FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Extending the Peer Model with Composable Design Patterns

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Gerald Schermann
Matrikelnummer 0828114

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o. Univ.Prof. Dr. Dipl.-Ing. eva Kühn

Wien, 24.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Extending the Peer Model with Composable Design Patterns

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering/Internet Computing

by

Gerald Schermann
Registration Number 0828114

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o. Univ.Prof. Dr. Dipl.-Ing. eva Kühn

Vienna, 24.04.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gerald Schermann
Gauermanngasse 20 L, 2700 Wiener Neustadt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Zuallererst möchte ich mich bei meinen Eltern für deren großartige Unterstützung bedanken. Ein weiterer Dank geht an meine beiden Schwestern, die genauso immer für mich da sind.

Weiters möchte ich mich bei meiner Betreuerin eva Kühn bedanken, die es mir erst ermöglichte diese interessante Arbeit zu verfassen und mir dabei mit gutem Rat zur Seite stand. Ebenfalls zu erwähnen ist die Space Based Computing Group, welche sich mit dem Peer Model beschäftigt und mir nützliches Feedback gab. Ein besonderer Dank geht dabei an Thomas Hamböck, der mir mit seinen Latex-Macros ein einfaches und schnelles Umsetzen meiner Modelle ermöglicht hat.

Ein besonderer Dank geht auch an meinen Kollegen Janos für das Korrekturlesen meiner Arbeit. An dieser Stelle sind auch meine geschätzten Kollegen Bernd, Jürgen, Michael und nochmals Janos zu nennen, welche ich schon seit HTL-Zeiten kenne und mich auch durch das gesamte Studium begleitet haben, sei es bei vielerlei Gruppenarbeiten oder beim Lösen unterschiedlichster Beispiele. Danke!

Abstract

In the area of software development, reuse is an essential factor. Developing each component of a new product from scratch is costly, includes risks and has negative influence on the time to market. Therefore, the strategic reuse of software components is an important factor for the success of companies. It can leverage existing software investment, companies can build systems out of well-tested components of proven quality which have been used a couple of times and thus, both risks as well as costs for development and testing can be reduced. The software product line approach follows this aim by creating a platform of flexible components which can be selected and combined to different products tailored to stakeholder requirements. This is achieved by software variability, which allows one to customize components for the use in a particular context. It enables cost efficient mass customization of components.

In this work, a pattern based approach is presented which provides this software variability for the *Peer Model*, a programming model for modelling highly concurrent and distributed systems. Patterns are introduced as new components and their interrelations with the original components are described. Similar to the software product line approach, design decisions are delayed and fixed only when concrete instances are created. The pattern concept allows defining generic patterns depending on certain parameters or properties which represent these delayed decisions. Such decisions are not limited to minor effects; the variability allows for a far-reaching influence on the functionality provided by the pattern and thus, it opens up the possibility for reuse on a large scale. Moreover, as composition is heavily used, patterns can be combined and nested to form more complex patterns and to encapsulate functionality.

The advantages of this approach are demonstrated by an example use case from the train traffic telematics domain where signals of approaching trains are transferred from a sensor over multiple network nodes to a level crossing unit. Various modelling concepts and tools ranging from more low-level to highly abstract approaches are selected and with each of them, the example use case is realized. In order to create a meaningful evaluation, a set of criteria is developed, emphasizing elements which are important for the design of highly distributed and concurrent systems and essential for the aspired-to variability. The final evaluation shows that the *Peer Model*, extended with the pattern concept, stands out from the other approaches in the domain of distributed and highly concurrent systems. Moreover, systems designed with the *Peer Model* can be adapted to changing requirements without modifying the underlying architectural design and thus, cost- and time-intensive remodelling and refactoring work can be avoided.

Kurzfassung

Die Wiederverwendung von Softwarekomponenten ist ein wesentlicher Faktor für den Erfolg von Unternehmen. Jede einzelne Komponente eines neuen Produkts von Grund auf neu zu entwickeln wäre ein kostspieliges Unterfangen, verknüpft mit einigen Risiken. Negative Auswirkungen auf die Produkteinführungszeit wären eine direkte Konsequenz. Die geplante Wiederverwendung von Komponenten greift hingegen auf bereits getätigte Investitionen zurück und erlaubt es, neue Systeme auf Basis bereits entwickelter Komponenten mit erprobter Qualität zu erzeugen. Wesentliche Entwicklungszeiten und -kosten können eingespart werden. Bei Software-Produktlinien wird dieser Ansatz strategisch verfolgt: Einzelne, bestehende Komponenten werden ausgewählt und maßgeschneidert an Stakeholder-Bedürfnisse zu neuen Produkten kombiniert. Um das zu ermöglichen, müssen die einzelnen Softwarekomponenten kombinierbar, an den jeweiligen Einsatzzweck anpassbar und entsprechend erweiterbar sein. Diese Eigenschaften sind unter dem Begriff der Software-Variabilität bekannt.

In dieser Arbeit wird ein Pattern-basierender Ansatz präsentiert, welcher diese Software-Variabilität mit dem *Peer Modell* kombiniert. Das *Peer Modell* ist ein Programmiermodell für die Modellierung von verteilten und nebenläufigen Systemen. Patterns werden als neue Komponenten in das *Peer Modell* aufgenommen und deren Zusammenspiel mit den bestehenden Komponenten erläutert. Ähnlich wie bei Software-Produktlinien sollen Design-Entscheidungen verzögert und erst bei der Verwendung konkreter Instanzen getroffen werden. Das in dieser Arbeit vorgestellte Pattern-Konzept erlaubt die Definition von generischen Patterns, welche von einzelnen Parametern abhängig sind. Diese Parameter entsprechen den verzögerten Entscheidungen, diese ermöglichen tiefgreifenden Einfluss auf die Funktionalität der Patterns und dadurch eröffnen sie die Möglichkeit für die Wiederverwendung im großen Stil.

Die Vorteile dieses Konzepts werden anhand eines Anwendungsfalls aus der Eisenbahndomäne demonstriert. Signale eines ankommenden Zuges werden über ein Netzwerk aus mehreren Knoten entlang der Schienen an den Bahnübergang weitergeleitet und dort von einer Komponente verarbeitet. Mehrere Modellierungswerkzeuge aus dem Gebiet der verteilten Systeme wurden ausgewählt und damit der Anwendungsfall umgesetzt. Für eine aussagekräftige Evaluierung wurde eine Liste von Kriterien aufgestellt, welche die wesentlichen Eigenschaften für die Modellierung von solchen Systemen und im Speziellen die gewünschte Software-Variabilität aufzeigen. Die Evaluierung zeigt, dass das um das Pattern-Konzept erweiterte *Peer Modell* hervorsticht. Weiters können damit modellierte Systeme flexibel auf sich ändernde Anforderungen reagieren und laufen nicht in die Gefahr von notwendigen zeit- und kostenintensiven Änderungen an der zugrundeliegenden Architektur.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

1.1 Motivation

The Industrial Revolution ushered in a new era in which the usage of machines supported the manufacturing process and lead to mass production. It replaced the predominant type of production in which skilled craftsmen built products from scratch and specifically for the customer's needs. The advent of mass production resulted in a faster and cheaper manufacturing process. The downside was that the uniqueness and individualism of products resulting from labor-intensive hand-crafting work disappeared. The production of large amounts of standardized products had the effect that there was no diversity any more; all products were the same.

In the twentieth century, the industry started to incorporate the customer's wishes and requirements into the production process; this is also known as *mass customization*. Companies broadened their product portfolio by introducing various models and types of their goods. The problem was that these individual products resulted in higher production costs. The difficulty was solved by the idea of *product lines*. "A product line is a set of products in a product portfolio of a manufacturer that share substantial similarities and that are, ideally, created from a set of reusable parts." [4, p. 4]. The idea is to a have a platform of common parts which can be selected and combined to form different variants of products. A typical example is the car industry where parts can be used in various models. For instance, parts of the engine, the radiator, or the exhaust system. These set of common parts can be produced in large amounts, and therefore, both the benefits of such standardized "reusable" parts as well as the diversity in the product portfolio are given.

Transferred to the software domain, both types of products, individual and mass produced, can be identified as well. Mass produced software is typically called standard software. [48] Especially in the 1990s a trend arose to integrate the idea of product lines with the domain of software. This resulted in *software product lines*. Northrop described a *software product line* in [42] as a "set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Typical core assets are artifacts which are

costly to develop, such as the software architecture, domain models, requirements analysis, test cases and so on.

In traditional software development, when creating a software system from scratch, one often attempts to encapsulate the functionality in a library for reuse in later projects. Over the years, such a reuse has evolved from the reuse of sub-routines to modules, and later, with the emergence of object-oriented techniques, to the reuse of classes. Nowadays, it can happen on the level of components and services as well. In [42], the focus of this reuse types is described as "small grained, opportunistic, and technology driven". In contrast, reuse in software product line development is referred to as "planned, enabled and enforced". That means that parts are not developed to solve single system requirements, but designed with the initial purpose to be used in more than one system. Thus, commonality is of central interest. In software product line engineering, these artifacts can be created from scratch or by deriving from another platform or earlier systems. [48].

The different artifacts are designed in such a way that various products can be built by combining and configuring them. When designing and developing these artifacts for the common platform, future changes and requirements have to be considered. This means that it is not possible that all decisions can be made in advance; design decisions have to be delayed and therefore determined at a later point. This kind of delay or configuration mechanism is achieved using *variability*. Svahnberg et al. [52] defined software variability as the "ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context". Variability opens the door to include mass customization; it allows creating tailored software satisfying stakeholder requirements. There exist various techniques and ways how variability can be implemented. In [4], a detailed classification of these techniques is given, containing techniques from traditional software engineering which have been adapted for software product lines, as well as approaches which are invented to fulfil the specific requirements of software product lines.

The aim of the software product line approach is to establish strategic reuse. Some of the benefits of software product line development can be identified when looking at typical core assets. Basically, a software architecture is a complex artifact which requires a lot of development and testing effort. Developing such an architecture from scratch involves high risks and in the worst case, these may even lead to the cancellation of the planned product or project. In contrast, a flexible architecture which has been used a couple of times in different products has known quality attributes and decreases both risks and costs for development and testing. Testing itself is another field in which the strategic reuse of components can lead to reduced costs: generic test plans, test cases and test data can be taken as a basis and adjusted to new products. There is a determined process regulating how and to whom detected problems shall be reported and how required fixes shall be incorporated. These mentioned benefits can be classified as software engineering benefits and, in a broader sense, as business benefits since they have positive effects on the budget. In [43], a detailed overview of the benefits and the level on which they occur can be found. It also provides information based on conducted case studies describing which business goals are positively influenced by the product line approach.

The software product line approach has not only positive aspects. The downside is that the development methodology of a company cannot be changed immediately. Such a change needs

long term management and includes risks since high investments are required and benefits might not be visible during the first phases.

1.2 Contribution

The aim of this work is to extend the functionality of the *Peer Model* [31] with the ability to design composable and configurable patterns. The *Peer Model* is a programming model which supports the modelling of concurrent and distributed systems. Central components are so-called *peers*, which contain both coordination and business logic. Peers can be arbitrarily nested providing means to encapsulate certain behaviour. In this work, the *Peer Model* is extended by a pattern concept, whereby the involved patterns can be classified as *design patterns*. Gamma et al. [16] described design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". The idea behind is to (re-)use a solution for re-occurring problems. Above, the importance of *variability* for the software product line approach has been explained. We can combine it with the design patterns of the *Peer Model*. In the *Peer Model*, patterns can be seen as a recipe containing the components required to attain a certain functionality. Like patterns in general, they can be instantiated arbitrarily often in various areas. In addition, the patterns proposed in this work are flexible in such a sense that they can be tailored to the user's demands. These flexible patterns are designed in a generic way and similar to the software product line approach: decisions are delayed and determined only when a concrete instance is created. These decisions are not limited to minor effects; the variability allows for a far-reaching influence on the functionality provided by the pattern. Decisions are expressed by a set of properties, which are specified each time a pattern is instantiated. As reuse is a central factor, a pattern which encapsulates certain functionality can be used by another pattern as well. Complex patterns can be defined by combining multiple other patterns. All these ingredients allow one to aim for the strategic reuse known from the software product line approach, bringing with it the previously described benefits. The outcome of this work supports the creation of a pattern concept/methodology which eases and accelerates the development of systems by just selecting and configuring the required patterns.

1.3 Methodology

The first step is to clarify what a pattern actually means in the context of the *Peer Model* and how a pattern relates to the existing components. This includes also the possibility to create nested patterns, thus the creation of patterns by combining one or more sub-patterns.

The next step is to develop a mechanism which supports this kind of variability known from the software product line approach. Different components of patterns which are "flexible" and therefore configurable on pattern instantiation will be elaborated and their potential influence on the pattern's behaviour will be analyzed. Further, the actual usage of patterns is clarified, including the ways to handle the flexibility when composing patterns as well as when deploying peers containing patterns.

Afterwards, a literature study is conducted identifying different modelling concepts and tools in the domain of distributed and concurrent systems. A use case from the train traffic telematics

domain is taken as a proper basis to evaluate the previously gathered tools as well as the extended version of the *Peer Model* on their ability to create highly concurrent distributed systems with the focus on variability and its requirements. In order to create a meaningful evaluation, a set of criteria will be developed emphasizing the strengths and weaknesses of the considered approaches.

The use case is split up into two concrete variants: a basic and a more complex version. Both variants will be realized with each of the modelling concepts and tools. The realizations of the variants provide the necessary information to assess the particular tools with respect to the single criteria.

1.4 Structure of the Master's Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 provides related work in the area of concurrent and distributed systems. Various modelling concepts and tools are introduced, from low level to highly abstract approaches.
- Chapter 3 introduces the example use case from the train traffic telematics domain and its two variants. Moreover, a set of different criteria is given which is used throughout the subsequent evaluation. The example use case is realized with each modelling concept or tool presented in chapter 2 and used as a basis for the evaluation. The chapter concludes with a summary of the evaluation results.
- Chapter 4 introduces the *Peer Model*, which represents the basis for this work. This chapter shall make the reader familiar with the components of the *Peer Model* as well as with its graphical notation. It also provides the required information for the reader to understand the subsequent chapters of this work.
- Chapter 5 presents the central part and contribution of this work: the pattern concept which extends the *Peer Model*. It describes the nature of patterns and their relation to existing components. Further, it introduces the mechanism to define flexible patterns and explains how the composition of patterns is achieved.
- Chapter 6 demonstrates the abilities of the concept introduced in chapter 5. Both variants of the example use case are realized, whereas the majority of the components of the first variant are reused for the more complex second variant.
- Chapter 7 emphasizes the flexibility of the proposed pattern concept regarding its capabilities to react to changing requirements. Two potential additional requirements on the use case scenario are introduced which would typically require architectural changes and refactoring. The chapter concludes with the evaluation of the extended version of the *Peer Model* with respect to the chosen criteria.
- Chapter 8 provides concluding remarks and outlook to possible future work.

CHAPTER 2

Related Work

This chapter introduces related modelling concepts and tools which can be used to model distributed and concurrent systems, and provides basic information about these concepts and tools. A comparison of these approaches is conducted in chapter 3.

2.1 Petri Nets

Petri nets are widely used to model concurrent activities and describe distributed systems. Compared to business process modelling languages and tools they operate on a rather low level, but their expressiveness allows them to be applied in various scenarios. The concept of Petri nets was first introduced in Carl Adam Petri's dissertation [47]. As stated in [41], Petri nets provide means to describe systems "characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic". Over the last decades various extensions of the core concept of Petri nets have been proposed. Some of them are also backward-compatible to the original design, meaning that it is possible to reduce such a higher-level, extended Petri net to a low-level, basic one.

Often when talking about Petri nets the so called *place/transition nets* are meant, a class of Petri nets providing a formalism that allows a higher level of concurrency than the traditional Petri nets as they permit a *place* to hold more than one *token*. For reasons of simplicity, in this work the term Petri net is used as a synonym for place/transition nets.

Before focussing on the mechanisms and possibilities regarding composition, a brief introduction to the components of Petri nets should be given: beside the already mentioned places, place/transition nets consist of *transitions* and *tokens*. Places can hold a number of resources called tokens. A place holding one or more tokens is called *marked*. Transitions are used to transfer tokens between places. The distribution of tokens in the net represents the *state* of the system.

Traditional Petri nets provide no means of abstraction including ways to form subnets or group certain components. Therefore, simple reuse of already modelled functionality is not

possible. A component which could be reused nevertheless has to be completely re-modelled, which leads to blown up and often confusing nets.

2.1.1 Colored Petri Nets

Colored Petri nets (or CP-nets) belong to the category of *high-level nets*. Tokens can be distinguished by their color and represent arbitrarily complex data. Further, the concept of *variables* and *bindings* is introduced allowing an additional level of abstraction (cf. [36]). Concerning composition and reuse, *hierarchical CP-nets* as explained in [28] provide an interesting approach which will be considered in the following:

Substitution transitions and *subpages* are the essential components for modularity. A substitution transition can be seen as the abstract part, whereas a subpage contains a CP-net (subnet). The substitution transition is used to model the CP-net and acts as a place holder to keep the model's size small. A substitution transition refers to a specific subpage, which provides the detailed description of the activity represented by the transition [28]. In order to achieve such a modularity, interfaces are needed to define how subpages communicate with their surrounding net. Therefore, so called *socket* and *port places* exist. The port places of the subpages are assigned to socket places of the substitution transition.

Nesting of such modules is also possible, meaning that a subpage itself can contain substitution transitions. Further, two or more substitution transitions can refer to the same subpage, allowing for the reuse of components. During execution, for each such reference a separate instance of the CP-net of the subpage is created, therefore the substitution transitions are completely independent of each other.

In [11], another concept based on transition fusion through *annotations* for structuring CP-nets is introduced. Annotations on transitions indicate which transitions should act synchronously and build a so called *synchronous channel*. CP-nets with such channels allow restructuring and splitting the nets into multiple parts in order to get clear structured and coherent nets.

2.1.2 Objects and Petri Nets

The "nets within nets paradigm" as proposed in [54] and the previously mentioned synchronous channels from [11] form the basis for the so called *Reference nets* explained in [15, 39]. "Nets within nets" means that a token can hold a reference to another net instance. Instances of the same net are similar to objects being an instance of a class in object-oriented programming. They share the same "template", but are completely independent of each other. In [39], the authors show that such *Reference nets* can increase the expressive power of Petri net based workflow languages as they allow modelling various workflow patterns in an elegant and easy way.

2.2 Reo

Reo [5] is a coordination model used for the composition of software components and web-services. The fundamental concept relies on the notion of *channels*. Reo does not focus on which computational entities are interacting, rather on how such *component-instances* communicate with each other through *connectors*. Channels are the most basic form of connectors.

Connectors can be composed to form more complex connectors. Component-instances which are a non-empty set of active entities, such as processes or threads, communicate with other entities by performing input and output operations on *channel ends* connected to them. As mentioned before, the "inter-component-instance communication" is in the focus of interest, but a single component-instance can as well consist of further component-instances which are linked together by channels/connectors.

Basically, in computer science, channels have two ends. A *source end*, where the data is written into the channel and a *sink end*, where the data is read from the channel. Reo does not limit a channel to have exactly one source and one sink end. It is possible to have either this classic variant of one source and one sink end, or two source ends, or two sink ends. Reo offers numerous different channel types, where each channel type specifies the sort of the two channel ends and the relation of the input and output operations at the two ends. For example, a *synchronous channel* has one source and one sink end and it requires that the write operation on the source end is synchronized with its matching read operation on the sink end. The *FIFO channel* is an example for an asynchronous channel with one source and one sink end. Furthermore, the channel contains a buffer. As long as the buffer is not full, write operations on the source end succeed, independent of the availability of a reader on the sink end. In case of a full buffer write operations are blocked. Read operations succeed as long as there is data in the buffer. The *synchronous drain channel* is a more exotic variant with two source ends. As there is no sink end, every data item written into the channel is lost. A write operation on one end blocks until there is a write operation on the other end as well. This can be used to synchronize two processes. There are further primitive channels whose exact semantics, and differences between the various channel types, are described in detail in [5].

2.3 Uppaal

Uppaal [37, 38] provides a toolbox for the modelling, simulation and verification of real-time systems. A system consists of multiple processes and each process is modelled as an automaton. Uppaal is based on the theory of *timed automata* which extends finite state machines with the concept of clocks. In Uppaal, a *process* is composed of *locations* and *transitions*. Transitions are edges and are used to change locations. Such edges can be annotated with *selections*, *guards*, *synchronizations* and *update* expressions. Guards represent boolean expressions or conditions on clocks. Synchronization is used to synchronize processes by means of *channels*. Primarily, synchronization was only possible between two single processes, one being a sender and the other the receiver. As many other features, later versions of Uppaal introduced also a concept to synchronize more than two processes ($\rightarrow$ *broadcast channels*). Selections are used to non-deterministically bind an identifier to a value in a given range. The scope of the selection is limited to its transition and therefore to the synchronization, guard and update parts. Updates are used to change the state of the system, for example the modification of clocks or other local or even global variables.

2.4 BPMN - Business Process Model and Notation

Business Process Model and Notation [46] is a standard for the modelling and design of business processes maintained by the Object Management Group[1]. The goal was to bridge the gap between the business side responsible for the creation of business processes and the technical side responsible for the implementation of systems executing these processes. Thus, the standard is designed in such a way that it is understandable by the different stakeholders. It also regulates the possibilities for model execution, which is not limited to the Web Services Business Process Execution Language (WS-BPEL), even though that is the standardized execution language.

BPMN contains five categories of elements: *flow objects*, *data*, *connecting objects*, *swimlanes* and *artifacts*. Flow objects represent the main graphical elements separated in *events*, *activities* and *gateways*. An activity can be either an *atomic task*, or a *sub-process* enabling hierarchical (de-) composition. Gateways are used to model the control flow, including branching and merging. Connecting objects are used to combine flow objects, and swimlanes (both *pools* and *lanes*) are a mechanism for grouping the modelling elements. Finally, artifacts allow providing additional information about the process, for example by specifying textual annotations. A more detailed information about these modelling elements can be found throughout the comparison in chapter 3.

2.5 WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [45] is a language for the specification of business processes. In contrast to BPMN, WS-BPEL is an XML-based textual language. WS-BPEL provides two ways for describing business processes: *abstract* and *executable*. Abstract business processes are only partially specified and have a more descriptive role. Executable business processes however, are fully specified and thus can be executed too. WS-BPEL is all about *web-services*. Processes are defined by the composition, orchestration and coordination of web-services, meaning that every kind of interaction is done by acting with web-services. Due to its tight coupling to web-services it is highly related to standards such as WSDL [56], SOAP [55] and UDDI [44]. In WS-BPEL every process has one main *activity*, which can be either a *basic* or *structured activity*. According to the specification [45], basic activities describe the elemental steps of the process behaviour, whereas structured activities represent the control-flow logic and can contain further basic and structured activities. Another essential element of WS-BPEL are so called *Partner Links*. They define the parties that interact with the business process. Every partner link specifies at least one role. If both interacting partners offer services to each other, then two roles are used. Roles simply indicate which services the participants provide. WS-BPEL processes can be categorized into two types: synchronous and asynchronous. When calling a synchronous process, the caller waits until it receives the response from the callee. This request-response interaction is a two-way operation modelled by a *receive-reply* pair in the callee's process. Asynchronous processes, in contrast, are realized by two one-way operations: the caller invokes the callee's service and can immediately continue

[1] Object Management Group (OMG): `http://www.omg.org/`

with its work. In order to receive a callback, the caller has to provide a service to the callee. Both calls are realized using single *invoke* operations.

2.6 Actor Model

According to [19], the Actor Model is a mathematical theory that treats so called actors as the universal primitives of concurrent digital computation. It originates in 1973, proposed by Carl Hewitt, Peter Bishop and Richard Steiger [20]. The model was further described and improved, among others, by Gul Agha [1]. In an interview [21], Carl Hewitt described an actor as the fundamental unit of computation which embodies the three essential elements of processing (to get something done), storage (to be able to remember) and finally communication. Similar to many object oriented programming languages, which follow the philosophy that everything is an object, the Actor Model follows the philosophy that everything is an actor. One of the core components of the Actor Model is messaging. Actors communicate by sending messages to one another. The following axioms describe what an actor can concurrently do in response to a message it receives:

- An actor can create a finite number of further actors.
- It can send a finite number of messages to actors including itself.
- It can influence its behaviour by specifying how it should handle the next message it receives.

These actions are not subject to a defined order and as already mentioned, they can be executed concurrently. Messages are sent asynchronously and delivered on a best efforts basis and at most once. Moreover, there are no guarantees on their ordering. A message can take arbitrarily long until it arrives at its target. In order to send a message to a particular actor, the target's address is needed. Address information is retrieved from incoming messages, i.e. the sender's address, or from actors, which are created while processing a message. In addition, the actor can use the addresses it had known before processing the message, i.e. from previous steps. In the Actor Model, addresses do not correspond to the actor's identities — there is a many-to-many relationship between actors and addresses. One actor can have multiple addresses and one address can encompass multiple actors, which has its use for replication purposes.

Conceptually, the Actor Model specifies that an actor processes one message at a time. Every message handling represents a side-effect free operation. The Actor Model is an inherently concurrent model, actors share no state with each other and their only way of interaction is by sending messages asynchronously.

According to [19], the Actor Model can be used as a framework for modelling of, understanding of and reasoning about concurrent systems. Further, it has influence on many programming languages. Some of them, like Erlang, have built their concurrency directly upon the Actor Model. Others, like Java, provide frameworks or libraries to incorporate such an actor-based programming style.

CHAPTER 3

Comparison of Related Work

In this chapter an evaluation of the previously presented modelling concepts and tools is conducted. The goal of this evaluation is to assess the ability of these tools to design highly distributed concurrent systems. To create a meaningful evaluation, a set of criteria is developed whereof the majority covers the essential elements which are important for software variability and therefore for strategic reuse.

