol factory of the *Mulan* agent.

control both, the part of the conversations for their agents and the internal activities.

### 4.3.4 *Mulan* Protocol

One conversation consists of a series of related messages send back and forth between a set of agents. In a simple setting including only two agents, two *Mulan* protocols, one for each agent, are sufficient to determine and control the conversations. In a more complex setting the control over the conversation is thus distributed over the agents. Each agent holds only the information for its part of the conversation in the *Mulan* protocol, and the conversation describes the agents' interactions whereas one or more *Mulan* protocols describe the behavior of one agent during the conversation.[4]

A protocol defines a certain behavior during an interaction. By following the protocol it is ensured that the interaction between the interacting parties is possible. In the agent-oriented view, a protocol determines the communicational behavior of agents.

Protocols can be compared to processes or workflows. The protocol defines the actions or activities of the agent at a certain time. It defines sequences, concurrency or decisions. The goal is that the communicating agent can follow the whole conversation successfully until the end is reached. *Mulan* protocols are – just like *Mulan* agents – Petri nets. An agent can use numerous protocols and it can instantiate multiple instances of various protocols

[4]This terminology differs slightly from the FIPA terminology, where the conversation is called protocol. They are described in interaction protocols while the *Mulan* protocols as described here have no equivalent and are parts of the FIPA interaction protocols.

at the same time. Petri nets are an appropriate method to model processes or workflows (van der Aalst and ter Hofstede 2002) because they show the dynamic behavior directly.

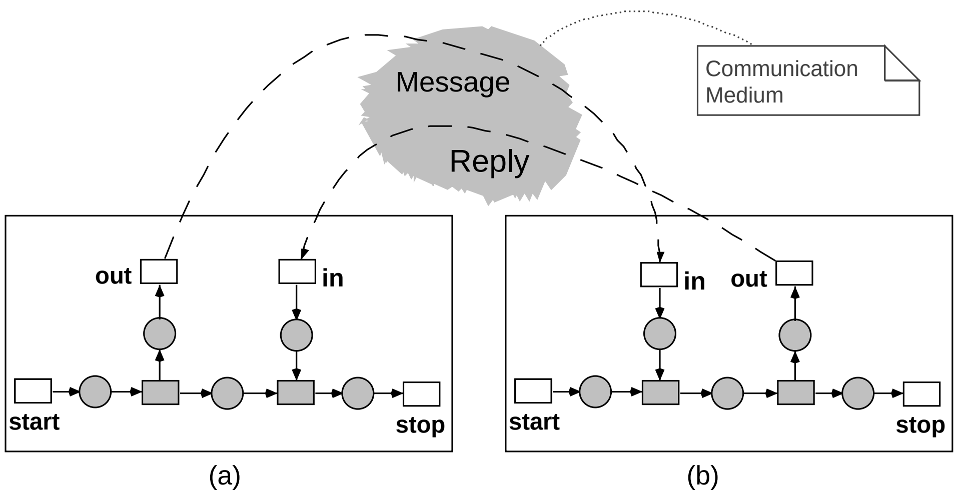


Figure 4.10: Two protocols exchanging messages. Protocol (a) initiates the conversation and protocol (b) replies. Together these two protocols form a conversation.

The abstract Petri net model, displayed in figure 4.10 for *Mulan* shows a simple scenario that is refined later just like the model for the agent. Two *Mulan* agents that communicate have to instantiate a protocol net each (at least one). The interaction or communication takes place between the two agents but the messages are passed from one protocol to the other. The transportation of the messages, of course, has to be accomplished via a medium. This medium is provided by the agents. The agent itself is using the communication medium of the platform.

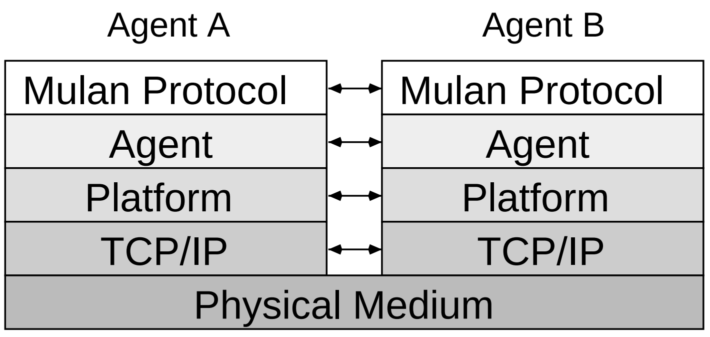


Figure 4.11: Layers of message transportation in *Mulan* for external communication across platforms.

This leads to an architecture comparable to a layered protocol stack like the TCP/IP-stack shown in figure 4.11. The word protocol is used in a different way in this context, so we use the term *medium* for a connection within one layer that can be used by a higher layer. In fact, when the two agents reside on one platform, the two lowest levels, the TCP/IP layer and the physical layer, are not used, instead there exists only one platform that handles the communication internally. In this view, we can now talk of vertical and horizontal communication. Vertical communication takes place between (vertical) layers, e.g. between the agent and the protocols or between the agent and the platform on which it resides. Horizontal communication is the communication of layers of the same level, e.g. the communication between two agents.

### 4.3.5 Description of *Mulan* Protocols

*Mulan* protocols are Petri nets, more specifically, they are reference nets. These protocols possess a clear, control flow-oriented character. They have a starting transition (inscription: `:start()`) and one ore more ending transitions (inscription: `:stop()`). These transitions define the life cycle of the protocol instances. There are input and output transitions (inscription: `:in(p)`, `:out(p)` which allow the protocol to pass messages from and to agents. These transitions contain *up-links* that constitute, together with the corresponding *down-links* in the agent nets, the synchronous channels that are the means of communication between reference net instances.

The messages received by an agent through the communication system are passed on to the responsible *Mulan* protocol through the synchronous channels. It is further processed by the protocol and its content can be extracted. Another message, e.g. a reply, can be formulated by the protocol and passed back to the agent, which sends it through the communication system to the new receiver.

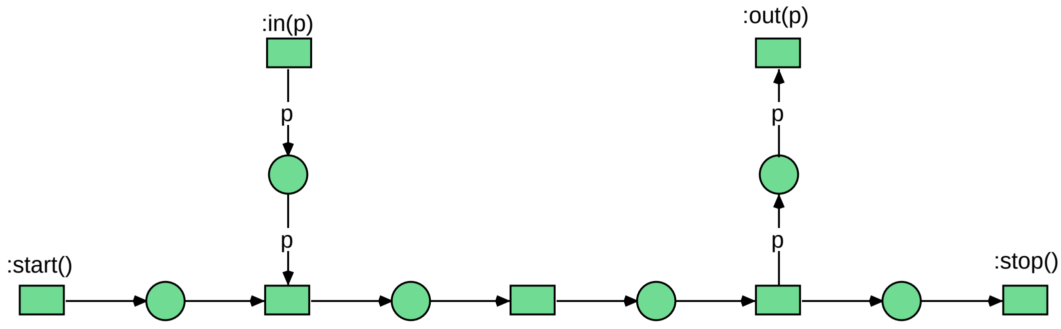


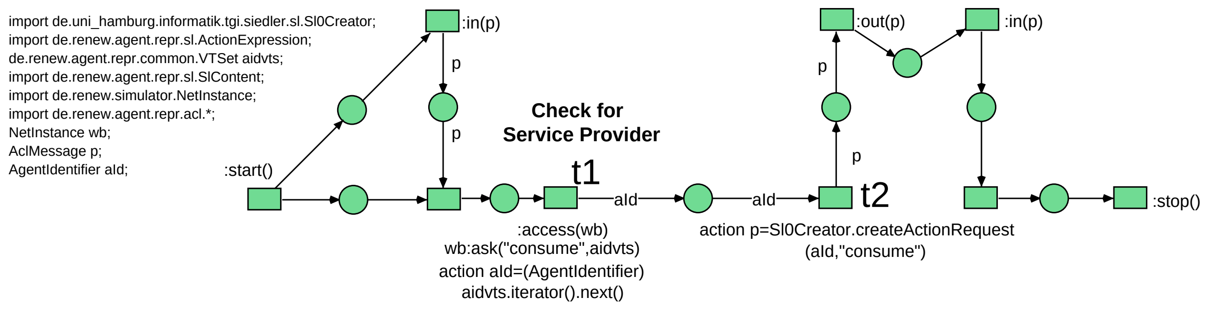
Figure 4.12: A scheme of a protocol.

The protocol defines, through its control flow, the behavior of the agent in the conversation for which it is ‘responsible’. Therefore it decides how a message is processed, which message is sent to which agent, and which internal actions are performed. However, the agents receive messages from or send them to other agents. The agents also decide with the knowledge from their knowledge base which protocols are being started to respond to a certain message, i.e. the agents choose the protocol that leads the conversation. Once a protocol is in charge of a conversation, it keeps the responsibility unless it passes the responsibility to another protocol or it finishes the conversation.

Figure 4.12 describes a scheme of a *Mulan* protocol. In the image the *Renew* conform inscriptions for the *up-links* are used (`:start()`, `:stop()`, `:in(p)`, `:out(p)`, `p` is a message also called performative). The flow of the process is directed from left to right; a *Mulan* protocol can therefore be read like a sentence. First the protocol has to be started (transition `:start()`) then a message is received. Now some other actions could be performed which are not shown in the scheme but signified by a transition. Then a message is send and finally the protocol is stopped which ends the conversation.

## 4.3.6 Implementing *Mulan* Protocols

The described Petri net model of figure 4.3 offers an overview over the agents infrastructure but it cannot be executed in a multi-agent environment. By refining all parts, a concrete executable model or implementation is constructed. To show an implementation of an application on the basis of *Mulan*, a version of a producer-consumer example is implemented as *Mulan* protocols. Figures 4.13 and 4.14 show executable Petri nets for the example. *Mulan* protocols are reference nets. They offer a Java-like inscription language. Communication between agents and *Mulan* protocols is realized via synchronous channels. This includes the starting and the stopping of the protocol and the message passing between protocols and agents. *Mulan* protocols can be read from start to stop, i.e. from the transition labeled `:start()` to one of the transition labeled `:stop()`.[5]



```
import de.uni_hamburg.informatik.tgi.siedler.sl.Sl0Creator;
import de.renew.agent.repr.sl.ActionExpression;
de.renew.agent.repr.common.VTSet aidvts;
import de.renew.agent.repr.sl.SlContent;
import de.renew.simulator.NetInstance;
import de.renew.agent.repr.acl.*;
NetInstance wb;
AclMessage p;
AgentIdentifier aId;
```

```
:access(wb)
wb:ask("consume",aidvts)
action aId=(AgentIdentifier)
aidvts.iterator().next()
```

```
action p=Sl0Creator.createActionRequest
(aId,"consume")
```

Figure 4.13: Producer-consumer: *produce* protocol as Petri net executable in *Mulan*.

In this example the producer agent pro-actively starts the *produce* protocol of figure 4.13 (`:start()`), a dummy message is received and ignored. At transition t1 the knowledge base is queried for an appropriate consumer agent. Assuming that there exists a list of identifiers of consuming agents, one – the first in the list – is chosen and its agent identifier

[5]Usually *Mulan* protocols can be read from left to right like a sentence or from top to bottom like a part of an extended sequence diagram. In this work most *Mulan* protocols is shown in the horizontal version.

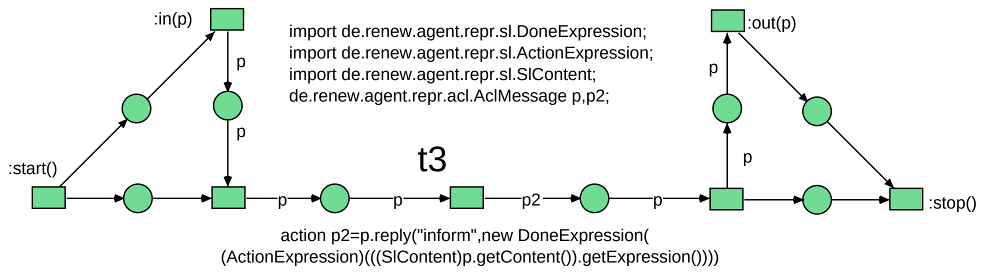

:in(p)
p
p
import de.renew.agent.repr.sl.DoneExpression;
import de.renew.agent.repr.sl.ActionExpression;
import de.renew.agent.repr.sl.SlContent;
de.renew.agent.repr.acl.AclMessage p,p2;
:out(p)
p
p
:start()
t3
p
p
p2
p
:stop()
action p2=p.reply("inform",new DoneExpression(
(ActionExpression)(((SlContent)p.getContent()).getExpression())))

Figure 4.14: Producer-consumer: *consume* protocol as Petri net executable in *Mulan*.

is moved to transition `t2`. Here a *consume* message is generated with the agent identifier as the receiver's name. This message `p` is handed to the own agent for sending, using the synchronous channel `:out(p)`. The *produce* protocol waits for an answer (`:in(p)`) to terminate the protocol (`:stop`).

At the consumer agent the message is received and the appropriate *consume* protocol of figure 4.14 is started. The message `p` is received and a reply is produced at transition `t3` and sent back.

## 4.4 Summary

The area of agent-oriented software engineering is still in an early phase of its development. The terms agent and multi-agent system are not yet fixed or agreed on in the research and development community. Many different views exist that have similarities and also dissimilarities, depending on the point of view.

In the software engineering view agents are active and autonomous software components that can be regarded as generalizations of objects or vice versa objects can be regarded as purely reactive agents. Agent can communicate with other agents. Furthermore, they have aims (or goals) that can be similar or in opposition to the aims of other agents, so they are in cooperation or competition to each other. Software systems that are formed of several communicating agents are called multi-agent systems.

In artificial intelligence the focus on agents is laid on intelligent methods that are incorporated by the agents to perform a certain task.

*Mulan* is a reference architecture or an agent platform based on Petri nets. It is a model of the platform as well as an implementation. The *Mulan* agents, their knowledge base, the platforms on which the agents reside and the protocols are Petri nets. *Mulan* protocols determine the behavior of the agents within the conversations. They are the main parts of the system that have to be modeled and constructed while implementing a multi-agent system.

# Chapter 5

# Structuring Mulan Protocols

This chapter introduces net components (Cabac 2002) and their concept. Net components are subnets that are meant to be combined with each other to form a Petri net. By this component-based approach of constructing Petri nets, the drawing of the nets is facilitated and the Petri nets are structured in a way that they become easily readable and unified. For the *Mulan* protocols, a set of net components exists that are called *Mulan* net components. Besides the fact that they provide the basic functionality regarding the communication that is done by *Mulan* protocols, also some simple patterns and programming artefacts are introduced into the development process. These *Mulan* net components are presented in this chapter as well as an example implementation of *Mulan* protocols using these net components.

## 5.1 Net Components

Net components (NC, see Cabac 2002 and Cabac et al. 2003) are subnets, which can be composed or combined to from a large net. They should provide general functionality that can be commonly used. However, a set of net components is only meant to serve for a special subset of similar Petri nets. Only if many similar nets of the same category are produced, the effort of designing a set of net components is rewarded and the benefits of net components are exploitable.

Net components should not be confused with the software engineering viewpoint on components of large software parts. Instead, the net components presented in this work resemble the control structures of structured programming languages like cycles or conditional statements.

### 5.1.1 Notions

A net component is a set of net elements that fulfills one basic task. The task should be so general that the net component can be applied to a broad variety of nets. Furthermore, the net component can provide additional help, such as a default inscription or comments. One of the used components contains a predefined but adjustable declaration node. In a formal way, net components can be seen as transition-bordered subnets. This suits the notion of net components covering tasks.

Every net component has a unique geometrical form and orientation that results from the arrangement of the net elements. A unique form is intended so that each net component can easily be distinguished from the others and identified. The geometrical figure also holds the potential to provide a defined structure for the Petri net. The unique form of the net components and the notion that Mulan protocols can be read from left to right results in structured *Mulan* protocols.

In the default implementation of the *Mulan* net components, places are added at the outward connecting transitions (*interface place*) for convenient net component connection. Only one arc has to be drawn to connect one net component to another. This is a simple and efficient method that also emphasizes the control flow. The connection of net components is provided by this place, which at all times should only contain anonymous tokens (or none at all).

Direct data exchange between net components is not desired in order to guarantee an easily connecting interface. Instead, data is handed to the data-containing places via virtual places. By adding an appropriate virtual place to the net component, data can be transferred indirectly to the transition that uses a variable. In the usual case a test arc does this. Data is handled and stored in a data block, which is located above the control flow part of the protocol. Annotations of the data-containing places should be adjusted to the appropriate name as well as the annotations of the corresponding virtual place.

### 5.1.2 Structure

Net components are transition-bordered subnets that can be composed to form a larger Petri net. Their purpose is to provide pre-manufactured solutions of reoccurring challenges. Moreover, they also impose their structure onto the constructed net. Like a snowflake's structure is determined through the underlying structure of the water molecules, the net is structured by the net components.

Through their geometric form, the net components are easily identified in a larger net. This adds to the readability of the net and to the clearness of the overall structure of the net, which is an accumulation of substructures.

Jensen (1996) describes several design rules for Petri net elements, which are based on work that has been done by Oberquelle (1981). These rules are concerned with the ways of drawing figures and give general advice for Petri net elements such as places, transitions and arcs. They are also concerned about combinations and arrangement of the elements.

Net components extend the rules by giving developer groups the chance to pre-define reusable structures. Within the group of developers, these structures are fixed and well known, although they are open for improvements. Conventions for the design of the code can be introduced into the development process, and for developers it is easy to apply these conventions through the net component-based construction. Furthermore, the developing process is facilitated and the style of the resulting nets is unified. Once a concrete implementation of net components has been incorporated and accepted by the developers, their arrangements (form) will be recognized as conventional symbols. This makes it easier to read a Petri net that is constructed with these net components. Moreover, to understand a net component-based net it is not necessary to read all its net elements, it is sufficient to read the substructures.

### 5.1.3 Requirements for Net Components

Net components have to be designed for their purpose. In any case, different kinds of nets require different sets of net components. However, within a set of net components that has been designed for a special purpose, the net components should remain as generic as possible. The net components should be easily inter-connectable so that the construction of nets is facilitated. Furthermore, net components should be designed to represent one syntactical entity. This means that a net component should represent one basic task that is decomposable. A net component should be easily identifiable to a reader of the net. This can be achieved by arranging the net elements in a unique (geometrical) form.

Finally, a net component should also provide solutions for challenges that frequently reoccur. Functionality that is thus implemented once can be used again without going through the process of 'low level' implementation again. Altogether these characteristics for net components are shown in the table 5.1:

Especially for *Mulan* protocols that are produced in numbers while designing *Mulan*-based application, some advantages can be seen in a component-based approach. The net components contribute not only to a clear structure of the nets, but also to a faster

| **Characteristic** | **Benefit** |
|---|---|
| Generic character | Net components are broadly applicable. |
| Interconnectivity | Net components are easily combinable. |
| Closeness | Net components have clear semantics. |
| Unique form | Net components are easily identifiable. |
| Located in repository | Net components provide pre-manufactured solutions. |

Table 5.1: Criteria for net components.

development of applications.

## 5.2 The Mulan Net Components

A set of net components for *Mulan* protocols exists (Cabac 2002, chapter 4.3) that has been successfully tested and used in a teaching project (*settler 2*, Moldt et al. 2001) at the University of Hamburg, Computer Science Department, Theoretical Foundations Group. The set of *Mulan* net components has been used excessively, and a large number of net component-based *Mulan* protocols have been designed during the project.

These *Mulan* net components provide the basic functionality to construct protocols. Those protocols that are constructed with the help of the *Mulan* net components are not restricted to the exclusive use of net components; however, it is unnecessary to use non component-based net elements, because the set is self-contained. The set provides structures for control flow management that includes alternatives, concurrency, cycles and sequences. In addition, the functionality for exchanging data is provided, which offers receiving or sending of messages. Furthermore, some basic protocol related structures are provided that handle the starting and the stopping of the protocols.

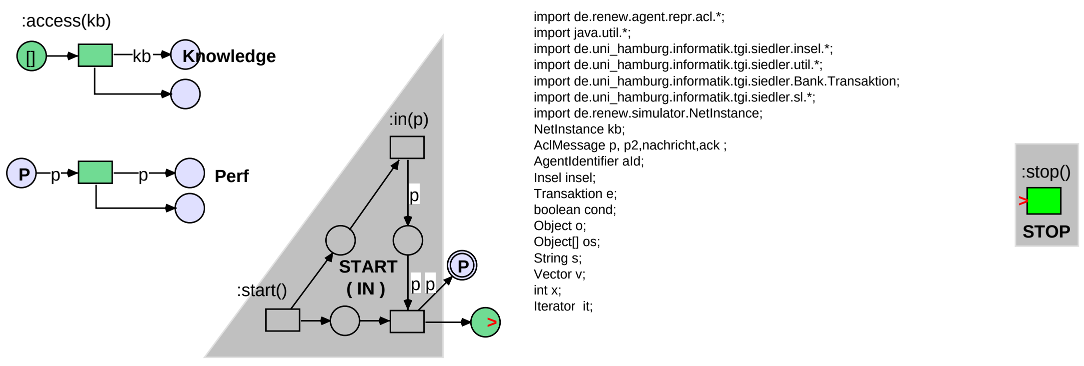
### 5.2.1 Description

A selection of *Mulan* net components is presented in this section. This is done to demonstrate what kind of functionality they provide for *Mulan* protocols and their form – by which they are identified – is introduced. In this section, the essential and most frequent net components for messaging and for basic flow control are presented. Further net components exist that cover sequences, sub calls and manual synchronization.[1]

#### Essential Net Components

Beginning (*NC start*) and Ending (*NC stop*) are needed in all protocols. There is exactly one start in every Mulan protocol, but there may be more than one stop. The protocol is started when the transition with the channel inscription `:start()` is fired and stopped when one transition with the inscription `:stop()` is fired. In addition, the *NC start* also

[1]The full set of net components can be found in (Cabac 2002).



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance kb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure 5.1: Essential net components: *NC start* and *NC stop*.

provides the declaration of the imports, all variables that are used by the net components and the access to the knowledge base (`:access(kb)`). At the beginning a *Mulan* protocol always receives a message, so this functionality is also provided in the *NC start* together with a data block[2] that holds the received message. The message (performative `"p"`) is received by the transition `:in(p)` and is handed to the data block of the net component by a virtual place (`"P"`). The message is finally held in the place `Perf` and information from the message can be extracted at the preceding transition and stored in additional places. The four transitions with the inscriptions `:start()`, `:stop()`, `:access(kb)` and `:in(p)` are the up-links of synchronous channels. Interfaces of the net components – i.e. the elements that can connect to other net components – are marked with ">".

Furthermore, the *NC start* provides a declaration for the net and the access of the knowledge base. The declaration already declares the variables that are used in all *Mulan* net components and the import statements. It can be supplemented with other variables or imports by the developer. The access to the knowledge base is realized as a synchronous channel. It has to be supplemented with access methods for the data that is stored in or retrieved from the knowledge base.

### Messaging Net Components

These are the net components that provide the means of communication. The *NC in* receives a message in the same manner as the *NC start* (described in the preceding section). The message is handed to the data block of the net component.

Additional data containing places can be added to the data block as desired. These places can contain elements that were extracted from the messages, for example the name of the sender or the type of the performative. The *NC out* provides the outgoing message task. The *NC out-in* is a short implementation for the combination of both *NC out* and *NC in*.

[2]Data blocks are net elements that hold and provide the desired data in places. The data is accessed through the usage of virtual places.

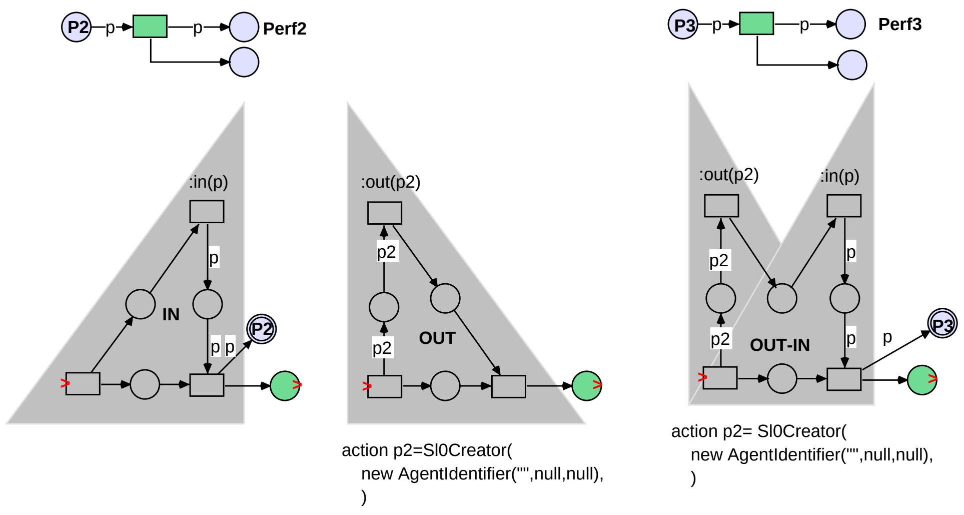


Figure 5.2: The net components for message transport: *NC in, NC out, NC out-in.*

It provides a send request and wait-for-answer situation but does not add functionality other than *NC out* and *NC in*. However, it shortens the protocol significantly.

### Control Flow Net Components: Alternatives, Concurrency

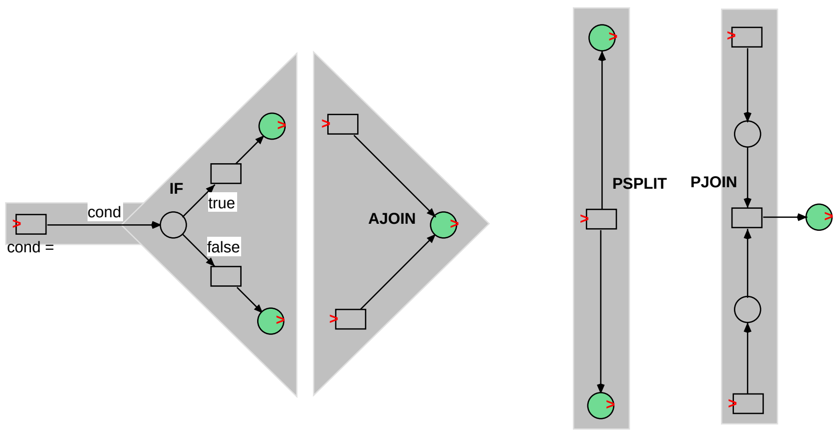


Figure 5.3: Conditional and concurrent processing: *NC cond*, *NC ajoin*, *NC psplit* and *NC pjoin*.

The conditional can be used to add an alternative to the protocol. It provides an exclusive or (XOR) situation. To resolve the conflict the boolean variable `cond` should be adjusted as desired. As a complement to the *NC cond* the *NC ajoin* (alternative join)merges the two alternative lines of the protocol. The *NC psplit* (parallel split) and the *NC pjoin* (parallel join) are provided to enable a concurrent processing within a protocol. Note that the forms of these differ significantly from *NC cond* and *NC ajoin* to have a clear separation of parallelism and alternatives.

### Loops

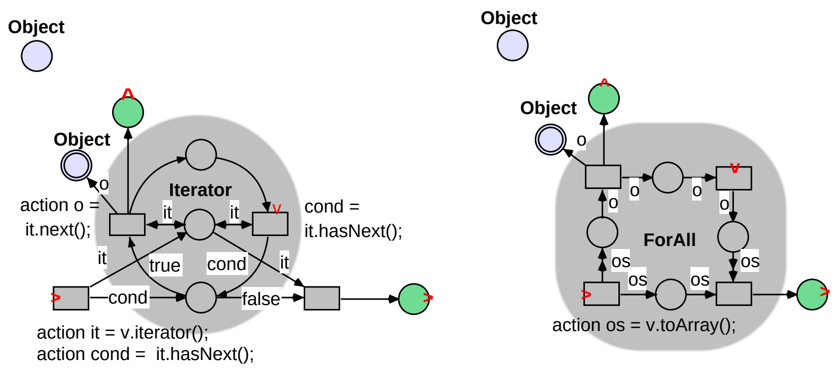


Figure 5.4: Loops. *NC iterator and NC forall.*

These are the equivalent to the basic loops. The *NC iterator* provides a loop through all elements of a set described by the `java.util.Iterator`. It processes the core of the loop in a sequential order. The *NC forall* uses flexible arcs to provide a concurrent processing of all elements of an array. Flexible arcs allow the movement of multiple tokens with one single arc (see Reisig 1997 and Kummer et al. 2001b). The number of tokens moved by the flexible arc may vary, thus its name. In *Renew* two arrowheads indicate the flexible arcs. A flexible arc puts all elements of an array into the output place and it removes all elements of a pre-known array from the input place. The cores of both loops, *NC iterator* and *NC forall*, are marked with $\wedge$ (beginning) $\vee$ (ending).