Beside the basic information provided in chapter 2, this chapter provides more detailed information about the presented approaches. Nevertheless, due to the sheer complexity of some of these concepts and tools and the limited resources for this work, the provided information might not be sufficient to completely follow all presented examples. For more details, the interested reader is enjoined to peruse the referenced literature.

3.1 Comparative Use Case

In order to identify the strengths and weaknesses of the modelling concepts and tools presented in the previous chapter, an example use case will be introduced. As the areas of application of the considered modelling concepts differ widely, the goal is to evaluate whether a respective tool or concept fulfils predefined criteria.

The use case is borrowed from a research project[1] in the train traffic telematics domain and has been described in [32]. The setting is the following: along a track multiple low-power nodes equipped with wireless communication functionality are positioned. A sensor node detects an approaching train and forwards this information in form of an event to its following (upstream) neighbor nodes. The event is forwarded along the track until the unit at the level crossing is reached. The transfer of the event shall be reliable; therefore, acknowledgement messages and potential retransmissions are necessary elements. The described use case is depicted in figure 3.1. In the following, two variants of the use case that are defined in [30], in the remainder of this work called variants A and B, will be summarized.

[1] www.loponode.org

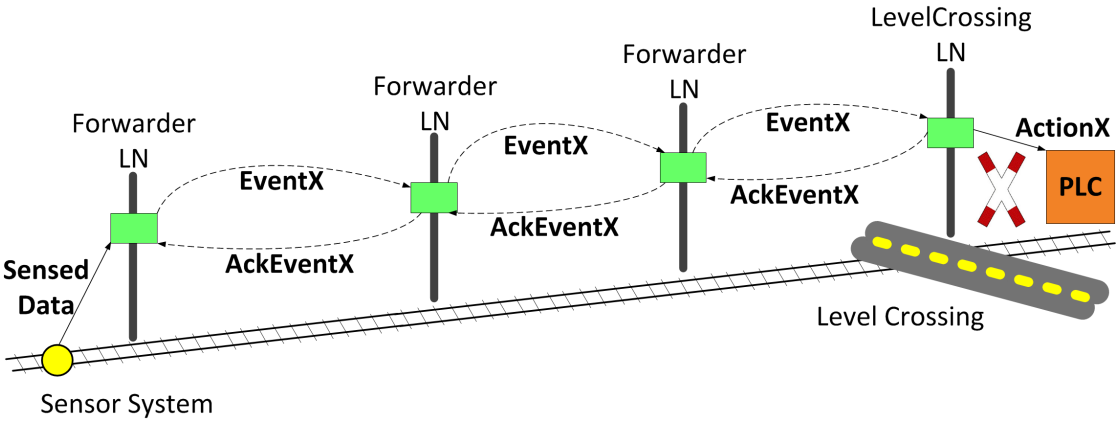

Figure 3.1: Train Traffic Use Case - Figure taken from [32]

3.1.1 Variant A

Variant A covers the basic setting of the explained use case where every forwarder node has exactly one upstream and one downstream neighbor. Each event must be acknowledged in a point-to-point way. This variant is depicted in figure 3.1.

3.1.2 Variant B

Variant B depicted in figure 3.2 provides a more complex example where a signal is forwarded to a group of nodes. This group-based protocol is described in [30]. The group members coordinate with each other in such a way that one member, also called the leader, is responsible for forwarding the signal to the upstream group and, in addition, for sending an acknowledgement to the downstream group or sensor. The other group members establish a failover mechanism such that they take over the event-handling after a specified timeout in case that the leader fails. Compared to the first variant, this one provides a higher reliability due to the grouping of network nodes, but requires group internal coordination as well.

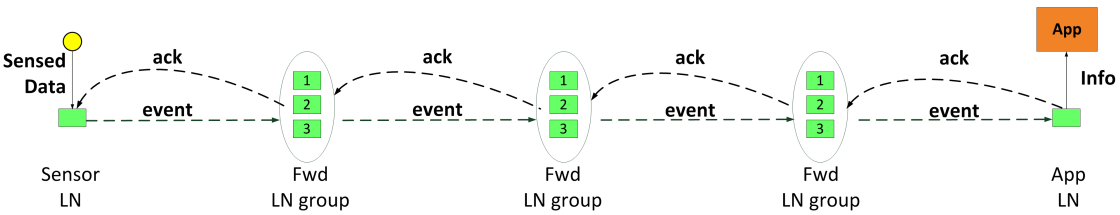

Figure 3.2: Use Case Variant B - Figure taken from [30]

One might wonder why such an acknowledgement mechanism, as we will see throughout this work, is necessary as there are a lot of protocols out there which support this functionality out of the box, for example TCP. The reason is, that these nodes extracted from the use case are

typically not connected to the internet and these are low-power nodes meaning their communication and processing load have to be kept at a minimum. Moreover, it is not the goal of this work to design ideal notification protocols; the existing example use case is taken as a common basis for the comparison of the mentioned modelling concepts with respect to the criteria presented in the following.

3.2 Evaluation Criteria

Both variants of the described use case shall be realized with the considered modelling concepts to allow us to gain insights about their strengths and weaknesses. The following criteria will be used throughout the context of this evaluation:

- Composition: The criterion whether the considered approach supports the "gluing" together (combination) of two or more modelled artifacts to form a new, more complex artifact. Composition allows the introduction of hierarchical structures into the model and provides layers of abstraction.

- Reuse: This criterion is about whether such a modelled artifact can be used more than once. This includes the application in different models as well as the manifold usage in a single model through multiple instances.

- Parametrization: Parametrization describes the ability to influence the behaviour of the modelled system, for example by specifying particular options on instantiation. In connection with composition and reuse it has to be distinguished whether a specified parameter affects all instances of the reused component or just a single instance (which results in a higher degree of control).

- Separation of concerns: This criterion describes whether the modelling concept supports the separation of business (or application) logic and coordination logic. Business logic describes what has to be done in order to fulfil a certain task, for example the calculation of the overall costs of an order. Coordination logic is about how the parts or components of the model interact with each other. A clear separation of concerns facilitates potential future extensions and expedites maintenance.

- Dynamics: This criterion is about how flexible and dynamic a modelling concept is. Does it allow adding or removing or even replacing certain components at runtime? A typical example would be the registration of an additional neighbor node in the underlying use case. A flexible tool or modelling concept would support the extension at runtime, whereas other ones would require a complete re-design of the model to meet the new demands.

- Addressing: Addressing specifies the capability of the underlying system to automatically deliver messages from a sender to arbitrary nodes in the system, where it is not required that both sender and receiver are directly connected. Therefore, it includes the possibility

of creating an implicit and dynamic connection between sender and receiver for the purpose of transmitting the data. Further, the receiver should be able to identify the sender of the message, which gives it the opportunity to reply.

- Scalability: (Design-) Scalability is about whether the modelling concept allows modelling large artifacts or systems and whether it scales naturally to any number of participating components. The more components involved, the more complex the models get. However, some modelling concepts provide mechanisms to abstract the number of components and therefore the models stay clear. For example, scaling up to five group members in variant B would require some tools to explicitly model all five group members, whereas others abstract the concrete number of components and need no adjustments.
- Time: This criterion embodies the capability to incorporate the factor time. This includes the possibility to define deadlines specifying the point in time when particular objects become invalid (or expire), as well as delayed execution, which allows controlling the time when objects will be available. Moreover, time triggered execution shall be supported. This allows, for example, the triggering of a certain escalation routine when a message has not arrived within a deadline.
- Toolchain and documentation: This criterion evaluates the tool support for the considered modelling concept. Is the modelling concept already shipped with a concrete tool or are there various implementations available? Is there a graphical tool easing the development of models and systems or is the toolchain limited to code-based development? Furthermore, the quality of the documentation is evaluated with respect to its support in solving the issues arising while realizing the variants of the example use case.
- Simplicity: This criterion describes the effort needed to gather an adequate level of knowledge to be able to use the considered approach both efficiently and effectively.

The first seven criteria are essential ingredients to provide a high level of software variability. The time criterion is necessary to involve time-based processing which is required in the realization of the presented use case. Otherwise, it would be impossible to resend a signal after a particular timespan if no acknowledgement has been received. Further, nodes maintain a list of processed signals to prevent additional work in case that a treated signal is received again. These lists are getting bigger over time and might lead to memory problems — thus, a time triggered mechanism to discard "old" signals is necessary. To keep the models readable, this memory problem is omitted throughout this evaluation, however, the time criterion describes whether it could be solved with a concrete tool or not. The last two criteria focus on the modelling process and how much effort is required to realize the variants of the use case.

3.3 Petri nets

3.3.1 Use Case Implementation - Variant A

The example was realized using *CPN Tools* [49], a graphical tool for modelling and simulating (Timed) Colored Petri nets. As will be seen, the implementation also makes use of the advanced

features of the ML language used by CPN. ML stands for metalanguage and is a general-purpose functional programming language. Listing 3.1 shows the colors, variables and functions required throughout the realization of variant A. The tokens used to transfer the event have the color *PACKET*. Such a packet is a tuple consisting of the event ID and the event itself. For simplicity, the color *EVENT* is one out of the three enumerated types 'A','B' or 'C', which correspond to example events. The majority of the variables and the function were used to simulate the network transmission, potential transmission failures and their resulting retransmissions. As this simulation is out of the scope for this work's evaluation, it won't be described in detail, but for the sake of completeness its corresponding subnets will be illustrated.

```
colset INT = int;
colset EVENT = with A | B | C;
colset PACKET = product INT * EVENT timed;
colset RESULT = list PACKET;
var n, k: INT;
var e : EVENT;
var p : PACKET;
var res : RESULT;
colset Prob = int with 0..10;
colset Range = int with 1..10;
var s : Prob;
var r: Range;
fun Ok(s : Prob, r : Range) = (r<=s);
```

Listing 3.1: CPN (Variant A) - Declarations

Figure 3.3 shows the top-level CP-net. *Network* and *Join* are examples for substitution transitions and demonstrate the ability of CP-nets to create hierarchical structures.

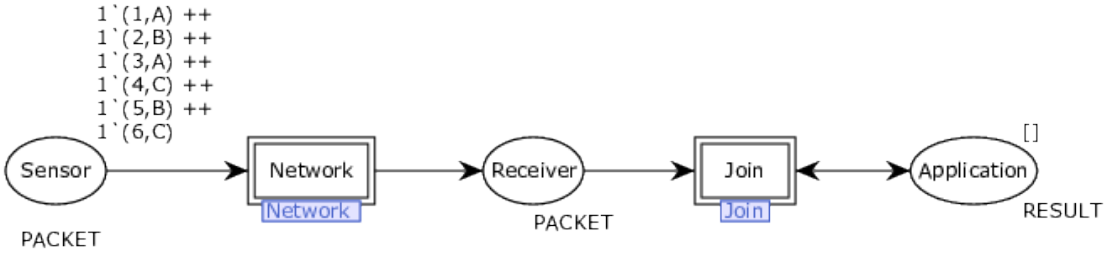

Figure 3.3: CPN - Top level net

The network subnet in figure 3.4 shows how the sensor node and the receiver unit are connected through multiple network nodes. According to variant A, every node has exactly one upstream and one downstream neighbor. The upstream neighbor of the sensor node is node *N1*. The actual transmission between two nodes is achieved using the subpage *SingleTransport*, which is referenced in multiple substitution transitions as can be seen in the figure. This demonstrates the ability to reuse a modelled component. As for parametrization, every instance of such a

subpage is an exact copy of the original one. A subpage may have some parameters specified as variables or functions in the declarations part, but as these are global, they affect all instances of the subpage.

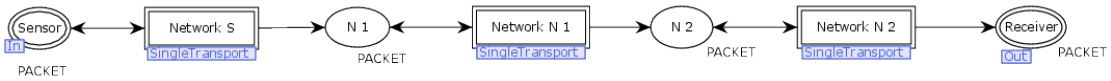

Figure 3.4: CPN (Variant A) - Network subnet

The already mentioned *SingleTransport* subpage is shown in figure 3.5; it contains the logic to forward signals. This net is based on the transport protocol presented in [23]. It forwards the signals with ascending event ID. The transport logic uses two further substitution transitions *PacketTransport* and *AckTransport*, which are both used to simulate transmission failures. The bidirectional arc between the place *NextSend* and the transition *SendPacket* is a graphical shortcut for two unidirectional arcs binding the same variable.

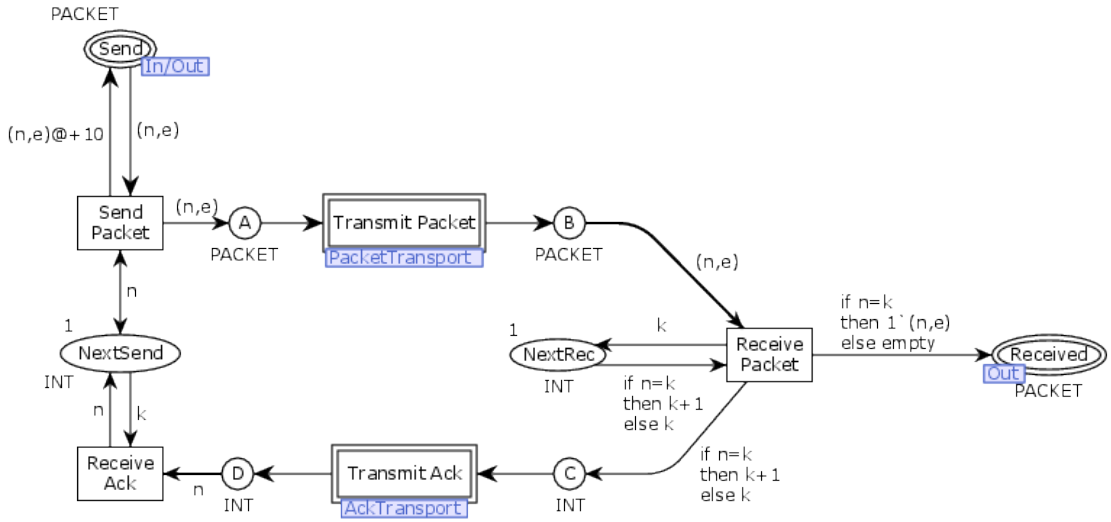

Figure 3.5: CPN - Transport subnet

The subpage *Join* used in the top-level net is depicted in figure 3.6. Its task is to filter already processed events. To achieve this, it makes use of the ML features of CP-nets, more precisely the functions for list manipulation. The *Application* place stores a list of already processed events, in case of an incoming event from the *Receiver* place via the transition *Join* it is checked for whether the incoming event has been treated before. If that is the case, the list is kept unaltered, otherwise the new event will be added to the list.

Figures 3.7 and 3.8 show the already mentioned subpages for the simulation of transmission failures.

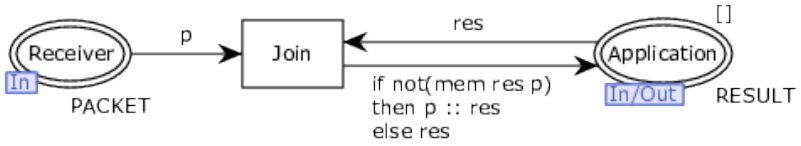

Figure 3.6: CPN - Join subnet

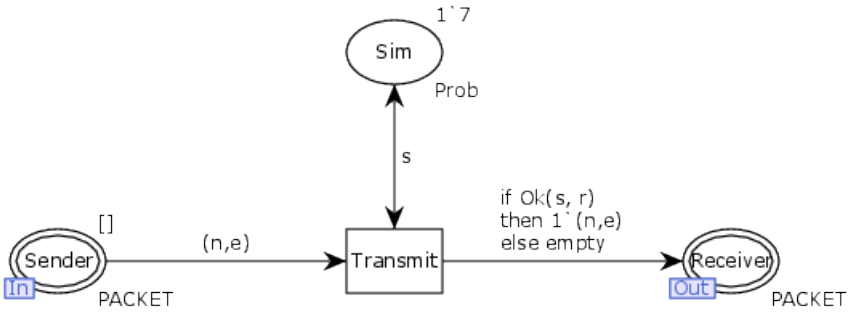

Figure 3.7: CPN - PacketTransport subnet

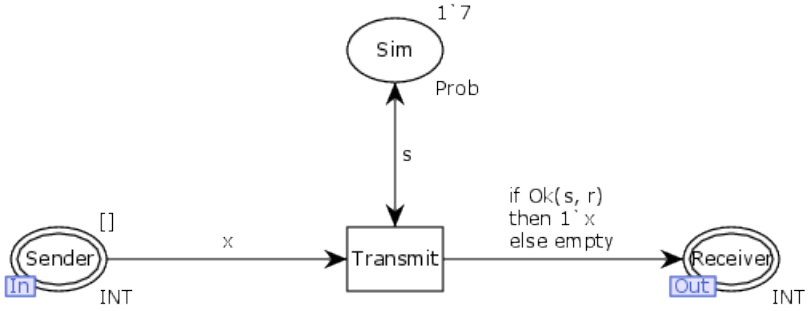

Figure 3.8: CPN - AckTransport subnet

3.3.2 Use Case Implementation - Variant B

The second variant is realized in the same fashion as the first one. In order to realize the group coordination, certain extensions are necessary: the type and variable declarations need to be extended by the color *NodeTimeout*, which corresponds to the amount of time a group member waits for the current leader to process the event until it takes over the group-leadership and handles the event on its own. The enhanced version of the type and variable declarations can be found in listing 3.2.

```
colset INT = int;
colset EVENT = with A | B | C;
colset PACKET = product INT * EVENT timed;
colset RESULT = list PACKET;
```

```
var x,n,k: INT;
var e : EVENT;
var p : PACKET;
var res : RESULT;
colset Prob = int with 0..10;
colset Range = int with 1..10;
colset NodeTimeout = int with 5..10;
var s: Prob;
var r: Range;
var to: NodeTimeout;
fun Ok(s : Prob, r : Range) = (r<=s);
```

Listing 3.2: CPN (Variant B) - Declarations

The top-level net and the subnets *Join*, *SingleTransport*, *PacketTransport* and *AckTransport* stay the same. The *Network* subpage is replaced by the net depicted in figure 3.9.

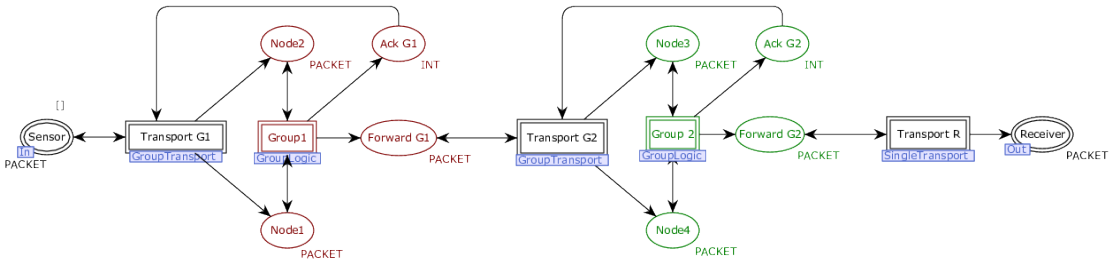

Figure 3.9: CPN (Variant B) - Network subnet

The example consists of two groups. For better visibility, the first group is colored red and the second green. Beside the actual group members *Node1* and *Node2* or *Node3* and *Node4*, there are two further places in each group. *Ack* and *Forward* are used by both group members. The *Network* subpage uses three different subpages (*GroupTransport, GroupLogic, SingleTransport*) in five substitution transitions.

The *GroupTransport* subpage shown in figure 3.10 contains the logic to distribute the signals wrapped with packets to the group members. Similar to variant A, the subpage *PacketTransport* is used to simulate transmission failures. So it might happen that both group members receive the packet, that only one receives it, or that none receive it. Further, the subpage contains the logic to deliver acknowledgement messages from the conjointly used *Ack*-place back to the sender. Again, the *AckTransport* is used to simulate failures when transmitting the acknowledgement messages.

Figure 3.11 depicts the subpage *GroupLogic*. The places and transitions which belong to the first group member are shown in red and for the second group in blue. The black colored places are used by both group members for coordination purposes. The *nextRec*-place holds the information about the event ID to be processed next, *InTurn* contains the ID of the current group leader which is by default 1. Basically, only the group leader is responsible to handle incoming events and to process them. The remaining group member buffers the incoming events too,

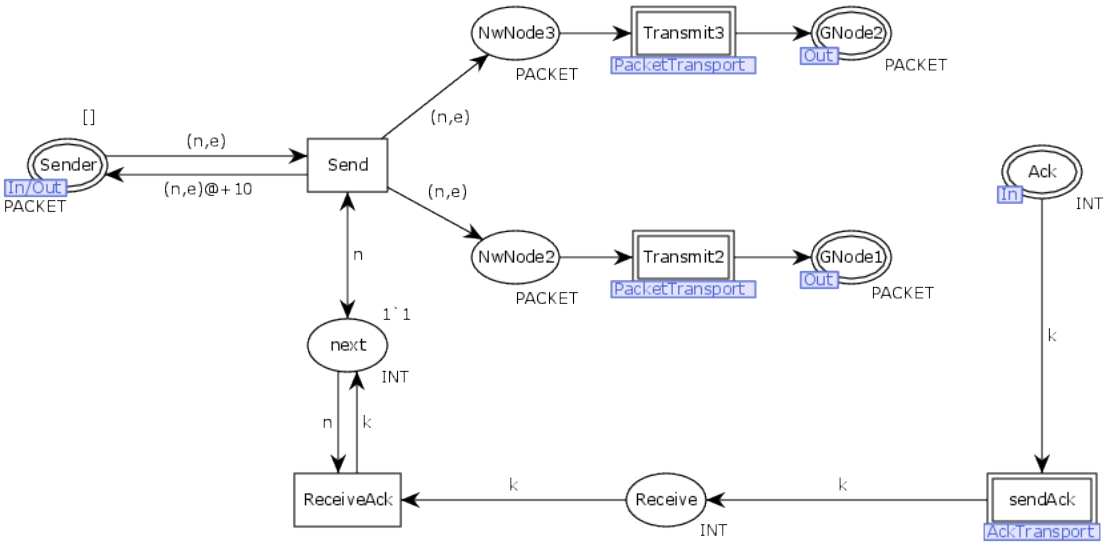

Figure 3.10: CPN (Variant B) - GroupTransport subnet

which can be seen on the *Buffer* transition and *Pending* place. To prevent the current group leader from buffering the event, the *Buffer* transition has an attached condition, i.e. $x <> 1$ for group member 1, where x corresponds to the ID of the group leader retrieved from the *InTurn* place. These buffered events become inactive for a random time which is achieved using the time-functionality of CPN. This random time, specified between 5 and 10 time-steps, is used to give the group leader enough time to process the event. If this failover time elapses the event gets active again in the corresponding *Pending* place and then the group member can gain the leadership of the group via the connected *TakeOver* transition. Such situations typically arise when the former group leader does not receive the new event and therefore the timeout occur. To prevent leadership changes due to already processed events, such treated events are removed from the pending list by using the *Shrink* transition. The actual event processing is carried out by the *Process* transitions; the counter for the event ID to be processed next is increased and the leader's turn-token is renewed. Finally, the event is transferred to the *Forward* place which is interconnected with the previously discussed *GroupTransport* subnet.

3.3.3 Conclusion

As already explained in the previous sections, substitution transitions allow one to incorporate hierarchical structures, and subpages can be used multiple times. Therefore, the criteria concerning composition and reuse are fulfilled. Parametrization is possible, for example by relying on global variables, but such parameters affect all instances of a concrete subpage — therefore, the criterion is considered only partially satisfied.

The main purpose of Petri nets is to describe distributed systems; they are not intended to create real world systems and applications. In order to analyze potential system states, however,

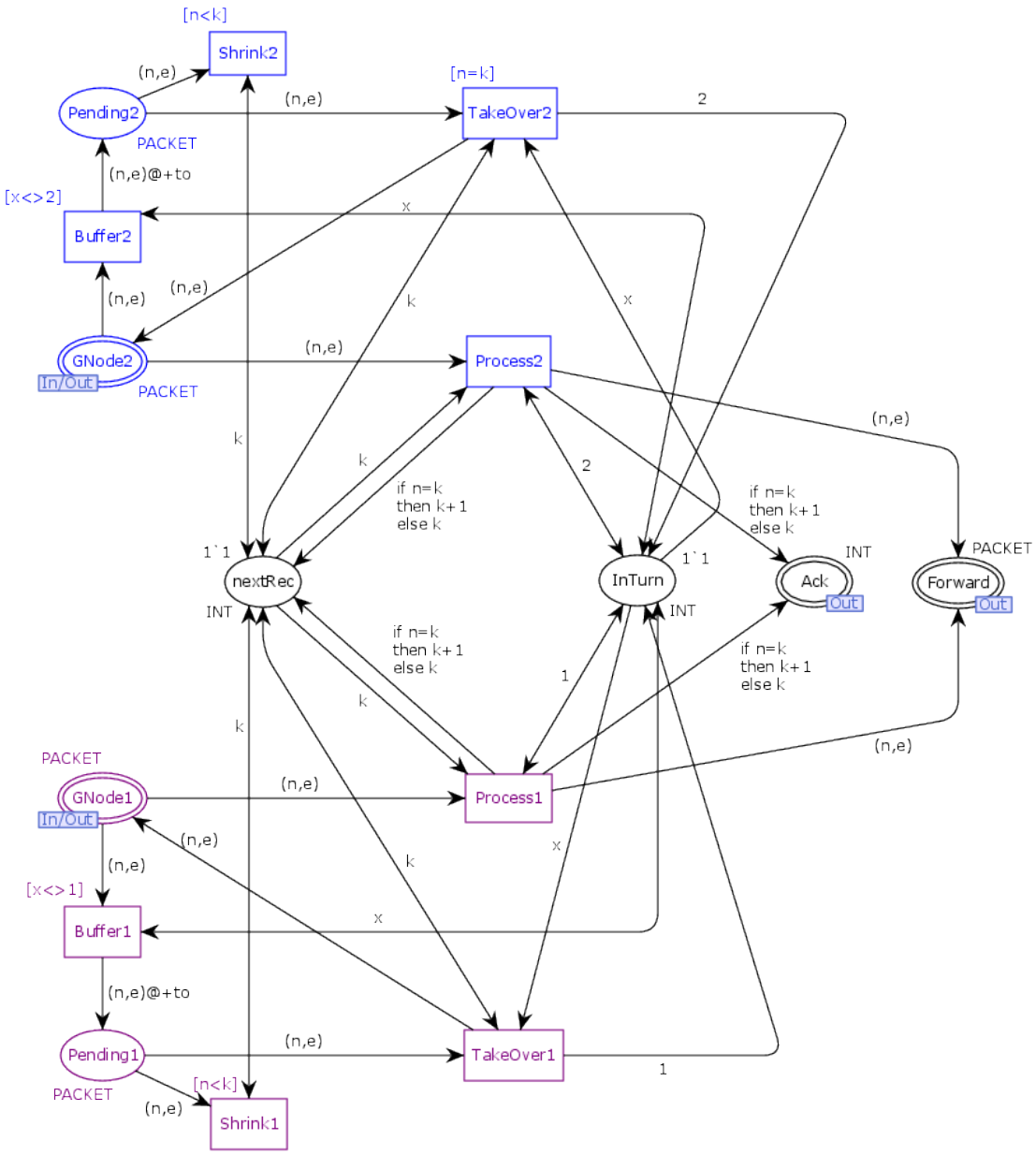

Figure 3.11: CPN (Variant B) - GroupLogic subnet

one often needs to include some kind of business-logic — therefore, Petri nets as well as Colored Petri nets do not separate between business and coordination logic.

Petri nets are of a static nature and not agile or flexible, as there are no possible changes at runtime. An additional member in group 1 in variant B would mean that the substitution transition *Group1* has to be replaced by a substitution transition and a subpage pair capable of handling three group members — a remodelling of the net would be necessary. Further, an additional group member would require a lot more places and transitions. Especially the conjointly used places *InTurn* and *nextRec* would be "overloaded" with arcs. This is a really good example to illustrate that Petri nets and Colored Petri nets as well have deficiencies concerning design-scalability, which also affects the readability of the models.

With Timed CP-nets the factor time can also be incorporated into nets. *CPNTools* allows the specification of timed color sets. This feature was used for the *PACKET* color set and thereby, it was possible to integrate the failover mechanism to the *GroupLogic* subnet. The timestamps are used to control when certain tokens are available. In *CPNTools*, timestamps can be specified on the initial marking inscription of places for each token, by putting them in a transition time inscription or by appending them to outgoing arcs of a transition. Basically, they are used to delay the execution of single tokens and therefore, they can also affect the firing of transitions. In CP-nets, tokens cannot expire; such a mechanism requires a workaround and has to be implemented into the particular net. For example by a state based approach such that after a specific amount of time a state change happens and as a result tokens are handled differently than before. Similarly, time triggered execution can be realized by a transition which is fired when a token with a previously defined timestamp becomes available. Due to this necessary workarounds the time criterion is only partially fulfilled.

Regarding addressing, Petri nets or CP-nets do not support such functionality that tokens can be automatically transmitted between two not connected nodes. Again, the reason lies in their static nature and no implicit transitions can be created at runtime.

CPN Tools, which was used throughout this evaluation to realize the variants provides good support for modelling CP-nets. It offers a direct visual verification and highlights certain inconsistencies and problems. Furthermore, it allows simulating the nets as well. The documentation is comprehensive and the tool itself is shipped with a couple of example nets, which allows the user to get familiar with the tool and its features.

For understanding Petri nets not much effort is needed, models can be created quickly. Colored Petri nets bring in a lot more functionality, which on the one hand allows creating less blown-up and clearer models, but on the other hand requires some time to get a feeling for the additional features and particularly when to use them.

3.4 Reo

3.4.1 Use Case Implementation - Variant A

Figure 3.12 shows the realization of variant A of the running example. The model was created using the *Extensible Coordination Tools* (ECT) [51], a set of plugins for the Eclipse platform allowing, amongst others, the graphical editing, simulation, and verification of Reo connectors.

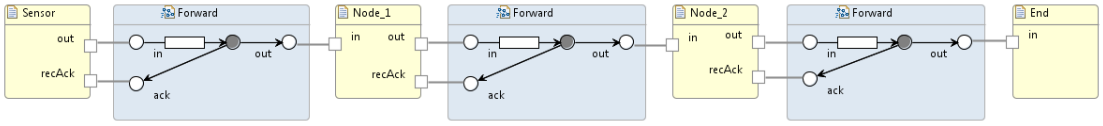

Figure 3.12: Reo (Variant A) - Realization

The example shown in figure 3.12 consists of four components (sensor, two network nodes and end-node) and three *Forward*-connectors. As previously described, composition is prevalent in Reo. The connector *Forward* is itself composed of four nodes, two synchronization channels, and one FIFO channel. The task of the connector is to transfer the event to the upstream neighbor. The event will be buffered in the FIFO channel until the neighbor reads from its incoming channel. Simultaneously, the event is returned to the sending node indicating the successful transfer (acknowledgement). It is the task of the node to cope with unavailable nodes, for instance when no acknowledgement arrives within a specified time. The component could for example log this problem to inform the user about the occurred unavailability of the upstream node. Basically, the event to be forwarded stays indefinitely long in the FIFO buffer. Time-based channels which will be discussed later allow discarding the pending event after a configured amount of time.

A more sophisticated realization of the transmission of certain signals has been presented in [8]. The corresponding connector is depicted in figure 3.13. It is based on a faulty FIFO channel which behaves like an ordinary one except that write operations might fail with a certain probability τ. Therefore, the connector contains logic to resend the signal until the receiver connected to the node *out* successfully obtains it. If that is the case, simultaneously an acknowledgement message is sent to the synchronous drain between nodes B and G. This synchronous drain together with the two synchronous channels between H and G, and G and E stop the data flow through the FIFO channels in the middle in the following and thus, also the resending process. The FIFO channels between nodes C and E, and nodes E and F, as well as the lossy synchronous channel between nodes F and C are responsible for resending the signal until it is stored in the buffer between nodes A and B, which simulates the successful transfer. A more detailed description of this connector can be found in [8]. The reason why the realization of the *Forward* connector is kept simple is that the *Extensible Coordination Tools* currently do not support such probabilistic channels.

The discussed *Forward* connector is used multiple times in the given model. Therefore, reuse is an available feature and further, each of the connectors is an independent instance. At creation time specific parameters can be passed to the corresponding constructor leading to parametrizable behaviour. For example, one may specify the queue-size of the FIFO-channel of the *Forward* connector.

3.4.2 Use Case Implementation - Variant B

The connector responsible for the transmission of signals between two groups and the group internal logic managing the processing and failover is depicted in figure 3.14. In the presented model each group consists of two components (members) which are more advanced compared to

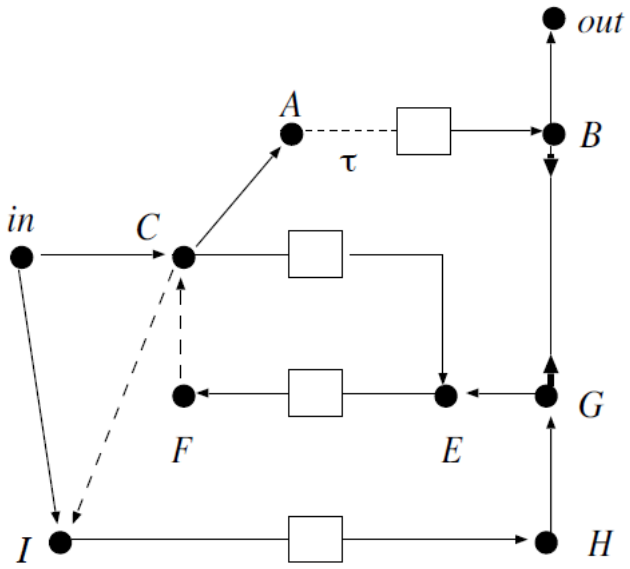

Figure 3.13: Repeated Transmission and Faulty Channels - Figure taken from [8]

the components of variant A. The connector *GroupBasedForwarder* is composed of two *Forward* connectors and a couple of additional channels and nodes for the group internal logic.

Basically, the receiving units (*G2_Node1* and *G2_Node2*) interact via four ports (channel ends) with the connector. The *input* port is used to receive incoming events. Events are available on this port if the member is the leader of the group, which depends on the state of the FIFO channel between nodes *K* and *Turn1* for the first member or the channel between nodes *L* and *Turn2* for the second member. In addition, an event is also forwarded to the "passive" member's *input* port when the timer channel between *E* and *I* (member 1), or *F* and *J* (member 2) is triggered after a timeout of 5 seconds. For both cases, normal and failover, the incoming event is buffered in the channels between *A* and *C* (member 1) or *B* and *D* (member 2). The port *createTurn* allows a component to renew its "turn-token" after an event was processed. Further, the port *resetTimer* allows one to reset both timers to prevent an unnecessary failover handling and thus to signal the successful processing of the event. This is achieved by sending a specific message to the timer channel. More about the behaviour of timers and time based channels can be read in [7]. Finally, the *ackGroup* port is used to "consume" the buffered event in the channel of the opposite member. The timer reset and the consumption of the buffered event form the group internal acknowledgement. After executing these three "post-processing" mechanisms, the overall connector is reset and therefore ready to receive the next signal.

Similar to the components shown in variant A, the sending units *G1_Node1* and *G1_Node2* interact with two ports: the first port forwards signals and the second receives acknowledgement messages. It should be obvious that network nodes/groups are responsible for both tasks: first, to receive and process the signal and second, to then forward it to the upstream group.

Finally, it should be mentioned that the connector between the initial unit (sensor) and the

first group must be slightly modified in such a way that it has only one *Forward*-connector which writes to the node *Input*.

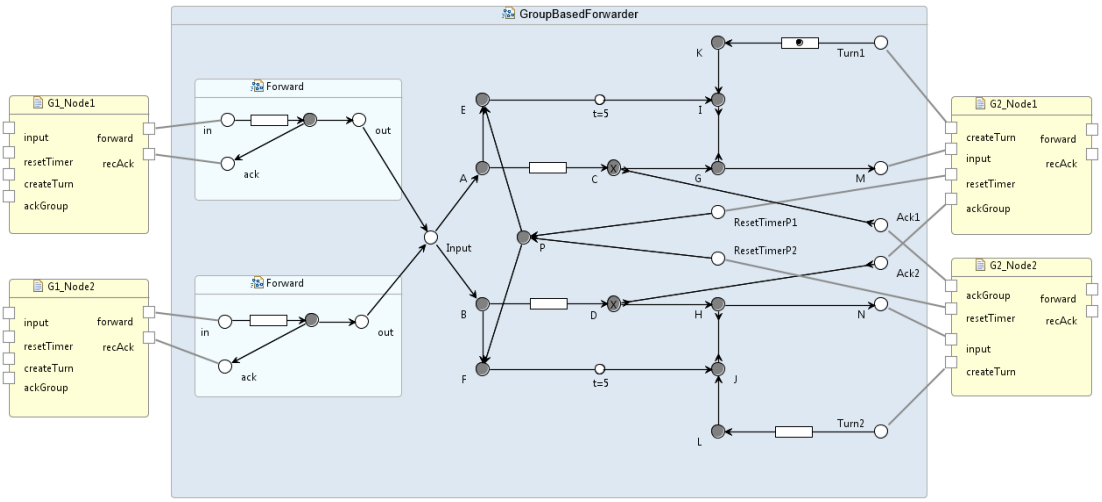

Figure 3.14: Reo (Variant B) - Realization

3.4.3 Conclusion

As already mentioned, Reo supports composition, reuse as well as parametrization to the desired extent. Concerning separation of concerns, the idea is that the business logic is accomplished by single components which can be arbitrary software components including web-services or can themselves be composed of yet other connectors and components. The connectors represent the coordination logic; therefore separation of concerns can be seen as being present as well.

Changes on connectors at runtime are covered in detail in [25–27], whereas [27] extends the other two approaches. Such a connector reconfiguration is based on the theory of so-called high-level replacement systems, which can be seen as an extension of the algebraic graph transformation theory to other high-level structures. Essential for these transformations is a set of rewrite rules defining left-hand side (LHS), right-hand side (RHS) and a mapping between them. Such a rewrite rule is applied on a source graph by searching for an occurrence of the LHS, which is then replaced by the RHS leading to the corresponding transformed graph. [27] contains a couple of examples, including a reconfiguration rule where a worker component is added at runtime to a connector. This technique can be applied in the running variants as well, for example when an additional group member should be added. Thus, Reo fulfils the dynamics criterion.

Thinking one step ahead, it is possible to add or modify an existing connector to create a dynamic connection between two or more components at runtime. That means that such an addressing mechanism is realizable, even though it requires a lot of work as the needed rewrite rules have to be defined. Addressing is thus not supported directly and this kind of workaround is required. The criterion is seen as not completely satisfied.

As already mentioned, the addition of a group member would require the reconfiguration of connectors. Further channels and nodes are introduced to connect the new member to its group. As a result, after the transformation, the graph is a larger than before. Adding a couple of additional group members leads to complex graphs which are hard to read as well. There is no abstraction mechanism; every additional component has to be added to the model and therefore Reo models do not scale to the number of participating components.

In the use case realization above, a timer channel with "off"-option has been used. As described in [7], such an option allows a timer to be stopped before the expiration of its delay when a special message is consumed by its source end. Timer channels allow the inclusion of delayed execution into a model as the timer's emitted signal can activate following circuits. Further, FIFO channel and timer channels can be composed to a simple and reusable connector which stores a particular data item for a specified period of time before it is emitted on the connector's sink end. With such a connector it is possible to delay the processing of single items as well. *Reo* does not directly support the expiration of data items. There are expiring FIFO channels, where data items are lost if they are not consumed within a specified amount of time, but the expiration time is limited to this single channel and there is now way to span it over a complete connector. A workaround would be to include the timestamp into the data item and delegate the removal of expired items to the particular software components. As item expiration is not completely supported, the time criterion is only partially fulfilled.

Concerning the toolchain and documentation, Reo provides the *Extensible Coordination Tools* plugin for the Eclipse platform. A more detailed description of its features and the corresponding research work can be found in [6, p. 199]. Unfortunately, the plugin is lagging behind the published research articles, not all features are yet fully integrated. As already mentioned, there is for example no support for probabilistic channels. Furthermore, when including time based channels, the creation of animations fail. This criterion is considered only partially fulfilled.

The last criterion to mention is simplicity. Though Reo has a manageable amount of predefined channels and each with clear semantics, the act of creating models is a rather complex task. The combination of different channels and their semantics makes especially larger models hard to understand.

3.5 Uppaal

3.5.1 Use Case Implementation - Variant A

Both variants of the example use case have been implemented using version 4.0 of Uppaal [9], which provides a Java-based graphical user-interface. The system of variant A consists of three different automata, which will be presented in the following.

Figure 3.15 depicts the sensor automaton. The outgoing transition of the initial state contains a selection and an update expression. The former is used to non-deterministically choose a value of the range of the type *e_t*, which is defined in the global declarations part of listing 3.3, and bind it to the identifier *e*. The latter assigns the selected value to the local variable *event*. This initial transition is used to simulate the detection of a signal. The automata uses then the broadcast

channel *forward* to signal an approaching train. After broadcasting, the automaton waits in the right location until it receives an acknowledgement (*ack* channel) from its upstream neighbor or until a timeout occurs, which would result in a retransmission of the event. To be precise, the shown automaton is a *template*. Uppaal provides a sophisticated concept for the reuse of automata based on templates. Templates allow one to design abstract automata with a set of defined *parameters*. These parameters can be seen as formal parameters which get substituted by the actual parameters on process instantiation. Parameters can have arbitrary types ranging from simple integers and channels to arrays and even more complex data structures. The concrete types and more information can be found in the Uppaal language reference [53]. Parameter passing can be handled either by value or by reference, whereas channels and clocks have to be passed by reference. The sensor automaton of figure 3.15 has the following parameters:

```
broadcast chan &forward, chan &ack, int &f_event
```

The identifier *timeout* is globally declared constant, whereas *event* and *x* are local variables. The identifier *x* is a local clock, which gets reset once the right location is reached. The upper transition with its green annotation is an example for a guard.

Uppaal's synchronization mechanism allows one to directly address specific nodes or locations in different automata. These nodes do not need to be explicitly connected via transitions. But with single synchronization statements it is not possible for the receiver of the signal to identify the emitter of it. Synchronization, coupled with update expressions, is the key to solve this "issue". Basically, both transitions are executed at the same time, but the update expression of the sender is evaluated before its counterpart. This allows transferring information between the sender and the receiver. This mechanism is used throughout this model to forward the event information. The value of the variable *event* is assigned to the reference parameter *f_event*, which is a global variable shared between the sensor and network node automata.

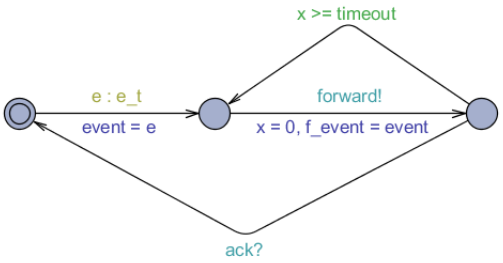

Figure 3.15: Uppaal (Variant A) - Sensor Automaton

The template for the network node automaton is depicted in figure 3.16. It listens on a specific broadcast channel (*input*) for incoming events; in case of an arrival it acknowledges (*send_ack*) it immediately. This is achieved by a so called *committed location*, graphically represented by a "C". In such a location, no time is allowed to pass and the next overall transition which is executed must involve an outgoing edge of at least one of such committed locations — so there is no interference with other "non-committed" transitions. After acknowledging, the node broadcasts the event to its own follower (*forward*) and waits until either a timeout occurs

or the upstream neighbor notifies about the successful transmission (*rec_ack*). The network node template has the following parameters:

```
broadcast chan &input, broadcast chan &forward, chan &send_ack,
chan &rec_ack, int &in_event, int &out_event
```

The parameter *in_event* corresponds to the shared variable used to exchange the incoming event. *out_event* is the counterpart on the forwarding side. In this examples, the handling of events happens sequentially. A new event is not processed before its predecessor event is not acknowledged. Concurrent event processing would require arrays of synchronization channels and user-defined functions for correlation-purposes. This would increase the complexity of this examples tremendously, which is the reason why in this work sequential processing is used.

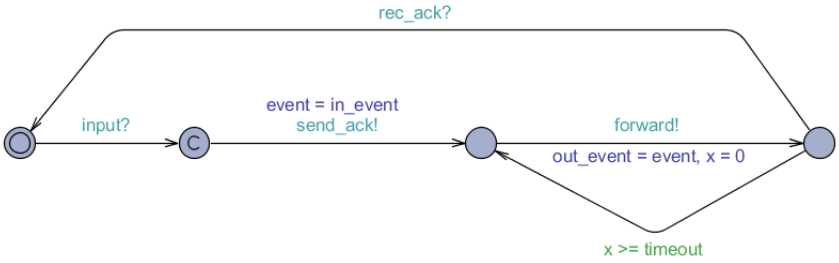

Figure 3.16: Uppaal (Variant A) - Node Automaton

For the sake of completeness, figure 3.17 shows the end node of the system. The logic for the level crossing is omitted. The parameters are given in the following:

```
broadcast chan &input, chan &ack, int &event
```

The parameters *input* and *ack* are used as before and *event* is the shared variable between the downstream neighbor (network node automaton) and the end node automaton itself.

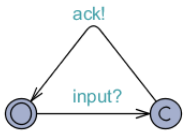

Figure 3.17: Uppaal - EndNode Automaton

The complete system consisting of a sensor, two network nodes and one level crossing node is specified in listing 3.3 and a graphical representation is given in figure 3.18. The sensor node is connected to node 1, node 1 is further wired with node 2. Finally, node 2 is connected to the end node.

```
// global declarations
typedef int[0,5] e_t;
```

```
const int timeout = 5;

// system declarations
broadcast chan sensorOut, node1Out, node2Out;
chan ackSensor, ackNode1, ackNode2;

int n1_event, n2_event, end_event;

// parameters: broadcast chan &forward, chan &ack, int &f_event
SensorNode = Sensor(sensorOut, ackSensor, n1_event);

// parameters: broadcast chan &input, broadcast chan &forward,
//    chan &send_ack, chan &rec_ack, int &in_event, int &out_event
Node1 = Node(sensorOut, node1Out, ackSensor, ackNode1,
             n1_event, n2_event);
Node2 = Node(node1Out, node2Out, ackNode1, ackNode2,
             n2_event, end_event);

// parameters: broadcast chan &input, chan &ack, int &event
End = EndNode(node2Out, ackNode2, end_event);

system SensorNode, Node1, Node2, End;
```

Listing 3.3: Uppaal (Variant A) - System declaration

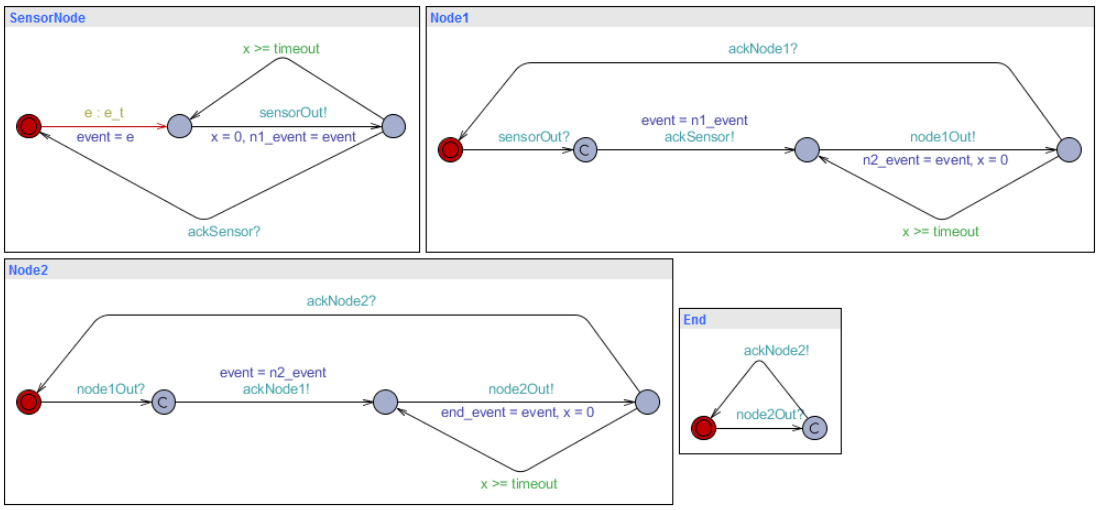

Figure 3.18: Uppaal (Variant A) - Example System

3.5.2 Use Case Implementation - Variant B

The template used for the sensor automaton is nearly the same as in variant A. Moreover, its graphical representation is identical to the figure 3.15. The only difference is that the channel *ack* is a broadcast channel instead of a binary channel. This modification has a semantical reason: a binary channel "fires" if and only if both sender and receiver are "ready". A broadcast channel can emit signals even if no receiver is ready. This behaviour is useful when multiple members of a group can emit such an *ack* signal, especially when timeouts occur and the failover mechanism takes effect. The remaining parameters and variables stay the same. The list of arguments is as follows:

```
broadcast chan &forward, broadcast chan &ack, int &f_event
```

The group member automaton depicted in figure 3.19 is the most complex graph contained in this variant. As the network node template in variant A, it comprises the logic for receiving and forwarding the event. In addition, the logic for handling the group internal coordination is contained as well. The current leader's ID is passed as an argument *leader* to every group member instance. Further, every instance gets its member ID *m_id*. In cases where the ID of the current leader and the member ID are equal, the automaton is responsible for processing the event and forwarding it. Beyond that, the leader notifies the other group members to prevent the execution of their failover mechanisms via the *g_ack* broadcast channel. Finally, after the group member has forwarded the event successfully, another synchronization *g_reset* is emitted to reset all group members such that a new incoming signal can be processed.

The failover mechanism for the passive group members is coupled to their local clocks x and a specific failover time *fo_time*. If the group leader does not synchronize the processing of the event via the *g_ack* synchronization and the timeout occurs such that the expression $x >= fo\_time$ evaluates to true, then a member claims the leadership and processes the event on its own. In order to prevent that multiple group members claim the group-leadership at the same time, the failover times shall be chosen differently.