Petri nets can be drawn with *Renew* in a fast and comfortable way. To be able to use net components in a similar way, it is desirable to have a seamless integration of net components in *Renew*. This is provided by a simple palette that is the usual container for the buttons of all drawing tools for net elements.

## 5.2.2 Realization

Renew supports a highly sophisticated plug-in architecture (Schumacher 2003). It is appropriate to extend *Renew* with a plug-in, so that the usual functionality is still completely available. The net components can be drawn in the same way as simple drawing elements by selecting the tool from a tool palette. Once the palette is loaded into the system, the net components are always available for drawing until the palette is unloaded again. Figure 5.5 shows the graphical user interface with the extension palette loaded.

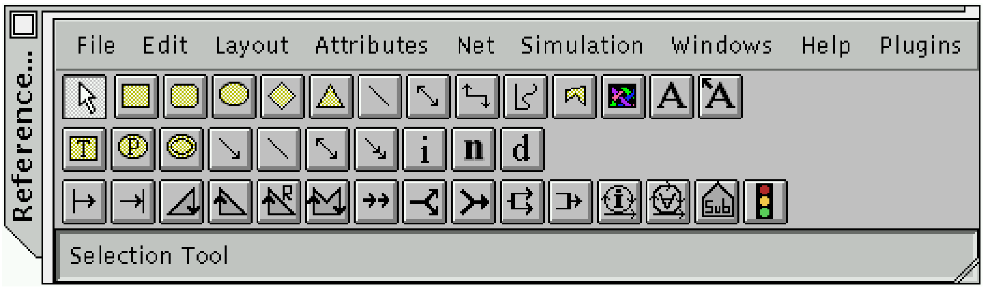


Figure 5.5: The graphical user interface of *Renew* with the net component extension as the bottom palette of tool buttons.

All net components are realized as *Renew* drawings, so they can easily be adjusted to the need of the programmer by editing within *Renew*. The net component drawings are held in a repository, thus a general set of net components can be shared by a group of programmers. Nevertheless, users can also copy and modify the repository to adjust the net components to their needs, or build new net components with *Renew*. It is also possible to use multiple palettes of different repositories.

Net components are added to the drawing in the same way as the usual net elements. The only difference is that after the new net component is drawn all elements of it are selected automatically. This provides the possibility to adjust the position of the net component in relation to the rest of the drawing.

## 5.3 Applying Net Components to *Mulan* Protocols

Implementations, such as *Mulan* protocols, demand complete models in order to obtain executable Petri nets. Consequently, a *Mulan* protocol built with net components has to contain exactly one *NC start* and at least one *NC stop*. The combination of these two are also the simplest *Mulan* protocol that can be built by using net components.

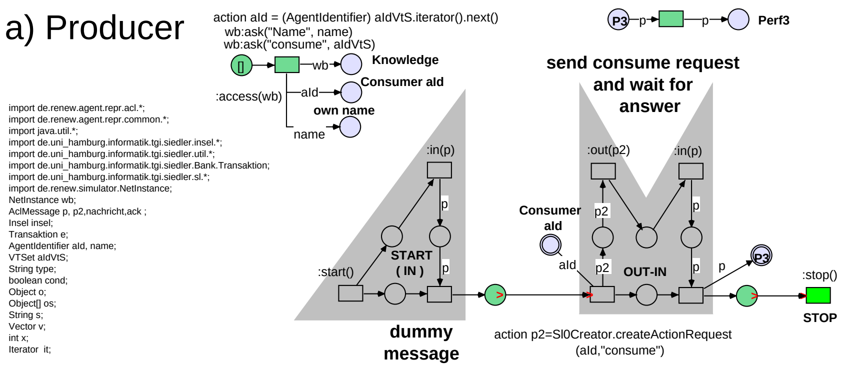
Each net component alone implements one specific task that should briefly be described by its name. However, net components are constructed to be combined with other net components to form a Petri net. To demonstrate the constructing power of net components, two examples of *Mulan* protocols are presented in this section. The first is a producer-consumer example that is presented in a very simple version and is afterward extended with more functionality. The second example is taken from the first implementation of the *settler* game.[3] In fact, this is a reimplementation of a *Mulan* protocol that was first implemented without the usage of net components. For this reason, it is a good object for inspection

[3] "The Settlers of Catan" is a popular card board game. It has been implemented in a series of teaching and research projects (Moldt et al. 2001 and Moldt et al. 2002) at the Theoretical Foundations Group of the University of Hamburg (Bosch et al. 2002). In this work the game is referred to as the *settler* game and the projects as the *settler* projects.

since the two versions can be compared. Thus, the advantages of net component-based *Mulan* protocols can be observed.

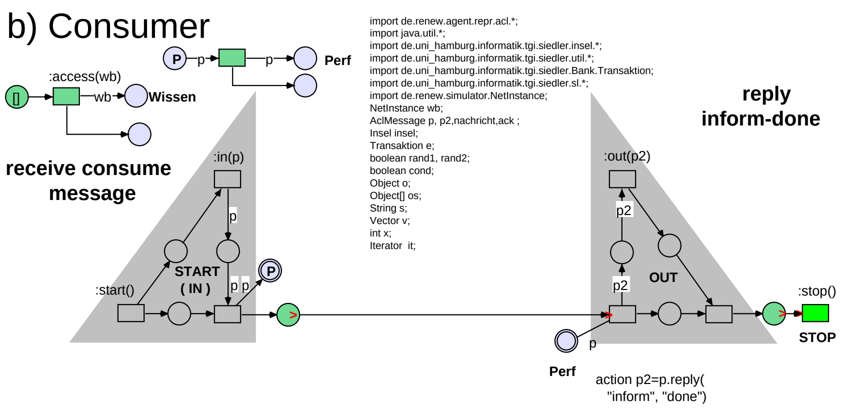
## 5.3.1 Producer-Consumer by Applying Net Components

The *producer-consumer* example from section 4.3.6, displayed in figures 4.13 and 4.14, can also be realized with net components. Although this is still a very simple example, it shows that the construction of *Mulan* protocols is facilitated. For both protocols, only three net components are sufficient to construct the *Mulan* protocols. Additionally, only one virtual place and some inscriptions have to be added to construct the full working *Mulan* protocol.



```
import de.renew.agent.repr.acl.*;
import de.renew.agent.repr.common.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
AgentIdentifier aId, name;
VTSet aIdVtS;
String type;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure 5.6: The *produce* Mulan protocol on the basis of net components.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
boolean rand1, rand2;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

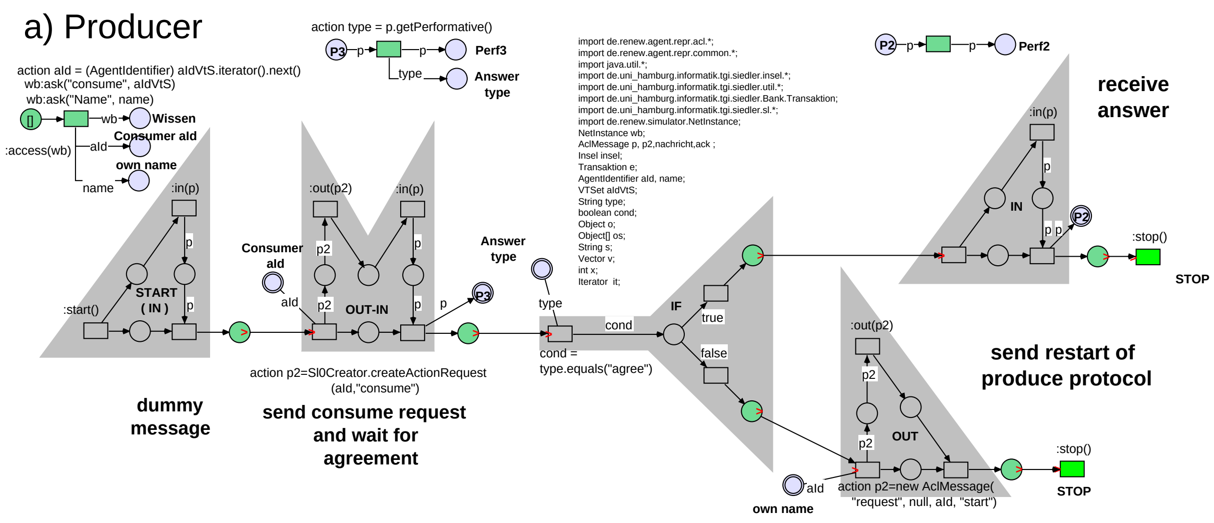
Figure 5.7: The *consume* Mulan protocol on the basis of net components.

Figure 5.6 shows the *produce* protocol with the three net component *NC start*, *NC out-in* and *NC stop*. Figure 5.7 shows the *consume* protocol with the three net component

*NC start*, *NC out* and *NC stop*. This is enough to construct the same functionality as was done in section 4.3.6.

With a little experience the comprehension of the nets is reduced to the reading of two elements each. Without the need to examine every net element, it is thus possible to understand the described processes.

Since this is a very simple example we are now looking into the possibility to extend the functionality by refactoring the nets. The consumer now, before it sends the consume message, can decide whether it wants to consume or not. So first it sends an agreement to the producer. This agreement can consist of an 'agree' message or a 'refuse' message. After it has sent an 'agree' message, a failure in consuming can still take place, so it can send, apart from the 'inform-done' message, also a 'failure' message.



```
action aId = (AgentIdentifier) aIdVtS.iterator().next()
  wb:ask("consume", aIdVtS)
  wb:ask("Name", name)
```

```
action type = p.getPerformative()
```

```
action p2=SI0Creator.createActionRequest
      (aId,"consume")
```

```
cond =
type.equals("agree")
```

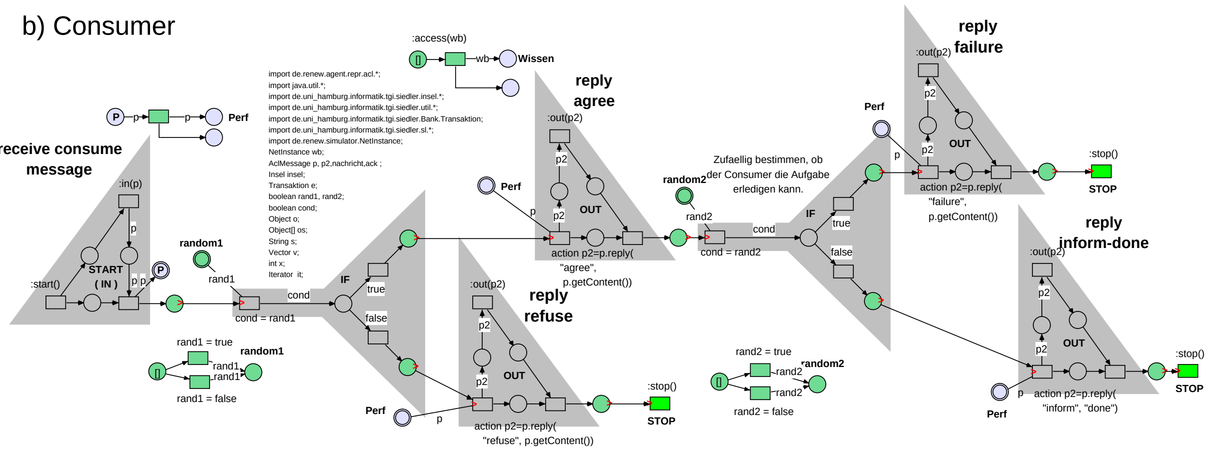
```
action p2=new AclMessage(
   "request", null, aId, "start")
```

```
import de.renew.agent.repr.acl.*;
import de.renew.agent.repr.common.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
AgentIdentifier aId, name;
VTSet aIdVtS;
String type;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure 5.8: The enhanced version of the *produce* Mulan protocol.

Both figures 5.8 and 5.9 show resulting Petri nets of the enhanced versions of the *produce* and the *consume* protocol. Note that the nets are not displayed here to be read as Petri nets, although they are fully operational and can be executed in *Renew* and Mulan without any changes. Instead, they are presented to get an impression of the structured layout and the application of the net components.

Note that the decisions whether it is agreed by the consumer or refused to fulfill the service is modeled as a random choice with the probability of fifty percent. After the producer has received a message that is not an 'inform-done' message, it starts a new produce protocol by sending a message to its own agent. This new protocol can now possibly choose another consumer from its list of consumer agents.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
boolean rand1, rand2;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure 5.9: The enhanced version of the *consume* Mulan protocol.

### 5.3.2 Player Registration

The visual aspects of net components play a crucial role in recognition and thus in readability. Since net components have a fixed geometrical structure they can always be identified without reading any details of the net elements. The geometrical form of the net itself becomes readable to the programmer without using any modeling abstraction.

Figure 5.10 show the original version of the *player registration* protocol implemented for the first *settler* game. Note that the net is not supposed to be readable in all its details. The important fact is that the overall and the mid size structures which are identified as groups of elements (later to be modeled in net components), are detectable. The net describes the registration of a *player* agent at the *game control* agent, showing only the part for the *game control*. The procedure can be described like this:

> The *game control* agent receives a message from a *player*. It sends a message to the *bank* agent to set up a new account for this *player* and waits for an answer. If the account already exists, the *player* is informed that he is already registered. If the account does not exist, the *island* agent gets a message that a new *player* is registered and the *bank* gets informed about the initial funds of the *player*'s account. The *game control* waits for the *bank* to acknowledge this. After that, the *game control* checks how many *players* have registered for the game and if a preset number of *players* are registered the *game control* sends a message to itself to start another protocol that informs all registered *players* of the beginning of the game. Finally the *game control* sends a message back to the currently registering *player* to inform it that the registration is completed.

This is quite a long description but it is necessary here to explain what happens in the net. The *Mulan* protocol, shown in figure 5.10, is already quite well structured. It can be read from left to right and synchronous channels for message passing are exclusively

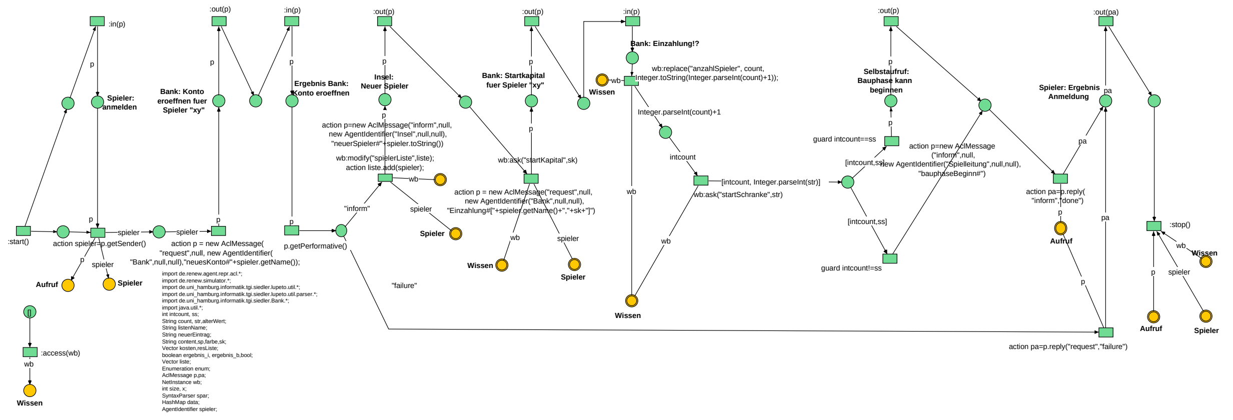


Figure 5.10: Mulan protocol for the registration of a *player* (original version).

located at the top on the net. Actually, this layout helps to understand the net. In the beginning of the protocol, structures that are very similar to the messaging net components are recognizable: the triangle for *NC start* and the ‘M’-figure for *NC out-in*. After the following conflict the clear structure is lost and the reader gets confused.

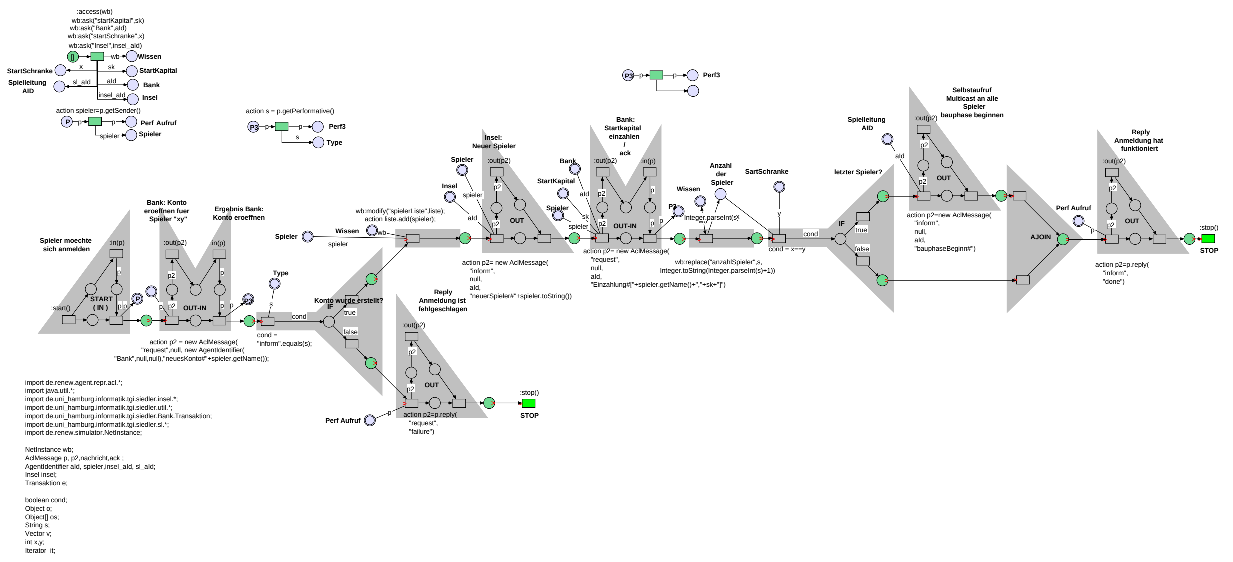


Figure 5.11: Mulan protocol for the registration of a *player* (net component version).

The refactored version of the registration protocol is shown in figure 5.11. It uses exclusively net components to implement the *Mulan* protocol. In addition to the already presented net components, this net also features a net component for representing sequences, *NC sequence*. It consists only of one transition and should be used for action inscriptions or expressions. Again, this net is not presented here to be read as a Petri net, instead we should focus on the overall structure and the net components.

Similar to the *producer-consumer* example in section 4.3.6 the registration protocol in figure 5.11 can be read like a sentence from left to right. If we follow the control flow from the start, we can reach two different stop transitions (*NC stop*) and on the way we pass three different sequences of net components representing three different scenarios. Since this is obvious, just one sequence is mentioned here the other two can be constructed accordingly. The sequence for the registration of the last *player* and the starting of the game can be described by *NC start, NC out-in, NC cond, NC sequence, NC out, NC out-in, NC sequence, NC cond, NC out, NC ajoin, NC out* and *NC stop.* It is the longest possible scenario within the protocol.

Scenarios can be derived from the *Mulan* protocol making it easy to understand the net. More important, though, is that mid size structures – as net components – can be identified from the first glance on the net. For instance, the two conditionals can easily be identified without searching for them.

Due to the usage of net components, the basic tasks performed by this protocol and the structure of the interaction can be read without the need to interpret the details of the net itself. It can be seen that the game control is involved in several communicative acts and two decisions. The general structure of the interaction can thus be derived from the net itself. Only the participants of the conversation are not identified yet. For this we recommend to add comments for each net component.

The example shows that a well-structured Mulan protocol, which uses net components exclusively, can be read without reading any of the net elements. Furthermore, the procedure to understand the net is easier than by looking at the unstructured (or only slightly structured) net of the original version in figure 5.10.

A net constructed with net components can be transformed directly into an agent interaction protocol diagram, as described in section 2.3. This can be useful for documentation or communication. The mapping of net component-based *Mulan* protocols onto agent interaction protocol diagrams is simple. Once the net components are identified, they can easily be mapped onto the corresponding elements of the agent interaction protocol diagram.

Figure 5.12 shows the full conversation between the four agents as the final agent interaction protocol diagram augmented with the geometrical symbols of the net components.

## 5.4 Summary

Net components are sets of net elements with geometrical arrangements that make it easy to identify them and to distinguish them form each other. Each net component is bounded by transitions and fulfills one basic task. As a means of structuring, the *Mulan* net components are capable of accelerating the development of *Mulan* protocols. The readability of net component-based *Mulan* protocols is increased significantly in comparison to *Mulan* protocols that do not use net components. This has been shown in this chapter

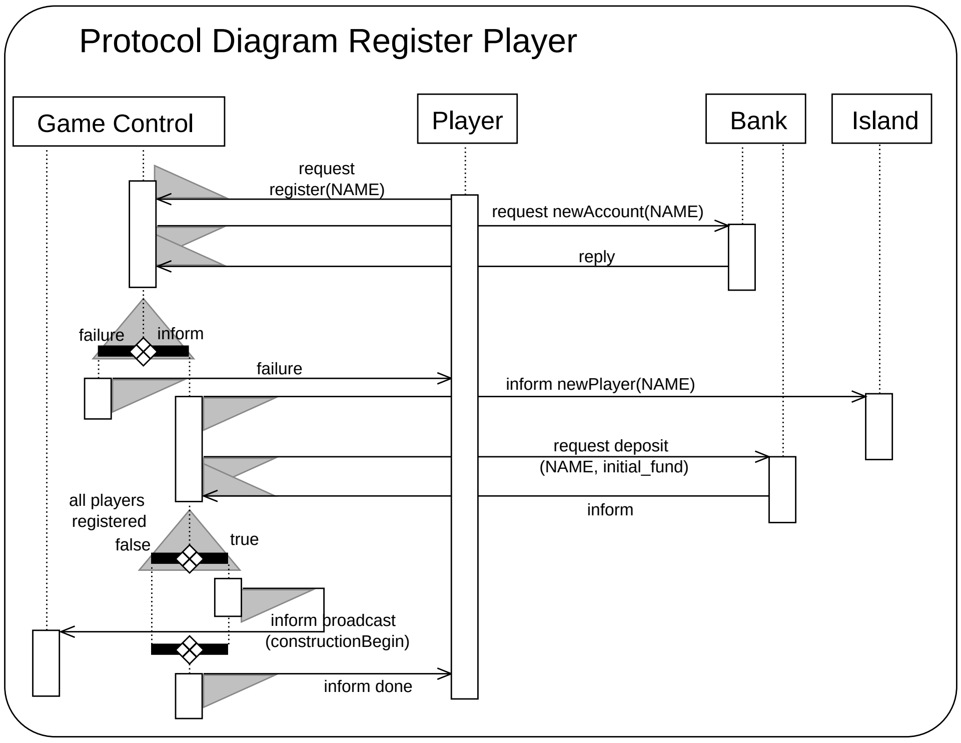


Figure 5.12: Mulan conform-structured agent interaction protocol for the registration of a *player*.

on two examples. One is a constructive example, the other one a comparison between the net component-based and the unsupported *Mulan* protocol.

The net components tool that is implemented as a plug-in for *Renew* makes it possible to apply net components easily to Petri nets. The net components that are held in a repository are editable in *Renew*, thus adaptable to the needs of the development team.

The *Mulan* net components have been used successfully in the second teaching project of the ongoing series of *settler* game projects ‘Adaptive Systeme’.[4]

[4]‘Adaptive Systeme’ (‘Adaptive Systems’, Moldt et al. 2002). For a description of the agent-oriented implementation of the second *settler* game “The Settlers of Catan” see Reese (2003, Chapter 7)

# Chapter 6

# Modeling Agent Interaction

The modeling of agent interaction is important for the designers of agents, so that the protocols can be constructed in a way that the agents' conversations are well defined and ordered. Without the predefining specification of the protocols, the agents would fail to succeed in their interaction. The model of the conversation holds the specification of the agent interaction.

In this section, three ways of modeling agent interactions are described. The focus is laid on the graphical modeling of agent interaction using UML Sequence Diagrams, Petri nets and agent interaction protocol diagrams (compare Cabac et al. 2003, Odell et al. 2000 and AUML 2003). The methods of modeling agent interaction are described in this chapter by modeling the producer-consumer example from section 5.3.1.

## 6.1 Sequence Diagrams

Sequence diagrams are a commonly used in process modeling. As described in section 2.2.2, sequence diagrams are part of the unified modeling language (UML). They are especially useful because with them it is easily possible to convert textual descriptions of proceedings or scenarios into graphical descriptions. In a textual description of a series of interactions, a natural order is supplied by time. A text is always sequentially ordered, so when the text starts with the beginning of the described proceeding and follows each action step by step, it can be translated into a sequence diagram directly.

The difference between a textual and a diagrammatic description can be seen in analogy to the description of a path that is given to reach a certain location compared to a map of a landscape. The map shows the whole area at one glance while the text only describes one way (with possible alternatives).

For the construction of a sequence diagram the interacting entities are ordered and distinguished by their position in the horizontal dimension. Then the interactions are entered into the diagram directly following an order from top to bottom. So the time is represented in the vertical dimension. By this the sequential description of a text is transformed into a two dimensional description that shows clearly the temporal progress and also, with the horizontal messages sent back and forth, the interactions between the entities.

Sequence diagrams are often used to model the message sequences of method calls between objects in the programming world. If agents are seen as generalizations of objects – in the software engineering view (compare section 4.1.3) – then using sequence diagrams for the modeling of agent interaction seems obvious. For agent interactions, the entities are agents and the interactions are messages that are sent and represented by horizontal arrows.

Agents only communicate by messages. This means the communication between the agents is asynchronous. In contrast, the communication between objects is usually synchronous. Since all communication between agents are asynchronous in sequence diagrams, there is no need to represent a synchronous message, so all messages exhibit only stick arrowheads. Furthermore, there are no return statements in the communication of agents. This makes the dashed line dispensable.

Apart from these restrictions, the diagrams look like normal sequence diagrams. Object identifiers now represent agents that act in a role suiting the conversation.

### 6.1.1 Example

To illustrate how sequence diagrams are built for agents, the producer-consumer example is used again. First, a description of the interactions is given in a textual representation and then as sequence diagrams. Two versions of the description are given here. The short version is striped of all conditionals.

**Producer-consumer (short version):** The producer initiates the produce protocol and sends a request message to the consumer to consume the message. To this the consumer can send an agreement message. Then the consumer can also send a message answering the request.

**Producer-consumer:** The producer initiates the produce protocol and sends a request message to the consumer. The consumer can refuse the request by sending a refuse message. If the consumer agrees to the request, an explicit agreement is required, so the consumer must send an agree message back to the producer. Furthermore, in the case of a positive agreement, the consumer must reply to the request by one of the two following methods: 1. The consumer sends a failure message if some exception has occurred and it cannot fulfill its task. 2. The consumer sends an inform-done message if the request has been fulfilled.

In the case of a refusal, the producer receives a refuse message and sends a message to itself to restart a new instance of the produce protocol. In the case of an agreement, the producer waits for the answer and, after having received it, terminates the protocol no matter what the answer is.

It is already clear that these descriptions do not describe one scenario; they describe much more, i.e. a whole set of scenarios (in this case three). One simple scenario that would satisfy the descriptions is: "the producer sends a request and the consumer sends a refusal." A second one would be: "the producer sends a request and the consumer sends an agree message back followed by a failure message." And the last scenario would be: "the producer sends a request and the consumer replies with an agree message followed by an inform-done message." This is the reason why figure 6.1 shows three sequence diagrams for the producer-consumer example. The refuse scenario is displayed in the first sequence diagram of the figure.