Compared to variant A the forwarding of the event is enhanced in such a way that the repetition of the forwarding-process is limited to the value of *total_retries*. Every retry decrements the counter *r* and if its value reaches zero, the event will be discarded and the group will be reset. The complete list of arguments of the group member template is as follows:

```
const int fo_time, broadcast chan &input, broadcast chan &forward,
broadcast chan &send_ack, broadcast chan &rec_ack, int &in_event,
int &out_event, int &leader, int m_id, broadcast chan &g_ack,
broadcast chan &g_reset
```

The graphical representation of the template for the end node automaton is the same as depicted in figure 3.17. Because the realization of variant B uses broadcast channels for signalling acknowledgements, the only difference compared to A's template is the type of the argument *ack* as can be seen in the following list of parameters:

```
broadcast chan &input, broadcast chan &ack, int &event
```

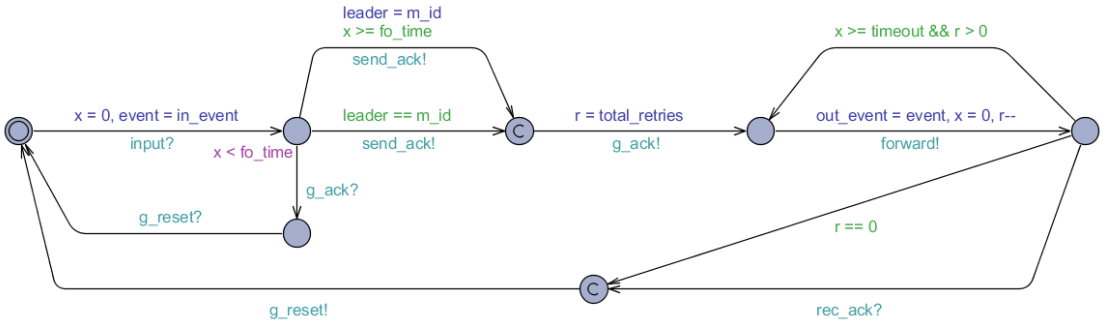

Figure 3.19: Uppaal (Variant B) - Group Member Automaton

Finally, listing 3.4 provides the specification of the example system. In total, there are six template instantiations, one sensor, four group members divided into two groups, and one end node. The sensor is connected to the first group, which is further wired to the second group. Additionally, the latter is connected to the end node.

```
// global declarations
typedef int[0,5] e_t;
const int timeout = 10;
const int total_retries = 5;

// template instantiations
broadcast chan sensorOut, group1Out, group2Out;
broadcast chan ackSensor, ackGroup1, ackGroup2;
broadcast chan g1_ack, g2_ack;
broadcast chan g1_reset, g2_reset;

int g1_exchange, g2_exchange, end_exchange;

const int g1m1_id = 11;
const int g1m2_id = 12;
const int g2m1_id = 21;
const int g2m2_id = 22;

int g1_leader = g1m1_id;
int g2_leader = g2m1_id;

//Parameters: broadcast chan &forward, broadcast chan &ack,
//            int &f_event
SensorNode = Sensor(sensorOut, ackSensor, g1_exchange);

//Parameters: const int fo_time, broadcast chan &input,
//            broadcast chan &forward, broadcast chan &send_ack,
//            broadcast chan &rec_ack, int &in_event,
//            int &out_event, int &leader, int m_id,
//            broadcast chan &g_ack, broadcast chan &g_reset
G1M1 = GroupMember(7, sensorOut, group1Out, ackSensor, ackGroup1,
                   g1_exchange, g2_exchange, g1_leader , g1m1_id,
```

```
                    g1_ack, g1_reset);
G1M2 = GroupMember(8, sensorOut, group1Out, ackSensor, ackGroup1,
                    g1_exchange, g2_exchange, g1_leader , g1m2_id,
                    g1_ack, g1_reset);

G2M1 = GroupMember(7, group1Out, group2Out, ackGroup1, ackGroup2,
                    g2_exchange, end_exchange, g2_leader , g2m1_id,
                    g2_ack, g2_reset);
G2M2 = GroupMember(8, group1Out, group2Out, ackGroup1, ackGroup2,
                    g2_exchange, end_exchange, g2_leader , g2m2_id,
                    g2_ack, g2_reset);

//Parameters: broadcast chan &input, broadcast chan &ack, int &event
End = EndNode(group2Out, ackGroup2, end_exchange);

system SensorNode, G1M1, G1M2, G2M1, G2M2, End;
```

Listing 3.4: Uppaal (Variant B) - System declaration

3.5.3 Conclusion

As already discussed, Uppaal's template concept provides a way to easily reuse models. Further, these templates can be supplied with various parameters. Therefore, the user can instantiate multiple instances of the same template with "configurable" behaviour.

Uppaal is not designed for the composition of single automata, as is possible with substitution transitions and subpages in CP-nets, for example. Uppaal provides so-called *partial template-instantiations*, where new templates can be created out of existing ones by specifying only a part of the existing template's parameters. But that does not cover composition in such a way as it has been seen in other modelling concepts.

Similar to Petri nets, Uppaal is not designed to create real-world applications, but to model, simulate and verify systems. Therefore, there is no separation between business and coordination logic. Once a declared system is instantiated according to its configuration, the simulation and verification engine is in the center of focus. Uppaal is static; there can be no changes on the model at runtime. Therefore, the described dynamics criterion is not supported. Due to the fact that Uppaal is based on the concept of finite state machines, it is not difficult to use and to create simple models with it — the criterion of simplicity is fulfilled.

Concerning addressing, Uppaal provides a sophisticated mechanism to synchronize processes and in connection with shared variables also to exchange information. But as Uppaal is static all these synchronizations have to be already considered in the modelling phase. Nevertheless, this criterion is seen as fulfilled as well.

In *Uppaal*, delayed execution can be realized such that a location is changed only if a certain condition on a clock evaluates to true. This prevents data items from being processed by the following locations. Furthermore, clock-based conditions can be used to switch the location and to execute certain routines. As seen in the implementations of the two variants of the use case, information between different automata is passed via shared variables. The only way to incorporate the "age" of this information is to include an additional clock for each data item

which is set to zero at the time the data item is created. Then, the difference between a global clock and this data item specific clock can be used in the following to determine whether the information is considered as expired or not. Basically, *Uppaal* is not meant for handling the age of single data items which are shared between variables. The focus lies on synchronization and not on the data items. In conclusion, the criterion regarding time is partially supported.

The advantages of broadcast channels coupled with Uppaal's template concept and its template instantiation mechanism help to keep the systems clear and small. For example, additional group member instances can be easily added without changing the underlying models. Therefore, Uppaal fulfils the scalability criterion too as it abstracts over the concrete number of template instantiations.

Regarding toolchain and documentation, Uppaal is a toolbox, which allows modelling, simulating, and verifying systems directly. The graphical user-interface supports the user while creating models and offers a syntax check for solving certain modelling issues. Beside the research articles, there is a comprehensive online documentation providing a central access point for information.

3.6 BPMN - Business Process Model and Notation

3.6.1 Use Case Implementation - Variant A

Figure 3.20 depicts the collaboration between sensor nodes and network nodes of variant A of the running train example realized using BPMN 2.0. The shown model contains two pools, each representing one participant of the variant. The focus of the model lies on the sensor node and its interaction with its neighboring network node. The network node pool is modelled as a black box, meaning that the internal structure is concealed. This allows us to describe the sensor node and to clarify its interaction with the network node, without confusing the reader with the internals of the network node pool.

The single activity in the sensor pool is a so called *Call Activity*. Basically, BPMN knows five types of sub-processes used for (de-)composition. These are: *Embedded*, *Event*, *Transaction*, *Ad-Hoc* and the already mentioned *Call Activity*. As only call activities are used in these examples, the remaining ones won't be further explained. Instead, the interested reader can find further information in the BPMN specification [46]. Call activities, formerly known (until BPMN 2.0) as *reusable sub-processes*, allow defining a reference to a globally defined process. As the process is global, it can be used by multiple activities across numerous pools. The corresponding global process *Forward* can be found in figure 3.21.

The process *Forward* is used to forward an event to a specified destination. This is achieved using the send task *Forward Event*. After sending the message, the process waits until it either receives an acknowledgement or a timeout occurs. This is realized using an event-based gateway coupled with a receive task (*Receive Acknowledgement*) and a timer intermediate event. In case that the timer fires it is attempted to forward the event again.

Figure 3.22 depicts the collaboration between a network node and the level crossing unit. Further, it shows the internal logic of the network node pool. As can be seen, the network node process uses also the global process *Forward*. The level crossing process contains a receive task,

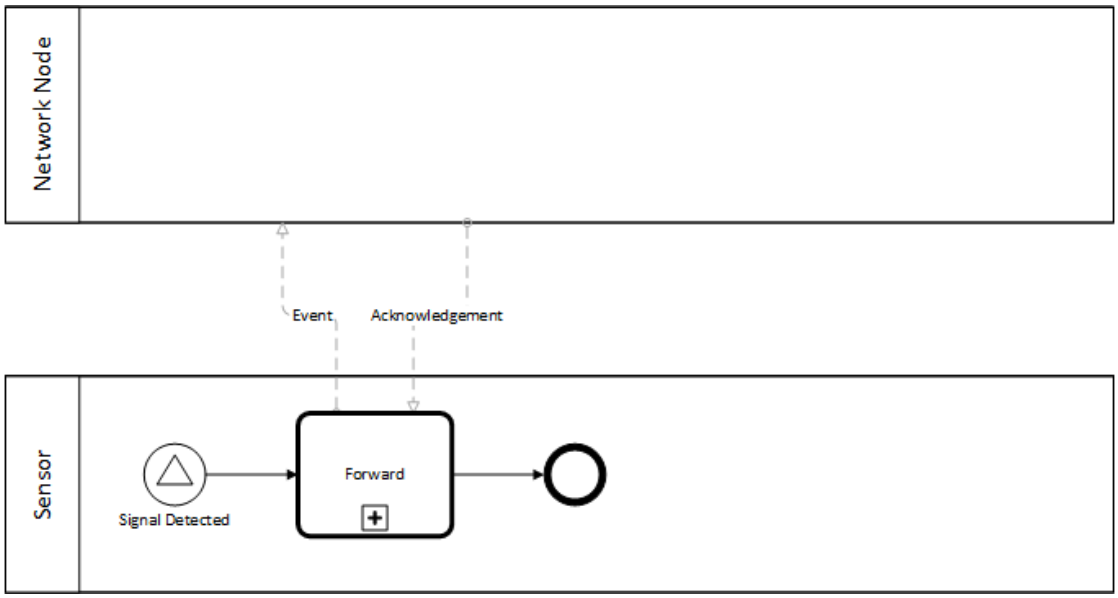

Figure 3.20: BPMN (Variant A) - Collaboration between Sensor and Network Nodes

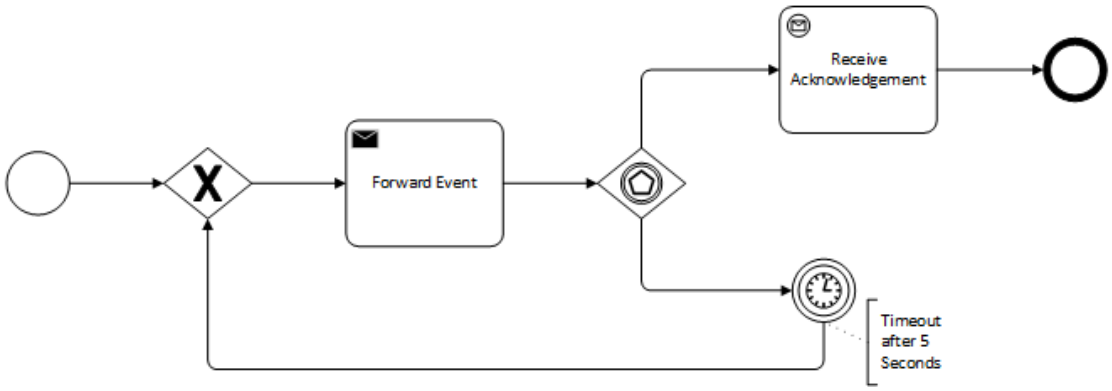

Figure 3.21: BPMN (Variant A) - Global Process *Forward*

a send task and a task for processing the event. Further, it distinguishes whether an event has already been processed or not. This is achieved using an exclusive gateway, which is represented as a diamond labeled with an "X". The collaboration model between two network nodes has been omitted, since the overall actions would be the same as between network node and level crossing unit.

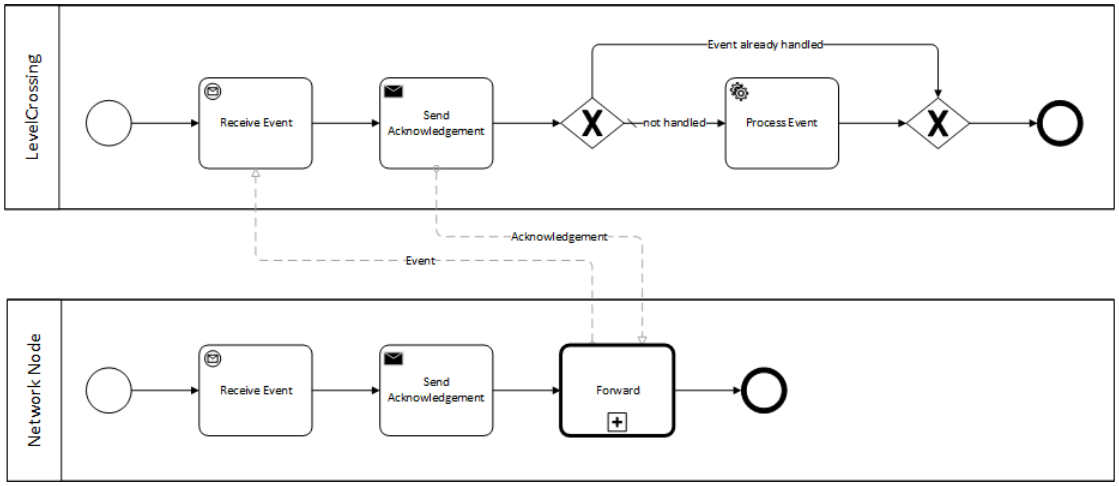

Figure 3.22: BPMN (Variant A) - Network and Level Crossing Collaboration

3.6.2 Use Case Implementation - Variant B

Figure 3.23 depicts the collaboration between the sensor and network nodes of variant B. It is almost identical to the model shown in variant A, except that the pool *Network Node* is renamed to *Group Member* and the black box pool is specified as a so-called *multi instance* pool which is pointed out by the three horizontal lines at the bottom. This allows expressing that a sensor node forwards an event to multiple members forming a group. The used global process *Forward* is the same as shown in figure 3.21.

The collaboration between level crossing and group member can be seen in figure 3.24. The group member pool contains two lanes, one for forwarding the event (*Event Processor*) and one for group internal coordination (*Group Coordinator*). Lanes are sub-partitions of pools and they allow to represent the roles involved in the process. Basically, the process is started when a group member receives an event. In the following, the current group status is retrieved and used as a basis for the decision whether the group member shall process the event because it is its turn or stay passive and let the group leader do the work. In the former case, the member sends an acknowledgement message to the sender of the event, followed by another acknowledgement message to its group members to notify them about the handled event. Finally, the event is forwarded to the upstream neighbor, in this case to the level crossing unit.

In the latter case, the group member process waits until it receives an acknowledgement message from the group leader. If a timeout occurs after a specified failover time, then the group

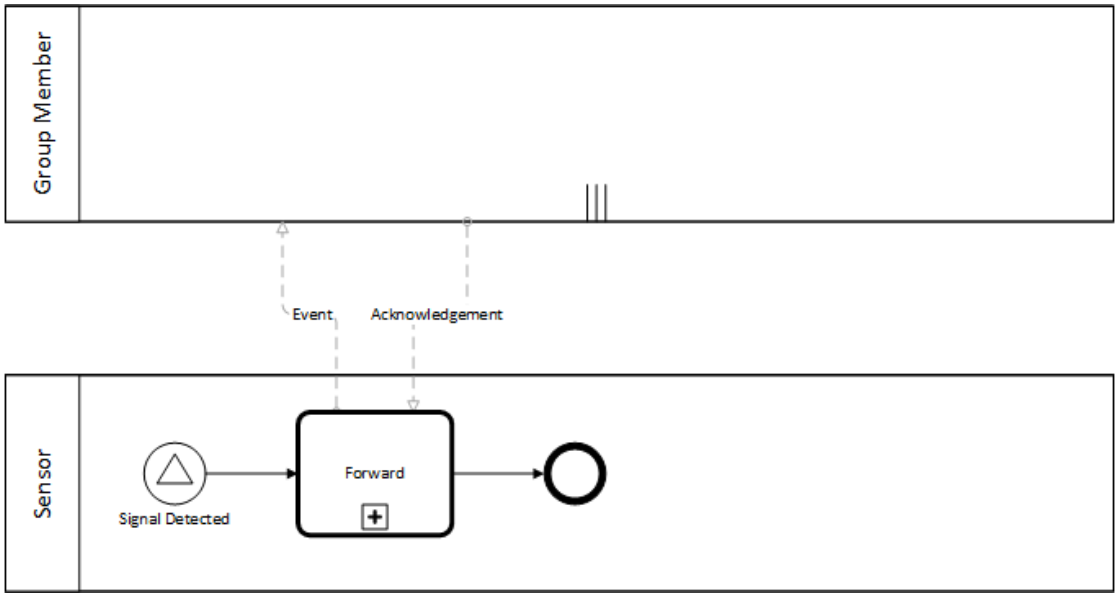

Figure 3.23: BPMN (Variant B) - Collaboration between Sensor and Group Members

member claims the leadership of the group and informs the other group members. After this, the process is resumed with the activities of the group leader described before.

In the left bottom corner of the figure the sequence for receiving and updating the group status can be seen. These kinds of messages are only sent inside a group. The task *Send Group Acknowledgement* of a member corresponds to the *Receive Group Acknowledgement* task of the other group members. In addition, the *Claim Group Leadership* task corresponds to the message start event *Receive Leadership Request*. Again, there is one member (the new leader) who sends the message and multiple receivers (the other group members).

In contrast to variant A, the processes in figure 3.24 start with a *message start event*. In variant A the processes start with an empty start event followed by a *Message Receive Task*. Both versions have the same effect. In BPMN there are almost always various modelling alternatives, however, these might be of different quality.

3.6.3 Conclusion

Compared to the previously discussed modelling concepts, BPMN is very high level and much more abstract. The various capabilities regarding composition and reuse have been already mentioned, but there is also further research ongoing in the area of process composition. The authors of [22] follow the strategy of composing more complex processes out of simple ones by introducing a set of so called *composition operators*.

As BPMN is more abstract, parametrization is completely different compared to the previous explained approaches. First of all, beside its graphical notation, BPMN 2.0 specifies also its execution semantics. The execution of BPMN models is not limited to its WS-BPEL mapping. The specification leaves the door open to execute models directly. Ultimately, the degree of support

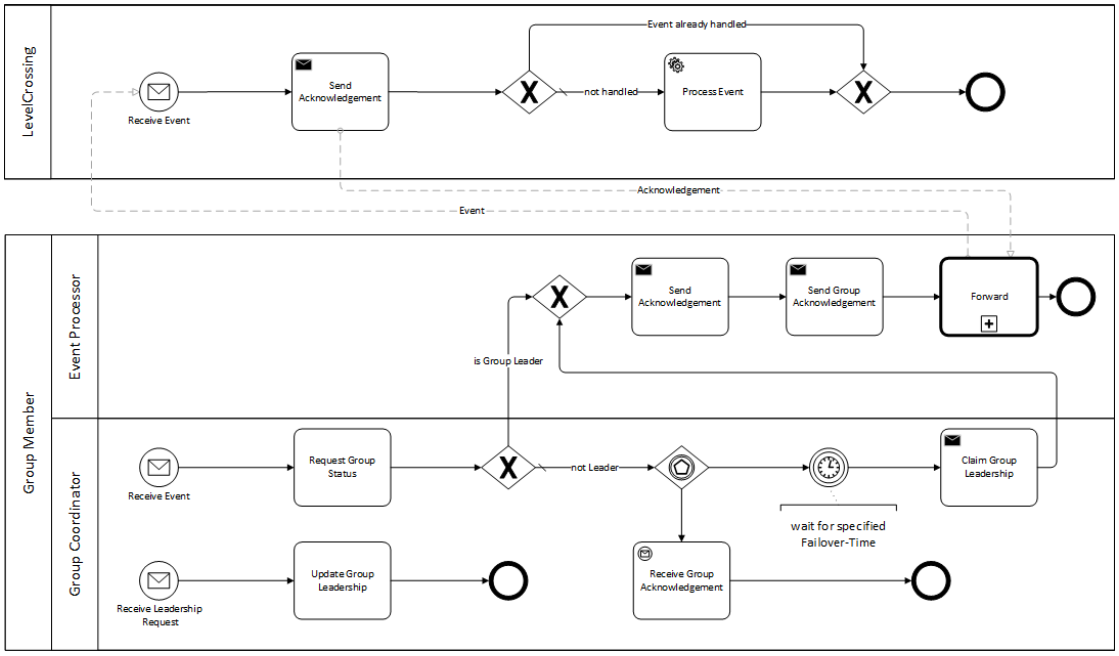

Figure 3.24: BPMN (Variant B) - Collaboration between Group Member and Level Crossing

depends on the concrete tool implementing the specification. As parametrization, dynamics and addressing are these criteria which are tightly coupled to the execution of the model, it is mostly up the concrete tool whether certain functionality is supported or not. *Data Objects* which are elements of the category of artifacts can be seen as way to include externally specifiable behaviour ($\rightarrow$ parameters) into the designed model. As the concept of *Data Objects* is clearly specified, parametrization is most likely available in the majority of such tools. Regarding dynamics, the specification is too "high level". The addition and removal of runtime elements is a factor where it depends on the realization of the concrete task or sub-process which interacts with the components whether it supports such kind of dynamics. Addressing is something which can be seen as realized as long as the modeller uses the specified send and receive tasks, message events, as well as message flows to explicitly illustrate the endpoints of communication.

The overall idea of BPMN is to model business processes; therefore the majority of its notational elements is designated to express business logic. (Atomic) Tasks and sub-processes are characteristical examples. Coordination is realized using gateways, sequences and message flows. In conclusion, it can be stated that BPMN supports the distinction between business- and coordination logic.

BPMN is "abstract enough" to cope with multiple instances of activities or pools. There are numerous so-called activity markers which allow controlling the execution behaviour and therefore also whether a specific activity is executed once or multiple times — such as when, for example, there is more than one neighbor. Further, there is an option to specify that a pool is a multi-instance pool meaning that the interaction is carried out with more than one partner without the need to explicitly visualize each of these partners in the model. Therefore, the

criterion for scalability is fulfilled as BPMN allows abstracting over the concrete number of components.

As seen in the implementations above, timer events allow triggering certain execution paths. The availability of concrete data items depend on the implementation of the specification. The specification contains no instruction whether the time when data items are available for flow objects shall be specifiable or not. The same holds for the expiration of data items. As the models are abstract, it is possible to include gateways checking whether data items have reached their deadline. The concrete realization is up to the tool implementing the specification. Therefore, the criterion describing the requirements regarding the integration of time is partially fulfilled.

There are multiple tools and systems implementing the BPMN standard. This starts from diagramming or vector graphic applications such as *Microsoft Visio*, which contain templates for drawing business processes, leading to complete bundles of various applications, which support also the deployment and execution of business processes such as the *Oracle Business Process Management Suite* or *ActiveVOS*. Regarding documentation, the standard itself is the first source of information. However, as Egon Börger remarked in [10], the standard contains, amongst others, numerous ambiguities and underspecifications which could, due to a lack of precision, lead to incompatibilities. Nevertheless, there is plenty of documentation available which supports users in learning the subtleties of the modelling language. Therefore, the criterion toolchain and documentation is satisfied.

Concerning simplicity, it is not a complicated task to model a simple business workflow. But as the complexity increases, more and more rules have to be observed and often various modelling alternatives might be suitable. The user is faced with modelling decisions and it is completely unclear whether one alternative should be favoured against another one. Thus, the criterion for simplicity is only partially fulfilled.

3.7 WS-BPEL

3.7.1 Use Case Implementation - Variant A

Figure 3.25 shows the graphical representation of the sensor node's WS-BPEL process. The corresponding WS-BPEL and WSDL code parts can be found in the appendix A.1. The example has been realized with *Apache ODE* [3] and the *BPEL Designer* plugin for the *Eclipse* platform.

The sensor process is implemented as a synchronous process which is initiated by invoking the *initiate* operation. This invocation is expressed by the *receive* activity named *receiveEvent* in the main sequence. The more interesting logic part is located in the *repeatUntil* loop. The signal is forwarded to the upstream node process by invoking (*ForwardEvent*) an asynchronous web-service of the registered partner link *ForwardEventPL*. Afterwards, a structured activity for selective event processing is used. Either the upstream node process responds with an acknowledgement message or a timeout occurs on the *onAlarm*-branch. The former results in the termination of the loop, the latter decreases a counter variable regulating how often a retransmission is tried. Listing 3.5 gives an excerpt of this mentioned selective event processing. The *pick*-activity has two branches, *onMessage* and *onAlarm*. In case of an incoming message, the flag (variable *endLoop*) for stopping the loop is set to *true* by using an *assign*-statement. The

alarm on the other branch is specified by the expression *'PT5S'*, which fires after 5 seconds and as a result the variable *counter* is decreased using an *assign*-statement as well.

The loop continues until a response arrives or the total amount of retries is exhausted. The if-statement following the loop is used to generate a respective response message for the caller of the sensor process. Finally, the synchronous process terminates after executing the *reply* activity named *replyOutput*.

```
<bpel:pick name="Pick">
  <bpel:onMessage partnerLink="ForwardEventPL"
                  operation="onAcknowledgement"
                  portType="np:NodeProcessCallback"
                  variable="ReceiveAckMessage">

    <bpel:assign validate="no"
                 name="endLoop">
      <bpel:copy>
        <bpel:from>
          <bpel:literal xml:space="preserve">true</bpel:literal>
        </bpel:from>
        <bpel:to variable="terminate"></bpel:to>
      </bpel:copy>
    </bpel:assign>
  </bpel:onMessage>

  <bpel:onAlarm>
    <bpel:for>'PT5S'</bpel:for>
    <bpel:assign name="decreaseCounter">
      <bpel:copy>
        <bpel:from
          expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"
              >
          <![CDATA[$counter - 1]]>
        </bpel:from>
        <bpel:to variable="counter"></bpel:to>
      </bpel:copy>
    </bpel:assign>
  </bpel:onAlarm>
</bpel:pick>
```

Listing 3.5: WS-BPEL - Sensor Process Excerpt

3.7.2 Use Case Implementation - Variant B

The graphical representation of the WS-BPEL process for a single group member is depicted in figure 3.26. The corresponding WS-BPEL and WSDL code is listed in the appendix A.2. In contrast to the previous example this process is realized as an asynchronous WS-BPEL process. It contains three partner links as presented in listing 3.6. The partner link *downstream* is used to communicate with the downstream nodes. It embodies two roles: one describes the services the group member provides and the other one describes the single service used as callback. The second partner link *group* provides access to the group coordination web-services. It allows

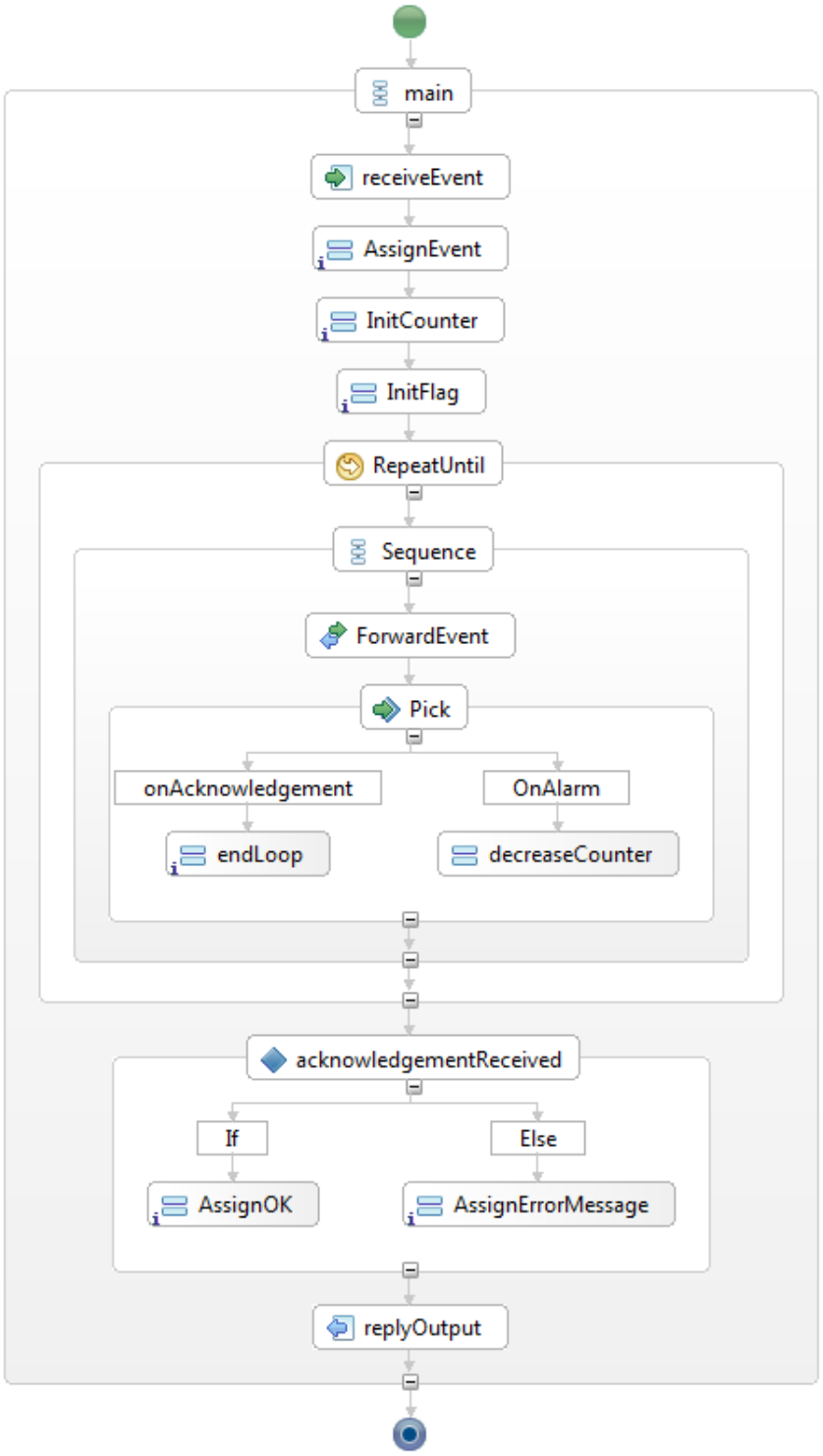

Figure 3.25: WS-BPEL (Variant A) - Graphical representation of the Sensor Process

the WS-BPEL process to gain information about the state of the group, claim its leadership or acknowledge the completed processing of a certain event. The final partner link *forward* is used to pass the event to an outsourced web-service responsible for forwarding it to the upstream group. This outsourced service is a modification of the previously described sensor process with the difference that the event shall be forwarded to multiple nodes instead of one.

```
<bpel:partnerLinks>
  <bpel:partnerLink name="downstream"
                    partnerLinkType="tns:BPELForward"
                    myRole="BPELForwardProvider"
                    partnerRole="BPELForwardRequester" />

  <bpel:partnerLink name="group"
                    partnerLinkType="tns:GroupCoordinationWS"
                    partnerRole="groupOps"/>

  <bpel:partnerLink name="forward"
                    partnerLinkType="tns:ForwardSignalWS"
                    partnerRole="forwardSignal"/>
</bpel:partnerLinks>
```

Listing 3.6: WS-BPEL (Variant B) - Group Member Process Excerpt - Partner Links

It is assumed that every group member has a specified node ID which is stored in the variable *nodeID*. This ID is used along the main sequence to check whether the member is the leader of the group or not. This is accomplished by calling a corresponding web-service of the *group* partner link. The following conditional statement splits the sequence of execution into two possible branches. If the current instance is the leader then the group is notified that the event is handled by it. The leader then delegates the event to a web-service managed through the *forward* partner link. The final action of this branch is sending an acknowledgement message to the caller of the process. The second branch of the condition starts with a selective event processing (*pick*). Basically, a group member waits a specified amount of time, after this time has elapsed, the member notifies the other group participants via the *group* partner link that it takes over the group leadership. The remaining steps are the same as in the other branch. The event is delegated to another web-service and an acknowledgement message to the caller is sent. In the case that an acknowledgement message from another group member arrives during the specified waiting period, the WS-BPEL process exits. This is possible because another group member has already handled the event.

It has to be noted that both variants were realized without using correlation. Correlation is used to identify the messages which belong together. Basically, events and signals will be handled concurrently. Therefore, the system must discern to which event certain messages (for example the acknowledge messages) belong. Multiple instances of WS-BPEL processes run in parallel and the messages have to be routed to the right instances. This is achieved using the correlation functionality of WS-BPEL where certain parts of the messages are specified as the properties used for correlation.

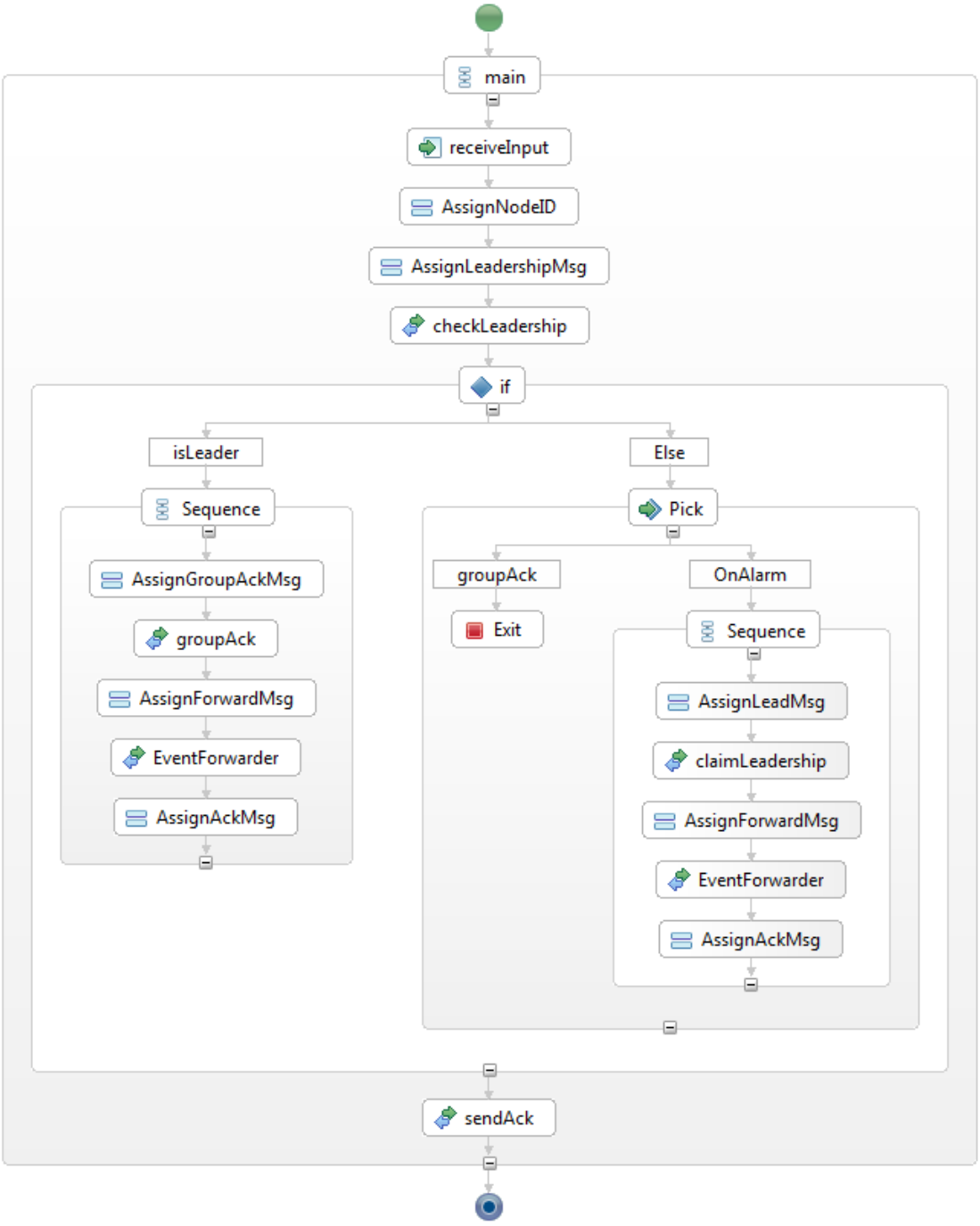

Figure 3.26: WS-BPEL (Variant B) - Graphical representation of the Group Member Process

3.7.3 Conclusion

Composition is a central factor in WS-BPEL — web-services can be composed to form more complex web-services. A small weakness is that WS-BPEL does not support the definition of process-fragments or sub-processes which could be reused within a process or across multiple processes as explained in [24]. The typical workaround is to outsource these process-fragments into a particular web-service, which could be called on a by-need basis. Then, the former "parent" process as well as other web-services can make use of it. Nevertheless, both composition and reuse criteria in the context of this evaluation are fulfilled. The authors of [24] examine various kinds of sub-processes and provide syntax and semantics for their WS-BPEL extension. Concerning parametrization, web-services provide a natural support for passing certain parameters. As seen in the sensor process example, the occurred event has been passed as a parameter. Parameters are not restricted to simple types as used in the presented examples — *WSDL* in conjunction with *XML-Schema* [57] allow to define much more complex messages.

Basically, business logic is encapsulated in web-services and the WS-BPEL process specifies in which order and how they are to be used. Coordination logic can be expressed using programming language-like constructs as seen in the example using loops, conditional statements, assignments and so on. Therefore, the separation between business- and coordination logic can be considered as satisfied as well.

Due to its support for composition, WS-BPEL allows the definition of flexible and dynamic processes. The example regarding the inclusion of a further group member has been solved in the second variant by outsourcing the group coordination and forwarding functionality to individual web-services. Therefore, it is not necessary to modify the original WS-BPEL process when changes in the group neighborhood occur. The outsourced services can be themselves WS-BPEL processes or any other kind of technology which can be published as web-services. WS-BPEL is code based and its control structures allow abstracting over the concrete number of components which are incorporated into the process. The criterion for scalability is fulfilled.

Single nodes are not directly connected. Instead, they interact by invoking web-services of each other. Such implicit connections as required for the addressing criterion are given. A network node running a WS-BPEL process can call any other network node as long as the target node provides a web-service to call. Further, the recipient can identify the caller and therefore it can also perform callbacks. The underlying technologies such as *SOAP*, *WSDL* and *WS-Adressing* provide the necessary basis for this.

The execution of a process can be delayed using the *wait* activity which allows one to specify a period of time or a even deadline. Moreover, the *pick* activity or the event handler of scopes support the execution of concrete branches if particular time-based conditions are met. There is no direct support for the expiration of data items. Timestamps can be added to the data types specified in the WSDL part and conditional statements can be used in the process implementation to filter out such expired items. A similar workaround is possible if data items shall not be processed before a particular point in time is reached. As the data item specific time handling is not directly supported, the criterion regarding time is only partially fulfilled.

Concerning simplicity and especially compared to the previously discussed modelling concepts, much effort is needed to create even simple processes. The formerly mentioned underlying technologies are a downside here since they require additional background knowledge. The act

of writing WS-BPEL processes and web-services is tremendously verbose.

Further, the used toolchain for solving the variants of the use case is far away from being technically mature. The *BPEL Designer* plugin which has been used as graphical designer for the processes has a multitude of bugs which make the creation of WS-BPEL processes hard work. It would have been faster to directly code the WS-BPEL process and the WSDL files. *Apache ODE*, which has been used as WS-BPEL engine, has its flaws too. For example, the current version does not support *WS-Addressing* which makes the testing of especially asynchronous WS-BPEL processes extremely cumbersome. However, there are a lot of WS-BPEL engines available, both Open Source and proprietary, which have been used successfully in practice. Compared to BPMN, the specification of WS-BPEL is better structured and therefore, it is easier to find certain information. Thus, the criterion toolchain and documentation is considered as fulfilled.

3.8 Actor Model

3.8.1 Use Case Implementation - Variant A

Both variants of the example use case were realized using the Java programming language and the Open Source library Akka [2] in the version 2.2.3. Beside its availability for Java, Akka is incorporated into the standard library for the Scala programming language.

Actors are components encapsulating state and behaviour; their only way of communication is by exchanging messages. Every actor has a so called mailbox in which the incoming messages are placed. In Akka, the mailbox is managed by the system; messages are processed by calling the actor's *onReceive* method. Akka's default behaviour is that messages are handled in FIFO order.

Basically, the Actor Model follows the "one actor is no actor" strategy. Therefore, actors "come in systems". Actors build hierarchies by creating child actors, split tasks into smaller ones and delegate them to their child actors. The hierarchy can be as deep as necessary for processing the tasks in manageable pieces. Child actors are supervised by their single parent actor. The handling of failures occurs in the opposite direction: if one actor is not able to handle a situation, it can send a failure message to its parent. Therefore, errors can be handled on the right level. Akka provides comprehensive support for this; different configurable supervisor strategies define what should happen with a child actor in case of a failure. For the concrete life cycle of actors in Akka and the various pre-defined strategies, the reader is enjoined to peruse the Akka documentation [2].

The code for the *Sensor* actor is provided in listing 3.7. Every *Sensor* actor is initialized with the address of its upstream neighbor. One of the benefits of Akka is that it does not distinguish between local and remote actors. As the documentation states "Akka is distributed by default". [2] Actors in Akka are location transparent. For the developer, there exists no difference whether a system of actors runs on a local machine or is distributed on multiple network nodes; the code to be written is the same. There is no need for an additional software layer as often seen in various programming languages when remote functionality is required. Further, Akka does not require that parent and child actors are located on the same network node. To

achieve this location transparency, actors are addressed using paths. The structure of a path and a concrete example is given in the following:

```
akka.<protocol>://<actorsystemname>@<hostname>:<port>/<actor path>
akka.tcp://train@10.0.0.1:2552/user/sensor/forwarder
```

The *user* actor of the example above denotes the parent-actor of all user-created actors. Its termination leads also to the shutdown of all created actors.

```
//...
public class Sensor extends UntypedActor {
   private LoggingAdapter log =
      Logging.getLogger(getContext().system(), this);
   private final String upstream;

   public Sensor(String upstream) {
      this.upstream = upstream;
   }

   public static Props makeProps(String upstream) {
      return Props.create(Sensor.class, upstream);
   }

   @Override
   public void onReceive(Object o) throws Exception {
      if(o instanceof EventMsg) {
         EventMsg m = (EventMsg) o;
         log.info(m.toString());

         this.getContext().actorOf(Forwarder.makeProps(this.upstream, m));
      }else {
         unhandled(o);
      }
   }
}
```

Listing 3.7: Akka - Sensor Actor

The *onReceive* method in listing 3.7 shows that the sensor actor handles only messages of type *EventMsg*. The actor's only task is to create a new instance of a *Forwarder* actor, which is responsible to forward the event. This structure allows a fast processing of incoming events. Every event is handled by a single lightweight child actor.

The code for the already mentioned *Forwarder* actor is given in listing 3.8. The actor is initialized with the event to forward and the address of the receiver. The *preStart* method is invoked after the actor is created and used to instantiate a scheduler which repeatedly resends the event every 5 seconds. In order to send a message, the receiver needs to be looked up beforehand. This is achieved using the *actorSelection* method of the actor's context. The path specified on the method call can also contain wildcards, which allows for multiple actors to be addressed and to receive messages. The *tell* method of the *ActorSelection* sends the message specified on the first parameter asynchronously. The second parameter transfers the actor's

address which allows the receiver to reply. Furthermore, a receive timeout is set which leads to a *ReceiveTimeout* message after 25 seconds.

The remaining *onReceive* method completes the actor's functionality. In case of an *AckMsg* message, the resending scheduler is stopped and the actor terminates. If a *ReceiveTimeout* message arrives after the specified amount of time, then the actor is stopped without forwarding the event.

```
//...
public class Forwarder extends UntypedActor {

   private LoggingAdapter log =
      Logging.getLogger(getContext().system(), this);
   private final String upstream;
   private final EventMsg event;
   private Cancellable handler;

   public Forwarder(String upstream, EventMsg event) {
      this.upstream = upstream;
      this.event = event;
      this.handler = null;
   }

   public static Props makeProps(String upstream, EventMsg event) {
      return Props.create(Forwarder.class, upstream, event);
   }

   @Override
   public void preStart() throws Exception {
      super.preStart();

      final ActorSelection selection =
         getContext().actorSelection(upstream);
      this.handler =
         this.getContext().system().scheduler()
             .schedule(
                   FiniteDuration.Zero(),
                   Duration.create(5L, TimeUnit.SECONDS),
                   new Runnable(){
                     @Override
                     public void run() {
                        selection.tell(event, getSelf());
                     }
                   }, this.getContext().dispatcher());
      this.getContext()
          .setReceiveTimeout(Duration.create(25L, TimeUnit.SECONDS));
   }

   @Override
   public void onReceive(Object o) throws Exception {
      if (o instanceof AckMsg) {
         log.info(o.toString());
         this.handler.cancel();
         this.getContext().stop(getSelf());
```

```
        } else if (o instanceof ReceiveTimeout) {
            log.info("Timeout occured, could not forward event!");
            this.handler.cancel();
            this.getContext().stop(getSelf());
        } else {
            this.unhandled(o);
        }
    }
}
```

Listing 3.8: Akka (Variant A) - Forwarder Actor

Listing 3.9 gives an excerpt of the *NetworkNode* actor. In case of an incomming *EventMsg* message, the actor replies to the sender a new acknowledgement message *AckMsg*. Moreover, if the received event has not been processed before, then a new *Forwarder* child actor is created and charged with forwarding the event. If any other message arrives, it is passed to the *unhandled* method which leads by default to debug output.

```
//...
public class NetworkNode extends UntypedActor {
    //...
    @Override
    public void onReceive(Object o) throws Exception {
        if(o instanceof EventMsg) {
            EventMsg m = (EventMsg) o;
            log.info(m.toString());

            //send ack
            this.getSender().tell(new AckMsg(m.getID()), this.getSelf());

            if(!this.processed.contains(m.getID())) {
                this.processed.add(m.getID());
                this.getContext().actorOf(
                        Props.create(Forwarder.class, this.upstream, m));
            }
        }else
            unhandled(o);
    }
}
```

Listing 3.9: Akka - Network Node Actor

For the sake of completeness, listing 3.10 shows the *onReceive* method of the *LevelCrossing* actor. Similar to the *NetworkNode* actor, the actor sends an acknowledgement message back to the sender. Furthermore, if the event has not been processed, then the corresponding business code has to be executed.

```
//...
public class LevelCrossing extends UntypedActor {
    //...
    @Override
    public void onReceive(Object o) throws Exception {
        if(o instanceof EventMsg) {
```

```
            EventMsg m = (EventMsg) o;
            log.info(m.toString());
            // send ack
            this.getSender().tell(new AckMsg(m.getID()), this.getSelf());

            if(!this.processed.contains(m.getID())) {
                this.processed.add(m.getID());

                //
                // call business code here
            }
        }else
            unhandled(o);
    }
}
```

Listing 3.10: Akka - Level Crossing Actor

3.8.2 Use Case Implementation - Variant B

In variant B, an event is forwarded to multiple nodes. Therefore, the *Sensor* actor is initialized with a list of addresses. This list is then, on each event, passed to the created *Forwarder* child-actor. Beside the fact that the single address string is replaced by an *ArrayList<String>*, the *Sensor* actor stays the same as already shown in listing 3.7.

The *Forwarder* actor needs to be adjusted in such a way that the event is forwarded to multiple actors. Therefore, the *preStart* method needs to be modified as shown in listing 3.11.

```
//...
public class Forwarder extends UntypedActor {
    //...
    private final ArrayList<String> upstream;
    //...
    @Override
    public void preStart() throws Exception {
        super.preStart();

        final ArrayList<ActorSelection> list =
            new ArrayList<>();
        for(String path : upstream)
            list.add(this.getContext().actorSelection(path));

        this.handler =
            this.getContext().system().scheduler()
                .schedule(
                    FiniteDuration.Zero(),
                    Duration.create(5L, TimeUnit.SECONDS),
                    new Runnable(){
                        @Override
                        public void run() {
                            for(ActorSelection selection : list)
                                selection.tell(event, getSelf());
                        }
```

```
                }, this.getContext().dispatcher());
        this.getContext()
            .setReceiveTimeout(Duration.create(25L, TimeUnit.SECONDS));
    }
    //...
}
```

Listing 3.11: Akka (Variant B) - Forwarder Actor

Between the *Sensor* actor and the *LevelCrossing* actor the event is passed by *GroupMember* actors which replace the *NetworkNode* actors of variant A. Each group is formed by one or more *GroupMember* actors. The group leader forwards the event to the members of the upstream group. For this purpose, each actor is initialized with a list of the upstream nodes, a list of its group members and a flag whether the actor is the group leader at start. In addition, a failover time is specified which corresponds to the amount of time a group member waits before it takes over an unprocessed event and the group leadership.

The *GroupMember* actor shown in listing 3.12 embodies the following functionality: if an *EventMsg* message arrives and the considered actor is the group leader, then the actor replies with an acknowledgement message to the sender. In addition, it creates a *Forwarder* actor as usual. Finally, the leader needs to notify the other group members about the processed event. Therefore, a *GroupAckMsg* message is sent to all group members. If the considered actor is not the current leader of the group, then the actor registers a handler which gets activated when the specified failover time elapses. In the handler's *run* method, the same message exchange is done as explained before. In order to stop the handler from being executed when the group leader has processed the event, a reference to the handler is stored in a hashmap. If a *GroupAckMsg* message is received and a handler is registered for the corresponding event in the mentioned hashmap, then the handler is deleted to prevent the execution of the failover mechanism.

```
//...
public class GroupMember extends UntypedActor {
    //...
    public GroupMember(ArrayList<String> upstream, ArrayList<String> members,
                       FiniteDuration failover, boolean isLeader) {
        //...
    }

    @Override
    public void onReceive(Object o) throws Exception {
        if (o instanceof EventMsg) {
            final EventMsg m = (EventMsg) o;

            if (this.isLeader) {
                //send ack
                this.getSender().tell(new AckMsg(m.getID()), this.getSelf());

                if (!this.processed.contains(m.getID())) {
                    this.processed.add(m.getID());
                    this.getContext()
                        .actorOf(Props.create(Forwarder.class, this.upstream, m));
```

```
                GroupAckMsg g_ack = new GroupAckMsg(m.getID());
                for(String path : members)
                    this.getContext()
                        .actorSelection(path).tell(g_ack, this.getSelf());
            }
        }else {
            final ActorRef sender = getSender();

            Cancellable c =
                this.getContext().system().scheduler()
                    .scheduleOnce(
                      this.failover,
                      new Runnable(){
                          @Override
                          public void run() {
                              log.info("failover starts");
                              sender.tell(new AckMsg(m.getID()), getSelf());
                              isLeader = true;

                              if (!processed.contains(m.getID())) {
                                  processed.add(m.getID());
                                  getContext().actorOf(
                                      Props.create(Forwarder.class, upstream, m));

                                  GroupAckMsg g_ack = new GroupAckMsg(m.getID());
                                  for(String path : members)
                                      getContext().actorSelection(path)
                                                  .tell(g_ack, getSelf());
                              }
                          }
                      }, this.getContext().dispatcher());
            this.handler.put(m.getID(), c);
        }
      } else if(o instanceof GroupAckMsg) {
        GroupAckMsg g = (GroupAckMsg) o;
        log.info(g.toString());
        if(this.handler.containsKey(g.getID())) {
            this.handler.get(g.getID()).cancel();
        }
        this.isLeader = false;
      }
      else {
        unhandled(o);
      }
    }
}
```

Listing 3.12: Akka - Group Member Actor

As the *LevelCrossing* actor does only receive events, no modifications are required and therefore, the actor is exactly the same as presented in listing 3.10 in variant A.

3.8.3 Conclusion

As explained, actors form hierarchies. Complex tasks are split up and are delegated to child actors which are supervised by their parent actor. According to the axioms, actors can create a finite number of child actors while processing a message. There is no limitation regarding the amount and type of child actors. Thus, there can be multiple instances of actors too. To sum things up, the possibility to create such hierarchical structures fulfils the composition criterion described earlier in this work. Moreover, multiple independent instances of actors can be used throughout the system and therefore also the reuse criterion is satisfied.