The sequence diagrams are displaying the scenarios in a clear way. The scenarios can be compared to each other and developers, who have to implement agent protocols, can use these diagrams as a basis for their communication. This example is rather simple and consists only of three scenarios. However, if the model had to deal with more scenarios, the number of diagrams would grow with the number of scenarios and the whole model would grow to a confusing number of diagrams. In the case of infinite numbers of possible scenarios, which is the case when dealing with cycles, the method would reach its limitation.

But since it is not possible to show neither alternatives nor cycles in sequence diagrams, each scenario has to be drawn into one single diagram. To deal with this, diagrams have to be combined or abbreviated or the missing scenarios can be assumed by the developers.

### 6.1.2 Advantages

The main advantage of sequence diagrams is their simplicity. The semantics of the elements are clear, easily understandable and comprehensible. UML is the modeling standard in the software research and developing community, and therefore almost every developer

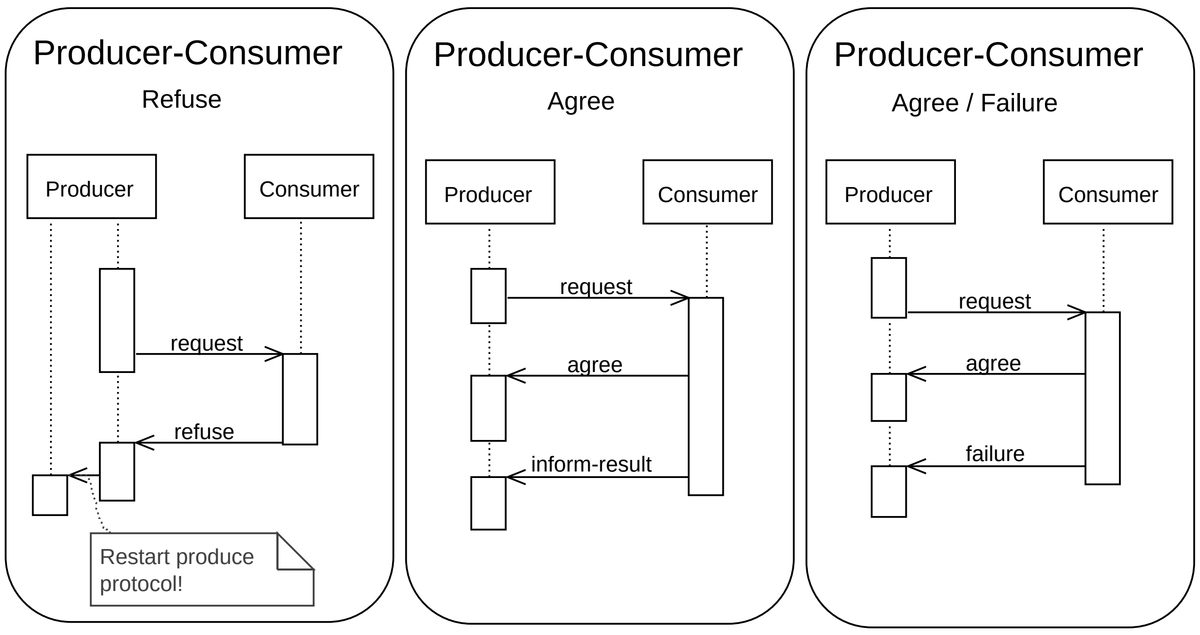


Figure 6.1: FIPA Request Protocol described with sequence diagrams (four scenarios).

has experience with UML and Sequence diagrams. Furthermore, the transformation from a textual description of one scenario into a sequence diagram is a very simple and straightforward procedure, so the conversion can be done easily. In addition to this, it should also be mentioned that with the diagrammatic description of the major scenarios the notions of the processes can be clarified and conveyed to the developers.

### 6.1.3 Restrictions

Sequence diagrams can only display one scenario. They do not provide any means to model alternatives or cycles. As a consequence, multiple iterations must be flattened or abbreviated if they are about to be modeled. It is possible to model simple processes with these diagrams. Test cases, for example, are simply scenarios. That is why it seems appropriate to specify these with sequence diagrams. When it comes to describing procedures, the capabilities of the sequence diagrams are exhausted.[1]

Since the usage of sequence diagrams to describe agent interactions is so limited, other approaches should be considered. One other possibility to model agent interaction is the use of Petri nets.

[1]In the newest proposal for UML version 2.0 (UML 2003b) some ideas of AUML have been included to extend sequence diagrams in a similar fashion. So the statements may not apply to future versions of sequence diagrams in UML. However, since this work actually compares the old version with the new proposals the statements are still valid.

## 6.2 Petri Nets

Petri nets are well suited for modeling processes. They provide sequences, alternatives, concurrency and cycles in a clear and simple diagrammatic representation.

Abstract modeling of agent interaction is possible and efficient. Good models can be built to specify the processes of protocols. Models can be built on different levels of abstraction and concrete models can be executed by virtual machines. *Renew* together with *Mulan* allows agents to execute their *Mulan* protocols, which can be seen as concrete executable models or model implementations of the agent's protocols.

### 6.2.1 Example

The same producer-consumer example of the previous section is modeled here with Petri nets. Using Petri nets for modeling the agent interaction protocols makes it possible to include cycles and decisions into the diagram. Figure 6.1 shows the producer-consumer example as an abstract Petri net model in two versions.

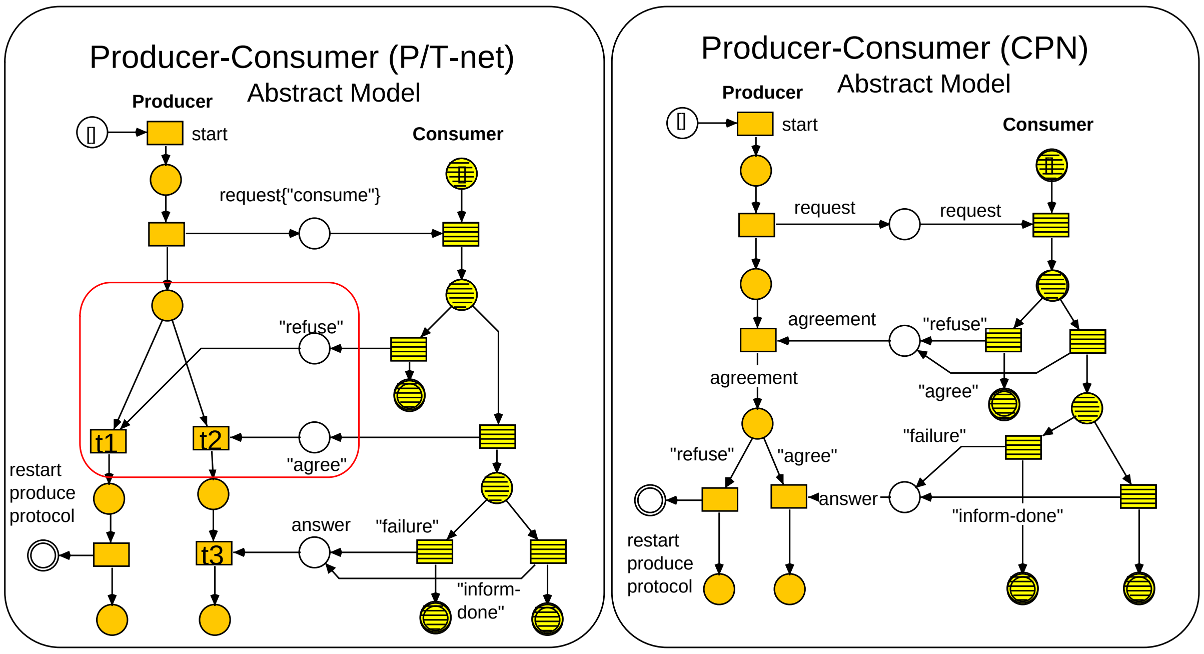


Figure 6.2: Producer-consumer example as P/T-net and as CPN.

The first version is achieved by modeling the producer-consumer example with a P/T-net. The second uses coloured Petri nets, in which identifiable (colored) tokens can be used. In both models the grayish net elements (left side) represent the producer's actions and states while the hatched net elements represent those of the consumer. White net elements represent the communication channels between the two agents and black places indicate the termination (stop) of a protocol.

The abstract Petri net model is much more compact than the sequence diagrams since Petri nets are more powerful in their expressiveness. But the model in itself is more complex and therefore not as easily readable as a sequence diagram. For developers with little experience in Petri nets it may even be impossible to understand the meaning of it.

Both nets model the behavior of the producer-consumer example. The producer sends the request and waits for the agreement. If the consumer agrees it sends an answer, if not a new version of the produce protocol is started again.

However, in the P/T-net model the agreement has to be made explicit by modeling two different places `refuse` and `agree` for the communication channel because all tokens are indistinguishable. This leads to a 'non-free choice' situation for `t1` and `t2` (indicated by the rounded rectangle) and, what is worse, into a dependency between the producer agents behavior and the identity of the communication channel. The conflict should not be solved by the identity of the channel, but by the type of message that is received.

The goal is to create a model that is valid for both agents and that can be converted into an implementation for each agent. The locations of decisions should be made explicit and they should be located at *one* point in the agent's part of the model – not in the communication channels. A separation of messaging and decision-making is also desirable. By using identifiable tokens in coloured Petri nets, the different message types can be modeled as different tokens and the decision can be separated from the receiving of the messages. Thus the decision can be made in one conflict without any additional information.[2]

### 6.2.2 Refinement

One question that has to be asked is: "when is the model good?" In fact, if a model is understandable and also fulfills its purpose, then it is effective and it can also be efficient. Here, though, one additional criterion has to be stated. The model has to be convertible to an implementation. We get one step closer to a convertible model by the separation of separate tasks. Since the net components already identify the separable tasks, it seems clear that a model that is already oriented towards those tasks can be converted easily.

The second model of figure 6.2 can be used to separate the two parts of the two agents. Figure 6.3 displays the two parts separated. The communication is simulated by the usage of virtual places. Dashed lines, in analogy to figure 4.10 in section 4.3.4, indicate the message paths. This model – and also both models of figure 6.2 – can be executed. This simulates the communication flow between the agents' protocols.

The geometrical representations of the net components have already been supplemented in this model. It can clearly be seen that by exchanging the basic net elements with the corresponding net components, the skeletons for the *Mulan* protocols that follow this conversation can be created.

[2]Since the answer of the consumer is not evaluated, it is possible to merge the two different answers at this point. Nevertheless, for the complete modeling a decision, instead of just one transition `t3`, similar to the previous (at `t1` and `t3`) would be more accurate and is only omitted her for the reason of simplicity.

This is essentially a way of implementation through refinement.[3]

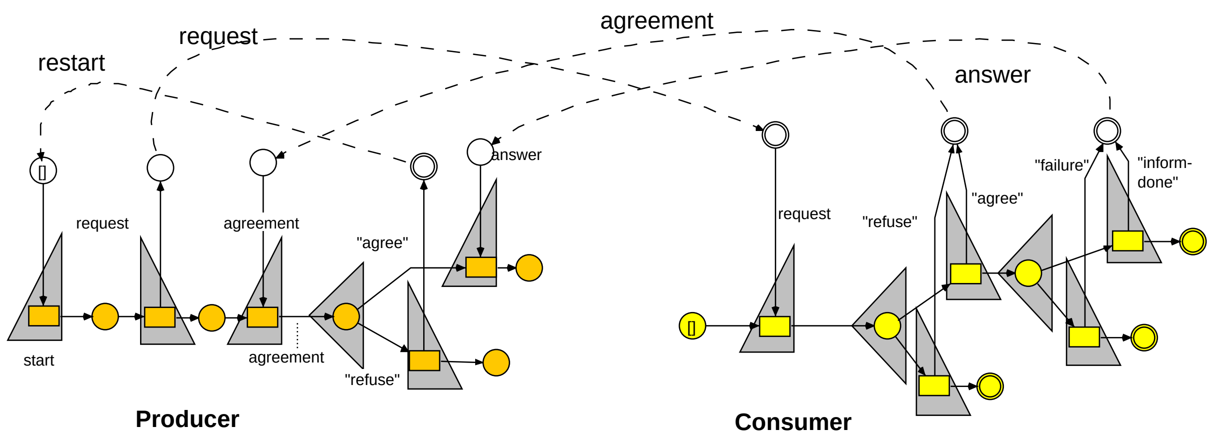


Figure 6.3: Producer-consumer example of figure 6.2 split up into the two parts of the two agents. Virtual places simulate the communication. Dashed lines are not Petri net arcs; they only indicate which virtual place belongs to which original place.

The resulting skeletons are the basis for the *Mulan* protocols that follow the specified conversation. The resulting – complete – *Mulan* protocols are presented in figures 5.8 and 5.9 except that the sending of the request and receiving of the agreement is combined into one net component (*NC out-in*).

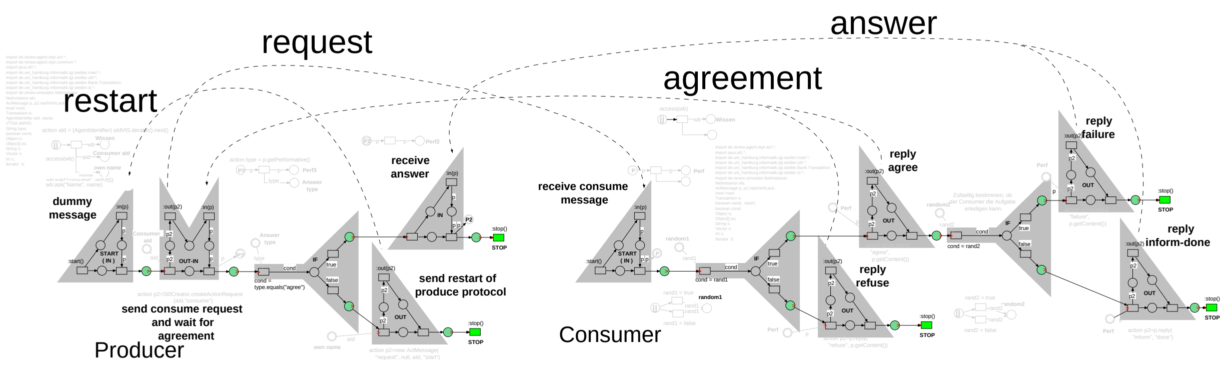


Figure 6.4: The resulting two *Mulan* protocols from figures 5.8 and 5.9 in an overview.

Figure 6.4 displays the two *Mulan* protocols in one image similarly arranged as the models of figure 6.3. In fact these *Mulan* protocols are refinements of the models. Irrelevant details have been faded to gray. Again, this net is not represented here to be read as a Petri net, instead it is sufficient to identify the net components. However, these *Mulan*

[3]Actually, this is not very accurate for the places that represent the conflicts, since net components are exclusively transition bounded. However, the *NC conf* is modeled in exactly the same way so that the desired result can be achieved by refinement solely.

protocols are fully operational. Note that, in contrast to figure 6.3, the dashed lines do not connect the matching virtual places but show, in a similar fashion, the exchanging of the messages through the messaging infrastructure provided by the agents and the platforms (not shown in the figure).

### 6.2.3 Advantages

One advantage of Petri nets is their dual representation as text and diagram. This means that they combine a clear denotation that can be visualized in the diagrammatic representation with an operational semantics that can be executed. Model and implementation melt together into one representation of the system. The models can already be executed for simulation. This can be of advantage in understanding the conversation and for verification (compare Lehmann 2003).

Nevertheless, Petri nets can also be used for abstract modeling that can show overviews (coarsening) or details (refinement) of the system. Although model and implementation merge together it is not always evident that the net can be the program code because traditionally Petri nets have mainly been used for modeling.

Another advantage is the expressiveness of Petri nets that are powerful enough to describe any algorithm. Even concurrent and distributed processes can be described and implemented, which gives Petri nets an advantage over conventional programming languages.

### 6.2.4 Restrictions

The greatest disadvantage of Petri nets as a modeling language is that they are not included in the standard modeling languages. Furthermore a great variety of Petri nets, their appearance and even different formalisms exist.

Concrete models that can also be executed tend to grow very big and complex. If those models are broken down into hierarchies or modules, the overall overview is not given anymore. Accordingly, abstract modeling is required and can be done using Petri nets. But if abstract modeling is done with Petri nets, the universality is lost since only developers with special knowledge are able to read those diagrams.

Implementations using Petri nets also tend to loose their clear structure (Cabac 2002) due to the complexity of the protocols, the variety of construction possibilities and other implementation related flaws.

## 6.3 Agent Interaction Protocol Diagrams

Sequence diagrams are simple and intelligible, but they do not provide any control flow elements. Petri nets have expressive power, provide flow control and can be executed, but the representation is not always well structured and Petri nets are not well known to many developers. In addition, there are many ways of representing the same circumstances. This is often not desired.

This section presents agent interaction protocol diagrams as extensions of sequence diagrams that try to combine advantages of both sequence diagrams and Petri nets by adding control flow elements to the first.

## 6.3.1 Example

By using the elaborated version of the agent interaction protocol diagrams with explicit split of life lines, it is possible to achieve a structure that can directly be transformed into a Petri net by using net components. Instead of applying message split figures for alternatives, the opposite is used, i.e. message join figures as already mentioned in section 2.3.

Especially for alternatively sent messages, a message join figure can at times be of advantage, for example when the message is a reply. This is also intuitive since the receiver of a reply expects only one answer. Thus the number of sent and received messages is directly represented in the model. This can be compared to the step that has been made in section 6.2.1 from P/T-net to coloured Petri net (compare with figure 6.2).

For messages sent from two concurrent paths of the process, neither split nor join figure is required since two messages also arrive at the receiver agent, leaving the numbers of messages unchanged.

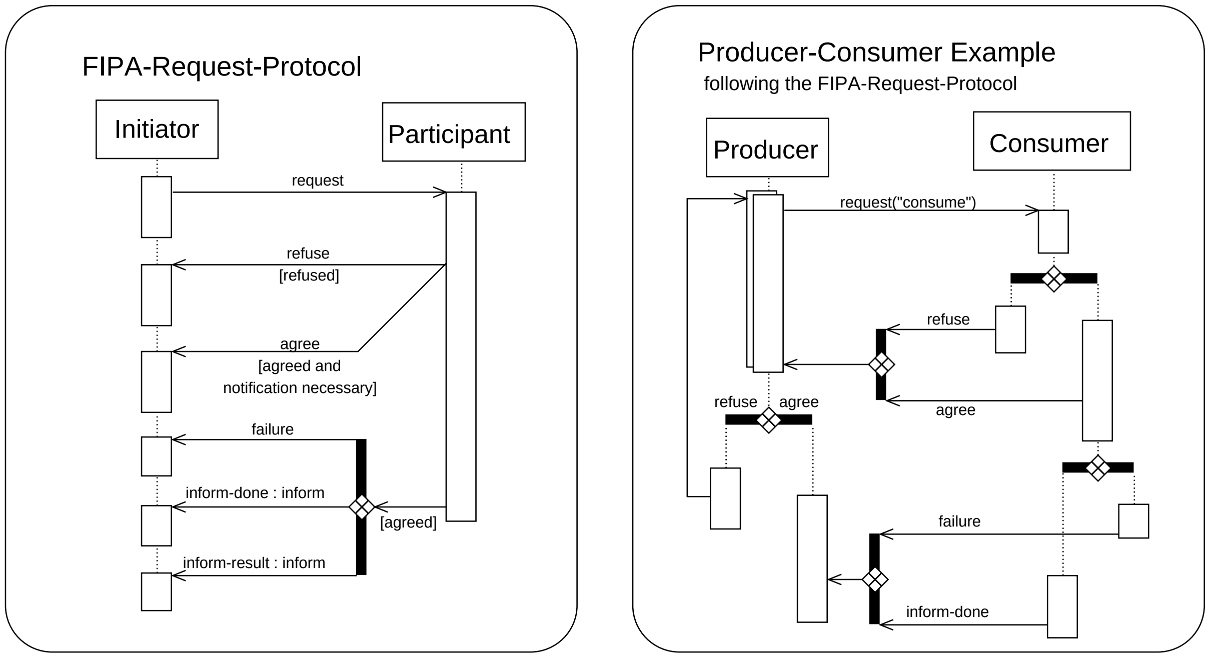


Figure 6.5: Interaction protocol diagram of the FIPA Request Interaction Protocol and a FIPA Request-compliant *producer-consumer* example.

Figure 6.5 shows the original FIPA Request Interaction Protocol specified in (FIPA 2001e). This is displayed here to show that the producer-consumer example, which is used in this chapter, is compliant to the FIPA request protocol. The first diagram uses the abbreviated version of agent interaction protocol diagrams while the second one uses the elaborated version of modeling described in section 2.3. This is done so that the original specification of the FIPA request protocol is displayed also in its original notation.

By using alternatives and also a message that point at the beginning of the initiator, indicating a message that calls a new instance of the protocol, many scenarios can be displayed in one diagram. All possible scenarios are thus folded into one diagram (compare figure 6.1).

### 6.3.2 Benefits

The benefits of the approach using the agent interaction protocol diagrams are quite clear. The simple modeling techniques of the sequence diagrams are combined with the power of full procedural expressiveness. With only a few additional elements added to the sequence diagrams, a notation is achieved that can be easily read and interpreted by any developer of agent protocols that is somehow skilled in UML modeling techniques and knows the sequence diagrams.

Nevertheless, the agent interaction protocol diagrams are denser and more complicated than the sequence diagrams, but they also manage to fold several scenarios into one diagram so the benefit of expressiveness has to be paid by complexity. However, if the modeling is done in a certain fashion, which is oriented towards the implementation done with net components, the structure of the *Mulan* protocols can be not only derived from the model but also automatically generated. This is shown in the following chapter.

### 6.3.3 Restrictions

Using agent interaction protocol diagrams the models can be expressive and simplistic at the same time. However, the problem of multiple possibilities of modeling similar or equal circumstances is not solved completely. Especially when regarding the different representations of the different versions of agent interaction protocol diagrams, the short and the elaborated version, great variations can be produced in the models.

## 6.4 Summary

Sequence diagrams are well suited to model scenarios in the area of software development. They provide a simple representation that is intuitive and comprehensible. However, when it comes to modeling procedures, the expressiveness of sequence diagrams is only very limited.

Petri nets, especially high-level Petri nets (in this case reference nets) have the kind of expressive power that is needed to model procedures. They are capable of representing alternatives and cycles. Furthermore, they are also capable of expressing concurrency or sequential behavior. However, the richness and the variability of Petri nets are far too great to model efficiently in a development team and to achieve conventions without restricting the varieties of possible solutions.

As an intermediate solution, agent interaction protocol diagrams are more expressive than sequence diagrams and more intuitive and closer to the modeling standards than Petri nets. However, Petri nets can be used to define clear semantics for agent interaction protocol diagrams. From either, Petri net models or agent interaction protocol diagrams *Mulan* protocols can be derived. However, when using Petri nets, the modeler has to take caution to achieve a model that is also convertible into an executable *Mulan* protocol.

# Chapter 7

# Generating Petri Nets from AUML Diagrams

Generating code structures, respectively skeletons, from agent interaction protocol diagrams is a conversion of representations of interactions. In this work, the code that is derived from agent interaction protocol diagrams is in the form of diagrammatic representation again, namely Petri nets that are composed of net components.

The goal is to obtain a method that allows an automated code structure generation from agent interaction protocol diagrams. The results of this generation are Petri net skeletons. However, the developers still have to complete the details of the implementations or refactor the skeleton according to their needs. In fact, this can speed up the development process significantly.

This chapter focuses on the design of agent interaction protocol diagrams and their conversion to *Mulan* protocols, while the next chapter presents the implementation of the prototype that allows the generation of the code structures of *Mulan* protocols.

## 7.1 Diagram Conversion

Bell (1998) distinguishes three approaches of model-based code generation for object models. These are the structural, the behavioral and the translative approach. The structural approach is extended by the behavioral approach, which is extended by the translative approach. The structural approach produces incomplete source code (skeletons) from templates that have to be revised. No behavioral aspects of the code are generated by this approach.

The behavioral approach is based on state machines. The behavioral aspects are modeled in these diagrams and source code is generated from them, which is complete and revising the source code is not necessary. The translative approach uses independent models for architecture and application. A complete application model of object structure, behavior, and communication is the base for code that is generated.

Bell (1998) describes the conversion of a model (diagram) to source code as mapping of translation. A similar view can be assumed when converting diagrams to other diagrams.

### 7.1.1 Agent Interaction Protocol Diagrams

Agent interaction protocol diagrams describe the external behavior of the agents in an application of a multi-agent system. So complete code cannot be generated from these diagrams. In the case of converting one sort of diagrams into another sort, it is possible to talk of *transformation*, *mapping* or *translation*. All these terms are well suited to describe the procedure of converting agent interaction protocol diagrams into Petri net skeletons.

From the mathematical point of view, *transformation* or *mapping* focuses on the fact that elements of a set of diagram elements are mapped onto other diagram elements of another set. The agent interaction protocol diagram elements would represent the domain and the Petri net elements would represent the co-domain of the mapping.

From the linguistic point of view, the agent interaction protocol diagrams can be seen as a description, i.e. their elements describe procedures or behavior. We could also talk of a description or modeling language as in the name *'UML'*. If the agent interaction protocol diagrams are recognized as a language to describe procedures then also Petri nets can be considered as a language. From this point of view, the conversion of representations is a translation from one language to another.

Both notions are applicable to the conversion of diagrams; in this chapter, however, the first point of view is assumed. Elements or constellations of elements of the domain of agent interaction protocol diagrams are mapped onto elements of the co-domain of Petri nets, which are, in fact, net component-based *Mulan* protocols.

The following sections describe the syntax and semantics of agent interaction protocol diagrams. The need for standards and conventions has been recognized in the preceding chapter, where solutions for that have been presented as well.

### 7.1.2 Syntax and Semantics

For programming languages the syntax is usually described by using the (extended) Backus-Naur form ((E)BNF). Hoare and Wirth (1973) used another approach of describing the syntax of a programming language, the syntax diagram, for the description of *Pascal*. These descriptions of description languages are actually metamodels.

For the unified modeling language (UML), a metamodel also exists that describes the syntax: the UML metamodel. In the specifications of UML (2003a), not only the abstract syntax of the models but also parts of the semantics are defined by using the UML metamodel and informal description. This is done by the following means. First the models' elements are described informally, then a class diagram is used to describe the syntax, followed by the 'well-formedness rules' given in the object constraint language (OCL). Finally, some more information is given for some of the elements describing the semantics informally again. For a detailed description of the UML metamodel and the metamodeling techniques, see (Baier 2000).

#### Syntax

Since the syntax of agent interaction protocol diagrams does not differ much from that of sequence diagrams, we do not go into its details. The only difference in syntax – so far – consists in the new diagram elements, split and join (see section 2.3.2). These are actually used in a similar fashion in activity diagrams, thus their syntax is clear. Regarding agent interaction protocol diagrams splits and joins can be applied to life lines and messages.

#### Semantics

For the understanding of diagrams, but also for the transformation of diagrams into program code or other diagrams, the semantics of the elements and their arrangement has to be clear and well defined. Similar characteristics are applicable to diagrammatic as for conventional programming languages. According to Louden (1993), programming languages should be efficient, universal, orthogonal, uniform, simple, expressive, accurate, machine-independent, safe, consistent, extensible and restrictable.

Agent interaction protocol diagrams can be examined regarding any of these attributes. However, here the main concerns are the accurateness, the uniformity and the consistency of the diagrams, their elements and the arrangement of the elements. This results in the elimination of ambiguous representations of similar circumstances that is crucial to the successful mapping of one description onto another. Ambiguous meanings of elements have to be eliminated in order to obtain a unique and clear interpretation of the diagram so that it can be transformed into program code structures.

A programming language as well as a diagrammatic language is an abstract representation of processes or procedures. The level of abstraction or of detail has to be decided by the modeler respectively developer. Nevertheless, Booch et al. (1999, pp. 132-133) point out that the level of detail should not be confused with accuracy of the description or model. A less detailed, i.e. less formal, model can be as accurate as a very detailed (formal) one. However, the level of detail directly influences the expressiveness. Regarding the generation of code this means that a less formal model that is correct can (as well as a very formal model) be mapped onto code skeletons, but a more detailed model can produce a more complete skeleton.