Parametrization depends on the concrete realization of the Actor Model, therefore this criterion will be evaluated with respect to Akka. As already presented but not yet mentioned, in Akka, actors are created by calling the *actorOf* method of the actor's context. See for example listing 3.7. The method's only argument is an instance of the class *Props*, which is a configuration class to specify options for the creation of actors. It allows the introduction of both system- and user-defined parameters. Furthermore, Akka makes heavy use of configuration files, which can be complemented by user-defined information as well. In conclusion, the criterion concerning parametrization is fulfilled.

Actors can be added and removed arbitrarily at runtime. In Akka, actors are lightweight components and according to its documentation about 2.7 million actors per GB RAM are supported. Therefore, systems created with Akka allow scaling up to meet the user's demands. Just like WS-BPEL, Akka is code based and its control structures allow abstracting over the concrete number of components which are incorporated into the process. Thus, the criteria regarding dynamics and scalability are both fulfilled.

In Akka, separation of concerns is not directly given. But as common in software development it can be introduced using a layered design, e.g. a particular business layer which is accessible using a defined interface. Then, the coordination logic comprising the messages, actor life cycles, schedulers and so on can be separated from the business logic. As encapsulation is a common task in software development, this criterion can be seen as supported too.

The Actor Model is all about messages, therefore addressing is ubiquitously used. As already explained, Akka is completely location transparent. The developer can write code without wondering about network communication issues. There is no difference in communicating with local or remote actors. Actors can be looked up using a path based mechanism where wildcards are supported as well. Thus, the criterion concerning addressing is satisfied.

The conceptual Actor Model makes no mention of the factor time. Actors create further actors, send messages and can change their behaviour. There are no rules regarding the temporal validity of messages or delayed execution. Therefore, it is up to the concrete realization of the Actor Model whether such time based processing is provided. As seen in the examples, Akka allows scheduling the execution of a given function or the sending of a message. Therefore, time triggered execution is supported. Moreover, a message which shall not be processed before a specific point in time is reached can be "delayed" with such a scheduler. On the other hand, Akka does not support the expiration of messages. There are two potential workarounds, both require to include a timestamp into the message data. The first is the implementation of an appropriate mailbox replacing the default mailbox of Akka which removes expired messages. The second is to ignore such invalid messages in the particular *onReceive* method. As there is

no direct support in Akka, the criterion regarding time is only partially fulfilled.

The Actor Model has a clear set of rules, the "usability" depends on the used tool implementing the Actor Model. The model itself has no graphical notation; systems have to be realized by coding. Thus, programming skills are a necessary prerequisite. Beside the required installation of Java or Scala and the Akka library itself no further preparations are required. The Akka documentation is comprehensive and supplemented with numerous examples. Non software developers would have their problems with the Actor Model (as well as with the previously presented concepts), but for the group of software developers the Actor Model and especially Akka can be learned easily. In conclusion, the criterion regarding simplicity is fulfilled and due to the lack of a graphical tool supporting the creation of actors, the criterion toolchain and documentation is partially fulfilled.

3.9 Classification Summary

In this section the previously conducted textual evaluation of the various modelling concepts regarding the initially described criteria is graphically summarized in form of a table. In table 3.1 the results of the classification can be found. A '$+$' denotes that the related approach fulfils the considered criterion, whereas a '$\sim$' denotes that the criterion is partially fulfilled. A '$-$' indicates non-fulfilment.

	CPN	**Reo**	**Uppaal**	**BPMN**	**WS-BPEL**	**Actor Model**
Composition	$+$	$+$	$-$	$+$	$+$	$+$
Reuse	$+$	$+$	$+$	$+$	$+$	$+$
Parametrization	$\sim$	$+$	$+$	$+$	$+$	$+$
SoC	$-$	$+$	$-$	$+$	$+$	$+$
Dynamics	$-$	$+$	$-$	$\sim$	$+$	$+$
Addressing	$-$	$\sim$	$+$	$\sim$	$+$	$+$
Scalability	$-$	$-$	$+$	$+$	$+$	$+$
Time	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
Toolchain & Docs	$\sim$	$\sim$	$+$	$+$	$+$	$\sim$
Simplicity	$+$	$-$	$+$	$\sim$	$-$	$+$

Table 3.1: Classification of the considered modelling concepts and tools

As can be seen in table 3.1, none of the considered approaches fulfils all presented and desired criteria. The Actor Model obtains the best results as it has only two criteria which are not completely satisfied. Table 3.2 provides an overview of the biggest strengths and weaknesses of the modelling concepts and tools gathered throughout the evaluation. The complexity of these tools (3 out of 6) can be considered as the biggest problem, as it requires often enormous knowledge to design even simple models. On the other hand, composition is the most commonly satisfied criterion (3 out of 6).

	biggest strength	biggest weakness
CPN	composition	scalability
Reo	composition	simplicity
Uppaal	reuse	composition
BPMN	composition	simplicity
WS-BPEL	addressing	simplicity
Actor Model	addressing	time

Table 3.2: Overview of biggest strengths and weaknesses

3.9.1 Concluding Remarks

As seen throughout this chapter, the use case realizations of the different modelling concepts and tools have some slight deviations in their functionality. The solutions for Uppaal, WS-BPEL, and the Actor Model contain further logic which stops the event forwarding process after a specified amount of failed attempts. Such a logic would have been possible in the CPN realization as well, but it would have ended in more complex nets and was therefore omitted. The same holds for Reo; an additional sub-connector would have been necessary which counts the attempts and influences the behaviour of the *Forwarder* connector. In BPMN, the global process *Forward* would require an additional exclusive gateway to integrate such a case distinction.

The realizations of the use case for Uppaal and Reo support the processing of events only in a sequential way. Reo is channel based, events are processed one after another. As mentioned, in Uppaal, a concurrent processing is possible; it requires more complex data structures and user-defined functions for event correlation.

WS-BPEL and Reo were these modelling concepts which required by far the most time to (a) get familiar with the semantics and (b) realize both variants of the example use case. Especially the semantics of Reo was hard to understand; many examples and a trial and error approach were necessary to get a feeling for it. As mentioned, several bugs in the used toolchain caused problems when solving the examples with WS-BPEL, and as a consequence these cost a lot of time. The Actor Model was the concept which was the “simplest” to learn, followed by Uppaal, which benefited from its automaton-based nature.

In the following chapters of this work a pattern concept for the *Peer Model* is introduced with the aim to provide a high level of software variability. This concept will be evaluated in the same fashion as the approaches in this chapter. Finally, in chapter 7, the *Peer Model* extended by the pattern concept and the Actor Model are compared in more detail with a newly introduced criterion.

CHAPTER 4

Peer Model

This chapter aims to provide the technical background for readers who are not familiar with the *Peer Model*. It gives a brief introduction to the important concepts and the graphical notation of the *Peer Model*, which will be relevant throughout the upcoming chapters of this work.

4.1 Basics

Typically, highly concurrent applications consist of multiple, distributed and parallel running nodes, where the communication and synchronization between nodes is the relevant factor to perform certain tasks. This administration of communication and synchronization between these nodes is what is generally known as *coordination*. The *Peer Model* [31] is a programming model for the modelling of concurrent and distributed systems with an emphasis on such coordination tasks.

In contrast to other modelling approaches such as Petri nets, which are designed to be domain independent in order to support a wide range of application areas, the *Peer Model* focuses on distributed environments and tries to bridge the gap between design and implementation. It connects the designer's view of clean and verifiable systems with the developer's aim of "getting things done" by providing a language to design, analyze and implement highly concurrent distributed systems.

Central components are so called *peers*, and models can consist of one or more such peers. Composition is an important factor as it allows forming more complex models out of "simpler" ones and encapsulating certain functionality. Another characteristic is the possibility to reuse such peers. To keep models readable at the one hand and maintainable on the other hand, the *Peer Model* follows a clear separation of coordination- and business-logic (application-logic). This is achieved by "outsourcing" the business logic from the designed coordination logic. Business logic is integrated with so called "services" at specific locations in the model.

Further, the *Peer Model* is characterized by its dynamism and scalability. The design can be scaled up without leading to blown-up models. *Peers* abstract over the concrete number of

instances, whereas such instances can be added at runtime as well. Models can be designed using a graphical notation or specified by a domain-specific-language. In the following, the essential components of the *Peer Model* together with their graphical notations will be presented.

4.2 Components

4.2.1 Entry

Entries are the items used to transfer information between peers. Each entry consists of two parts: *application* and *coordination data*. The former is used to transport application-specific data throughout the system, which can be processed by services encapsulating business- or application-logic. As the name suggests, the latter is used for coordination purposes, and holds meta-information for system-internal mechanisms like entry selection and transactions. [31] The application data can be seen as a black box for the system. Coordination data is a set of properties represented by key-value pairs, e.g. the entry type is represented by the *type* property. A further distinction is made between user-defined and system coordination-data. As the name implies, user-defined properties are supplemented by the programmer to express certain coordination logic. System defined coordination properties are interpreted by built-in system functionality and lead to specific behaviour.

4.2.2 Flow

Flows are used for correlation purposes. A flow comprises a set of entries which belong together. The system defined coordination property *Flow ID* is used to indicate that an entry belongs to a specific flow. This approach can be compared to typical internet packet or web-server mechanisms, where such ID information is used to specify which packets belong together or which response message corresponds to which request. Moreover, it is highly related to the correlation functionality of WS-BPEL.

4.2.3 Container

Containers are those elements of a model which hold entries. Basically, a container in the *Peer Model* is realized using a *space-container* from the asynchronous and blackboard-based communication model XVSM [13, 34, 35]. XVSM generalizes the original Linda [17] tuple space communication and extends it with various concepts. Containers in the *Peer Model* support configurable coordination mechanisms [33]. Common coordination models are for example *FIFO* (first in first out), *LIFO* (last in first out) or *random*, which returns stored entries in arbitrary order. A container can be accessed using the operations read, write and take. Read selects one or more entries based on a query, take is a consuming read that also removes the queried entries from the container and write is used to put entries into the container. Containers are addressable; they are referenced by a URI in the network. The graphical representation of a container is a grey shaded box.

4.2.4 Peer

According to [31], a *peer* is a structured, re-usable and addressable component encapsulating application (business) and/or coordination logic. A peer provides two stages for input and output, the so called *Peer-In-Container* (PIC) and *Peer-Out-Container* (POC). PIC and POC correspond to containers as defined previously. Between the containers for input and output the peer's internal logic is modelled. This is achieved using subsidiary peers and wirings (see 4.2.5). Roughly speaking, the internal logic describes how the information in form of entries flows from the input to the output-container. As peers can use other peers to express encapsulated behaviour, this is a clear indication for re-use and modularity. Typically, peers which are used to model the logic of another peer are called *sub-peers*. Application logic can be integrated with services. A peer is physically deployed on one single site of the network. A peer can be deployed multiple times, meaning that there can be multiple instances of the same peer on several nodes in the network. Such a deployed peer is called *peer-instance*.

Figure 4.1 represents the graphical notation of a peer. As can be seen, the containers for input and output are located on the left and the right side, respectively. The name of the peer is entered into the label on top. The internal logic is modelled within the white area labeled with "behaviour".

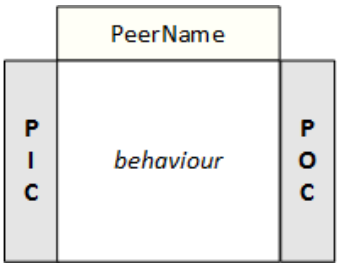

Figure 4.1: Peer

Space peers are a special form of peers where the PIC and POC are merged. They are used to store data shared by multiple threads/processes and act as a medium for communication and synchronization [31]. The graphical notation of a space peer is given in figure 4.2. The grey shaded area represents the single container used to store the entries. As a space peer does only maintain entries, it does not contain internal coordination logic.

During deployment of a peer, a unique address is assigned (e.g. an URI). PIC and POC are addressable using the peer's address supplemented with the relevant container. Internal components of a peer such as sub-peers are addressed by composing the peer's URI with the sub-peer's name. Therefore, a peer forms a kind of namespace for its components.

4.2.5 Wirings

Wirings are the transport mechanism of the *Peer Model*. They are used to incorporate services (see section 4.2.6) into a peer and transport entries between components. The scope of a wiring is bounded by its surrounding peer and it can create connections between the peer's containers

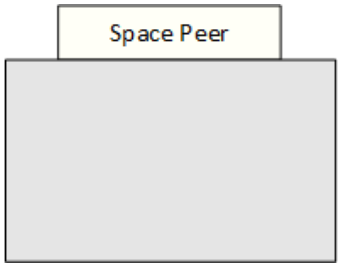

Figure 4.2: Space Peer

and all containers of the peer's sub-peers. Each wiring has a name and consists of the following three sections:

1. Guard:
   This section can be seen as the pre-conditions that must be fulfilled in order for the wiring to be activated. It consists of one or more ordered *input links* $G_1, \ldots, G_k$ with $k >= 1$. An input link is a connection starting from one of the containers in the scope of the wiring leading to the wiring's control box. An input link is labeled with a query and an operation which are both executed on the source container. The query denotes which entries (zero or more) should be selected and the operation specifies whether the selected entries should be just read, meaning that they will remain in the container, or consumed (taken) which has the consequence that they are removed from the container. Queries are specified using the semantics defined in [12]. The query results are passed to the so called *entry collection* (EC), which is a temporary container of the wiring. A wiring is activated if and only if all input links are satisfied, a side-condition is that the first input link must be a take-operation. This side-condition is ignored if there is a "none-test" input link which checks that a specific entry is not available and the wiring has an output link which writes such an entry to the input link's source container. Both conditions prevent the wiring from entering a potentially infinite execution loop.
   The usages of flow IDs (FID) influence the guard sections. Input links must query entries with compatible FIDs, which mean that they must have the same FID, whereas entries without specified FID can be seen as neutral elements compatible with every flow.

2. Service:
   The service-section is optional and allows calling services. Similar to the input links in the guard-section, the services are also ordered and therefore executed sequentially in the given order. They have access to the wiring's entry collection. More about services can be found in section 4.2.6.

3. Action:
   The action section is the final section and is optional as well. It is used to distribute the results of the wiring. An action section consists of ordered *output links* $A_1, \ldots, A_k$ with $k >= 1$. An output link connects the wiring's control box and one of the containers in the scope of the wiring. Analogously to input links, output links are also labeled with

a query and an operation. They are executed on the wiring's *entry collection* and the resulting entries are passed to the output link's targeted container. In contrast to input links, output links don't have to be fulfilled, but they are processed in the specified order. If one link cannot be fulfilled the next one is executed. After executing the last output link, the remaining entries in the entry collection are deleted.

The three mentioned sections are executed sequentially. Thus, the processing of output links starts only after the last service has completed its work. As services are integral parts of wirings, the graphical notation for wirings that has been defined in [29] is summarized after introducing the concept of services in the following section. For the figures the LaTeX macros developed by [18] are used.

4.2.6 Services

Services are used to add business- or coordination-logic which is not modelled within the *Peer Model*. Basically, the majority of services are user-defined services. A service can take multiple entries as input and emit multiple entries as its output. For both, the entry collection serves as the basis. Similar to the input and output links, the entries can be queried with a non-consuming read or a consuming take. Take operations are graphically represented by a filled arrowhead, whereas read operations have an unfilled head.

A typical application-specific service would be the processing of a certain task entry. The result is then written back to the entry collection from where output links can distribute the resulting entry to specific locations.

4.2.7 Sample Peer

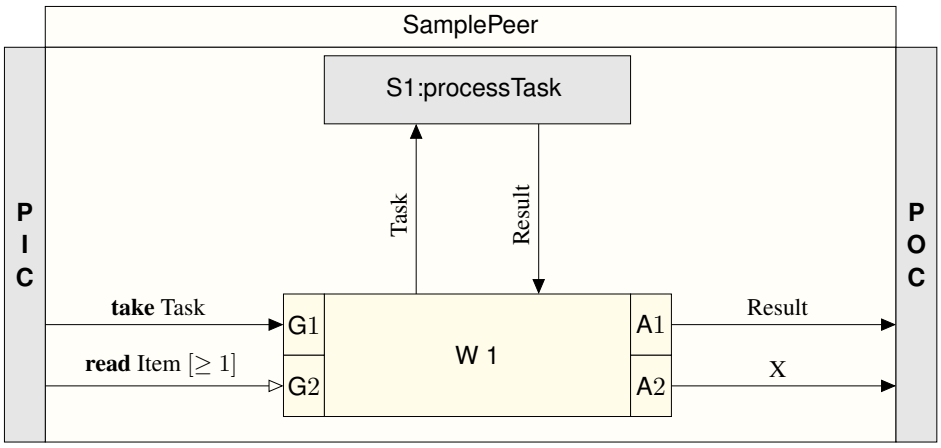

Figure 4.3: Sample Peer with Wiring and Service

Figure 4.3 shows an example for a peer containing a wiring with an attached service. The wiring $W1$ has two guards (input links), namely $G1$ and $G2$. In addition it contains the service

$S1$ and two actions (output links) $A1$ and $A2$. The input link $G1$ takes exactly one *Task*-entry, $G2$ reads one or more *Item*-entries from the PIC of *SamplePeer*. The expression "≥ 1" in the square brackets is used to filter the result. It is possible to apply multiple such filters, where the result of the left part is passed to the right part. This can be compared to the XVSM query mechanism proposed in [13]. For the concrete example shown in the figure, first all available *Item*-entries are selected and as a second condition it is checked whether there are more than zero. The query for $G1$ shows an abbreviation: "take Task", which is a shorthand for "take Task[$= 1$]". The wiring $W1$ is activated if and only if both input links can be satisfied meaning the take operation for the *Task*-entry and the read operation for one or more *Item*-entries succeed.

Since there is the service $S1$ available, a call to the user-defined function *processTask* is performed. The function takes one *Task*-entry as input and creates one entry of type *Result*. As already mentioned, services can read or take multiple input entries from the entry collection and emit multiple entries back into the entry collection.

Finally, the output links of the action-section are performed in the corresponding order. If there is an *Result*-entry available in the entry collection (output link $A1$), then the query is fulfilled and exactly one *Result* entry is transferred to the POC of *SamplePeer*. An entry of type *X* would be also written to the POC according to output link $A2$.

4.3 Advanced Concepts

As already explained in the paragraph about entries, the coordination data is divided into user-defined and system-defined coordination data. The latter makes use of built-in system behaviour as the *Peer Model* automatically interprets such properties. In the following, some of these frequently used system properties that were introduced in [31,32] and their effects are summarized:

- TTS (Time-to-Start): The TTS specifies the time when entries are ready to be selected by the querying mechanism. Before the specified point in time is reached, the entries are invisible for the wirings. Thereby, the TTS can be used to delay the processing of such an entry. The TTS can be specified for flows as well, which has the effect that the TTS of each of its entries is influenced.

- TTL (Time-to-Live): The TTL indicates the amount of time that can elapse before the entry is considered as invalid and as a consequence is wrapped into an exception entry. This can be used to prevent entries from pending indefinitely long in containers. On the flow level, every entry of the flow is wrapped into an exception entry when its TTL expires.

- DEST (Destination): DEST is a special system property which is based on implicit transport functionality. If an entry has a defined DEST value and it resides in a POC, then the system automatically transfers the item to the specified destination. It is possible to specify more than one address — in that case, the entry is distributed to multiple receivers. Other wirings intending to read or take such entries from the POC have no effect in a situation where DEST is set. It makes no difference whether an address in the property corresponds to a local peer or a remote peer, thus this transportation mechanism is location transparent.

Exceptions are special types of entries used to handle failures. Timeout exceptions occurring when the TTL of an entry expires are an example for such exceptions. An exception entry is wrapped around the entry on which the exception has occurred during processing. Exception entries can be queried as typical entries including the retrieval of the type of the exception as well. This allows incorporating fine grained exception handling. The *Peer Model* supports also the configuration of default rules, for example that exception entries are automatically removed from containers or moved to specific peers.

CHAPTER 5

Pattern Concept

The following sections discuss the concept of patterns in the context of the *Peer Model*. It covers the questions what a pattern actually is, how patterns can be composed and what their deployment looks like. In addition, it will be outlined which types of patterns exist, what differences they have and how they relate to the existing *Peer Model* components.

5.1 What is a Pattern?

In the *Peer Model*, a pattern is a loose collection of components. It isn't specified if single components are local or remote — they can be seen as virtual and location-free. Their real location is defined during deployment. It should be obvious that this set of components is not an unrelated bunch of elements, but that they are tightly related from a semantic point of view. Components in the context of patterns are:

- Wirings:
  Wirings are used to connect particular components.

- Sub-patterns:
  Existing patterns can be used to create compositions of patterns. A previously created pattern used to define a new pattern is called a sub-pattern.

- Services:
  A pattern can also contain services which will be attached to existing wirings on deployment. For example, consider a simple logging service echoing every time the wiring gets executed. Such a fine grained extension of functionality is related to the concept of aspect oriented programming.

- Guards/Actions:
  Beside services, it is also possible to add guards and/or actions to existing wirings.

5.2 Pattern Parametrization

The pattern concept distinguishes between design time and configuration time. At design time the user specifies the basic structure and functionality of a pattern. The more flexible the design, the more possibilities are there to adjust the pattern to various situations. This adjustments happen when the pattern is used, which is why this time is called configuration time. At design time the user can rely on certain parameters which will be specified at configuration time. These parameters introduce variability to the *Peer Model* and thus an increased reuse of patterns is achieved. In the scope of this concept, such parameters are called properties. Properties can be defined for the following purposes:

- Location/Address Information:
  As mentioned before, patterns are location-free, virtual. On usage, the location has to be specified. Typical location properties are a guard's or an action's source and destination container. Moreover, addresses used for the DEST property of entries represent location information.

- Operation Information:
  Such properties allow one to specify whether an input link should use a consuming read (also known as take) or a normal non-consuming read operation. Depending on the concrete operation, such an option can have deep influence on the peer's behaviour.

- Entry Information:
  This allows specifying the entry types which will be processed by a pattern's components. Such options increase the reusability of patterns as they can be applied in various scenarios without the need to create a model with similar functionality where the only difference lies in the processed entry type.

- Query Information:
  Beside the entry type, the exact query of certain operations can be specified too. Compared to the specification of the entry type, the declaration of additional query information provides the possibility for fine-grained configurations.

- Service Information:
  Such parameters allow specifying the concrete service methods including the order in which they are called if the corresponding wiring is activated. This gives the user the opportunity to design generic patterns where the exact service implementations and as a result the wiring's behaviour is defined at configuration time.

- Guards and actions:
  Beside the input and output links which are created at design time, users have the ability to add guards and actions also at configuration time. Since the order of input and especially output links has significant influence on the behaviour of wirings and the system, it can be specified as well. To prevent potential conflicts, a priority has to be specified along with the position of the input or output link. Higher priority values have precedence during the positioning process. In combination with configurable service methods, the adding

of input and output links gives the user the chance for a far-reaching modification of the pattern's behaviour at configuration time. However, it is important that a pattern is only "modified" to a certain degree, such that the pattern's original purpose and semantics is retained.

In many cases, patterns are used with almost identical configurations. Therefore, it makes sense to allow the definition of default values which will be used if a particular property value is not set at configuration time. Another pleasant side-effect is that this leads to a reduced configuration effort for the user. As will be seen in the sections about composition and deployment, the concept of properties serves also as the instrument for connecting the pattern's components with the elements of the peer where the pattern will be used.

5.3 Pattern Types

Until now, the potential components of patterns and the pattern's configuration possibilities have been discussed. In the following, the focus is moved on to the structure of patterns. There exist two types of patterns: *basic patterns* and *peer patterns*. A basic pattern corresponds to the loose collection of components mentioned before. A peer pattern is a special variant of a basic pattern, which is accompanied by input and output containers. Figure 5.1 illustrates this difference.

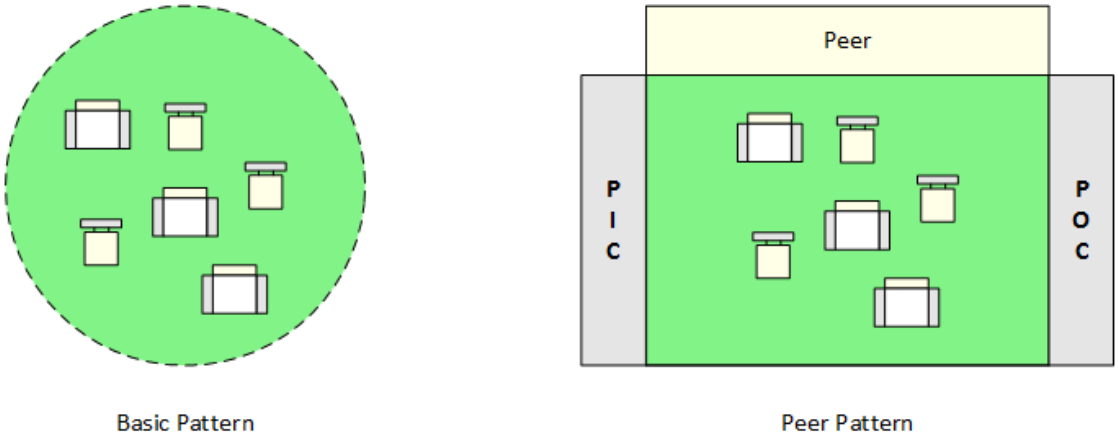

Figure 5.1: Pattern Types

The dashed green circle on the left should illustrate this bunch of loose components forming a *basic pattern*. It consists of three sub-peers and three wirings. As opposed to figure 5.1, the pattern's components may also be connected with each other. Often all components or at least some of them are interconnected, although the concept itself does not require it. When deploying the pattern, the pattern's properties are used to connect its parts to the components of a specific peer. The right part of the figure shows a *peer pattern* as the pattern's components are surrounded by a peer. Thereby, a self-contained component is created.