#### Agent Interaction Protocol Diagram Semantics

Specifying the semantics of programming languages is a difficult task (Aho et al. 1986). However, a transformation into a description for which an operational semantics already exists defines an operational semantics for the original description. So by specifying the mapping onto the code skeletons of *Mulan* net components – at least partially – operational semantics is defined for the diagrams. It is partial because the skeletons are not executable (yet).

For agent interaction protocol diagrams this means that the concrete mapping from the diagram elements to Petri net elements or *Mulan* net components have to be determined to obtain an operational semantics. The elements of the diagrams are the role descriptors, the life lines, the activations (of tasks), the messages and the split and join figures. For these elements, a mapping onto the desired Petri net figures has to be defined.

### 7.1.3 Restricting Agent Interaction Protocol Diagrams

The manner of drawing agent interaction protocol diagrams is restricted because of two reasons.. The first is the need for conventions that have to be made for a clear and unambiguous description. The second is of a pragmatic nature and enables to build a prototype that is able to perform the task without the need to implement every variation of modeling style. By this, the style of modeling is restricted but the expressiveness of the diagrams is not reduced.

Although agent interaction protocol diagrams offer control flow elements, they still focus only on the external communication between entities, just like sequence diagrams. The internal behavior is not reflected in the diagram.

#### Types of Splits and Joins

In this work, agent interaction protocol diagrams are restricted to use only life line splits and joins, and message XOR joins as additional elements. By using the life line splits it is possible to explicitly show alternative or concurrent threads in the diagrams. For example, the choice (1) of producing and sending one of two possible messages is represented as two possible threads using the XOR life line split. In addition, the concurrent (2) producing and sending of two messages are represented as two threads, but these threads are both concurrently active (AND life line split). In the first scenario (1) only one message is send from one of the threads. This is represented by a message XOR join, so the receiver only receives one message. In the second example (2) – the concurrent threads – two messages are send and thus two messages are received and no join figure is required.

Regarding the join figures it is possible to synchronize two concurrent threads by using the life line AND join. The life line XOR join is a simple merge and allows folding different scenarios into one diagram.

The restriction of the model to split figures for the life lines and join figures for alternative messages makes it possible to draw a model that reflects the same amount of incoming and outgoing messages as a net component-based *Mulan* protocol has messaging net components. In the *Mulan* protocol that is sending one of two possible messages two messaging net components have to be provided but in the receiving protocol only one receiving net component is needed. Thus the two (alternative) threads in the *Mulan* protocol are displayed explicitly just like the two threads in the agent interaction protocol diagrams.

#### Multiplicity of Splits/Joins

Another restriction in this work is that split and join figures are only used for the splitting or joining of two threads (binary splits/joins). This also does not reduce the expressiveness of the diagrams because multiple splits or joins can be constructed by a composition of binary splits or joins.[1]

#### Number of Protocols per Role Descriptor

A protocol should be uniquely identifiable. However, the inscription of the role descriptor is the identifier, so a role descriptor can only identify one *Mulan* protocol. Hence there can only be one life line connected to each of the role descriptors (compare with section 7.2.2). The first activations following this life line mark the start of the protocol.

## 7.2 Realization

Regarding the generation of code or code structures, the style, design and semantics of diagrams have to be considered carefully. Furthermore, two points have to be considered to manage the task of conversion. The first is to define the mapping for each element or constellation of elements and the second is to preserve the order of the elements. The two points are discussed in the next two sections.

### 7.2.1 Approach

If a conversation of agents is modeled as agent interaction protocol diagrams in the fashion mentioned in the preceding sections it is possible to generate code structures from these

[1] In fact, this does not even reduce the convenience of drawing diagrams much, because binary splits are by far the most common. In the *settler 2* project (Moldt et al. 2002) the need for multiple splits and joins in net components and agent interaction protocol diagrams elements was recognized but none of these constructs was applied in the resulting *Mulan* protocols.

diagrams. This is done by mapping the diagram elements to their corresponding net components. This generates a net component-based *Mulan* protocol skeleton. Roughly this means:

- A message arc is the abstract representation of the basic messaging net components (*NC out* and *NC in*).
- A split figure is the abstract representation of the conditional (*NC cond*) or a parallel split (*NC psplit*).
- A life line between a role descriptor and an activation marks the start of a protocol (*NC start*)

Combining the mapping with net components that act as templates for the code generator is a feasible approach. The net components define the basic tasks, only the mapping from diagram elements to these templates have to be defined. The next section describes the mapping in detail.

### 7.2.2 Mapping

The translation from agent interaction protocol diagrams to *Mulan* protocols is a mapping process. Like a discrete function, the mapping for each element has to be defined, the source being a diagram. The results are skeletons of *Mulan* protocols that consist only of net components. In addition to this, the connections among the net components and their arrangement also have to be chosen carefully. Furthermore, the structure of the resulting net has to be considered. However, the structure that is achieved results from the arrangement of the net components as well as their forms.

Furthermore, it has to be taken into account that a source of an agent interaction protocol diagram does not result in only one *Mulan* protocol, but in several. The diagram describes a conversation and the *Mulan* protocol describes and implements the behavior of one agent during the conversation. Hence one *Mulan* protocol has to be generated for each participant's role in the conversation.

The following list of mappings of diagram elements shows that not only elements have to be mapped on net components, but also constellations of elements.

#### Role Descriptors

The role descriptors represent the role of an agent during the described conversation. It marks the behavior of the agent and should name the agent and also purpose or action. The role descriptor figure is mapped on a *Mulan* protocol net template.

More precisely, the role descriptor is the descriptor that connects the activities and all other elements with the role of the agent. Therefore the image should be a reference to a drawing object that is the container for the *Mulan* protocol. The drawing also identifies the *Mulan* protocol through its name. The drawing holds the Petri net template from which an instance can be instantiated.

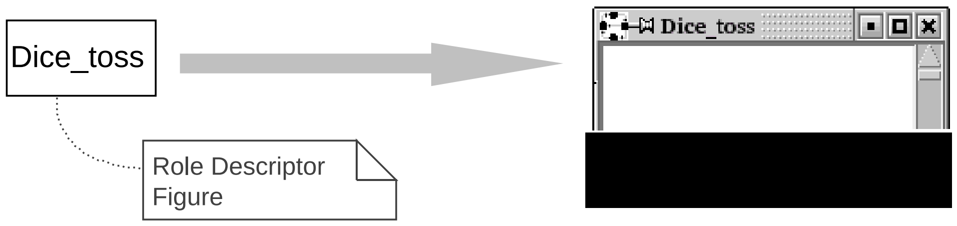


Figure 7.1: Role descriptor figure are mapped onto *Mulan* protocol drawings. Names in inscription of the figures and of the drawings coincide.

Figure 7.1 shows the mapping for a role descriptor figure. The drawing is empty because no action is defined in the diagram yet.

In the *settler 2* project the need for specific naming conventions has been recognized and established to distinguish the affiliation of the *Mulan* protocols to their agents. According to the convention, the name of the protocol should be one word that consists of two parts divided by an *underline* ('_'). The first part is the name (role) of the agent starting with a capital letter. The second part describes the action performed by the protocol starting with a small letter. As an example a player agent uses a '*Player_register*' protocol to register for the game at the game control agent, while the latter uses the '*GameControl_register*' protocol to register a player agent for the game. Figure 7.1 shows the role descriptor figure and the (empty) drawing frame of the generated *Mulan* protocol for the 'Dice' agent tossing the dice.

The name of the protocol is thus defined in the inscription of the role descriptor figure and should satisfy the convention.[2]

### Life Lines

Life lines represent the partial order between the active elements. Two diagram elements that are connected with a life line are in direct successive order. So wherever a life line exists in an agent interaction protocol diagram, a connecting arc has to be used in the *Mulan* protocol in between the corresponding net components. However, some active elements can be in successive order without displaying a life line in between. For example two message arc elements represented in the resulting *Mulan* protocol as messaging net components can be in direct succession by being connected to one activation element.

So the partial order between all net components in the resulting *Mulan* protocol has to be determined by the partial order given through the life line and through the order given by the connecting points of messages to activations.

[2]The generator names the resulting *Mulan* protocol with the inscription of the role descriptor figure.

### Activations

In sequence diagrams activations represent the active phases of objects. Since agents do not cease to be active this notion is not valid for them. Here activations merely mean that the agents are actually carrying out a task at the moment of the activation. The activation could actually be omitted, if the notion is adopted that an agent is always active. However, they can be useful if regarded as containers for sequences of actions or tasks. Actually, all tasks that can be performed in activations are the sending and receiving of messages. So an activation, as a container, can hold a sequence of messages being exchanged.

This is precisely the notion that is used in the generation tool. The activations define the precedences of the messaging net components. More exactly, the precedences are defined by the vertical position of the message arcs that are connected to the activations.

### Protocol Start

The role descriptor holds the reference to the *Mulan* protocol drawing. The (first) life line that is connected to the role descriptor marks the beginning of the protocol. For reasons of simplicity the net component *start* has been striped of the additional messaging functionality. In addition to the transition that holds the synchronous channel `:start()`, the access to the knowledge base is also provided[3] by this net component.

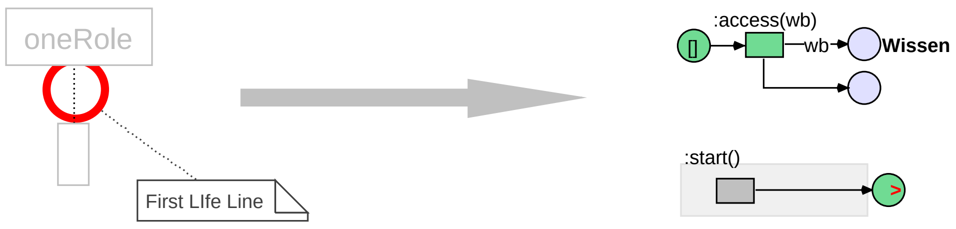


Figure 7.2: Mapping of the first life line end onto *NC start*.

### Messages

When mapping agent interaction protocol diagrams onto *Mulan* protocols, it becomes clear that for each role of an agent, i.e. at least for each role descriptor figure, one *Mulan* protocol has to be generated. Messages do not belong to one *Mulan* protocol. They are the objects that are sent from one *Mulan* protocol to another. In the agent interaction protocol diagrams this is clear since message arcs go from one activation that belongs to one role to an activation that belongs to another role. This means that message arcs must be mapped onto two net components in two different *Mulan* protocols.

Figures 7.3 and 7.4 show the mapping from the message arc end and tip to the net components *NC out* and *NC in*

[3]Also the declaration is provided by this *NC start*, but not shown in figure 7.2.

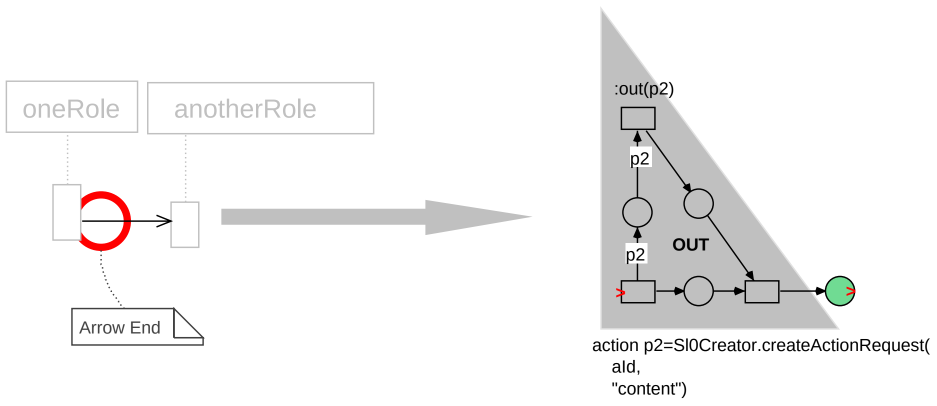


Figure 7.3: Mapping of the message arc end onto *NC out*.

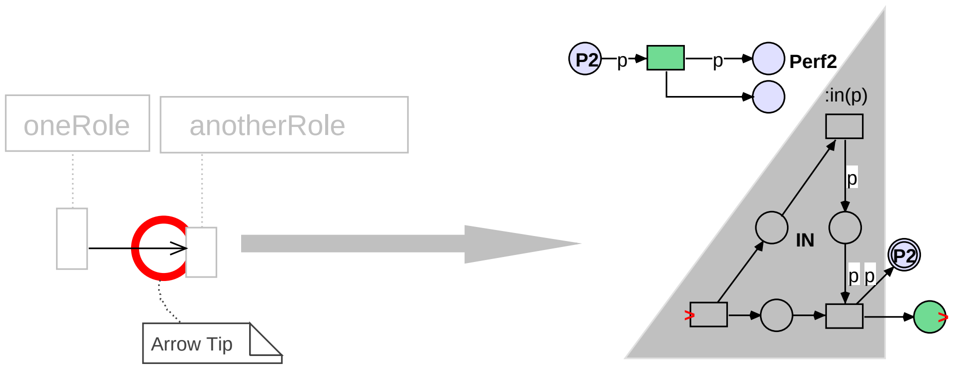


Figure 7.4: Mapping of the message arc tip onto *NC in*.

### Split Figures

Split figures are mapped onto splitting net components (*NC cond*, *NC psplit*). To obtain a consistent structure the design of the two net components has been slightly altered to fit into the same area with the same connecting points.

Figure 7.5 shows the mapping from an alternative split to the net component *NC cond*. The mapping of the concurrent split is done accordingly and is not shown as image.

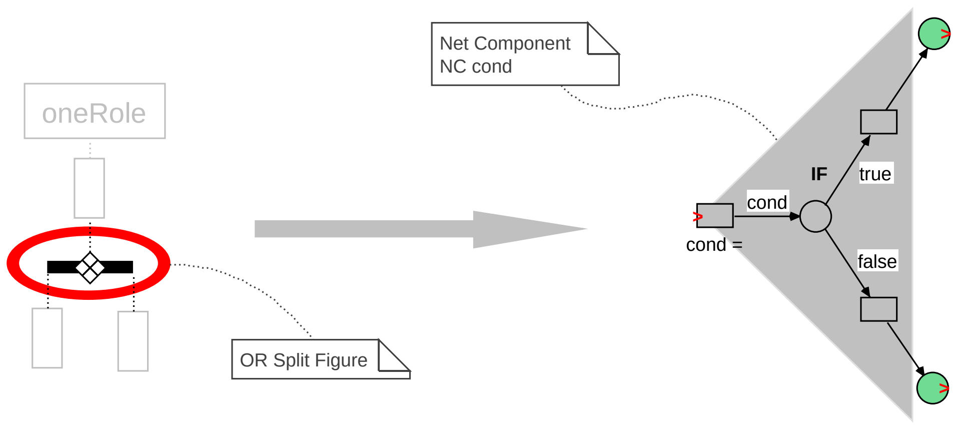


Figure 7.5: Mapping of the conditional split onto *NC cond*.

### Join Figures

The join figures are very similar to the split figures. The images for these elements are the join net components (*NC ajoin*, *NC pjoin*). One challenge, though, is that these are the only figures that have more than one predecessor that makes it difficult to determine the connectors. Figure 7.6 shows the mapping of an AND join figure onto a net component *NC pjoin*.

## 7.2.3 Mapping Table

The mapping table 7.1 summarizes all mappings described in this section. Most diagram elements that can be transformed are represented as net components in the *Mulan* protocols. One exception, though, exists. The role descriptor figure is mapped onto a *Mulan* protocol and can be conceptualized as a reference to this drawing.

The mapping described in the table 7.1 defines – partially – the operational semantics in the context of *Mulan* protocols.

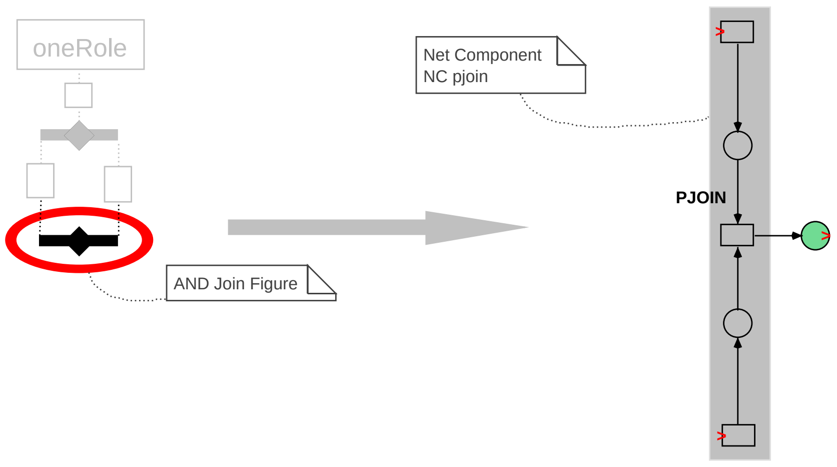


Figure 7.6: Mapping of the parallel join (synchronization) onto *NC pjoin*.

| **Diagram Figure** | **Net Component / Mapped Element** |
|---|---|
| Role descriptor figure | *Mulan* protocol drawing |
| First activation figure | *NC start* |
| Message arc figure end | *NC out* |
| Message arc figure tip | *NC in* |
| OR split figure | *NC cond* |
| AND split figure | *NC psplit* |
| OR join figure | *NC ajoin* |
| AND join figure | *NC pjoin* |

Table 7.1: The full table of mapping of diagram elements of *Mulan* protocol elements.

## 7.3 Pragmatics and Considerations

In the context of programming languages, besides the syntax, semantics and style, the pragmatics also have to be considered. The pragmatics of programming or modeling describes whether a way of describing certain procedures makes sense. Of course, this is even more difficult to decide than to describe the semantics of programming/modeling languages. However, some suggestions can be made in regards to how or when to use certain constellations of diagram elements in the model.

### 7.3.1 Concurrent Threads

Regarding concurrent threads and concurrently sent messages, the example of section 2.3.2 figure 2.8, also presented in figure 7.7 of explicit parallelism, does not make much sense. The diagram is neither syntactically nor semantically wrong, but why should one agent send two messages to the same agent concurrently? This situation is only applicable in exceptional cases. Figure 7.7 shows explicit parallelism in two versions of parallel sent messages that are not pragmatic (a, b) in a usual setting. In addition, one unproblematic version of concurrently sent messages (c) is displayed.

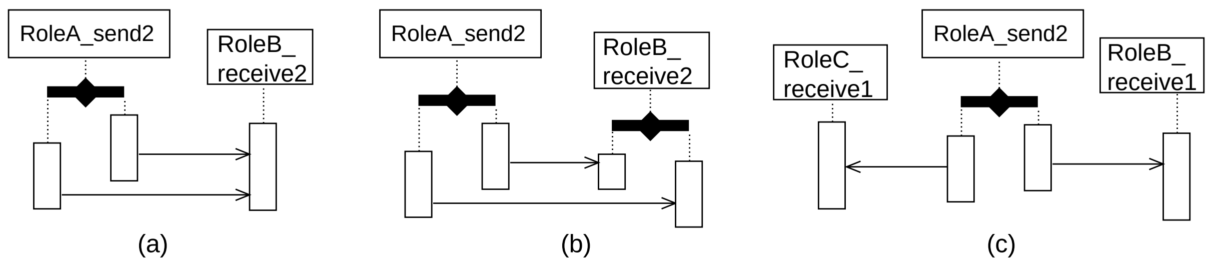


Figure 7.7: Two unlikely situations for explicit parallelism (a, b) and one like situation (c).

One example in which such a situation appears is the FIPA English Auction Interaction Protocol (FIPA 2001b). A parallel split figure is used – as a message split figure – in connection with a multicast message. The multicast message informs all participants of the auction's end while one extra message is sent concurrently to the winner of the auction to start the transaction of selling the goods.[4] The messages that are sent concurrently can be described as implicitly parallel, while the message for the winner is modeled in the diagram explicitly. However, the information about the winning of the auction could be included in the auction's end message. Thus the explicit parallelism is unnecessary.

### 7.3.2 Alternative Threads

In the same manner as described regarding the parallel threads, the alternative threads have to be examined with respect to their applicability. In a common scenario, an alternative split is used to produce and send a message (or set of messages) as an answer to a request according to the decision that has been made. There is only little sense in sending two messages of two alternative threads to different agents, if these are responses to a request.

Figure 7.8 displays a version of alternative sent messages that is not pragmatic in a usual setting (a) while the second setting (b) is common and useful. Situation (a) is not pragmatic because the decision which message is received in which thread cannot be made before the message is received. Situation (b) allows the receiving agent to receive a message and by examining the content it can decide how to proceed.

Exceptional situations exist that might demand a constellation of alternative diagram elements that send two alternative messages to different destinations. Furthermore, decisions of an agent before receiving a message are possible. However, this is not the usual case and should be considered carefully.

[4]This is, in fact, the sole appearance of a parallel split figure in a FIPA interaction protocols (FIPA 2001c).

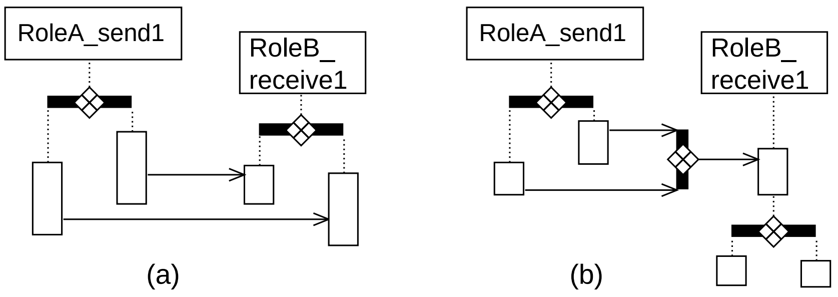


Figure 7.8: One unlikely situation for alternative send messages (a) and one more plausible situation (b).

### 7.3.3 Multiple Instances

Multiple instances of agents appear frequently in the interaction protocols. Examples for this are the participants of the auctions in the auction protocols of the FIPA or the players in the *settler* game. This is a kind of implicit parallelism in contrast to the explicit parallelism mentioned above. If multiple instances of agent roles are involved in a conversation, implicit multicasts have to be made. A notation for multicasts exists, but has not been included in the agent interaction protocol diagrams yet. Actually, it is not necessary to adopt multicast messages in the current version, since the generator does not support processing of inscriptions. However, multicasts can be modeled in *Mulan* protocols explicitly using cyclic net components or they can be accomplished by the message transport service (MTS) of the agent platform.

### 7.3.4 Multiple Participants

In the FIPA interaction protocols (IP, FIPA 2001d), only two participating roles appear in the diagrams. In more complex settings, several agent roles have to be considered. Only in such a situation, the problem described above of sending messages (alternative or parallel) to different agent roles emerges.

### 7.3.5 Coarsening of Diagrams

Especially in complex protocols with many agent roles, abbreviated presentations of activations are useful to preserve the clearness of the diagram. The focused agent role can be modeled in detail while the other agent roles can be abbreviated. This is similar to the coarsening of Petri nets and has been done intuitively by the participants of the *settler* projects. By this method, internal structures can be hidden and thus complexity of diagrams can be reduced. However, in the case of parallel threads – and not only then – an abbreviation can be ambiguous.

Figure 7.9 displays an abbreviation of a question/answer situation (a, b) which is very useful to reduce complexity in the diagram. The figure also displays the abbreviation of a

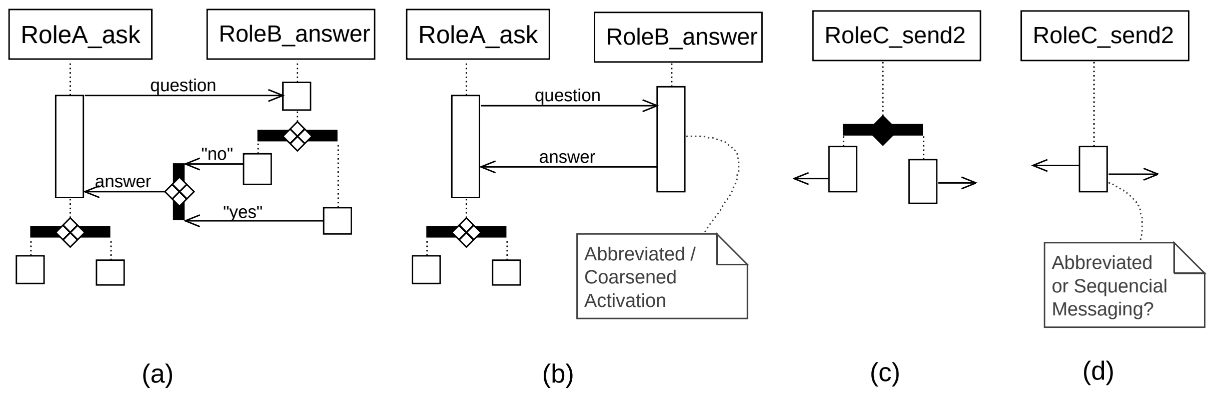


Figure 7.9: Abbreviation of diagram activations/threads (coarsening).

parallel sent message that is ambiguous in the abbreviated version. A sequential order of sending the messages is modeled in the same way.

## 7.4 Summary

The operational semantics of the agent interaction protocol diagrams – in the context of *Mulan* protocols – is defined through the mapping of diagram elements onto Petri net-based net components. For an unambiguous interpretation of agent interaction protocol diagrams, several restrictions of their design are proposed. If the mapping is clear, then code structures can be derived directly from the diagrams. However, the possibilities of drawing extended sequence diagrams, as they are proposed by the FIPA, are numerous and sometimes their syntax and semantics are not clear. So the restrictions are obligatory to achieve a basis from which code skeletons can be derived.

# Chapter 8

# Tool Development

The preceding chapter describes the general approach of drawing and designing agent interaction protocol diagrams as well as the transformation of these diagrams into *Mulan* protocols. This chapter presents the technical realization of the drawing tool plug-in for *Renew*. The development process for the three different aspects of the implementation is described in detail. These are (1) the design of the drawing elements, their arrangements to form a diagram and their implementation, (2) the design of the graphical user interface (GUI) and (3) the realization of the generator of *Mulan* protocols from these diagrams including the redesign of the net components.

So far three versions of the diagram plug-in have been implemented. Version 0.1, which was the initial version, has been constructed on the first prototype of the plug-in system for *Renew*. In this version the diagram plug-in served also as a prove of concept for the plug-in architecture. After the plug-in concept in the prototype had proven successful, the architecture of *Renew* was completely refactored by Schumacher (2003). This resulted in a refactored version (0.2) of the diagram plug-in, in which a better configuration of the plug-in and improved drawing capabilities were implemented. The current version (0.3) incorporates the generation of code structures.

## 8.1 Diagram Drawing Plug-in

The aim of the implementation is to obtain a prototype of a drawing/generating tool with which agent interaction protocol diagrams can be drawn. The diagrams shall be used to support communication between the developers, realize the documentation of the developed system and generate code skeletons in the form of *Mulan* protocols. In fact, this means that the diagram plug-in integrates modeling and development support for Petri net-based agent-oriented programming in *Renew*. During the course of this chapter, the design decisions that were made during the developing process of the plug-in are explained.

The general approach behind the developing process is that of rapid prototyping. Developing basic functionality that is directly used, tested, reviewed and refactored leads to an evolutionary development process. This approach obtains many versions of the software that is permanently refactored and supplemented with new functionality.

### 8.1.1 History

The first decision, that has been made, was the decision to implement the drawing tool for agent interaction protocol diagrams in *Renew*. Other platforms have been considered and seemed suitable for the cause. The first idea was to integrate the AUML diagrams into an existing UML tool. For this *ArgoUML* (ArgoUML 2003) and also *Poseidon* (Gentleware 2003) were examined. *ArgoUML* is an open source project and open for extensions. *Poseidon* is a commercial product, which is built on the basis of *ArgoUML*.

One important reason against both tools is that they are already heavy weight programming and modeling environments, providing elaborated modeling support for established UML techniques. Moreover, new modeling techniques in these tools would have to compete with established ones. Finally, creating support for modeling on heavy weight modeling tools adds to the technical overhead requiring more resources. A lightweight solution is preferable so the tool is implemented as a plug-in for *Renew*.

### 8.1.2 Extensibility

*Renew* is built on the drawing framework *JHotDraw*, which is based on a design of Erich Gamma and Thomas Eggenschwiler. *JHotDraw* is maintained and available at *SourceForge.net* (Gamma et al. 2003). This approach has some advantages. First, *Renew* is used anyway to run the Petri nets and *Mulan*, so this approach does not require any new heavy framework or tool for realization. Second, the design of the diagrams and their elements is not influenced or restricted by the design decisions or the philosophy of other tools. Third, the tool could be as light as it could be, supporting the developer of the protocols directly in the same environment. And forth, the extension can be easily realized within *Renew* because it has a pluggable extension mechanism since version 1.7 developed by Schumacher (2003).

Figure 8.1 displays the *Renew* GUI including the palette for the drawing tools of the diagram elements at the bottom. Following figures can be created with the tools (from left

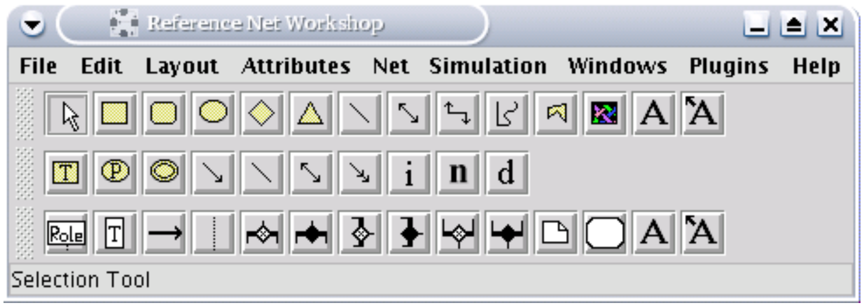

Reference Net Workshop
File Edit Layout Attributes Net Simulation Windows Plugins Help
Selection Tool

Figure 8.1: *Renew* GUI with the palette of the diagram plug-in (version 0.3$\alpha$).

to right): role descriptors, activations (tasks), messages, life lines, XOR splits, AND splits, message XOR joins/splits, message AND joins/splits, XOR joins, AND joins, notes, diagram frames, diagram text and connected diagram text. Role descriptors, notes, diagram frames, diagram text and message joins/splits can be drawn without restriction. Messages and life lines can only be drawn between node figures. Activations, splits and joins can only be drawn south[1] of other node figures so that a life line connection can be drawn between the newly drawn figure and the one north of it. The join figures have to be drawn by pressing the mouse button south of one figure, then drag and then release the mouse button south of another figure to identify two preceding figures for the joins.

### 8.1.3 User Interface

*Renew* can roughly be divided into two parts: the editor and the simulator. Both parts hold shares for the user interface. With the editor, Petri nets and also other drawings are constructed. The user interface offers input and modification mechanisms of various fashion. The simulator offers visualization (token game) and manipulation of the simulation. Regarding the drawing of diagrams, we have to have a look at the editor's capabilities.

The philosophy behind the user interface of *Renew*'s editor can be described as a policy of the *least possible interaction* or a *shortcut policy*. The editor offers several unobtrusive shortcuts that help the developer to draw a Petri net in a fast way. One shortcut is the figure handle[2], with which several new elements are drawn with one mouse action.

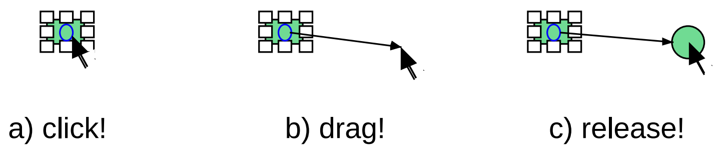


Figure 8.2: Drawings can be extended with one mouse action using figure handles.

[1]The four main directions on a screen are described as north, south, west and east.
[2]*Poseidon* and *ArgoUML* offer a similar functionality called rapid-button.

Figure 8.2 displays the way figure handles work. A selected object displays one or more handles (circle in the center of the transition), which offer context sensitive functionality. In the displayed activity the drawing is extended by dragging an arc out of the transition. With the release of the mouse button a new place is created. So two elements are drawn by one mouse action: the arc and the place. This works also vice versa; an arc dragged from a place will create a transition. So in a fast sequence, a whole Petri net can be created without the usage of any tool buttons or menu bars. The mouse pointer can stay inside the drawing and the function, which is also accessible through conventional means, is accessed through the shortcut. If, however, the arc that is dragged from the figure handle, and the mouse button is released over an appropriate figure, no new figure will be created. Instead only the arc is drawn to connect two already existing figures. So the figure handle works in a context sensitive manner.

Another shortcut is the right mouse button that produces, when clicked on a figure, a figure inscription with a context sensitive default inscription. Again the developer can add drawing elements without using any tool buttons from the menu bars.

An example for drawing manipulation support is the possibility to snap corners of arcs into a position so that the lines of the arcs are horizontal and vertical. This is achieved by pressing the control key while dragging the corner with the mouse pointer.

On the one hand, an objective regarding the drawing tool for agent interaction protocol diagrams is that it should fit into the philosophy of *Renew* so that the look and feel of the plug-in matches that of *Renew* and the integration in *Renew* is consistent. On the other hand, the shortcuts that are used in *Renew* are so useful that it is obviously a good idea to use the functionality that is implemented in *Renew* and adapt it to the needs of drawing diagrams (compare with section 8.1.5). This approach of functionality reuse has been followed and some of the shortcuts have been successfully applied to the diagram plug-in.

### 8.1.4 Imbedding Diagrams

The first step in design was the development of the diagram elements and the implementation of a graphical user interface to draw the elements. For the first version of the drawing tool, figures of the Petri net drawing facilities were adapted and presented in a manner to be able to draw the diagrams. The original aim was to support the modeling, communication and documentation by using agent interaction protocol diagrams and the possibility to distribute electronically stored diagrams among the developers.

The first elements were role descriptors, activations, messages, life lines, diagram frames, one style of horizontal split figures and one style of vertical split figures. They could be applied to drawings in *Renew* by the usual method via tool buttons provided in a palette. For the node elements (activations, role descriptors and splits) new connectors had to be developed because the usual connectors in *Renew* did not allow connections at fixed relative locations.

With the new connectors it is possible to draw horizontal messages and life lines that are connected at the top center of an activation figure, respectively at the bottom center.

In addition, the connectors for split figures were constructed, enabling horizontal messages for horizontal splits or vertical lifelines for vertical splits.

### 8.1.5 Details of Development

Especially the introduced connectors enabled the developers to draw diagrams that were already usable as a communicating and documentation means. However development of these diagrams were difficult and time-consuming. Therefore, the drawing capabilities and the design of the drawing elements were improved in several steps.

- Manually attachable decorations for the split figures were provided but did not prove easy in handling. So decorated split figures were introduced – as seen in this work – but the palette instead of two now had to hold six split figure tool buttons; one for each kind.
- The connectors of all figures were improved to reflect an intuitive arrangement of arcs. For example the connectors of split figures were changed. Arcs can only connect to the ends or to the center.
- The message tips were altered from filled solid small tips to stick arrow tips that reflected the fact that messages are asynchronous means of communication.

At this stage of the diagram plug-in (version 0.1) the diagrams were usable and the drawing capabilities were just sufficient enough to draw – with some effort – diagrams that were used to convey the design of the system that was developed. The agent interaction protocol diagrams were the grounds for the discussion of the specification of the agents' interactions. All drawings were made available to all developers in a concurrent versioning system (CVS 2003), so all developers were able to access a fresh document at any time and modifications could be synchronized.

The first success was achieved with the simple diagrams but soon it became obvious that some extensive refactoring was in need. Since the diagrams are not Petri nets they should not be treated as Petri nets in *Renew*. A new diagram type was introduced to separate Petri net drawings form diagram drawings. This required refactoring of *Renew*'s capabilities to handle drawing types. A drawing type manager was introduced along with different file name extensions for these drawing types.

At the same time, the need for better drawing support was urgent and the first preparations towards generating capabilities were made. The first benefit was that the drawing of diagram was extremely facilitated through three improvements.

- Life lines are now drawn automatically between node elements.
- A message can be drawn directly from a selected activation figure by using a message connection handle, making it possible to draw the message without using the message tool.[3] By dropping the message on the background, a new activation figure is drawn at the place of releasing the mouse button. In combination with the first improvement, this allows us to draw three figure elements (message arc, activation figure and life line) simultaneously with only one action (press mouse button, drag mouse,

[3]This functionality is copied from *Renew*'s figure handles for Transitions and Places.

release mouse button). This increases the speed of drawing the agent interaction protocol diagrams (see figure 8.3).

- To improve the layout capabilities of the diagram, a ‘snap-to-fit’ functionality is included. This lets the figures (activations and splits) snap horizontally to a position so that the life lines are vertical.
- An alternative move of the figures has already been tested which includes all connected figures that are below the dragged figure in the movement. This preserves the layout of a diagram.

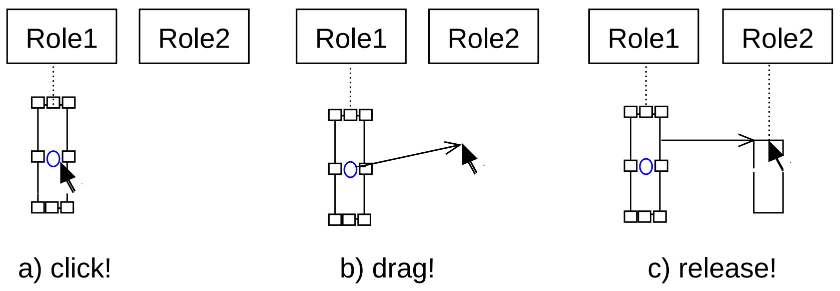


Figure 8.3: Diagrams can be extended with one mouse action using figure handles.

This reflects the present state of the diagram drawing capabilities of the prototype. In addition to these improvements, the need to separate life line split figures from life line join figures has been recognized because the automatic connection of life lines is different. These figures also do not yet implement the ‘snap-to-fit’ function.

### 8.1.6 Towards an Integrated Development Environment

Some functions are not yet implemented in the prototype but considered. These might be implemented in the next version of the tool. However, since the main aspects considered here are (a) to obtain a tool for drawing agent interaction protocol diagrams, (b) to generate code skeletons from these diagrams, and (c) to examine the way of modeling the agent interactions, the functionality – in that respect – of the current version of the tool is not only sufficient, but also quite comfortable. A short list of not yet implemented functions:

- Automatic coarsening/refinement of structures.
- Patterns or diagram components as design patterns (similar to the net components, but on another level of abstraction).
- Post layout ‘snap-to-fit’.
- Exchangeable generator.
- Parameterizable net components.
- Representation and conversion of message inscriptions into *Mulan* protocol message inscriptions.
- Generation of message classes.

- Round trip engineering.
- Integration of actions into the diagrams.

All the items from the list of possible improvements would add to the CASE functionalities of the diagram drawing and generating tool. Each item has its own challenge. For instance the last item in the list: there are two points that have to be considered. The first one is, whether it is appropriate to include internal functionality to agent interaction protocol diagrams. This would alter the function of agent interaction protocol diagrams. As an extension of sequence diagrams these diagrams still focus on the interactions between instances. If internal behavior is directly presented in the diagrams, the focus shifts from the outside view to an overall view.

The second point is the visualization of the actions. Duvigneau (2002) proposes actions to be visualized as activation decorations. Little shaded squares or rectangles could be attached to the activations. They can also hold an inscription describing the action and their position can be adjusted vertically. Figure 8.4 shows the proposed action decoration for activations.

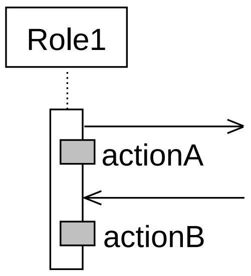


Figure 8.4: Proposal for a representation of actions in agent interaction protocol diagrams.

## 8.2 Generator

While the drawing capabilities of the plug-in are already very satisfying, the support for the conversion of diagrams to *Mulan* protocols is still at its beginning. In the current version only code structures (skeletons) can be generated. This means for *Mulan* protocols that only Petri net elements are generated, but inscriptions remain generic. Due to the usage of net components as templates for the generated code, the appearance of the *Mulan* protocol skeletons is well structured (compare with Cabac et al. 2003). Thus it is possible to understand the generated Petri nets easily and refactor them to obtain an executable protocol.

The generator uses the mapping that is described in section 7.2.2 and summarized in table 7.1 and the net components as templates for the generated code structures.

In general, a generator or compiler has to pass several stages or phases to perform the desired generation. These phases are described as scanning (lexical analysis), parsing (syntactical/semantical analysis), generating and optimizing (Aho, Sethi, and Ullman 1986).

The scanner searches for tokens in the program text and passes these tokens on to the parser that organizes the tokens according to their syntax. The result is usually a parse tree from which the code is generated.

If the source is available as agent interaction protocol diagrams, this procedure has to be altered slightly. One reason is that the form of the diagram reveals already the partial order of the diagram elements. Another reason is that the diagram elements are already available as objects, so no lexical analysis is required. However, the objects are only available in a list of elements (Java `Vector`), so the partial order has to be derived from the diagram.

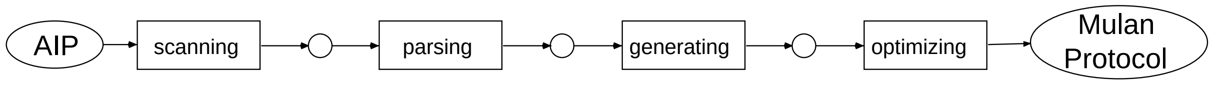


Figure 8.5: Phases in generation.

Figure 8.5 displays the phases of the whole generating process. Although the image displays the optimization as fourth phase, the prototype does not support any optimization. It is not needed at this stage of tool development for two reasons. First, the code that is produced is efficient enough for the kind of applications that are built within *Mulan* at the moment. Second, the prototype is meant to be a prove of concept; for this the unoptimized code is sufficient. However, if the demands on the software performance increases some optimization can be provided regarding the shorter *NC out-in* for combinations of *NC out* and *NC in* or even net components can be optimized.

So only three phases exist for the generation of *Mulan* protocols: the scanning, where the relationships between the node figures are derived from the diagram, the parsing, where the structure of the diagram is made explicit, and the actual generation of the code. These phases will be explained in the following sections.

### 8.2.1 Scanning

In analogy the first step in analyzing the diagram is called scanning. In this step the information about the relationships is extracted from the life lines and messages, and stored in the nodes. Node figures are the figures that can be connected by arcs, i.e. role descriptors, activations, splits and joins. We can express the relationship of the elements, connected through a life line, as a *parent-child* relationship, with the *children* as successor to their *parents* in the temporal order. After the scanning process, each node figure has two lists – one for each arc type – of its directly connected node figures. However, the unidirectional relationship between *parents* and *children* is not found yet. This is done in the next step. The scanning is very simple and is done by checking for each node, whether it is an arc. If an arc is found, the connected nodes are notified.

### 8.2.2 Parsing

In the second step that is called parsing the structures of the diagram are analyzed. Each structure starts with a role descriptor and each role descriptor identifies a structure. These structures are equivalent to the *Mulan* protocols later.

The algorithm of parsing the net structure is now described. The parsing of one structure starts at a role descriptor. This is now the active figure and the head of the structure.

(1) Each connected figure is registered as *child*.

(2) Each *child* registers the active figure as its *parent*.

(3) Each *child* registers the *head* that is known to its *parent* as its own *head*.

(4) Each *child* is registered in the *head* of the structure as a *tail* figure, if it is not already registered there.

(5) Each child is added to the parsing list of active figures, if it is not already in this list. This list is a queue.

(6) If the parsing list is empty, the algorithm stops. Else the next element of the parsing list is made the active figure and the algorithm continues with step number (1).

This is done for all structures, i.e. for all role descriptor figures. After this algorithm, the role descriptor figure 'knows' which node figures belong to its structure and each node figure 'knows' its predecessors (*parents*) and its successors (*children*). The structure or the partial order that exist among the node figures is thus analyzed.

### 8.2.3 Generating

After the structure of the diagram elements is analyzed, the generation of Petri net code according to the mapping of table 7.1 can begin. This is again done in two parts. First, node elements or constellations are mapped onto peer elements that represent the net components. The second step actually draws the net components and manages the Petri net drawing's layout.

The first part of the generation is simple for role descriptors, splits and joins. These are mapped onto the corresponding net components. For activations, a series of peers have to be created that reflect the number and kind of message arcs (tips or ends) connected to the activation. Thus the activation acts as a container for these peers. In addition, the order of the peers has to be preserved.

The second step creates a new Petri net drawing for a role descriptor, named according to its inscription. The role descriptor's *child* figure is put into a generating list that holds the figures for which net components should be generated, its peer provides a location for the first net component and the following algorithm is performed.

(1) As long as there are elements in the generating list do:

(2) Take the first figure from the list and make it the active figure.

(3) Each peer of the active figure draws its net component at the location provided by the *parent*'s peer.

(4) The new location(s) for the next net component(s) is (are) calculated and stored in the peer(s) of the active figure.
(5) All *children* of the active figure are put into the generating list and continue at step (1).

The figures that are located at the end of the procedure, i.e. the figures at the bottom of the diagrams have no *children*. This leaves the ends of the *Mulan* protocols unfinished. They have to be terminated by *NC stop* net components.

### 8.2.4 Layout and Connections

One of the crucial points about generated code is the layout of the code. This is even more valid for graphical representations. Due to the net components, a layout for the *Mulan* protocols is already pre-defined.

The general layout of a net component-based *Mulan* protocol is that of a sentence, it starts at the left side and ends ant the right. Each net component is drawn at a position to the right of an already drawn net component (except for the *NC start*), its peer providing the new location. If the net component is a split figure, then its peer provides two follow-up locations.[4] The vertical positions are adapted to fit both connecting elements. A join figure takes the two positions of its *parent*'s peers and calculates a position for itself vertically in the middle and horizontally at the maximum. Although this layout mechanism is very simple, it produces satisfying results for a large number of diagrams.

After the net components have been drawn, they also need to be connected by arcs with each other. For this, the connectable interface places and transitions[5] of the net components are identified and connected with a *Renew* arc connection.

## 8.3 Examples

This section presents the results that are produced by the generator for three examples. These are the *producer-consumer* example, the *player registration* example from the *settler 1* game and the *game control round* protocol from the *settler 2* game. The first two were already presented in chapter 5.2 and are presented here to demonstrate how the generator works. The latter is a more complex example that is shown here to test the capabilities of the generator.

Since the size of the diagrams and Petri nets increases with the complexity of the protocols and the space on the paper is only limited, it is impossible to present a readable image of the more complex *Mulan* protocols. However, the results should not be withheld in this work.[6]

[4]Actually, the connecting point can be of an arbitrary number, but in the prototype the split figures are designed only for two following net components and the calculation only provides two follow-up locations.

[5]In net components interface places and transitions are marked with a red '>'.

[6]The interested reader can examine the bigger versions of the figures in the attachment.

It should also be stated that the important results that can be examined in the images are the arrangement of the net components. The net components can be recognized by their gray shadow even if the Petri net elements are too small to be recognized.

In the next sections, the generated *Mulan* protocol skeletons for the examples are presented and confronted with the earlier presented custom made *Mulan* protocols. The last example is compared to the original *Mulan* protocol that is used in the *settler 2* game. The skeletons have not been modified except that the gray of the net component shadow has been darkened.

### 8.3.1 Producer-Consumer

The generator produces nice code skeletons from the producer-consumer source displayed in figure 8.6. Since inscriptions are not yet supported in the generation process the ‘pure’ diagram suffices as source.

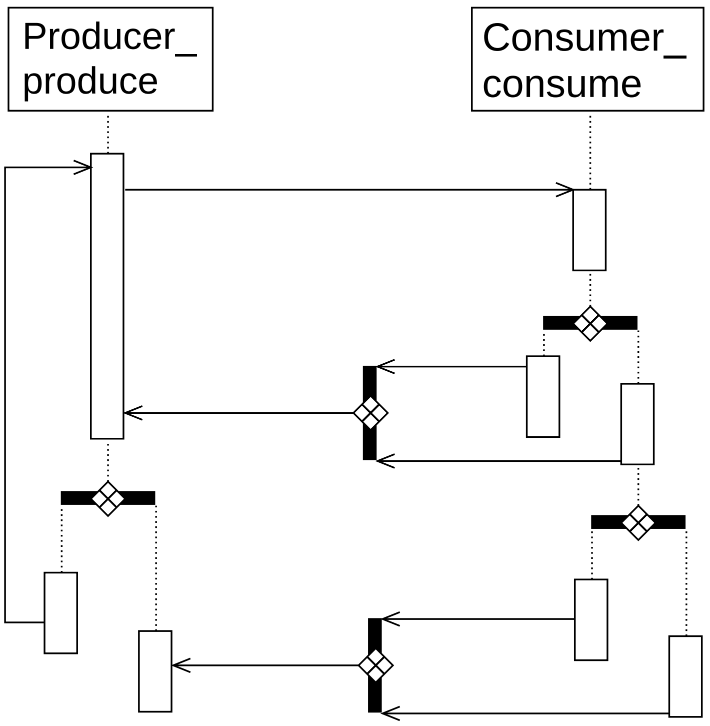
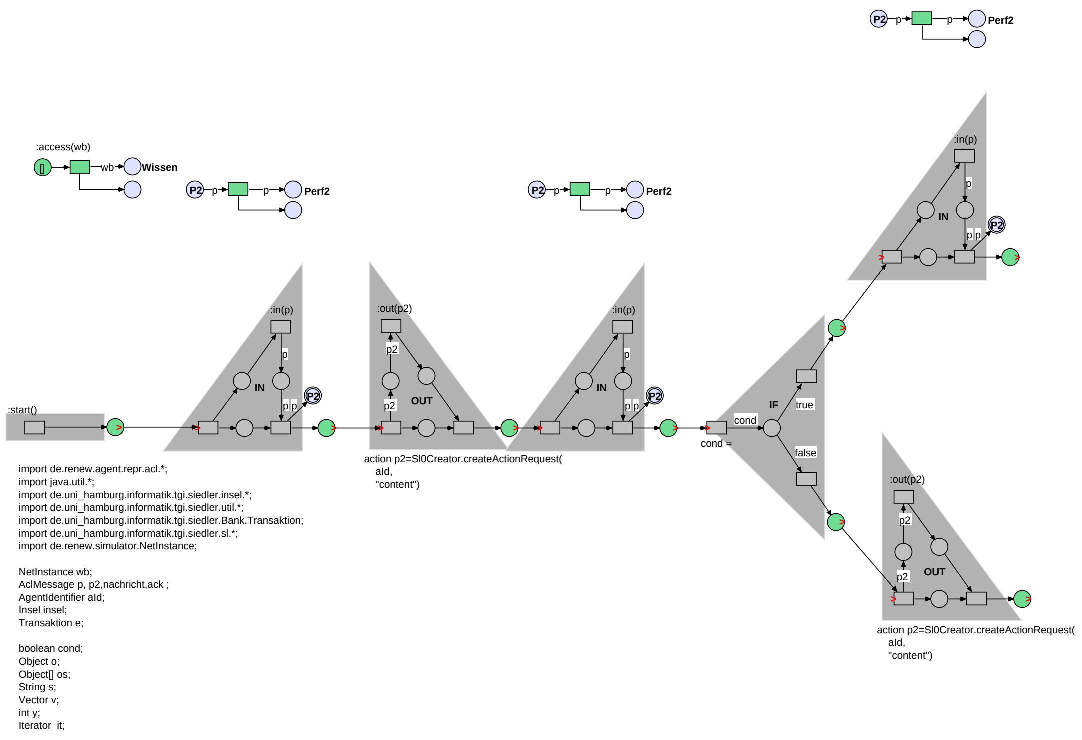


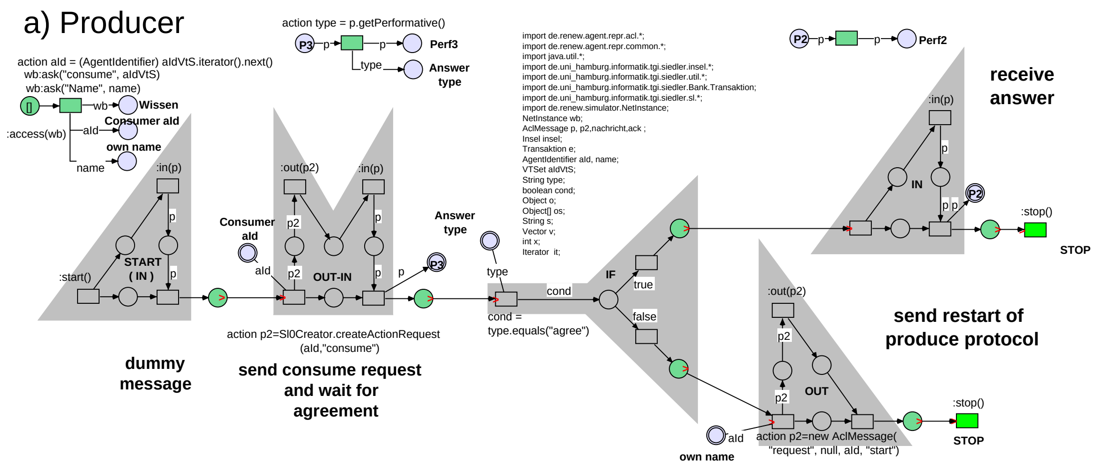
Figure 8.6: The source for the producer-consumer code generation.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

```
action p2=SI0Creator.createActionRequest(
    aId,
    "content")
```

Figure 8.7: *Produce* protocol code skeleton generated from the source in figure 8.6



```
import de.renew.agent.repr.acl.*;
import de.renew.agent.repr.common.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
AgentIdentifier aId, name;
VTSet aIdVtS;
String type;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

```
action aId = (AgentIdentifier) aIdVtS.iterator().next()
  wb:ask("consume", aIdVtS)
  wb:ask("Name", name)
```

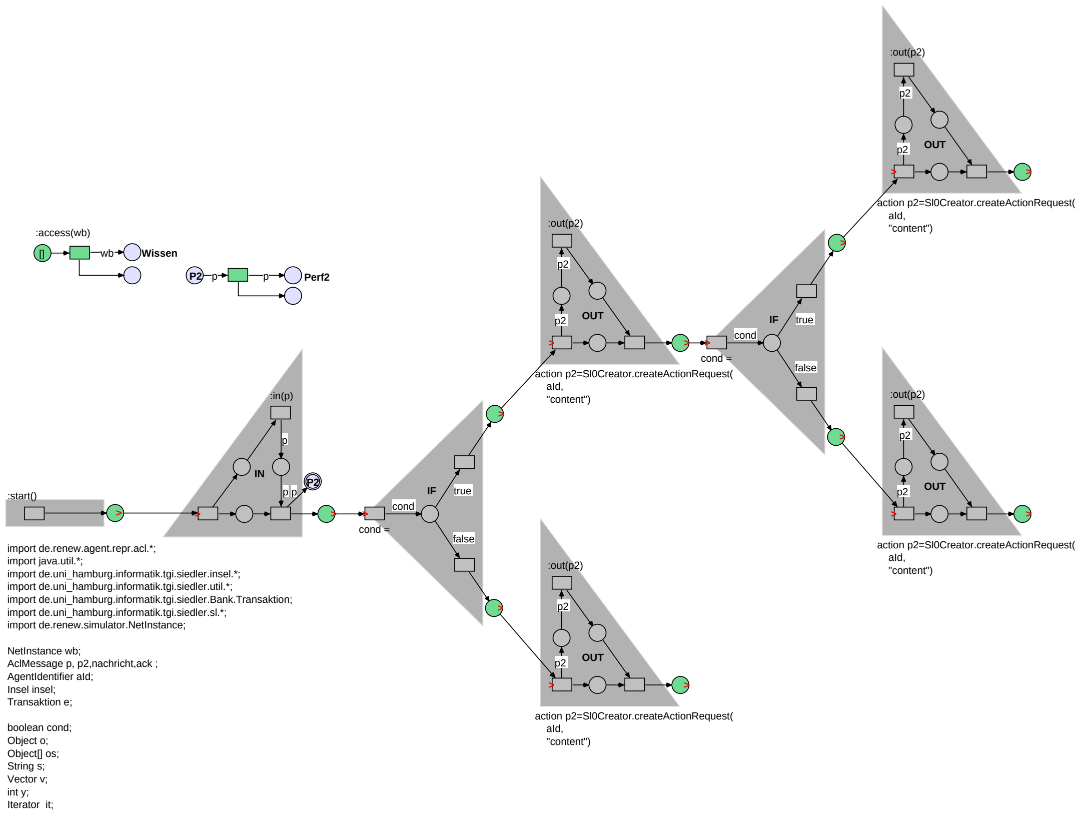
```
action type = p.getPerformative()
```