For better understanding, figure 5.2 shows an example for a basic pattern, the *Replicator* pattern. It consists of a single wiring with two input links, one output link, and an attached service. The pattern's purpose is to replicate a given entry *item* a specified number of times. The concrete number is provided by the *count* entry. In order to be used in various situations, the pattern allows the following configuration options:

- Entry type of *item*: The *item* entry used in the pattern is a placeholder and can be replaced by a specific entry type. This placeholder is shown in red in the figure.
- *Guard 1* source container: The source container of the first guard can be specified. By default, the source container is the PIC of the peer to which the pattern is deployed.
- *Guard 2* source container: The source container of the second guard can be specified in the same way. Again, the default source is the PIC of the peer to which the pattern is deployed.
- *Action 1* destination container: Similar to the source containers of the guards, the destination container of the action can be specified. By default, the destination is set to the POC of the peer to which the pattern is deployed.

The first option is an example for the specification of entry information, whereas the last three represent location information.

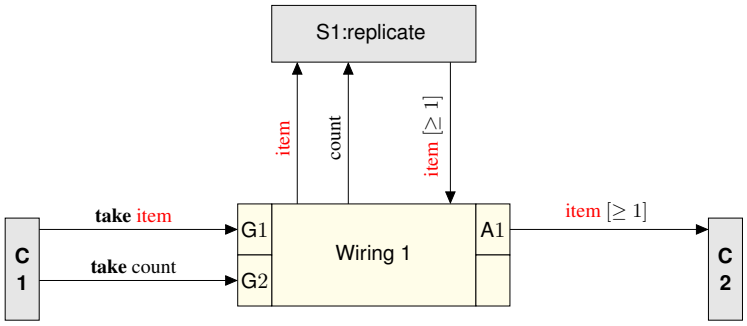

Figure 5.2: Basic Pattern Example - Replicator

Figure 5.3 shows an example for a *peer pattern*, the *ExtendedReplicator*. It contains a sub-peer and three wirings. The two wirings labeled with "***move** item*" and "***move** condition*" are used to shift entries of types *item* and *condition* to the sub-peer *FilterPeer*. Such a "move" is a graphical shortcut for a wiring which takes one entry of a specified type from a source container and moves it to a concrete destination. The *FilterPeer* sub-peer contains a configurable logic which filters entries based on the given *condition* entry. Appropriate *item* entries are then moved to the sub-peer's POC and replicated by the attached wiring "*Wiring 1*". The *ExtendedReplicator* pattern provides the following properties:

- Entry type of *item*; shown in red in figure 5.3

- Service method *filterService* used by the *FilterPeer* sub-peer. The service method is not visible in figure 5.3, it is an internal method of the *FilterPeer* sub-peer.

The *item* entry type is used as a placeholder. When using this pattern, the concrete entry type has to be specified. Moreover, a service method containing the filter logic has to be specified as well. This service method is called within the sub-peer. As can be seen in the figure, the pattern is surrounded by the PIC and POC of a peer. Therefore, such *peer patterns* are self-contained components.

When viewing the *ExtendedReplicator* pattern, it is not hard to detect the previously described *Replicator* pattern. As explained earlier in this chapter, patterns can be created by composing wirings, services, guards, actions, and sub-patterns. The *ExtendedReplicator* pattern uses the *Replicator* pattern as its sub-pattern. Pattern composition is discussed in the next chapter in more detail.

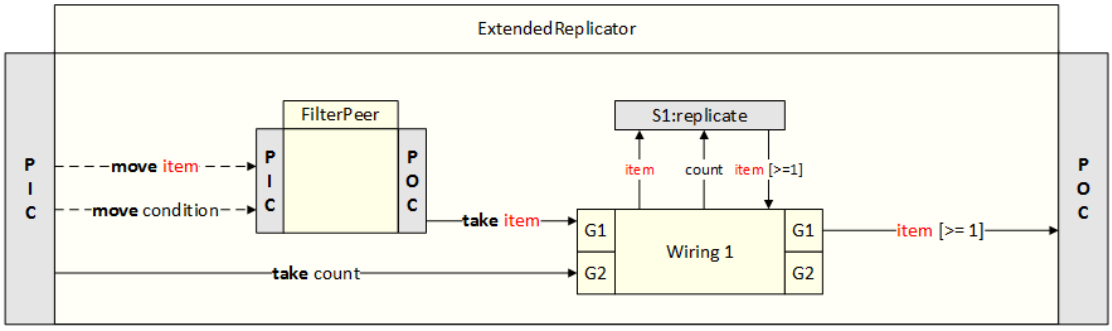

Figure 5.3: Peer Pattern Example - Extended Replicator

5.4 Pattern Composition

As said, a pattern is composed of multiple components. The most interesting of these components are other patterns, called sub-patterns, because they allow nested and parametrized functionality. Each pattern can contain further patterns and due to the passing of properties, these can be highly flexible. Basically, the adding of another pattern as a sub-pattern enhances the functionality of the pattern. The two pattern types have to be handled differently with regard to composition:

- Basic Pattern:
  A basic pattern is characterized by its loose components. In order to use them for the definition of another pattern, these components need to be connected to the components of the considered pattern. The properties of the sub-pattern therefore have to be specified before one can use it. Concerning the shown *Replicator* pattern, this means that the user needs to at least specify the type of the entry which should be replicated. For the remaining properties the default values are applied unless otherwise specified. For the specification of such properties, two possibilities exist. Either the value of a property of the employed

sub-pattern can be directly specified in the context of the pattern at design time or the value of the property has to be passed through by relying on a property of the considered pattern which is then specified at configuration time. The *ExtendedReplicator* pattern presented previously requires the entry type of *item* to be defined. This is an example for a property which is passed to a sub-pattern, as the *Replicator* pattern requires this option to be specified as well. In contrast, the source and destination containers of the *Replicator* pattern are directly specified within the *ExtendedReplicator* pattern.

A basic pattern is not a self-contained component and therefore, single parts of the pattern may be scattered across the components of the considered pattern. In other words, not all components of a pattern are necessarily connected.

- Peer Pattern:
  If a pattern contains a peer pattern, it can treat it as if it were a regular (sub-)peer. The pattern can define wirings starting at and leading to the containers of the peer pattern. In order to use a peer pattern as such a self-contained component a.k.a. a sub-peer, the properties of the peer pattern have to be specified in the same manner as required for basic patterns.

5.4.1 Naming

As the combination of patterns results in the integration of several components like wirings or peers into a new pattern, name-conflicts between these components may occur. To prevent such issues, names have to be unique. This can be achieved by combining the name of a sub-pattern component with the name of the sub-pattern. As an example, recall the special "move"-wirings with their graphical shortcut. These entry-shifting wirings can be seen as a simple basic pattern as well, the *Move* pattern. It allows specifying the entry to be moved, the source container and the target container. A graphical representation of the pattern is provided in figure 5.4.

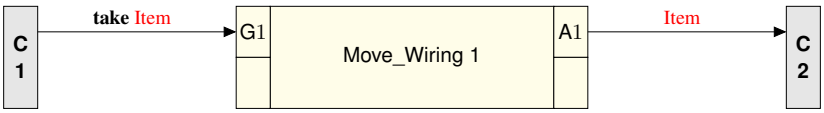

Figure 5.4: Move Pattern

The pattern contains a *Wiring 1* which is a general and therefore often used name. To prevent inconsistencies regarding the names of components, the name of the sub-pattern component is combined with the name of the sub-pattern, e.g. "Move_Wiring 1". As seen in figure 5.3, the *Move* pattern has been used two times in the *ExtendedReplicator* pattern. If there are multiple instances of the same sub-pattern, then a unique counting number is required as well, for example "Move_1_Wiring 1" for the first occurrence of the pattern and "Move_2_Wiring 1" for the second. By following such a strategy, potential naming issues can be circumvented.

Beside potential naming issues, such a convention will be relevant for the deployment of patterns as well. As explained, patterns are location-free, therefore the name of a pattern which

consists of a bunch of components is a key element which allows distinguishing to which pattern a component belongs. In case of a distributed deployment, the name provides the basis for implementing a lookup-alike mechanism to discover the distributed parts of the pattern.

5.5 Pattern Deployment

So far, the two types of patterns — *basic* and *peer pattern* — were introduced. Further, it was explained with which components a pattern can be formed. It was not yet discussed, however, how a pattern can be deployed. Basically, the deployment of a pattern such that peer instances can be created is similar to the composition of patterns. In order to create a runtime instance of a peer which embeds one or more patterns, the required properties need to be specified in the same fashion as it was done for the composition of patterns. *Peer patterns* are special in that sense as they form a self-contained component and can be deployed directly as long as all configurations are provided. As *basic patterns* are just a loose collection of components, they cannot be deployed directly. They have to be part of a normal peer or a *peer pattern*.

As patterns are location-free, it is possible to deploy parts of a pattern to different nodes. This "part of a pattern" can be seen as a sub-pattern, either basic or peer pattern. Suppose the *Split-Join* pattern where there is a pattern for each part — split and join. Then, the *Split* sub-pattern can be part of a peer pattern which is deployed to node X and the *Join* sub-pattern is part of another peer pattern deployed to node Y. As mentioned previously, the unique name of the pattern-instance allows identifying the parts and thus, the scope of the pattern. The concrete handling of distributed patterns, including the discovering of pattern parts and their effects on the flow of entries is out of the scope of this thesis and will be subject of future work.

5.6 Patterns, Peers, Peer Instances and their Relationship

As mentioned in the previous sections, a peer instance is a runtime instance of a specific peer. A basic pattern is a collection of several components and a peer pattern wraps such a collection with an input and output container. In order to create a runtime instance of a peer which embeds two or more patterns, all required properties need to be specified. Peer patterns are special in the sense that they are self-contained peers and therefore, runtime instances can be created directly subject to the condition that all configuration values are provided.

Keeping this information in mind and viewing the proposed concept of patterns and their mechanism to include sub-patterns, it can be concluded that every single peer can be modelled as a peer pattern. Every arbitrary peer can be designed as a peer pattern with the special characteristic that it does not require configuration information for the creation of runtime instances. Thus, a peer is just a special form of a peer pattern which has no properties. Recall the philosophy of the *Actor Model* covered in chapter 2.6 which states that "everything is an actor". Following the proposed pattern concept, this philosophy can be adopted as "everything is a pattern" in the context the *Peer Model*.

Figure 5.5 summarizes these findings graphically. The pattern ellipse depicts both types of patterns, basic and peer patterns, and how they are composed. Patterns are formed by wirings, guards, actions, and services, which is illustrated by the blue box on top. To create more complex

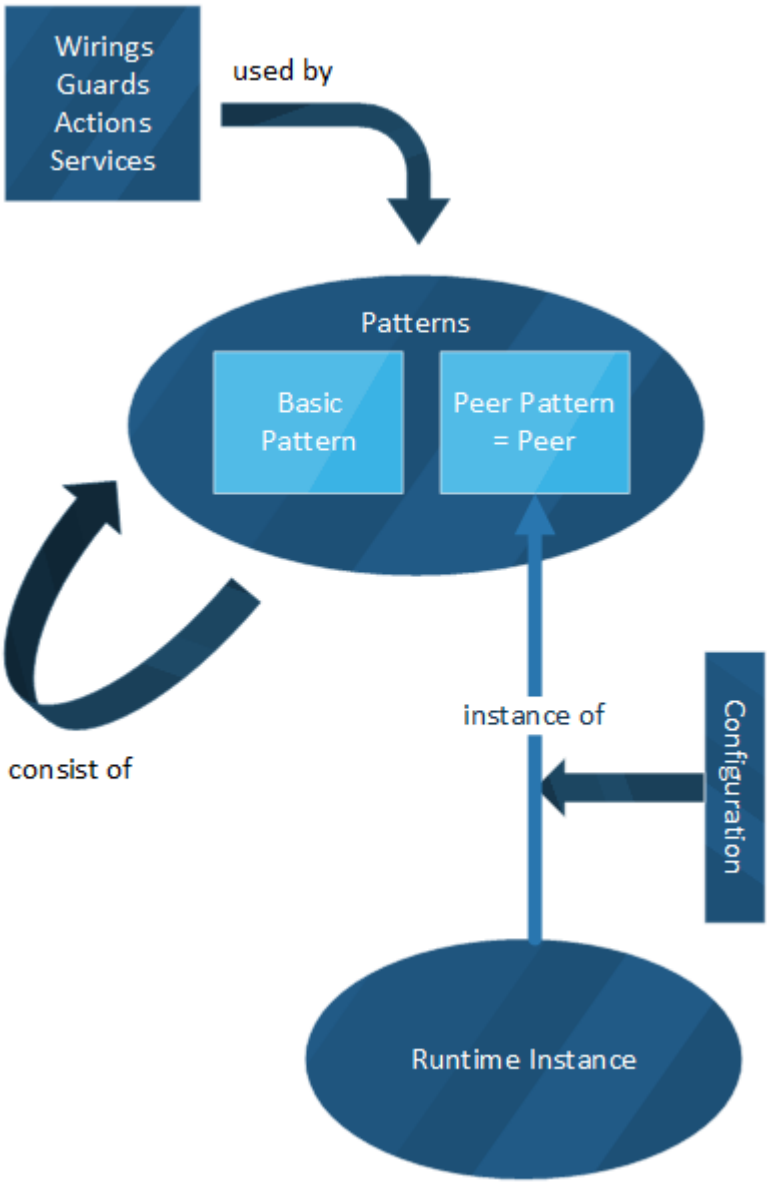

Figure 5.5: Pattern Concept - Overview

patterns, a pattern can use other patterns as so called sub-patterns. This allows a deep nesting of patterns as expressed by the arrow on the left. Runtime instances of peers or peer patterns can be created by specifying the required configuration information. This is illustrated by the arrow between the two ellipses and the configuration rectangle.

CHAPTER 6

Use Case Implementation

This chapter describes the implementation of the example use case following the previously presented pattern concept for the *Peer Model*. It starts with the realization of variant A and introduces various patterns, which will be also seen and reused in the implementation of variant B provided later in this chapter. The protocols and figures used in this chapter for variant A and B were defined in [30] and are used and adapted here to demonstrate the flexibility of the new pattern concept.

6.1 Variant A

This variant makes use of the patterns *Treat Event*, *Send and Retry*, *Send Acknowledgement* and *Process Event*. The first three were presented in [30] and partially covered in [32]. These patterns will be modified in such a way that they provide the core functionality for the components (*Sensor*, *Network Node* and *Level Crossing*) of this variant and build also the basis for the extended version of variant B.

With respect to the particular components, each pattern has to be deployed with a slightly different configuration. As can be seen later, the components' desired behaviour is achieved by plugging in variable service functionality. In the following, the modified versions of the patterns used throughout this use case will be introduced.

6.1.1 Treat Event Pattern

The *Treat Event* pattern is responsible for handling incoming events. Thus, it represents the starting point of the event processing. The pattern contains one wiring with two input links and five output links and is depicted in figure 6.1. The wiring is activated if an *event* entry is available and no entry of type *treated* with a corresponding flow ID. Such *treated* entries are used to keep track of already processed events and ensure that the wiring is executed only once for a single event. If the input links are fulfilled, the configured service methods are called. Depending on the concrete configuration, the event is "treated" by multiple services and entries are emitted

to the wiring's entry-collection. Afterwards, the output links of the wiring transfer the entries satisfying their queries to the corresponding target containers.

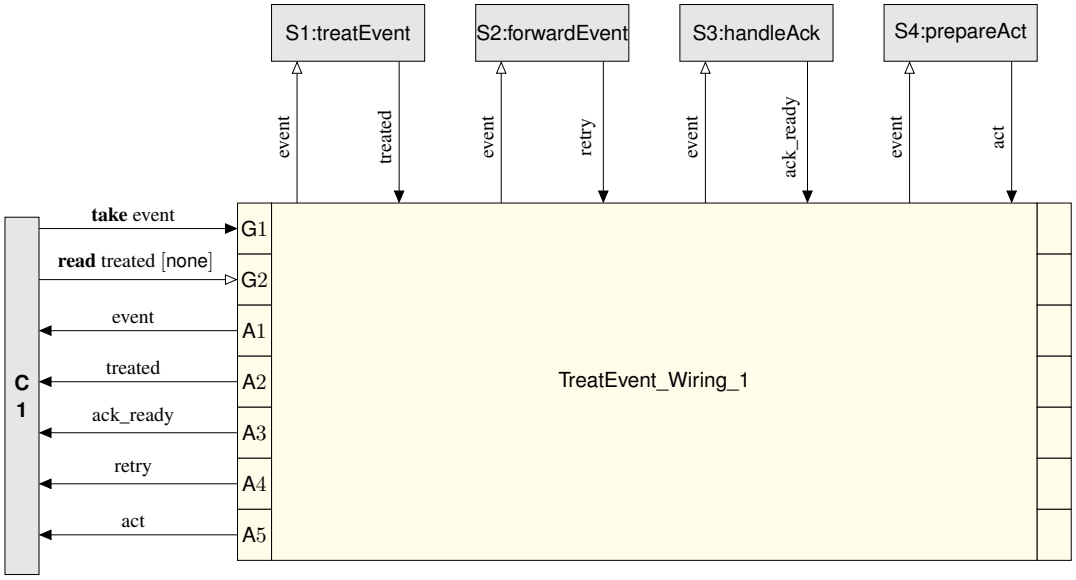

Figure 6.1: Treat Event Pattern

As already explained in chapter 5, these patterns allow fine grained parametrization. The pattern is shipped with four arbitrarily combinable service methods serving different purposes. Moreover, these service methods can be replaced by user-defined services and additional services can be plugged in. Beside the service functionality, the *Treat Event* pattern can be configured in such a way that the source container of each of the two input links can be specified as well as the target container of the five output links. To speed up the configuration, default values are used if specific parameters are not specified. In this case, the default source- and target-containers of the input links and output links are the PIC of the peer where the pattern is deployed to.

This pattern is used by every component of variant A, but each of them requires a different combination of the provided service methods to meet the demands. The implementations of the provided service methods are shown in listings 6.1–6.4.

The service *treatEvent* is used by all components. It reads the particular *event* entry and emits an entry of type *treated*. The entry *treated* is more or less a clone of the *event* entry containing the event information. Its purpose is to remember already processed events. For correlation the flow ID is copied as well.

The service *forwardEvent* is responsible for inducing the event forwarding process conducted by the *Send and Retry* pattern. For that purpose, a *retry* entry is created. Its flow ID is set to the event's flow ID and the TTS is set to the current time such that the entry is immediately available for other wirings when pushed back into a container. The service is used by *Sensor* and *Network Node* components.

The service *handleAck* is used to initiate the acknowledgement process undertaken by the *Send Acknowledgement* pattern. Therefore, an *ack_ready* entry is created with the flow ID set to the event's flow ID. The entry indicates that the component is ready to acknowledge the sender of the event about the successful transfer. As the sensor is the initial point in the network, there is no need to send an acknowledgement anywhere, therefore this service is only used by *Network Node* and *Level Crossing* components.

The last service is *prepareAct*, which "prepares" the event entry for the final processing. An *act* entry is created which encapsulates the event. The service is used at the *Level Crossing* component where such an *act* entry triggers the *Process Event* pattern which contains the actual business code to handle the event.

```
service treatEvent
   read event as e;
   emit treated;
begin
   treated t;
   copy_app_data(e, t);
   t.FLOW = e.FLOW;

   emit(t);
end service treatEvent;
```

Listing 6.1: Service Method *treatEvent*

```
service forwardEvent
   read event as e;
   emit retry;
begin
   retry r;
   r.FLOW = e.FLOW;
   r.TTS  = NOW();

   emit(r);
end service forwardEvent;
```

Listing 6.2: Service Method *forwardEvent*

```
service handleAck
   read event as e;
   emit ack_ready;
begin
   ack_ready ar;
   ar.FLOW = e.FLOW;

   emit(ar);
end service handleAck;
```

Listing 6.3: Service Method *handleAck*

```
service prepareAct
   read event as e;
   emit act;
begin
   act a;
   copy_app_data(e, a);
   a.FLOW = e.FLOW;
   emit(a);
end service prepareAct;
```

Listing 6.4: Service Method *prepareAct*

6.1.2 Send and Retry Pattern

The *Send and Retry* pattern is used to forward a treated event. The pattern embodies one wiring with three input links and two output links and is illustrated in figure 6.2. As previously described, the service methods used in this variant are designed in such a way that the different patterns interact with each other. The *Send and Retry* pattern reliably forwards an event to a specified peer in the network. Potential data transport failures are compensated by repetition. An event is periodically resent until an acknowledgement message from the receiver of the event arrives.

The pattern is designed in such a way that each source container of the wiring's input links as well as each target container of the output links can be configured. By default, C1 = PIC

and C2 = POC. Furthermore, the service can be specified as well as the two properties *UPSTREAM_NODE* and *RETRY_INTERVAL*. The former represents the address of the peer to which the event should be forwarded, the latter controls the time interval regulating how long the pattern waits until the event forwarding is repeated.

The service *sendRetry* given in listing 6.5 is used by both *Sensor* and *Network Node* peers. The pattern's wiring is activated if a *retry* entry as well as a *treated* entry are available and no entry of type *ack* with a flow ID which corresponds to the flow ID of the *retry* and *treated* entries has been received. Then, the specified service is executed which reconstructs an *event* entry out of the *treated* entry and sets its *DEST* property according to the receiver determined by the pattern's parameter *UPSTREAM_NODE*. In addition, a user defined property is used to attach the peer's address such that the receiver can reply with its acknowledgement message. Moreover, the TTS of the *retry* entry is set to a point in time in the future regulated by the pattern parameter *RETRY_INTERVAL*. This is used to trigger another execution of the wiring and therefore the resending of the event if no *ack* entry has been received in the meanwhile. Similar to all patterns presented here, this default service is replaceable, for example by an extended version which counts and stores the total number of retries in a property of the *retry* entry and discards the event if a certain threshold is reached.

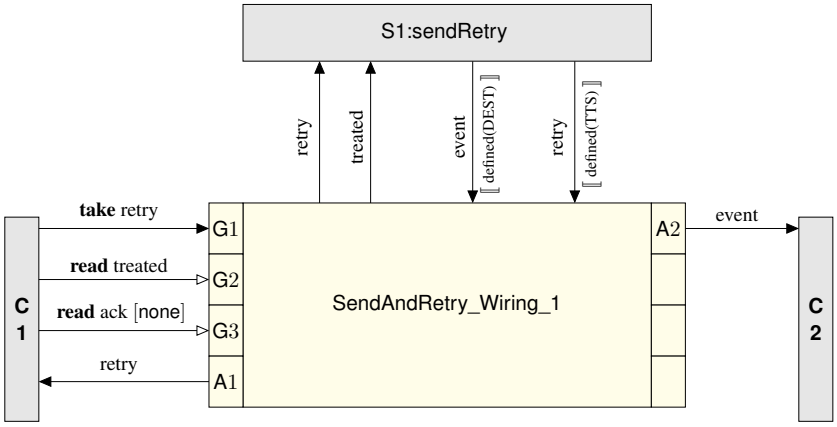

Figure 6.2: Send and Retry Pattern

```
service sendRetry
   take retry as r;
   take treated as t;
   emit retry;
   emit event;
begin
   event e;
   copy_app_data(t,e);

   e.FLOW = t.FLOW;
   e.DEST = UPSTREAM_NODE;
   e.set_user_property(ORIGIN, this_peer());
```

```
    r.TTS  = NOW() + RETRY_INTERVAL;
    emit(e, r);
end service sendRetry;
```

Listing 6.5: Service Method *sendRetry* used by Sensor and Network Node components

6.1.3 Send Acknowledgement Pattern

Figure 6.3 depicts the *Send Acknowledgement* pattern. The pattern is embodied by a wiring with two input links and one output link. It is used to transfer *ack* entries to the sender of the event and thus to acknowledge the receipt of the event. The wiring is activated if an *event* entry as well as an *ack_ready* entry are available. Then, the attached service is called. The result of the service is distributed by the output link.

Again, each source or target container of the input- and output links can be specified when deploying the pattern to a peer. By default, the source container of both input links is the peer's PIC. Concerning the output link, the source container is by default the peer's POC.

In variant A, the *Send Acknowledgement* pattern is used by the *Network Node* and *Level Crossing* peers. They use the same service implementation shown in listing 6.6. The service creates an *ack* entry. Furthermore, the *DEST* property is specified to make use of the *Peer Model's* transport functionality to transmit the entry to the sender of the event. The sender's address is retrieved from a property attached to the event. For correlation purposes the flow ID has to be set according to the flow ID of the corresponding event. This allows the recipient of the entry to determine to which event the acknowledgement belongs.

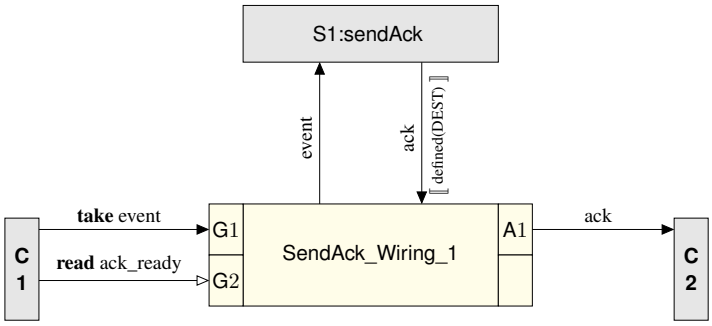

Figure 6.3: Send Acknowledgement Pattern

```
service sendAck
   take event as e;
   emit ack;
begin
   ack a;

   a.DEST = e.get_user_property(ORIGIN);
   a.FLOW = e.FLOW;
```

```
    emit(a);
end service sendAck;
```

Listing 6.6: Service Method *sendAck* used by Network Node and Level Crossing peers

6.1.4 Process Event Pattern

This pattern is the final one in variant A. Its purpose is to execute the user-defined business code to handle the event. The pattern is shown in figure 6.4 and consists of exactly one wiring with a single input link and a service.