```
action p2=SI0Creator.createActionRequest
        (aId,"consume")
```

```
cond =
type.equals("agree")
```

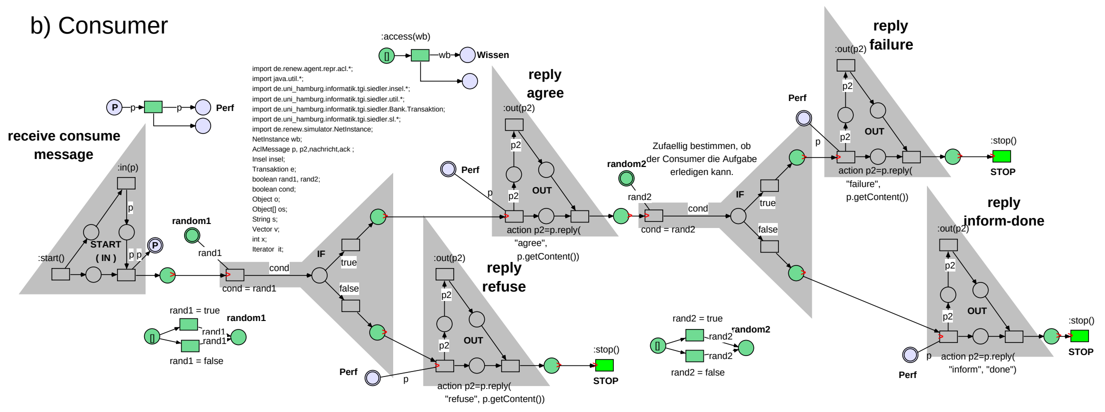
```
action p2=new AclMessage(
    "request", null, aId, "start")
```

Figure 8.8: Custom made *produce* protocol from section 5.3.1.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

Figure 8.9: *Consume* protocol code skeleton generated from the source in figure 8.6



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
boolean rand1, rand2;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure 8.10: Custom made *consume* protocol from section 5.3.1.

### 8.3.2 Player Registration

The player registration example from section 5.3.2 produces satisfying results. Note that in the custom made protocol, two sequence net components are included that hold actions on transitions. This is one example how actions can be applied to *Mulan* protocols. The new feature of decorating activations with actions, explained in section 8.1.6, could be applied here to generate these actions from the diagrams.

Figure 8.11 displays the agent interaction protocol diagram for the registration of the player at the game control of the *settler 1* game. Figure 8.12 shows the code skeleton generated from the agent interaction protocol diagram. This is compared to the code that was produced manually using net components. Note that the generation of *Mulan* protocols are suppressed if the role descriptor name is prefixed with an exclamation mark.

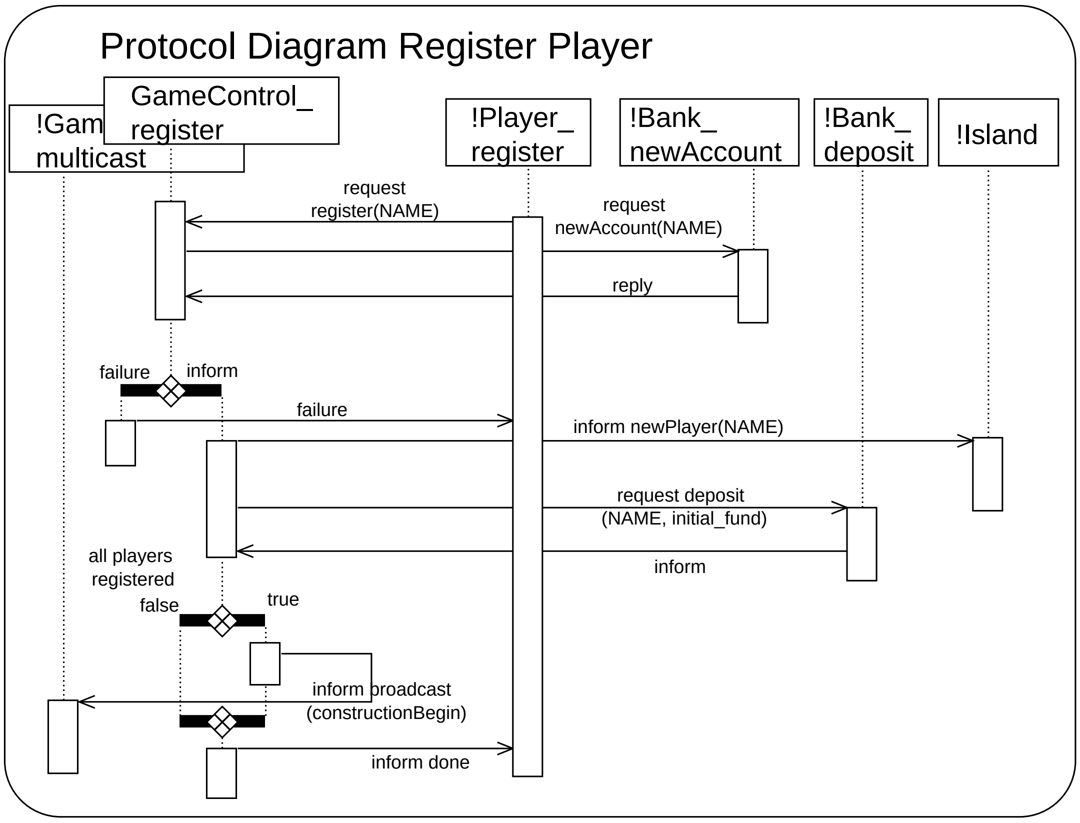
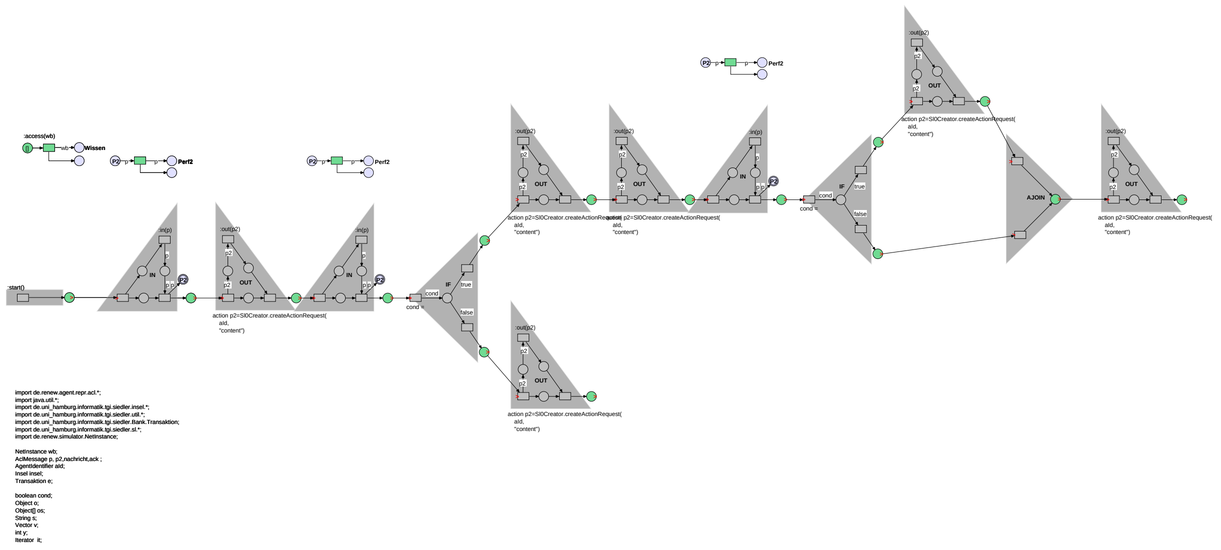


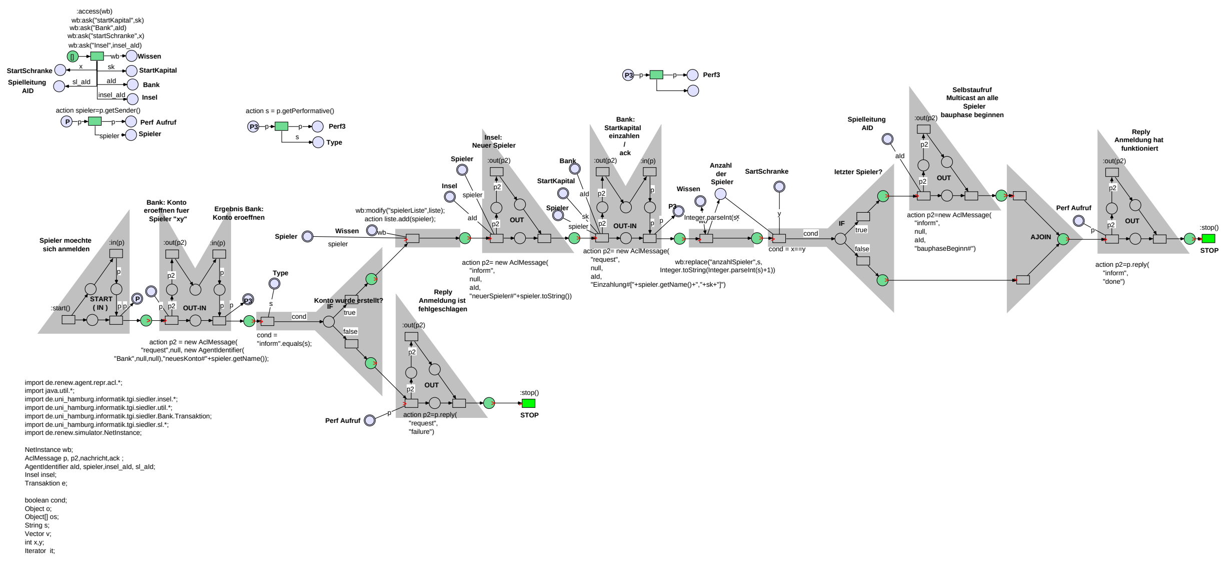
Figure 8.11: Player registration agent interaction protocol diagram; source of generation.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

```
action p2=SlOCreator.createActionRequest(
   aId,
   "content")
```

Figure 8.12: *Register* protocol skeleton generated from the source in figure 8.11



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId, spieler,insel_aId, sl_aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x,y;
Iterator  it;
```

```
:access(wb)
wb:ask("startKapital",sk)
wb:ask("Bank",aId)
wb:ask("startSchranke",x)
wb:ask("Insel",insel_aId)

action spieler=p.getSender()

action s = p.getPerformative()

action p2 = new AclMessage(
   "request",null, new AgentIdentifier(
 "Bank",null,null),"neuesKonto#"+spieler.getName());

cond =
 "inform".equals(s);

wb:modify("spielerListe",liste);
action liste.add(spieler);

action p2= new AclMessage(
   "inform",
   null,
   aId,
   "neuerSpieler#"+spieler.toString())

action p2= new AclMessage(
   "request",
   null,
   aId,
   "Einzahlung#["+spieler.getName()+","+sk+"]")

wb:replace("anzahlSpieler",s,
Integer.toString(Integer.parseInt(s)+1))

Integer.parseInt(s)

cond = x==y

action p2=new AclMessage(
   "inform",
   null,
   aId,
   "bauphaseBeginn#")

action p2=p.reply(
   "inform",
   "done")

action p2=p.reply(
   "request",
   "failure")
```

Figure 8.13: Custom made protocol for the registration of a player (*settler 1*).

### 8.3.3 Game Round

Finally, the complex agent interaction protocol diagram *round* of the game control in *settler 2* produces usable code skeletons and reveals also a minor bug in the layout/connecting mechanism (at the last *NC ajoin*). However, even though the generated code does not use the *NC out-in*, which is much smaller than the equivalent *NC out* and *NC in*, the whole net is not bigger than the custom made net. This is a satisfying result.

Figure 8.14 displays the agent interaction protocol diagrams while the following page displays in figure 8.15 both *Mulan* protocols. The generated code is displayed on the top. The custom made *Mulan* protocol – on the bottom – is displayed in its original version, divided in two parts. The end of the first part sends a message to its agent to start the second part.

Even though the Petri nets are not readable in detail, they are nevertheless displayed here to show the structure of the protocols.

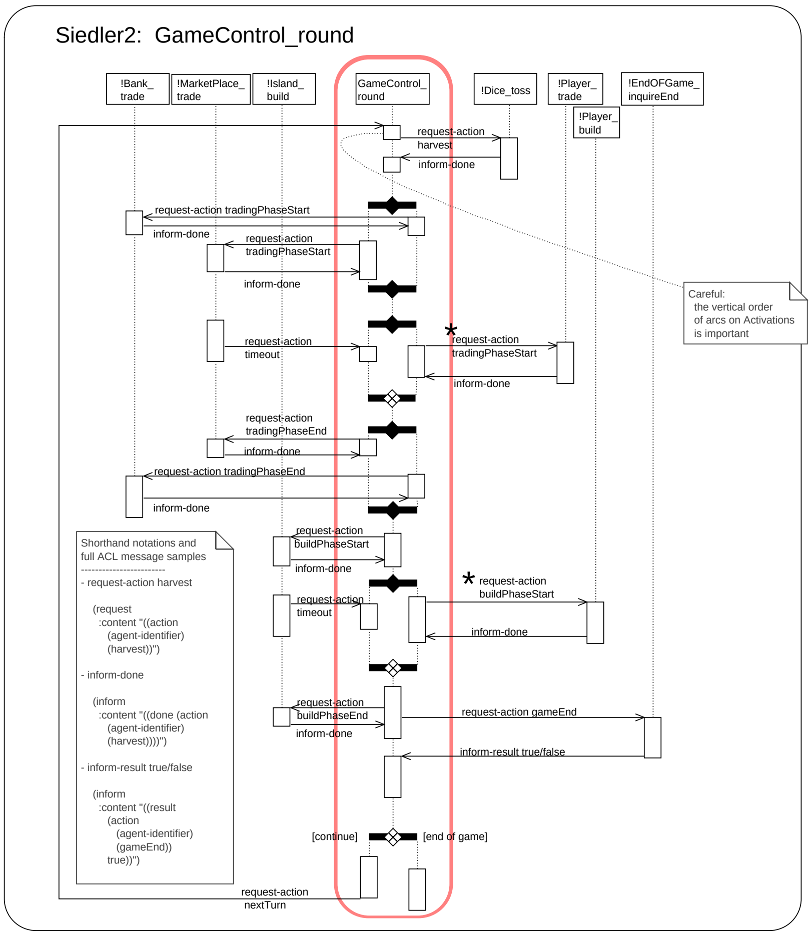
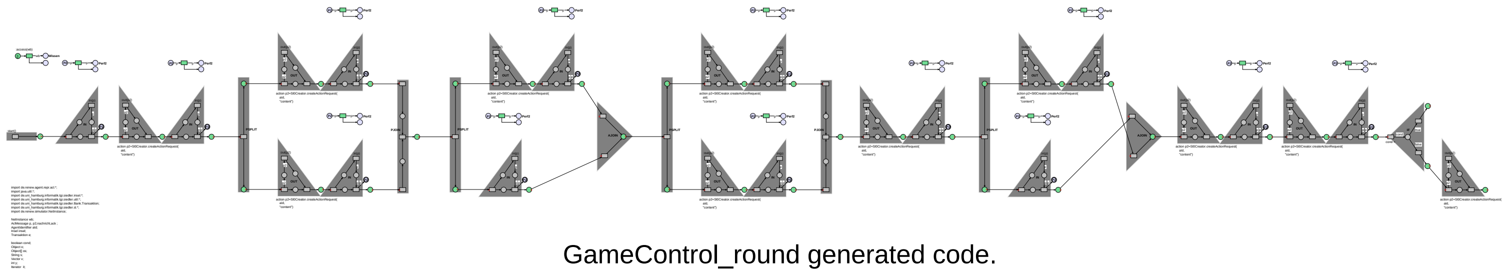
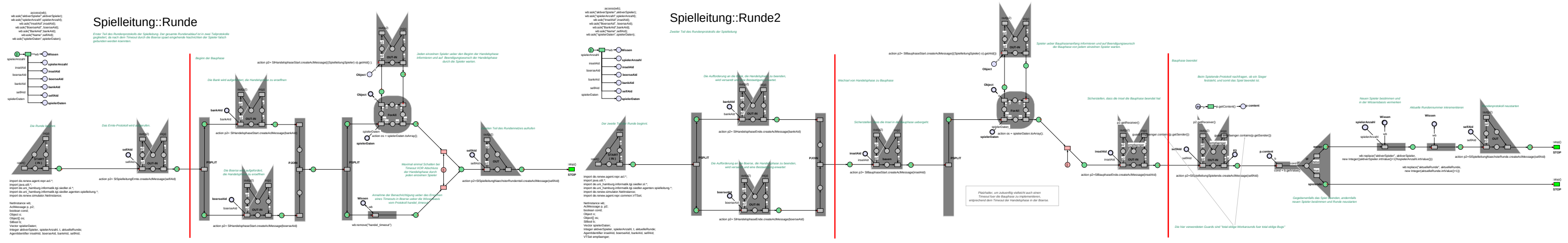

Siedler2: GameControl_round
!Bank_ trade
!MarketPlace_ trade
!Island_ build
GameControl_ round
!Dice_toss
!Player_ trade
!EndOFGame_ inquireEnd
!Player_ build
request-action harvest
inform-done
request-action tradingPhaseStart
inform-done
request-action tradingPhaseStart
inform-done
request-action timeout
request-action tradingPhaseStart
inform-done
Careful: the vertical order of arcs on Activations is important
request-action tradingPhaseEnd
inform-done
request-action tradingPhaseEnd
inform-done
Shorthand notations and full ACL message samples
- request-action harvest
(request :content "((action (agent-identifier) (harvest))")
- inform-done
(inform :content "((done (action (agent-identifier) (harvest))))")
- inform-result true/false
(inform :content "((result (action (agent-identifier) (gameEnd)) true))")
request-action buildPhaseStart
inform-done
request-action buildPhaseStart
request-action timeout
inform-done
request-action buildPhaseEnd
inform-done
request-action gameEnd
inform-result true/false
[continue]
[end of game]
request-action nextTurn

Figure 8.14: Agent interaction protocol diagram for the round of the *settler2* game. The focus lies on the game control agent. (See the appendix for a larger version.)



GameControl_round generated code.



GameControl_round original code.

Figure 8.15: Round protocol of the game control agent (*settler2*, generated code structure and custom made net)

### 8.3.4 Discussion

The resulting code skeletons of the generation process are satisfying. The layout of the generated nets for these examples is well structured so that a developer can easily read and refactor the nets. In comparison with the manually created *Mulan* protocols one difference is that the generated net does not use the *NC out-in* yet. Instead two net components are used *NC out* and *NC in*, which provide the same functionality.

To convert the generated code into executable code, several adaptation have to be made. First, the access to the knowledge base has to be adapted to gather the needed information. Second, the data that is included in the received messages has to be extracted. Third, the messages that are about to be sent have to be created. Fourth, the conditions on the alternative splits have to be defined. And fifth, on every end of the protocol a *NC stop* has to be added. All this is achieved by adding inscriptions or manipulating default inscriptions. Sometimes also new net elements have to be added. These are for example a new place to hold some kind of extracted data or a virtual place that provides access to some data for a net component.

Finally, internal processes have to be implemented manually or functionality that is provided by the code skeletons has to be altered. In the *round* protocol example of the Game control in figure 8.15 twice a message is multicasted to all the players that participate in the game. For this the *NC forall* is used in both cases to produce as many messages as there are players. To add this functionality in the generated code the skeleton must be refactored including the net components at the appropriate positions. However, this can also be done by handing the list of recipients to the message transport system, which has a multicast functionality already implemented. Whether the multicast is made explicit or done by the system is a question of design.

## 8.4 Summary

Agent interaction protocol diagrams are included seamlessly into the *Renew* environment by the drawing plug-in. The drawing capabilities as well as the look and feel match those of *Renew* so that the plug-in extends *Renew* in a natural and unobtrusive fashion. The integration of advanced modeling capabilities and development support for the construction of *Mulan* protocols is a step towards the goal of extending *Renew* and *Mulan* to serve as an integrated development environment for agent-oriented applications based on Petri nets.

The three examples demonstrate that it is possible to generate code structures from agent interaction protocol diagrams. Through the usage of net components, which impose the structure on the skeletons, the *Mulan* protocols become readable to the developers. The readability is important because the generated code has to be refactored or extended to become executable. However, the layout mechanism still needs further improvement to be able to generate very complex protocols.

# Chapter 9

# Conclusion

The aim of software engineering is to handle the problems of software development – especially – for large and complex systems. For this, solutions have been found, of which object-orientation and the unified modeling language (UML) are the most successfully applied methods, so far. Agent-orientation is a new approach that takes systems into account that are concurrent and/or distributed. The focus in agent-oriented modeling and development is laid on the agents' behavior, especially their communication or interaction. In general, modeling agent behavior can be done with interaction diagrams. In the Agent UML (AUML), the Foundation for Intelligent Physical Agents (FIPA) has proposed several extensions to the UML sequence diagrams, which take agent related modeling challenges into consideration and allow for a comfortable modeling of agent interactions.

*Mulan* (**Mul**ti-**A**gent **N**ets) is a reference architecture for a multi-agent system that is implemented with reference nets. In addition to the agents and the agent platform, also the behavior of *Mulan* agents is described by reference nets. These *Mulan* protocols, which are implementations of agent behavior during conversations, can be composed of net components to provide a unified structure.

While each *Mulan* protocol describes the behavior of one agent, agent interaction protocol diagrams focus on interactions between agents and thus offer a good overview over the whole conversation. This work presents a combination of both approaches.

The agent communications are modeled with agent interaction protocol diagrams and source code skeletons of the *Mulan* protocols are generated from the agent interaction protocol diagrams by mapping diagram elements on net components. While the overview over the conversation that shows the interactions between agents is covered by the agent interaction protocol diagrams, the behavior of one *Mulan* agent in a conversation is described by the parts of the diagrams that are related to a role descriptor and implemented as a (net component-based) *Mulan* protocol.

## 9.1 Summary

This work starts, after an introduction in chapter 1, by introducing UML in chapter 2 (Unified Modeling Language) as a means of modeling software systems. Modeling with UML is an approach to handle the complexity of software systems as well as their development process. In addition to the class and the sequence diagram, the extended sequence diagram, of which the agent interaction protocol diagram is a variant, is presented. The extensions of the UML sequence diagrams are proposed by the Foundation for Physical Agents (FIPA 2003) in the agent unified modeling language (AUML), which is aimed at an improvement of agent interaction modeling capabilities in UML.

Chapter 3 (Reference Nets and *Renew*) familiarizes the reader with Petri nets, reference nets and *Renew* (**Re**ference **Ne**t **W**orkshop), which is an editor and simulator for reference nets. *Renew* is also the basis for the reference architecture of a multi-agent system *Mulan*. Agents, multi-agent systems and *Mulan* are presented in chapter 4 (Agents and *Mulan*).

The main part of application development based on *Mulan* has do deal with the implementation of *Mulan* protocols. In chapter 5 (Structuring *Mulan* Protocols) the reader is acquainted with the *Mulan* net components, which are used to structure *Mulan* protocols, i.e. to obtain a unified and easily readable net structure.

In chapter 6 (Modeling Agent Interaction) three different ways of modeling agent interaction are compared. This comparison shows that modeling with agent interaction protocol diagrams is of advantage compared to pure sequence diagrams or reference nets.

Chapter 7 (Generating Petri Nets from AUML Diagrams) describes the operational semantics of agent interaction protocol diagrams for modeling *Mulan* agent interactions by defining a mapping from the diagrams to net component-based *Mulan* protocols. The realization of the agent interaction protocol diagram-based modeling technique and the conversion of these diagrams into *Mulan* protocols are presented.

Finally, chapter 8 (Tool Development) describes the *Renew* plug-in for drawing agent interaction protocol diagrams as well as the generation of code skeletons from these diagrams. Furthermore, three examples of generated code skeletons demonstrate the capabilities of the generation algorithm.

## 9.2 Results

The unified modeling language (UML) and Petri nets both have advantages when it comes to modeling software systems. While Petri nets have a formal definition and thus defined operational semantics, UML is actually only semi-formal. Nevertheless, UML has as its great advantages its broad acceptance, the great variety of models and the power of being the standard for application modeling.

However, UML is also very much integrated into object-oriented development. When it comes to new technologies, e.g. agent-oriented development, limitations in the modeling techniques exist. The need to extend UML is recognized by the agent-oriented research and development community and several proposals have been made to improve UML to fit agent-oriented development. This is especially true for modeling techniques focused on interactions.

UML and object-oriented development are backed up by powerful tools and software engineering methods, which support the developing process. These methods are developed to enhance the construction of large software systems and they are used and applied successfully. These methods can also be applied for software development based on high-level Petri nets. With more extensive use of these conventional techniques, the process of Petri net-based software developing can be improved. The advantages of Petri nets lie in their inherent concurrency; UML is a powerful modeling language that is well accepted and widely spread. UML and Petri nets can both contribute to the construction of large distributed and/or concurrent systems. Combining their advantages results in a powerful method to develop applications. However, the gap between traditional system modeling and the use of high-level Petri nets is relatively large. This work integrates both directions.

In this work the notion of net components and a corresponding set for *Mulan* are presented. The seamless integration of the net components into *Renew* provides a simple but powerful tool to support net component-based *Mulan* protocol implementation. The advantages of tool supported *Mulan* protocol development are clear. By using net components the *Mulan* protocols are structured and their structure is unified. This increases the readability of *Mulan* protocols and the software development is accelerated. Unification is especially desired when implementing software in project teams. Through the net components, a common language and style among developers can be achieved. Refactoring of protocols is facilitated because the components are loosely interconnected. Therefore, remodeling of *Mulan* protocols is supported.

The integration of UML-based modeling in the developing process has contributed to the clearness of the system and its overall structure. Besides the developing process, using AUML diagrams also altered the focus of development. The center of focus shifted from the agent's process to the communication between the agents. AUML provides methods for modeling interactions, which can be used to facilitate and structure the agents' interactions.

Net components are also used as templates by the generator for *Mulan* protocols from agent interaction protocol diagrams. They provide the structures of the skeletons that are generated. Thus, the generated code is readable and can be easily refactored with net components. The introduction of code generation into the development process of *Mulan* protocols alters this process significantly. If diagrams are only used in an early phase of development for communication and specification, developers tend to be less formal with their models. If, however, the diagrams serve as source for generation of code, more care has to be taken in their design as when implementing in Petri nets.

For the development of large applications on the basis of the Petri net-based reference architecture of a multi-agent system *Mulan*, tool support is needed on different levels of abstraction. This includes the construction of *Mulan* protocols, the modeling of agent interactions and the debugging of the system during development. The first two points are covered by the plug-ins for net components and agent interaction protocol diagrams. The last point is covered by the *Mulan Viewer* (Carl 2003), which visualizes the states of the agents' knowledge bases as well as the messages of the agents while the application is running.

Together the three plug-ins support the development process. They are integrated in *Renew* and they are thus three steps towards an integrated development environment (IDE) for the construction of *Mulan*-based application software.

## 9.3 Outlook

The introduction of UML-based modeling into the developing process and the unification of net structures turned out to be a successful approach. Nevertheless, the integration of conventional methods (UML), new approaches (AUML) and development of software with Petri nets can be driven further. More support is needed at all stages of development. These stages are modeling, specification, implementation, debugging, testing and documentation. In addition, integration is also needed for project management and team work support.

Regarding the agent interaction protocol diagrams and the generation of code from these diagrams, many functions can still be added to the tool. The generation of the message helper classes could be integrated in the plug-in's generation process – maybe as another plug-in for *Renew*. This would change another frequent task to an automatic generation of (Java) code. The expressions for the creation of the messages in the nets could then be added automatically in the messaging net components. Together with the inclusion of actions, data handling and automatic knowledge base manipulation in the generating process, the diagrams could be executed without manual refactoring of the generated code.

The generation algorithm creates the inscriptions. In order to be able to include these inscriptions in the generated code, the net components have to be parameterizable. This could be achieved through place holders in the templates (net components).

The handling of net components – especially while refactoring the *Mulan* protocols – can be improved by introducing a group behavior of the net elements that belong to a net component. If a move of one of the net elements that belongs to a net component results in a move of the whole net component, the handling would be much better and more intuitive. This behavior can also be applied to deletion. However, since the inscriptions of the Petri net elements have to be adjustable, the usual grouping functionality in *Renew* is not usable for this kind of grouping. Another grouping functionality has to be integrated into *Renew*. In this case – as in many other cases – the question rises where to include the functionality. Is this functionality general and should therefore be included in the main components of *Renew* or is it exclusively used by a plug-in and thus excluded from the main parts of *Renew*? This is a question of design matters and cannot be answered here.

Another improvement that should be realized is an exchangeable generator algorithm. This could be implemented as a *Renew* plug-in that extends the functionality of the diagram plug-in. For its realization a new set of net components has to be created and a new mapping from the diagram elements to these net components has to be defined. The generation functionality can easily be separated from the plug-in because the mapping is defined in the peer classes of the diagram elements. Nevertheless, some parts of the algorithm that are included in those classes have to be separated from the diagram element classes in order to use an exchangeable generator.

The drawing capabilities of the plug-in can be improved by adding more figure handles so that more elements can be drawn in diagrams effortlessly. In addition, diagram components – in analogy to net components – can be defined to provide whole structures of diagrams as a component-based approach to create the agent interaction protocol diagrams. Furthermore, the arrangements of the diagrams while editing diagram elements should not hamper the layout.

Regarding the modeling of agent interactions, the techniques that are proposed in the AUML are satisfying. However, even with the proposed extensions of UML, some complicated situations are not well supported in conversation modeling.

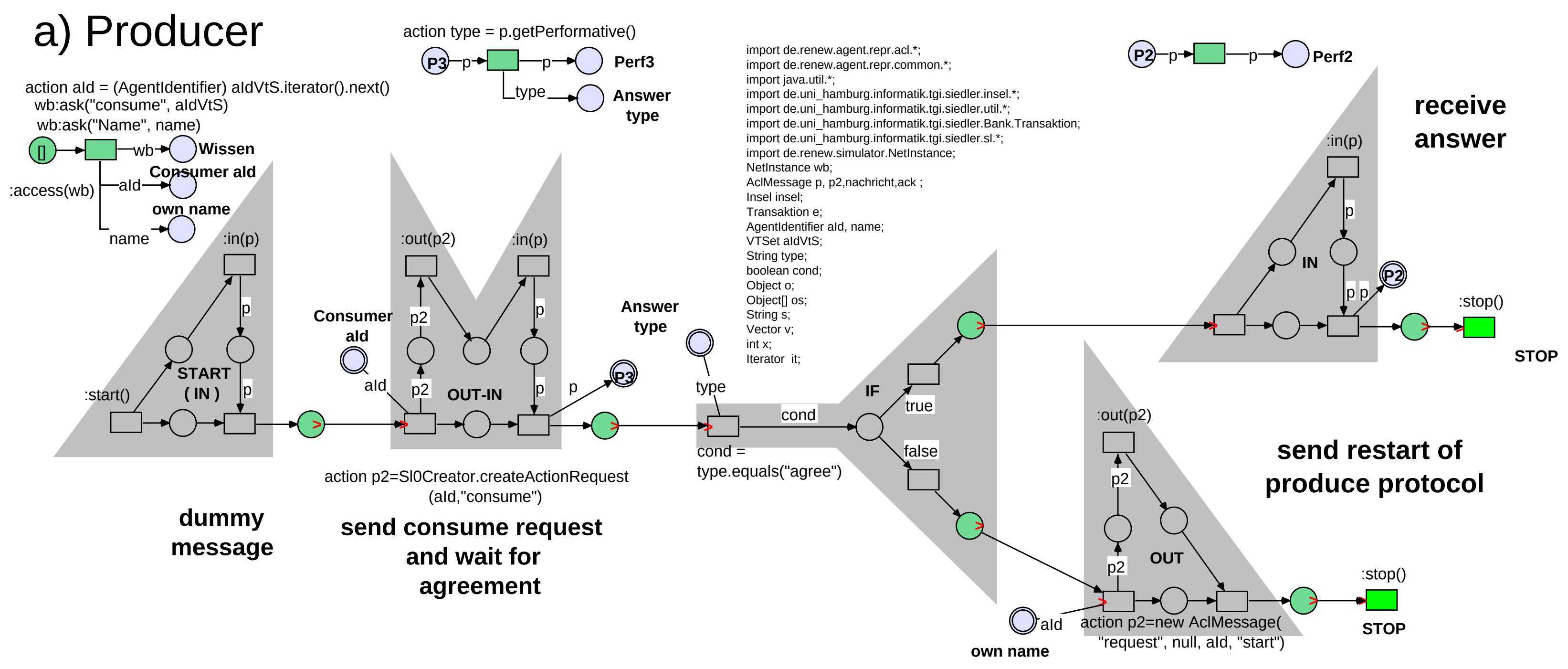
Net components were presented here as a part of *Mulan* protocols. Nevertheless, it is possible to apply the same principles to other domains. As an example the obvious solution for a Petri net-based workflow engine is mentioned here (see Moldt and Rölke 2003 and Braker 2004). It is possible to realize workflow patterns for that domain using the *Renew* extension presented in section 5.2.2.

The development process of an application in a team is a distributed process. The developer team has to take into account that the organization of this process is somehow complex and occupies much of the development time. Since we develop distributed processes as multi-agent systems, it seems obvious that an IDE for application development could be realized as multi-agent system.

# Appendix A

# Images

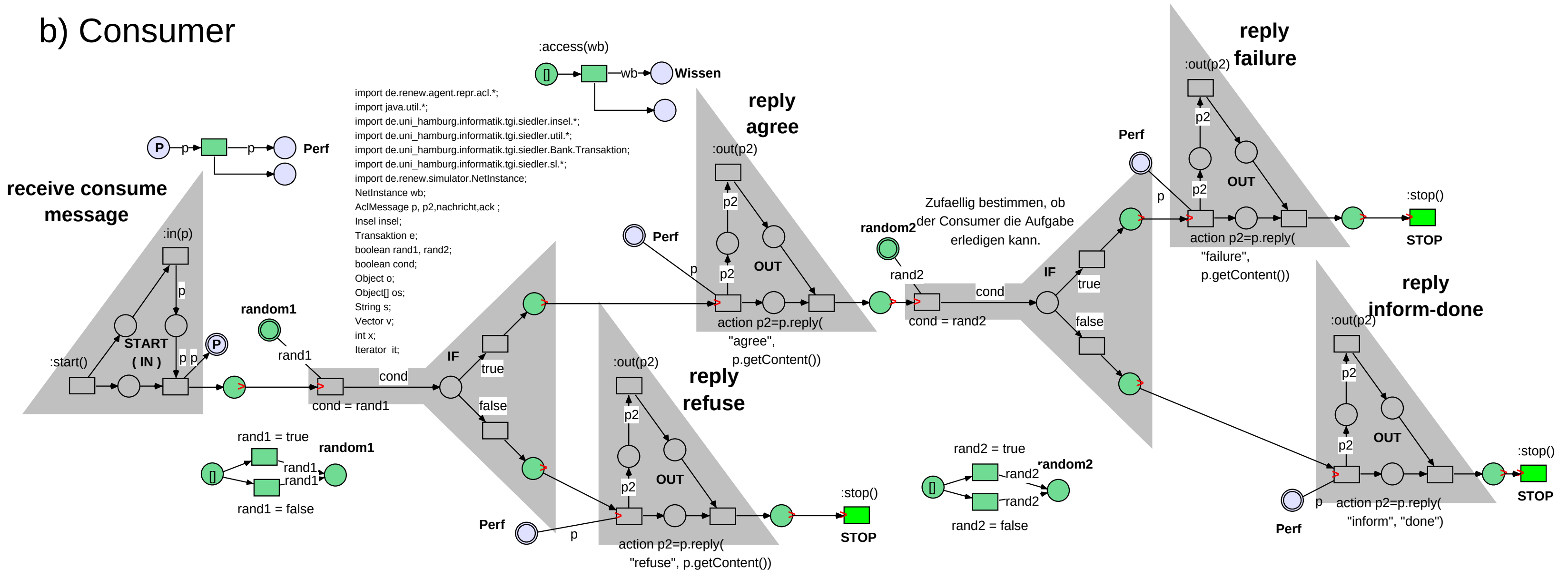
On the following pages several selected *Mulan* protocols and the agent interaction protocol diagrams of section 8.3.3 are again presented in a larger scale. This is done, so that the nets can be read. The images have not been altered except that the color of the net component shadows have been changed and the large nets of figures A.7 and A.8 have been wrapped. However, especially these large nets are not readable in detail, but the important fact is that the layout and structure of the *Mulan* protocols can be recognized.



```
import de.renew.agent.repr.acl.*;
import de.renew.agent.repr.common.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
AgentIdentifier aId, name;
VTSet aIdVtS;
String type;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

Figure A.1: *Produce* protocol, net component version.

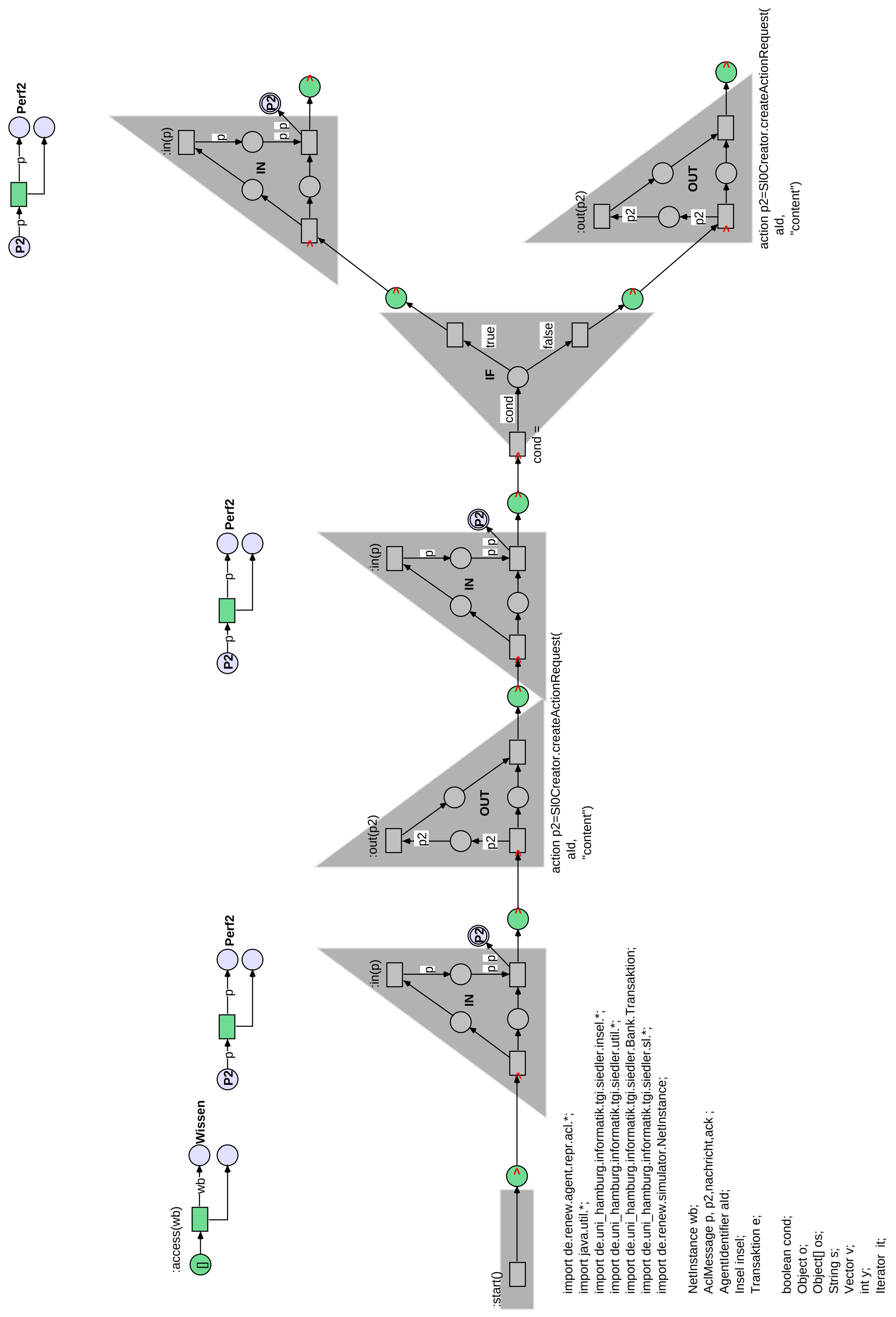
b) Consumer



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;
NetInstance wb;
AclMessage p, p2,nachricht,ack ;
Insel insel;
Transaktion e;
boolean rand1, rand2;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x;
Iterator  it;
```

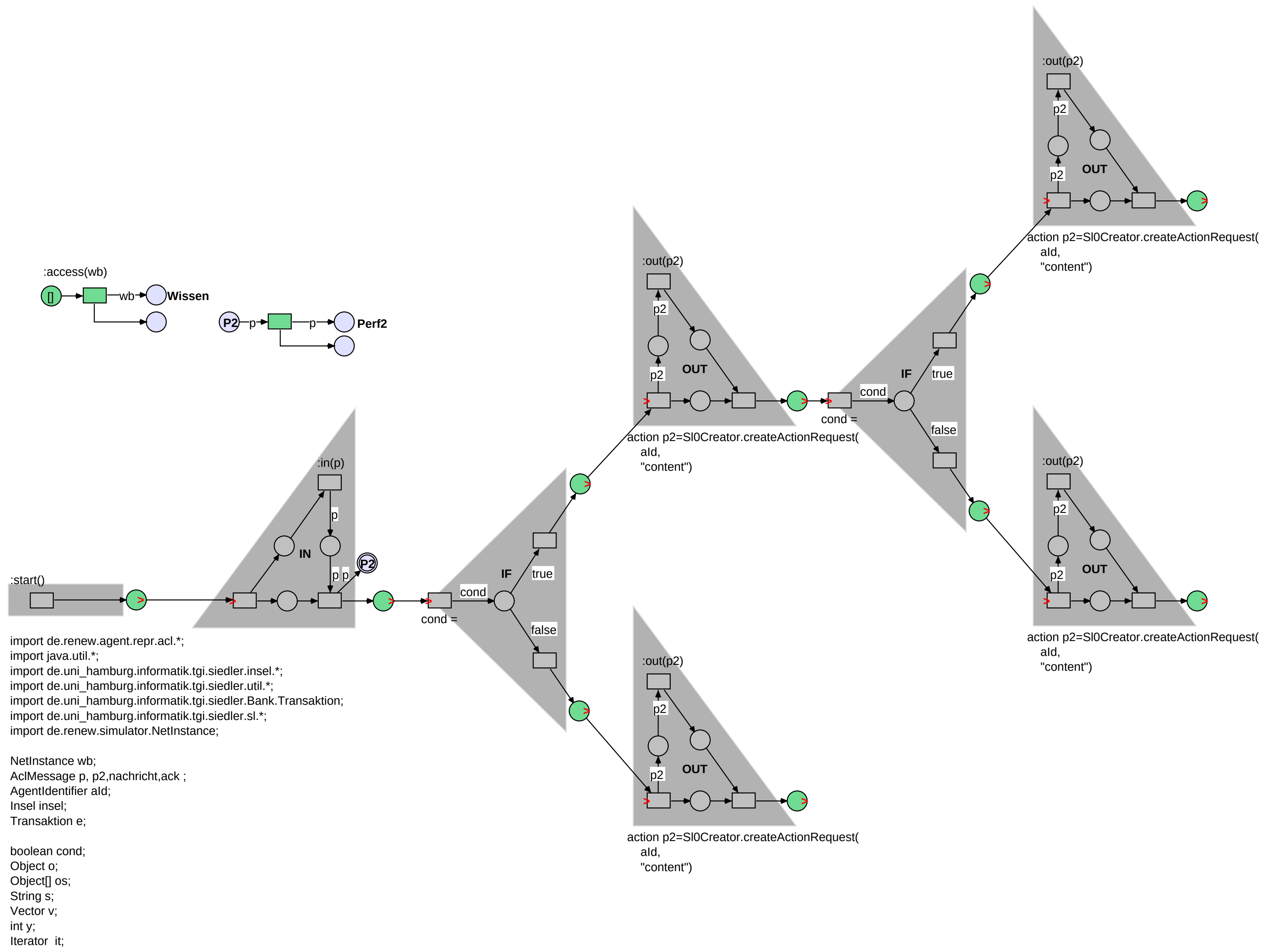
Zufaellig bestimmen, ob der Consumer die Aufgabe erledigen kann.

Figure A.2: *Consume* protocol, net component version.



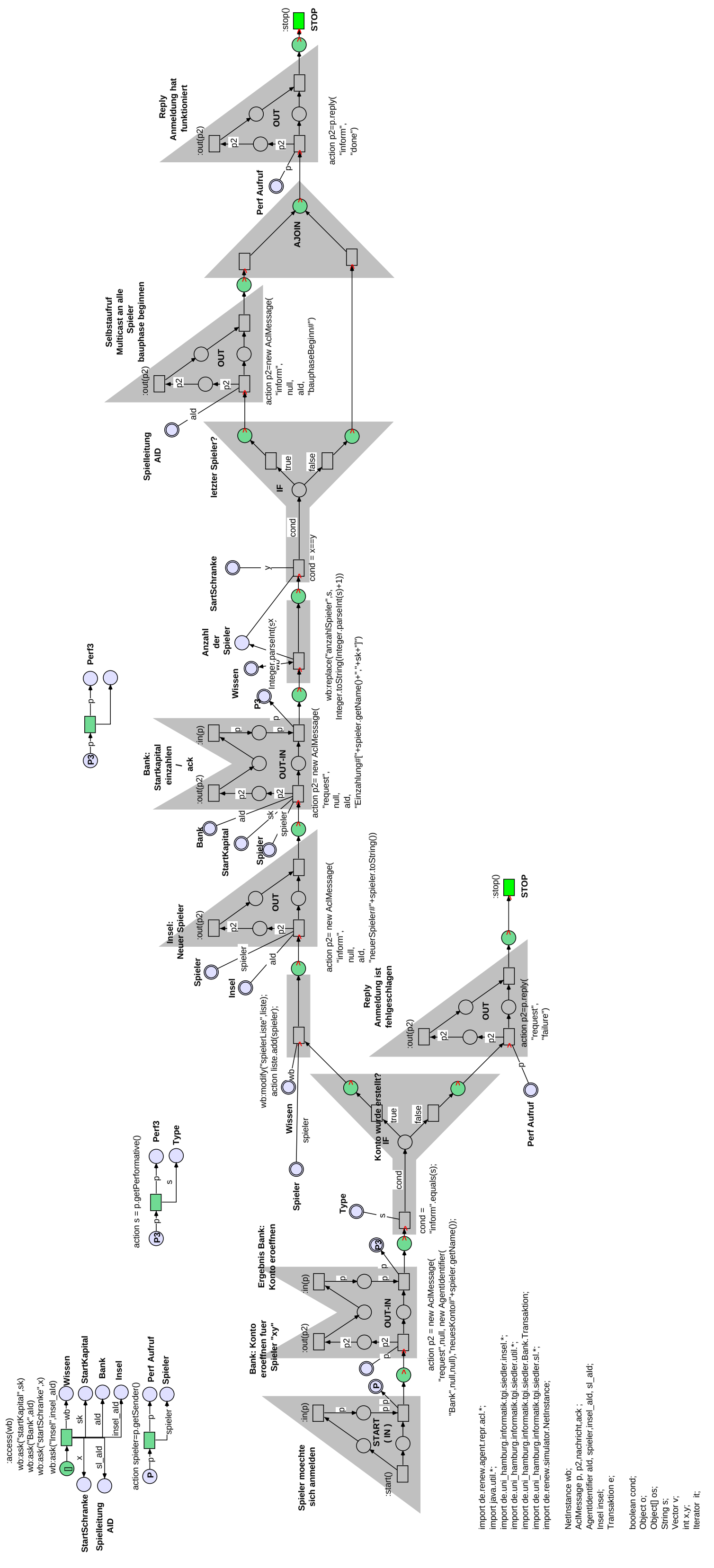
```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

Figure A.3: Generated code skeleton for the *produce* protocol.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

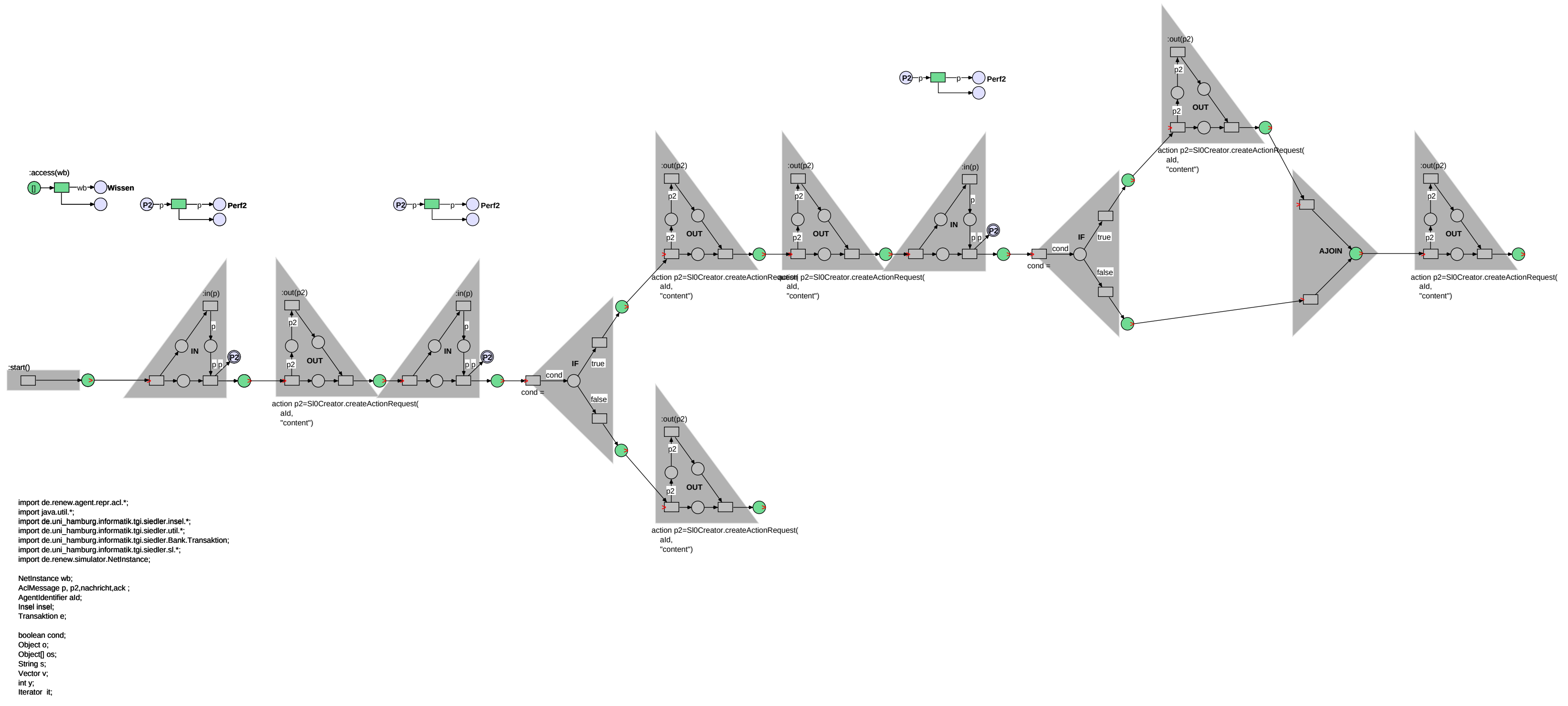
Figure A.4: Generated code skeleton for the *consume* protocol.



:access(wb)
wb:ask("startKapital",sk)
wb:ask("Bank",aId)
wb:ask("startSchranke",x)
wb:ask("Insel",insel_aId)
Wissen
StartSchranke
StartKapital
Spielleitung AID
Bank
Insel
action spieler=p.getSender()
Perf Aufruf
Spieler
action s = p.getPerformative()
Perf3
Type
Spieler moechte sich anmelden
:start()
START ( IN )
:in(p)
Bank: Konto eroeffnen fuer Spieler "xy"
Ergebnis Bank: Konto eroeffnen
:out(p2)
OUT-IN
action p2 = new AclMessage( "request",null, new AgentIdentifier( "Bank",null,null),"neuesKonto#"+spieler.getName());
cond = "inform".equals(s);
Konto wurde erstellt?
IF
true
false
Reply Anmeldung ist fehlgeschlagen
OUT
action p2=p.reply( "request", "failure")
:stop()
STOP
wb:modify("spielerListe",liste);
action liste.add(spieler);
Insel: Neuer Spieler
action p2= new AclMessage( "inform", null, aId, "neuerSpieler#"+spieler.toString())
Bank: Startkapital einzahlen / ack
action p2= new AclMessage( "request", null, aId, "Einzahlung#["+spieler.getName()+","+sk+"]")
Anzahl der Spieler
Integer.parseInt(s)
wb:replace("anzahlSpieler",s, Integer.toString(Integer.parseInt(s)+1))
SartSchranke
cond = x==y
letzter Spieler?
Selbstaufruf Multicast an alle Spieler bauphase beginnen
action p2=new AclMessage( "inform", null, aId, "bauphaseBeginn#")
AJOIN
Reply Anmeldung hat funktioniert
action p2=p.reply( "inform", "done")

```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId, spieler,insel_aId, sl_aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int x,y;
Iterator  it;
```

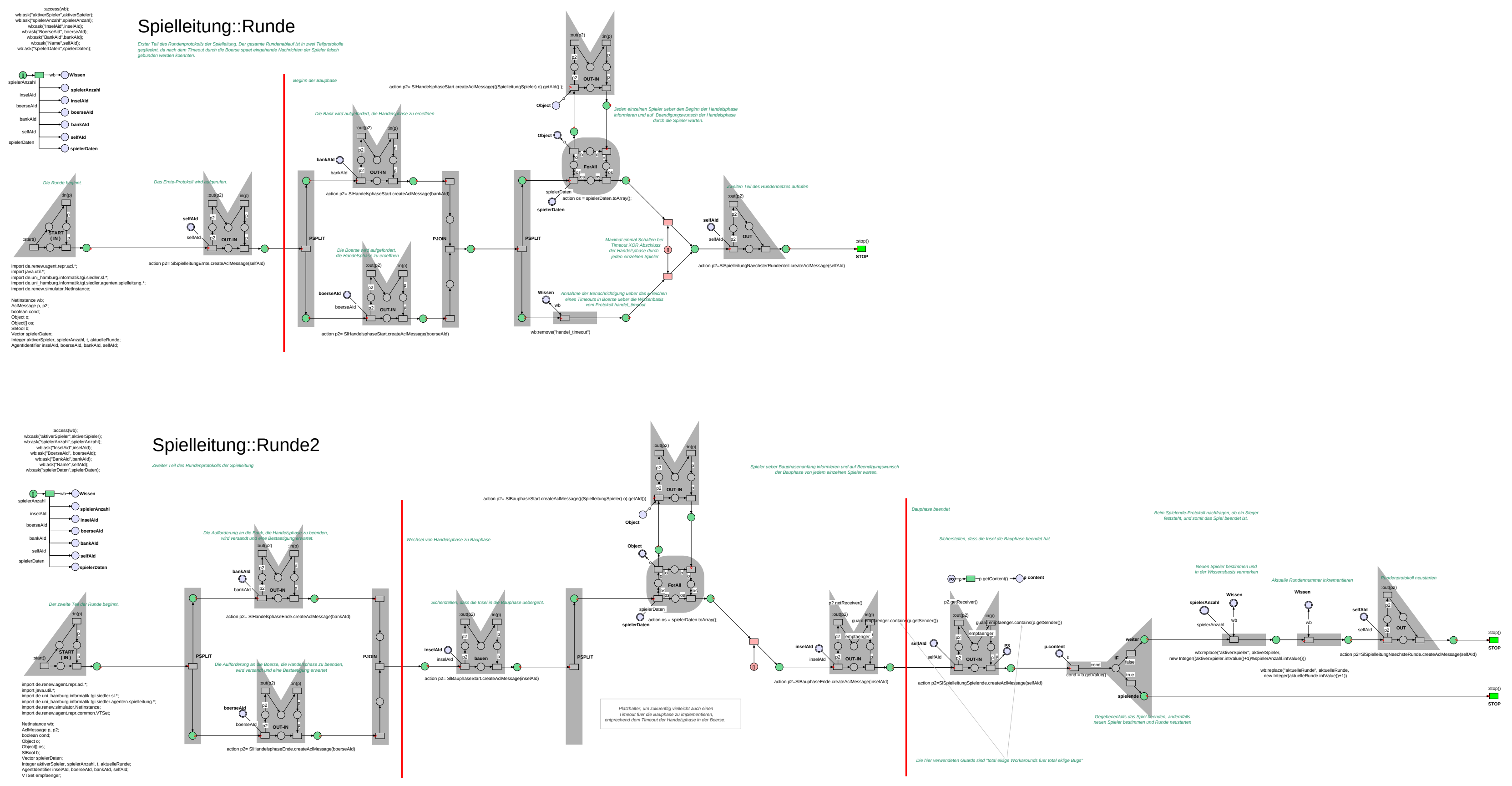
Figure A.5: Game control *register* protocol, manual net component version.



```
action p2=Sl0Creator.createActionRequest(
    aId,
    "content")
```

```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

Figure A.6: Generated code skeleton for the *register* protocol.


Spielleitung::Runde
Erster Teil des Rundenprotokolls der Spielleitung. Der gesamte Rundenablauf ist in zwei Teilprotokolle gegliedert, da nach dem Timeout durch die Boerse spaet eingehende Nachrichten der Spieler falsch gebunden werden koennten.
Wissen
spielerAnzahl
inselAId
boerseAId
bankAId
selfAId
spielerDaten
Die Runde beginnt.
:start()
:in(p)
START ( IN )
Das Ernte-Protokoll wird aufgerufen.
:out(p2)
OUT-IN
action p2= SlSpielleitungErnte.createAclMessage(selfAId)
Beginn der Bauphase
PSPLIT
PJOIN
Die Bank wird aufgefordert, die Handelsphase zu eroeffnen
action p2= SlHandelsphaseStart.createAclMessage(bankAId)
Die Boerse wird aufgefordert, die Handelsphase zu eroeffnen
action p2= SlHandelsphaseStart.createAclMessage(boerseAId)
action p2= SlHandelsphaseStart.createAclMessage(((SpielleitungSpieler) o).getAId() );
Object
ForAll
action os = spielerDaten.toArray();
Jeden einzelnen Spieler ueber den Beginn der Handelsphase informieren und auf Beendigungswunsch der Handelsphase durch die Spieler warten.
Maximal einmal Schalten bei Timeout XOR Abschluss der Handelsphase durch jeden einzelnen Spieler
Annahme der Benachrichtigung ueber das Erreichen eines Timeouts in Boerse ueber die Wissenbasis vom Protokoll handel_timeout.
wb:remove("handel_timeout")
Zweiten Teil des Rundennetzes aufrufen
OUT
action p2=SlSpielleitungNaechsterRundenteil.createAclMessage(selfAId)
:stop()
STOP
Spielleitung::Runde2
Zweiter Teil des Rundenprotokolls der Spielleitung
Der zweite Teil der Runde beginnt.
Die Aufforderung an die Bank, die Handelsphase zu beenden, wird versandt und eine Bestaetigung erwartet.
action p2= SlHandelsphaseEnde.createAclMessage(bankAId)
Die Aufforderung an die Boerse, die Handelsphase zu beenden, wird versandt und eine Bestaetigung erwartet
action p2= SlHandelsphaseEnde.createAclMessage(boerseAId)
Wechsel von Handelsphase zu Bauphase
Sicherstellen, dass die Insel in die Bauphase uebergeht.
bauen
action p2= SlBauphaseStart.createAclMessage(inselAId)
action p2= SlBauphaseStart.createAclMessage(((SpielleitungSpieler) o).getAId())
Spieler ueber Bauphasenanfang informieren und auf Beendigungswunsch der Bauphase von jedem einzelnen Spieler warten.
Platzhalter, um zukuenftig vielleicht auch einen Timeout fuer die Bauphase zu implementieren, entprechend dem Timeout der Handelsphase in der Boerse.
p2.getReceiver()
guard empfaenger.contains(p.getSender())
empfaenger
action p2=SlBauphaseEnde.createAclMessage(inselAId)
Bauphase beendet
Sicherstellen, dass die Insel die Bauphase beendet hat
P3
p.getContent()
p content
action p2=SlSpielleitungSpielende.createAclMessage(selfAId)
Die hier verwendeten Guards sind "total eklige Workarounds fuer total eklige Bugs"
p.content
cond = b.getValue()
IF
false
true
weiter
spielende
Gegebenenfalls das Spiel beenden, andernfalls neuen Spieler bestimmen und Runde neustarten
Beim Spielende-Protokoll nachfragen, ob ein Sieger feststeht, und somit das Spiel beendet ist.
Neuen Spieler bestimmen und in der Wissensbasis vermerken
wb:replace("aktiverSpieler", aktiverSpieler, new Integer((aktiverSpieler.intValue()+1)%spielerAnzahl.intValue()))
Aktuelle Rundennummer inkrementieren
wb:replace("aktuelleRunde", aktuelleRunde, new Integer(aktuelleRunde.intValue()+1))
Rundenprotokoll neustarten
action p2=SlSpielleitungNaechsteRunde.createAclMessage(selfAId)

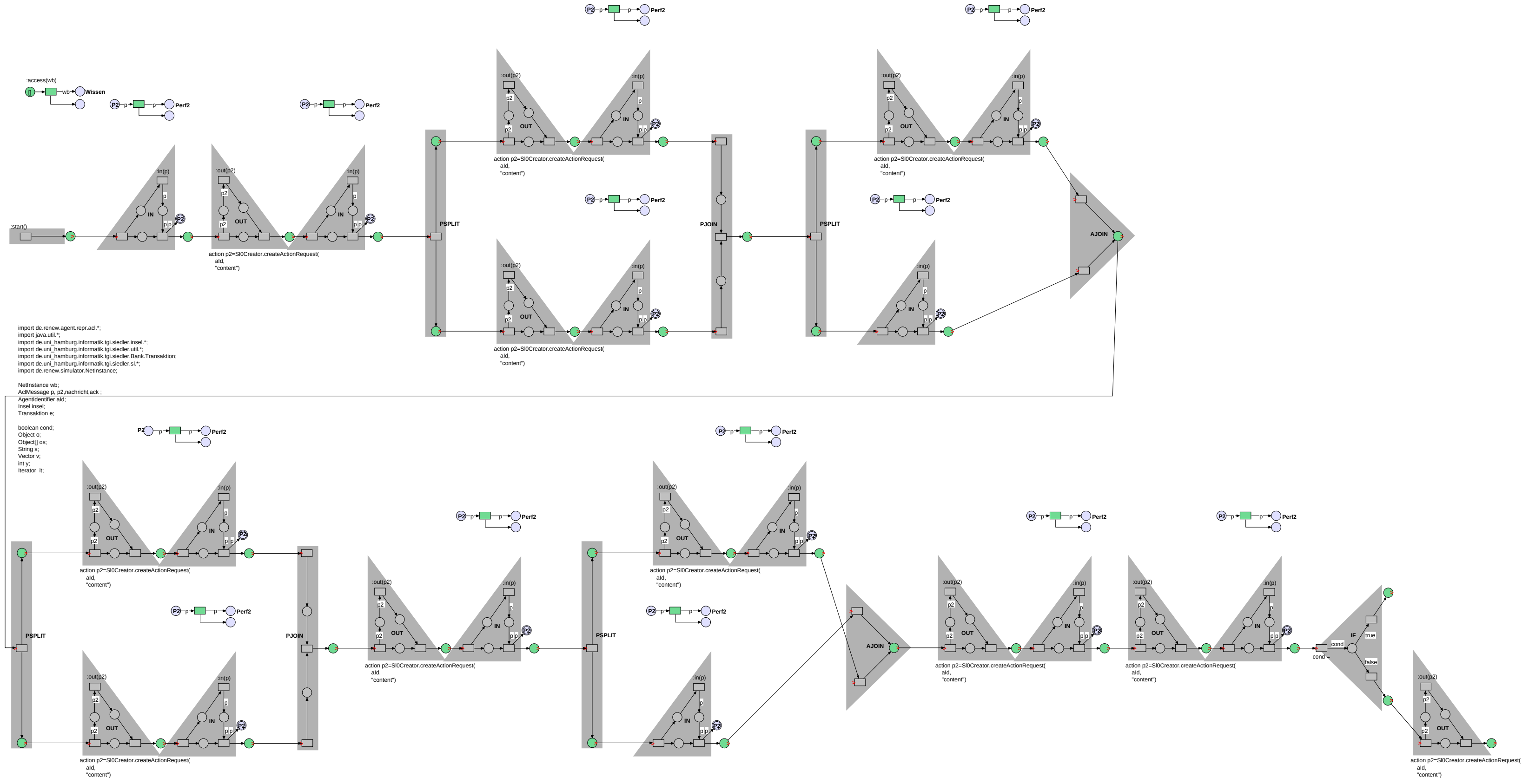
```
:access(wb);
wb:ask("aktiverSpieler",aktiverSpieler);
wb:ask("spielerAnzahl",spielerAnzahl);
wb:ask("InselAid",inselAId);
wb:ask("BoerseAid", boerseAId);
wb:ask("BankAid",bankAId);
wb:ask("Name",selfAId);
wb:ask("spielerDaten",spielerDaten);
```

```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.uni_hamburg.informatik.tgi.siedler.agenten.spielleitung.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2;
boolean cond;
Object o;
Object[] os;
SlBool b;
Vector spielerDaten;
Integer aktiverSpieler, spielerAnzahl, t, aktuelleRunde;
AgentIdentifier inselAId, boerseAId, bankAId, selfAId;
```

```
:access(wb);
wb:ask("aktiverSpieler",aktiverSpieler);
wb:ask("spielerAnzahl",spielerAnzahl);
wb:ask("InselAid",inselAId);
wb:ask("BoerseAid", boerseAId);
wb:ask("BankAid",bankAId);
wb:ask("Name",selfAId);
wb:ask("spielerDaten",spielerDaten);
```

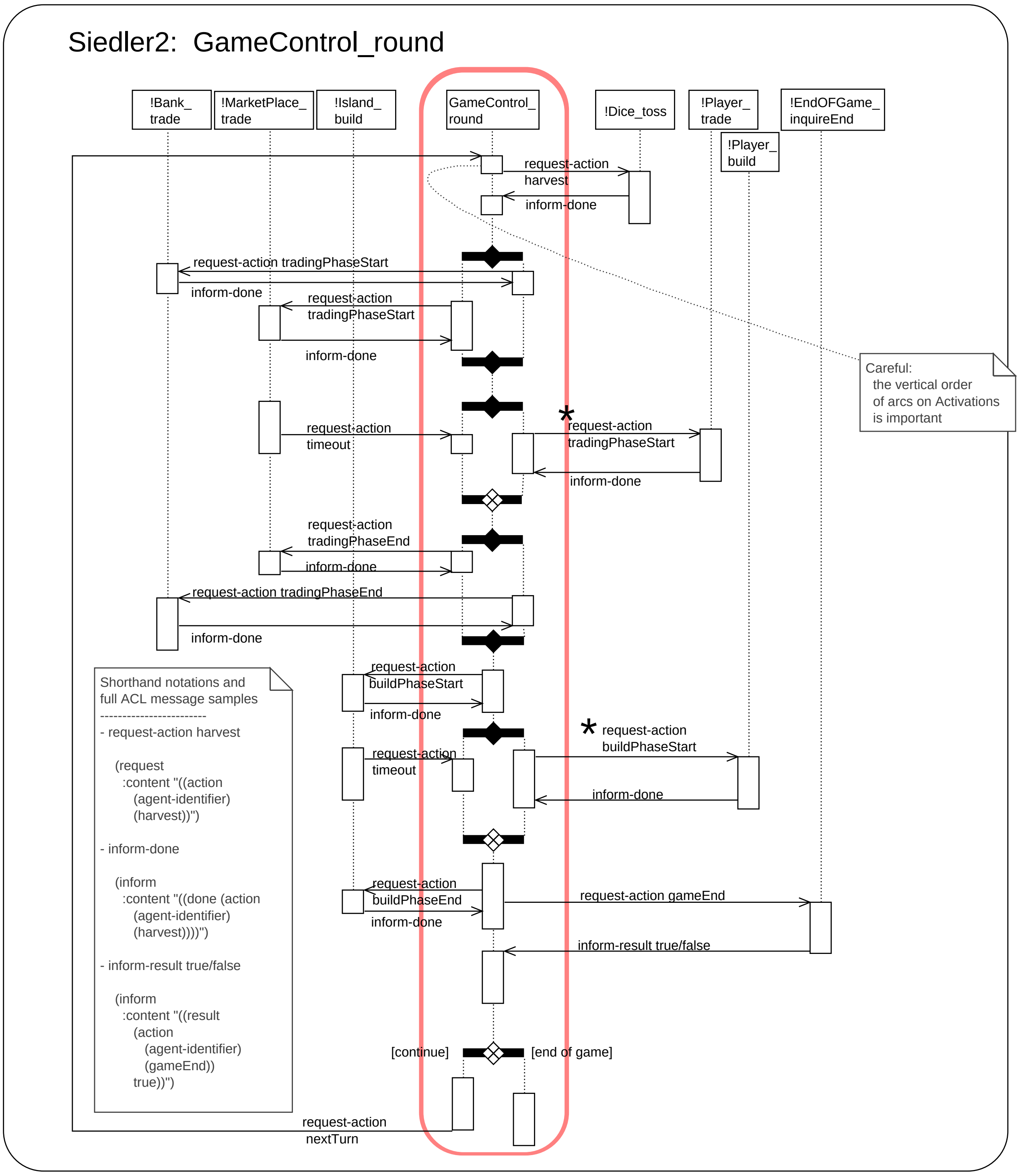
```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.uni_hamburg.informatik.tgi.siedler.agenten.spielleitung.*;
import de.renew.simulator.NetInstance;
import de.renew.agent.repr.common.VTSet;

NetInstance wb;
AclMessage p, p2;
boolean cond;
Object o;
Object[] os;
SlBool b;
Vector spielerDaten;
Integer aktiverSpieler, spielerAnzahl, t, aktuelleRunde;
AgentIdentifier inselAId, boerseAId, bankAId, selfAId;
VTSet empfaenger;
```

Figure A.7: Game control *round* protocol, manual net component version.



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.uni_hamburg.informatik.tgi.siedler.insel.*;
import de.uni_hamburg.informatik.tgi.siedler.util.*;
import de.uni_hamburg.informatik.tgi.siedler.Bank.Transaktion;
import de.uni_hamburg.informatik.tgi.siedler.sl.*;
import de.renew.simulator.NetInstance;

NetInstance wb;
AclMessage p, p2,nachricht,ack ;
AgentIdentifier aId;
Insel insel;
Transaktion e;

boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y;
Iterator  it;
```

Figure A.8: Generated code skeleton for the *round* protocol.



Shorthand notations and full ACL message samples
------------------------
- request-action harvest

```
(request
  :content "((action
     (agent-identifier)
     (harvest))")
```

- inform-done

```
(inform
  :content "((done (action
     (agent-identifier)
     (harvest))))")
```

- inform-result true/false

```
(inform
  :content "((result
     (action
        (agent-identifier)
        (gameEnd))
     true))")
```

Figure A.9: Agent interaction protocol diagram for the round of the *settler 2* game. The focus lies on the game control agent.

# Appendix B

# Used Tools

This text is written with the support of:

- *Xemacs*
- *LaTeX*

The images are produced with:

- *Renew*
- *Ghostscript*
- *The Gimp*

Operating systems:

- *Solaris*
- *Linux*

Programming environments:

- *Eclipse*
- *Renew*
- Java 1.4.1
- *Poseidon*

All sequence diagrams and all agent interaction protocol diagrams were drawn with the diagram drawing plug-in (within *Renew*) that is presented in chapter 8.

# Appendix C

# Abbreviations

| **Abbreviation** | **Full Name** |
|---|---|
| AI | Artificial intelligence |
| AIP | Agent interaction protocol (diagram) |
| API | Application programmers interface |
| AUML | Agent UML / agent unified modeling language |
| CAPA | Concurrent agent platform Architecture |
| CASE | Computer-aided software engineering |
| CPN | Coloured Petri net |
| (E)BNF | (Extended) Backus-Naur Form |
| FIPA | Foundations for Intelligent Physical Agents |
| GUI | Graphical user interface |
| IDE | Integrated development environment |
| IP | Interaction protocol |
| LNCS | Lecture Notes in Computer Science |
| MTS | Message transport service |
| *Mulan* | **Mul**ti-**A**gent **N**ets |
| NC | Net component |
| OMG | Object Management Group |
| P/T-net | Place/Transition- net |
| *Renew* | **Re**ference **Ne**t **W**orkshop |
| SL0 | Semantic language level 0 |
| SWE | Software engineering |
| TCP/IP | Transport control protocol / internet protocol |
| UML | Unified modeling language |

Table C.1: Abbreviations used in this work.

# Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers. Principles, Techniques, and Tools* (World Student Series Edition ed.). Addison-Wesley series in Computer Science. Reading, Massachusetts: Addison-Wesley Publishing Company.

ArgoUML (2003, November). Markus Klink. `http://sourceforge.net/projects/argouml/`.

AUML (2003). Agent UML. Webpage. `http://www.auml.org/`.

Baier, T. (2000). Ein Metamodell für eine UML-basierte Entwicklungsumgebung für verteilte und nebenläufige Systeme. Diplomarbeit, University of Hamburg, Department of Computer Science. `http://vsis-www.informatik.uni-hamburg.de/publications/readstu.phtml/DA/36/Diplomarbeit-Toby-B`

Bell, R. (1998, March). Code generation from object models. *Embedded Systems Programming 11*. `http://www.embedded.com/98/9803fe3.htm`.

Bergenti, F., O. Shehory, and A. Sturm (2003, February). Agent-oriented software engineering. In M. Luck, W. van der Hoek, and C. Sierra (Eds.), *AgentLink easss 2003*, pp. 125–158. AgentLink.

Booch, G. (1993). *Object-Oriented Design* (2. ed.). Benjamin/Cummings Redwood City, CA.

Booch, G., J. Rumbaugh, and I. Jacobson (1999). *The Unified Modeling Language User Guide: The ultimate tutorial to the UML from the original designers.* Addison-Wesley Object Technology Series. Reading, Massachusetts: Addison-Wesley.

Bosch, T., O. Gries, H. Kausch, M. Klenski, K. Lehmann, M. Morales, V. Seegert, and A. Vilner (2002). *Agentenorientierte Implementierung des Spiels "Die Siedler von Catan"*. Internal report, University of Hamburg, Department of Computer Science.

Braker, M. (2004, March). Workflowpetrinetze Hierarchisierung mittels Netzen-in-Netzen. Diplomarbeit, University of Hamburg, Department of Computer Science. not yet published.

Cabac, L. (2002). *Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen.* Studienarbeit, University of Hamburg, Department of Computer Science.

Cabac, L., D. Moldt, and H. Rölke (2003, June). A proposal for structuring Petri net-based agent interaction protocols. In W. van der Aalst and E. Best (Eds.), *Lecture*

*Notes in Computer Science: 24th International Conference on Application and Theory of Petri Nets, ICATPN 2003, Netherlands, Eindhoven*, Volume 2679, Berlin Heidelberg: Springer, pp. 102–120.

Carl, T. (2003, August). Evaluation und beispielhafte Erweiterung einer referenznetzbasierten Agentenumgebung. Studienarbeit, University of Hamburg, Department of Computer Science.

Christensen, S. and N. D. Hansen (1992, April). Coloured Petri nets extended with channels for synchronous communication. Technical Report DAIMI PB–390, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark.

Costello, R. B. (Ed.) (1996). *Webster's College Dictionary.* New York: Random House.

CVS (2003, November). Concurrent versioning system. `http://www.cvshome.org/`.

Dewey, J. (1980). *Art as Experience.* New York: Perigee Books, The Berkley Publishing Group.

Duvigneau, M. (2002, December). Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diplomarbeit, University of Hamburg, Vogt-Kölln Str. 30, 22527 Hamburg, Germany. `http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/wimis/duvigneau/diplomarbeit.ps.gz`.

Duvigneau, M., D. Moldt, and H. Rölke (2003, January). Concurrent architecture for a multi-agent platform. In F. Giunchiglia, J. Odell, and G. Weiß (Eds.), *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002*, Volume 2585, Berlin Heidelberg: Springer, pp. 59–72.

Edelmann, W. (1994). *Lernpsychologie: Eine Einführung* (4. ed.). Weinheim: Psychologie Verlags Union.

Ferber, J. (1999). *Multi-Agents Systems - An Introduction to Distributed Artificial Intelligence.* Boston: Addison-Wesley.

FIPA (2001a, August). FIPA Contract Net Interaction Protocol. `http://www.fipa.org/specs/fipa00029/XC00029F.pdf`.

FIPA (2001b, August). FIPA English Auction Protocol. `http://www.fipa.org/specs/fipa00031/XC00031F.pdf`.

FIPA (2001c, August). FIPA Interaction Protocol Library Specification. `http://www.fipa.org/specs/fipa00025/XC00025E.pdf`.

FIPA (2001d, August). FIPA Interaction Protocols Repository. `http://www.fipa.org/repository/ips.php3`.

FIPA (2001e, August). FIPA Request Interaction Protocol. `http://www.fipa.org/specs/fipa00026/SC00026H.pdf`.

FIPA (2003). Foundation for Intelligent Physical Agents. `http://www.fipa.org`.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns – Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley Publishing Company.

Gamma, E., J. Quante, and W. Kaiser (2003, November). JHotDraw. `http://jhotdraw.sourceforge.net/`.

Gentleware (2003, October). Gentleware Homepage. `http://www.gentleware.com/`.

Girault, C. and R. Valk (2003). *Petri Nets for Systems Engineering — A Guide to Modeling, Verification, and Applications.* Berlin: Springer Verlag.

Hoare, C. A. R. and N. Wirth (1973). An axiomatic definition of the programming language Pascal. *Acta Informatica 2*, 335–355.

Jacob, T. (2002). Implementation einer sicheren und rollenbasierten Workflow-Managementkomponente für ein Petrinetzwerkzeug. Diplomarbeit, University of Hamburg, Department of Computer Science.

Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard (1992). *Object-oriented Software Engineering; A Use Case Driven Approach.* Wokingham, England: Addison-Wesley.

Jennings, N. R. and M. J. Wooldridge (1998). Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge (Eds.), *Agent Technology: Foundations, Applications and Markets*, Berlin Heidelberg New York, pp. 3–28. Springer.

Jensen, K. (1996). *Coloured Petri Nets* (2nd ed.), Volume 1. Berlin: Springer-Verlag.

Jessen, E. and R. Valk (1987). *Rechensysteme; Grundlagen der Modellbildung.* Berlin: Springer-Verlag.

Köhler, M., D. Moldt, and H. Rölke (2001). Modeling the behaviour of Petri net agents. In *Proceedings of the 22nd Conference on Application and Theory of Petri Nets*, pp. 224–241.

Kummer, O. (2002). *Referenznetze.* Ph. D. thesis, University of Hamburg, Department of Computer Science, Logos-Verlag, Berlin.

Kummer, O., F. Wienberg, and M. Duvigneau (2001a). Renew - The Reference Net Workshop. In *Tool Demonstrations - 22nd International Conference on Application and Theory of Petri Nets.* See also `http://www.renew.de`.

Kummer, O., F. Wienberg, and M. Duvigneau (2001b). Renew - user guide. Documentation, University of Hamburg, Department for Computer Science. `http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf`.

Lehmann, K. (2003, November). Analyse und Bewertung von Agentenprotokollen auf Basis von Petrinetzen. Diplomarbeit, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany.

Louden, K. C. (1993). *Programming Languages: Principles and Practice.* Boston: PWS-Kent.

Moldt, D., M. Köhler, and H. Rölke (2001, WS). Lehre Projekt: Agenten-orientierte Softwareentwicklung.

Moldt, D., M. Köhler, and H. Rölke (2002, WS). Lehre Projekt: Adaptive Systeme.

Moldt, D. and H. Rölke (2003). Pattern based workflow design using reference nets. In W. van der Aalst, A. ter Hofstede, and M. Weske (Eds.), *International Conference on Business Process Management.* `http://www.springerlink.com/app/home/issue.asp?wasp=mgat4equvjdkvwwywq3m&referrer=parent&backt`

Oberquelle, H. (1981). Communication by graphic net representations. Fachbereichsbericht IFI-HH-B-75/81, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany.

Odell, J., H. V. D. Parunak, and B. Bauer (2000). Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu (Eds.), *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pp. 3–17. `http://www.jamesodell.com/ExtendingUML.pdf`.

Oestereich, B. (2001). *Objektorientierte Softwareentwicklung, Analyse und Design mit der Unified Modeling Language.* (5. ed.). Oldenbourg Verlag.

OMG (2003). Object management group. `http://www.omg.org/`.

Petri, C. A. (2003, June). Net Modeling – Fit for Science? Booklet: Keynote Lecture Petri Nets 2003, Eindhoven University of Technology, Eindhoven, The Netherlands. At the 24th International Conference on Application and Theory of Petri Nets, ICATPN 2003.

Pettijohn, T. F. (1998, October). *Psychology: A ConnecText* (4. ed.). Ohio State University-Marion: McGraw-Hill/Dushkin. `http://www.dushkin.com/connectext/psy/ch08/conform.mhtml`.

Reese, C. (2003, December). *Multiagentensysteme: Anbindung der petrinetzbasierten Plattform CAPA an das internationale Netzwerk Agentcities.* Diplomarbeit, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany.

Reisig, W. (1982). *Petrinetze - Eine Einführung.* Berlin: Springer-Verlag.

Reisig, W. (1997, October). *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag New York.

Renew (2003). Renew Homepage. `http://www.renew.de`, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany.

Rölke, H. (2003). *Agenten und Multiagentensysteme.* Dissertation, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany. Preliminary version, to be published in 2004.

Rumbaugh, J., M. Blaha, W. Premeralani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey 07632: Prentice Hall.

Russell, S. and P. Norvig (1995). *Artificial Intelligence a Modern Approach.* AI. Prentice Hall.

Schumacher, J. (2003, October). *Eine Plug-in-Architektur für Renew: Konzepte, Methoden, Umsetzung.* Diplomarbeit, University of Hamburg, Department of Computer Science.

Sommerville, I. (1996). *Software Engineering* (5. ed. ed.). International computer science series. Wokingham: Addison/Wesley.

UML (2003a, October). UML 1.5 Specifications. `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf`.

UML (2003b, October). UML 2.0 Specifications. `http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf`.

UML Resource Center (2003, October). `http://www.rational.com/uml/index.jsp`.

Valk, R. (1995, June). Petri nets as dynamical objects. In G. Agha and F. D. Cindio (Eds.), *Workshop Proc. 16th International Conf. on Application and Theory of Petri Nets, Torino, Italy.*

van der Aalst, W. M. P. and A. H. M. ter Hofstede (2002, August). Workflow patterns: on the expressive power of (Petri-net-based) workflow languages. In *Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, August 28-30, 2002 / Kurt Jensen (Ed.)*, pp. 1–20. Technical Report DAIMI PB-560. `http://www.daimi.au.dk/CPnets/workshop02/cpn/papers/Aalst.pdf`.

Wienberg, F. (2001). *Informations- und prozeßorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen.* Dissertation, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany. `http://vsis-www.informatik.uni-hamburg.de/documents/papers/Dissertationen/Wienberg/Dissertatio`

# Index