The pattern allows specifying the source container of the input link at configuration time. By default, the source container is the peer's PIC where the pattern is deployed to. The pattern has one mandatory property, the user has to specify the business logic which is called by the pattern's service method given in listing 6.7.

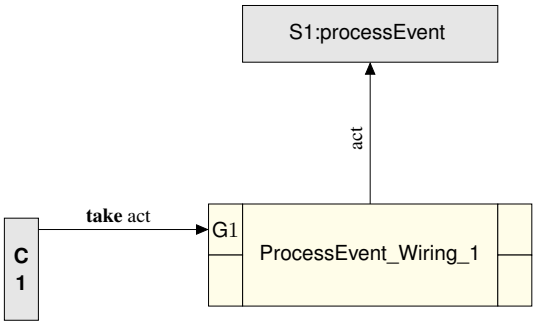

Figure 6.4: Process Event Pattern

```
service processEvent
    take act as a;
begin
    call(business_logic);
end service processEvent;
```

Listing 6.7: Service Method *processEvent* of the Process Event Pattern

6.1.5 Components and Pattern Usage

This section shows how the single components of the variant are modelled as peers, i.e. what patterns and what additional wirings or even sub-peers they require. Unless otherwise stated, the default values for source and target containers of input- and output links of the used patterns apply. The first component to be discussed is the *Sensor* presented in the following.

Sensor Peer

The *Sensor* peer embodies the following patterns:

- *Treat Event* with the service methods *treatEvent* and *forwardEvent* presented in listings 6.1 and 6.2.
- *Send and Retry* with the service method shown in listing 6.5.

In addition to the patterns' components, the wiring depicted in figure 6.5 is contained in the *Sensor* peer. The purpose of this wiring is to keep the load of the PIC small. Events, which have been forwarded and where the recipients have replied with an *ack* entry can be seen as completely processed from the peer's point of view. Therefore, the correlating *retry* and *ack* entries can be removed which is achieved by the two take operations.

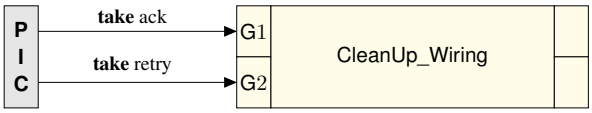

Figure 6.5: Variant A - Clean Up Wiring

Network Node Peer

Network Node peers make use of the following patterns:

- *Treat Event* with the service methods *treatEvent*, *forwardEvent* and *handleAck* presented in listings 6.1–6.3.
- *Send and Retry* with the service method given in listing 6.5.
- *Send Acknowledgement* with the service method of listing 6.6.

The *Send Acknowledgement* pattern is used to acknowledge the receipt of event to the configured downstream peer. The *Send and Retry* pattern's task is to forward the event to the configured upstream peer. Analogously to the *Sensor* peer, the *Network Node* peer uses an additional wiring to clean up no more needed event fragments. Therefore, the wiring presented in figure 6.5 is used as well.

Level Crossing Peer

The *Level Crossing* peer represents the final peer in the "event-stream". It embodies the following patterns:

- *Treat Event* with the service methods *treatEvent*, *handleAck* and *prepareAct* specified in listings 6.1, 6.3 and 6.4.
- *Send Acknowledgement* with the service method shown in listing 6.6.
- *Process Event*, where the input link's source container is the peer's PIC. The user has to specify the business logic which is called within the *processEvent* service method presented in listing 6.7.

6.2 Variant B

The *Level Crossing* peer used in variant B is the same as in variant A. The *Sensor* peer requires a slight modification in such a way that the *UPSTREAM_NODE* property which holds the address of the neighboring peer is now a list of addresses. This is because the event has to be forwarded to a group of peers. When the system-defined coordination property *DEST* of an entry contains more than one address then the entry is automatically delivered to every single specified address. The *Network Node* peer has to be extended with the necessary group logic. For this reason a couple of new patterns are introduced in the following.

6.2.1 Group-based Treat Event Pattern

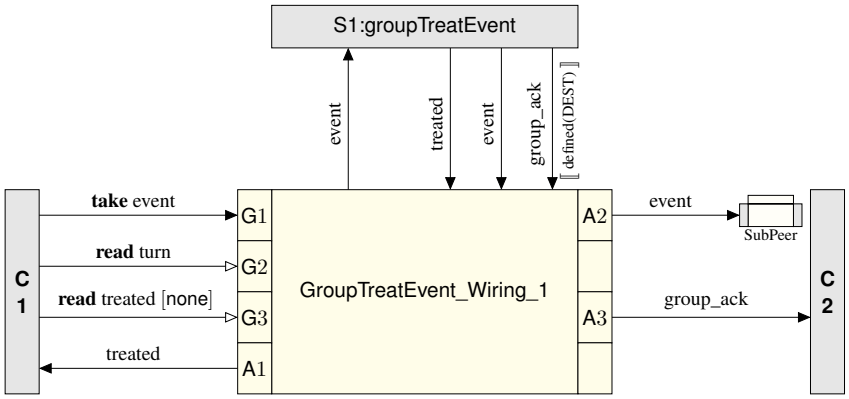

Figure 6.6: Group-based Treat Event Pattern

The pattern is depicted in figure 6.6. The purpose of it is to "treat" *event* entries if the particular peer is currently the leader of its group. Group leadership is indicated by the availability of a *turn* entry. The pattern's wiring is activated if and only if an *event* entry and such a *turn* entry are available and the event has not been treated before. Then, the service shown in listing 6.8 is executed. The service creates *group_ack* entries which are sent to the group members of the peer to signal that the leader processes this specific event. The group members' addresses are contained in the parameter *GROUP_MEMBERS*. Therefore, the entry is delivered to each specified group member. Moreover, a *treated* entry is created to prevent the event from being processed again. The particular event is forwarded to a special sub-peer which corresponds to the *Network Node* peer presented in variant A. This quite clearly shows the PM's capability for composition and reuse. The complete logic for acknowledging the sender and for forwarding the event is encapsulated in the sub-peer. The *Network Node* sub-peer is a peer pattern where the *UPSTREAM_NODE* property of the included *Send and Retry* pattern depends on a property of the *Group-based Treat Event* pattern. When deploying the pattern, the concrete addresses of the upstream group members have to be specified.

To make the distinction easier, the peer which contains both group logic and the *Network Node* sub-peer will be called *Group Member* peer in the following. The *Network Node* sub-peers forward events to the configured upstream *Group Member* peers or to the final *Level Crossing* peer if the end is reached. Acknowledgement messages are directly sent from sub-peer to sub-peer by relying on the attached *ORIGIN* property explained in variant A. To keep this pattern flexible, the source containers of all input links can be specified at configuration time. By default, they are set to the peer's PIC, thus C1 = PIC.

```
service groupTreatEvent
  take event as e;
  emit event;
  emit group_ack;
  emit treated;
begin
  group_ack ga;
  ga.FLOW = e.FLOW;
  ga.DEST = GROUP_MEMBERS;
  treated t;
  copy_app_data(e, t);
  t.FLOW = e.FLOW;
  emit(e, t, ga);
end service groupTreatEvent;
```

Listing 6.8: Service Method *groupTreatEvent*

6.2.2 Register Failover Pattern

The next series of patterns cover the various cases when a *Group Member* peer is not in the leadership position. The first pattern discussed is the *Register Failover* pattern illustrated in figure 6.7. It constitutes the starting point of two possible execution branches as will be seen later. The task of this pattern is to consume a newly arrived *event* entry and wrap it with a *pending* entry. The TTL of this entry is limited such that after its expiration a failover mechanism takes place, which is realized by the *Process Failover* pattern. The creation of the described *pending* entry happens in the *registerFailover* service given in listing 6.9. The concrete TTL is regulated by the *FAILOVER_TIME* property and is therefore separately configurable for every single peer which deploys this pattern. Again, source and target containers of input- and output-links can be specified, by default C1 = PIC.

```
service registerFailover
   take event as e;
   emit pending;
begin
   pending p;
   copy_app_data(e, p);
   p.TTL = FAILOVER_TIME;
   p.FLOW = e.FLOW;
   emit(p);
end service registerFailover;
```

Listing 6.9: Service Method *registerFailover*

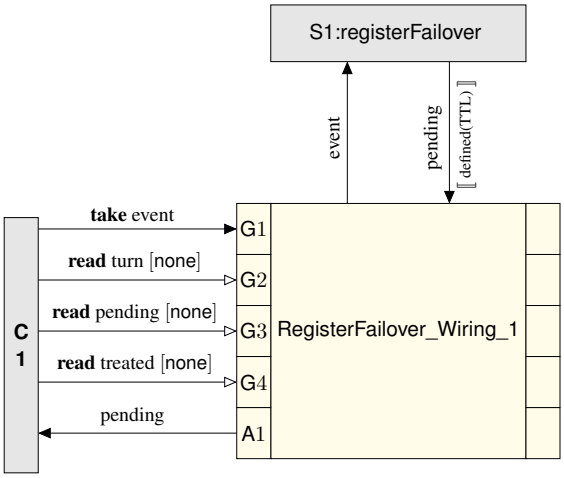

Figure 6.7: Register Failover Pattern

6.2.3 Process Failover Pattern

The *Process Failover* pattern is activated if the TTL of a *pending* entry has expired and as a consequence the entry has been wrapped with an *exception* entry. Then, the attached service *processFailover* shown in listing 6.10 is executed. It consumes the *exception* entry, unwraps it such that the *pending* entry can be put back into the container C1 with a newly specified TTL and generates a *turn* entry. Moreover, a *consensus* entry is created and sent to all group members signalling that the peer has taken over the group leadership. This pattern initiates the "negative" execution branch, in which the group leader has not processed the event.

The pattern's wiring is executed only once in order to create a *turn* entry and to take the leadership. The transformation of *pending* entries back to *event* entries such that the *Group-based Treat Event* pattern can process them in the following is the task of the *Process Pending* pattern. Moreover, there may be multiple *exception* entries which need to be consumed and processed, this is also the task of that pattern. Beside the already described properties *FAILOVER_TIME* and *GROUP_MEMBERS*, the service depends on the property *TURN_INTERVAL* which specifies the TTL of the *turn* entry. This property can be used to regulate how long a peer leads the group. Source and target containers of input- and output-links can be specified, by default C1 = PIC and C2 = POC.

```
service processFailover
   take EXC as x;
   emit turn;
   emit consensus;
   emit pending;
begin
   turn t;
   consensus c;
   pending p;
```

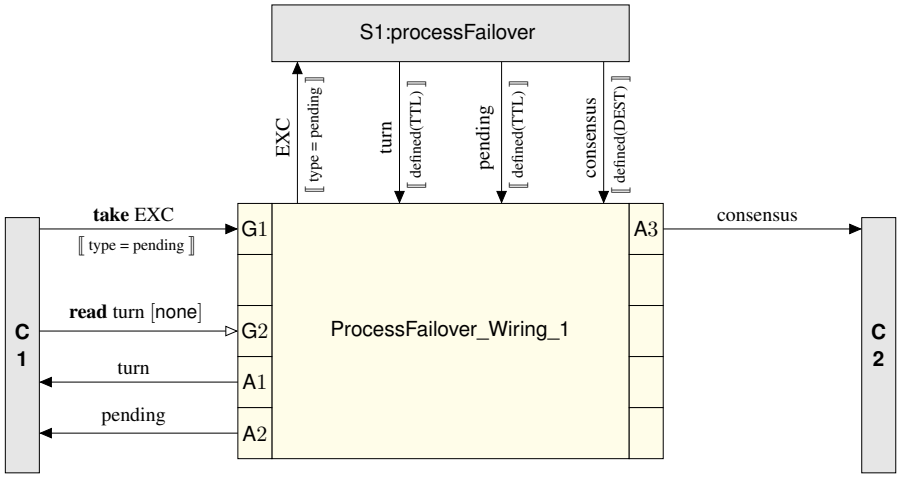

Figure 6.8: Process Failover Pattern

```
    t.TTL = TURN_INTERVAL;
    c.DEST = GROUP_MEMBERS;
    c.FLOW = x.FLOW;

    copy_app_data(x.pending, p);
    p.FLOW = x.FLOW;
    p.TTL = FAILOVER_TIME;

    emit(t, c, p);
end service processFailover;
```

Listing 6.10: Service Method *processFailover*

6.2.4 Process Pending Pattern

There may be multiple *pending* entries which need to be processed when a peer takes the group leadership. These pending entries are transformed back to *event* entries, such that the *Group-based Treat Event* pattern can consume them. This is achieved by the first wiring of the *Process Pending* pattern shown in figure 6.9. In addition, there may be multiple untreated *exception* entries as well. These have to be consumed and the underlying *events* have to be reconstructed. This is done by the second wiring of the pattern. The corresponding service methods of the wirings are given in listings 6.11 and 6.12.

The *Process Pending* pattern completes the failover process (“negative” execution branch) which has started with the registration of the failover mechanism in the *Register Failover* pattern and has been activated within the *Process Failover* pattern. The second possible execution branch during the processing of *event* entries is covered by the next pattern. As usual with the patterns presented here, the source and target containers of input- and output-links are configurable. By default, the PIC is applied.

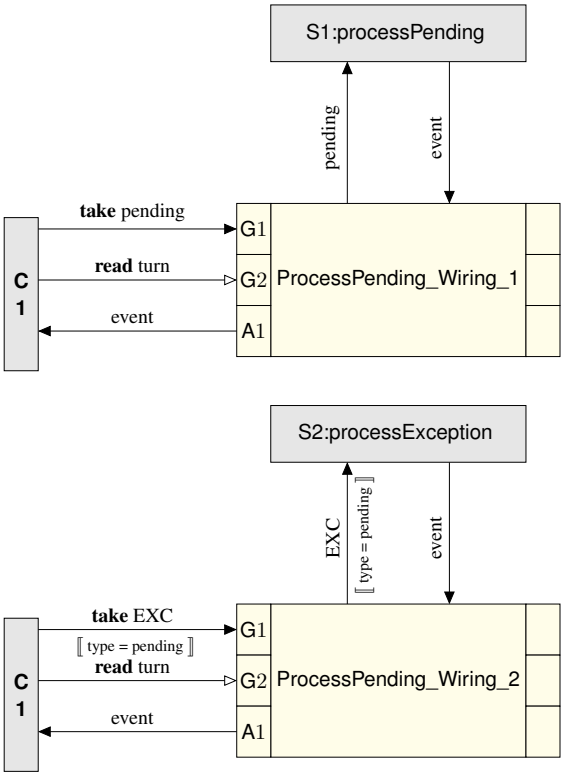

Figure 6.9: Process Pending Pattern

```
service processPending
   take pending as p;
   emit event;
begin
   event e;
   copy_app_data(p, e);
   e.FLOW = e.FLOW;
   emit(e);
end service processPending;
```

Listing 6.11: Service *processPending*

```
service processException
   take EXC as x;
   emit event;
begin
   event e;
   copy_app_data(x.pending, e);
   e.FLOW = x.FLOW;
   emit(e);
end service processException;
```

Listing 6.12: Service *processException*

6.2.5 Clear Pending Pattern

The *Clear Pending* pattern represents the "positive" branch when no failover is necessary. The pattern is activated in the following situation: the *Group Member* peer receives an *event* entry, but the peer is not the leader of its group. Therefore, the *Register Failover* pattern executes and creates a *pending* entry. Now the peer waits until the TTL set for this entry expires or a *group_ack* entry from the group leader arrives, indicating that the event has been successfully processed.

The former presents the previously covered “negative” branch, the latter the “positive” one. The peer receives a *group_ack* entry and can therefore remove the *pending* entry from the specified container. This is achieved by the wiring illustrated in figure 6.10 and the attached service given in listing 6.13. In order to keep track of the already processed events and to prevent unnecessary additional operation, a *treated* entry is created and distributed by the wiring’s output link. The property *ARCHIVE_TIMESPAN* can be used to specify the amount of time before the item is packed into an *exception* entry and can be removed to reduce the load of the particular container. The source containers of the two input links and the target container of the output link can be configured, by default C1 = PIC.

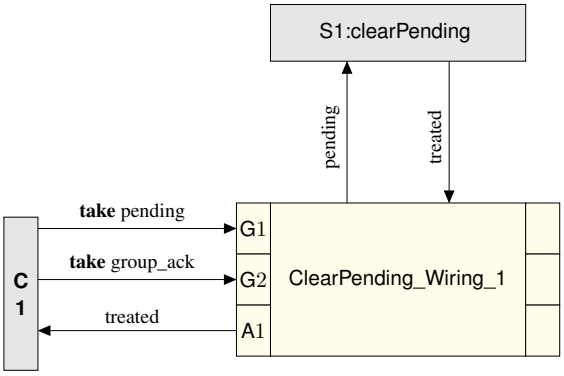

Figure 6.10: Clear Pending Pattern

```
service clearPending
   take pending as p;
   emit treated;
begin
   treated t;
   copy_app_data(p, t);
   t.TTL = ARCHIVE_TIMESPAN;
   t.FLOW = p.FLOW;
   emit(t);
end service clearPending;
```

Listing 6.13: Service Method *clearPending*

6.2.6 Clear Acknowledgement Pattern

This pattern is used to the keep the load of the *Group Member’s* PIC small. It may happen that multiple group_ack messages are sent, especially when two or more peers start with their failover mechanism at the same time. This pattern shown in figure 6.11 removes such *group_ack* entries for already treated events. By default C1 = PIC, however, both input links can be configured to retrieve the items from arbitrary containers.

Figure 6.11: Clear Acknowledgement Pattern

6.2.7 Release Turn Pattern

To prevent multiple *Group Member* peers of one group from being leader at the same time *consensus* entries are sent in the *Process Failover* pattern. To actually limit the group leadership to a single peer the *Release Turn* pattern is required. In case of an incoming *consensus* entry the peer removes its turn entries. There may be more than one even if this is a rare case when the wiring of the *Process Failover* pattern is executed for multiple different *exception* entries at the same time, then also multiple *turn* entries can be created. The pattern is illustrated in figure 6.12. The source containers of the input links can be specified, by default C1 = PIC.

Figure 6.12: Release Turn Pattern

6.2.8 Group Member Peer

As already described, the *Group Member* peer embodies all of the patterns introduced here for variant B:

- Group-based Treat Event which includes the *Network Node* peer presented in variant A as sub-peer
- Register Failover
- Process Failover
- Process Pending
- Clear Pending
- Clear Acknowledgement
- Release Turn

Each of them is used in its default configuration regarding source and target containers. The presented service methods are attached. During the deployment process, the user has to specify the concrete properties regulating failover times, resend intervals, group members, upstream neighbors and so on.

Variants A and B consisting of the presented patterns and their default services have been realized and tested with the C#-based implementation of the *Peer Model* called *PeerSpace.NET* presented in [50]. For the figures the LaTeX macros developed by [18] have been used.

CHAPTER 7

Flexibility

Business and especially the IT business is characterised by permanent changes. The authors of [40] define new requirements which cannot be adapted to the underlying system architecture as *architecture breakers* — these demand cost- and time-intensive evaluation and changes to the architecture. Further, *architecture limiters* are solutions which prevent simple and direct adaptations and require some workarounds. We can see that, an essential factor is that the underlying architecture is capable to adapt to changing requirements and business processes without introducing architecture breakers or limiters.

This chapter shows how the *Peer Model* extended with the pattern concept, with its focus on variability, proposed in this work facilitates the handling of changing business requirements without leading to such architecture breakers or limiters. For this reason, two potential additional requirements for the running use case are introduced. The necessary adaptations to the *Peer Model's* solution are then compared to the changes required in the Actor Model's realization of the use case. The Actor Model was chosen for this additional investigation because it obtained the best results in the previously conducted evaluation.

7.1 End-to-End Acknowledgement

Until now, the use case presented in 3.1 has been realized with immediate acknowledgement messages. Every component receiving an event has directly replied to the sender with a message confirming the reception. The new requirement demands for end-to-end acknowledgement messages. Therefore, a component waits with the confirmation of the receipt until the upstream node to which the event has to be forwarded sends an acknowledgement message. Thus, when the event arrives at the *Level Crossing* component, the acknowledgement messages are sent downstream in a cascading way as proposed in [30]. For the *Peer Model's* realization of this requirement the following adaptations are necessary:

- Extension of the *Send Acknowledgement* pattern used by the *Network Node* and *Level Crossing* peers.

- Different configuration of the *Treat Event* pattern used by *Network Node* peers.

The *Send Acknowledgement* pattern is "extended" with an additional output link for *ack_ready* entries. As already explained, the pattern concept allows adding additional services, guards and actions at configuration time. Beside the new output link, a different service method is used. The extended version of the pattern is illustrated in figure 7.1, the new service method which replaces the default one is given in listing 7.1.

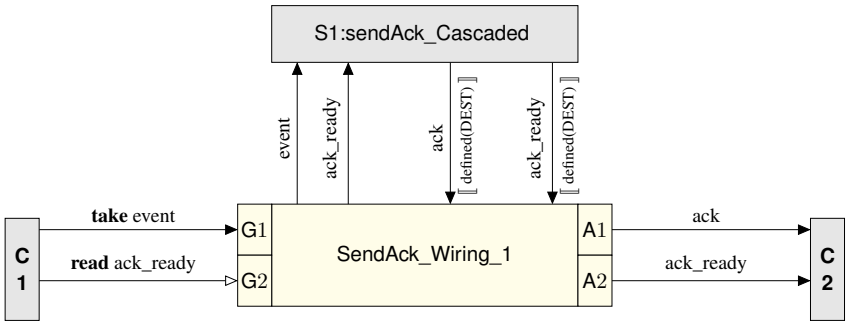

Figure 7.1: Extended Send Acknowledgement Pattern

```
service sendAck_Cascaded
   take event as e;
   take ack_ready as ar;
   emit ack;
   emit ack_ready;
begin
   ack a;

   a.DEST = e.get_user_property(ORIGIN);
   a.FLOW = e.FLOW
   ar.DEST = e.get_user_property(ORIGIN);

   emit(a, ar);
end service sendAck_Cascaded;
```

Listing 7.1: Service Method *sendAck_Cascaded*

The *Treat Event* pattern used by *Network Node* peers is configured such that compared to the original version the *handleAck* service method is removed. Figure 7.2 shows a visual representation of the reconfigured pattern.

The different configuration of the *Treat Event* pattern for *Network Node* peers has the effect that no *ack_ready* entries are created when processing events. Therefore, the wiring of the *Send Acknowledgement* pattern is not subsequently activated. The execution of this pattern's wiring is triggered downwards, starting at the *Level Crossing* peer. In addition to the *ack* entry, which is sent to the downstream neighbor by the *Send Acknowledgement* pattern as usual, the extended version of the pattern attaches an *ack_ready* entry. When such an entry is received by

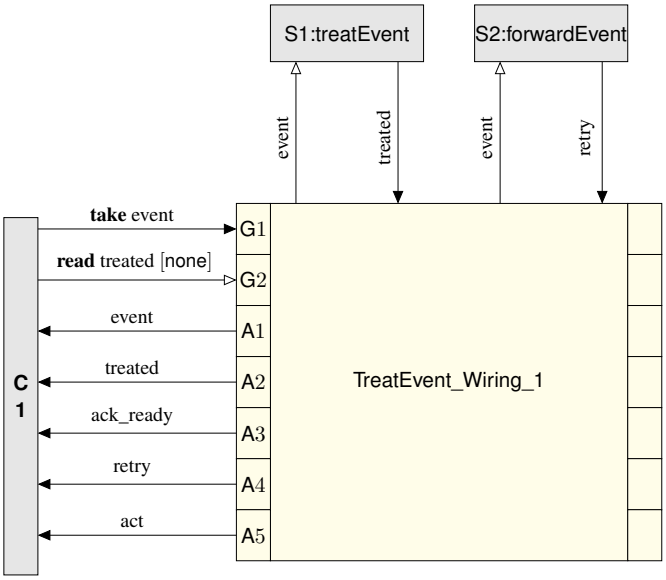

Figure 7.2: Configured Treat Event Pattern

a *Network Node* peer, the peer's extended *Send Acknowledgement* pattern is triggered and a pair of *ack* and *ack_ready* entries is sent downstream too. This process is repeated until the final acknowledgement message arrives at the *Sensor* peer.

As can be seen, the parametrizable and adjustable patterns allow for a great deal of modification of the peers' behaviour without changing the overall models. The concrete behaviour is specified at configuration time before the peers are deployed. The behaviour is not carved in stone; one could easily switch between the normal and the end-to-end acknowledgement mode, it is just a matter of configuration.

Basically, there are two ways to realize the end-to-end acknowledgement in the solution created using Akka (see secion 3.8.1), which is again the representative for the Actor Model. The first is to create a new *Network Node* and *Forwarder* actor pair capable of these end-to-end messages. The second variant is to incorporate the end-to-end acknowledgement into the existing code and make its usage depend on a concrete flag read from a configuration file. The former has the advantage that the code stays clean, readable and easier maintainable. The downside is that this strategy has its problems when multiple additional requirements need to be combined. This will be covered in the following section where a further requirement is introduced. In the appendix B, two actors are presented where both end-to-end and basic acknowledgement strategies are included and controlled using a boolean flag *end2end*.

7.2 Event Filtering

As the name implies, the purpose of this extension is to filter particular events. It shall be possible to place such filters on arbitrary nodes within the network with the effect that the filtered

event shall not be further forwarded (*Sensor* and *Network Node*) or processed (*Level Crossing*). Clearly, to prevent the repeated resending of the event, the necessary acknowledge messages have to be sent to the downstream nodes. The filter condition is part of the business code and can therefore be specified by the user. As the condition is replaceable, arbitrary variants are possible. An example could be the filtering of events which arrive in a certain time window — then, multiple events could be treated as a single event.

In the *Peer Model*, the filter is realized as an additional service for the *Treat Event* pattern. This filter is designed to be executed after the configured service methods. Figure 7.3 depicts the wiring of the *Treat Event* pattern configured for *Network Node* peers and enhanced with the filtering mechanism. The reconfigured pattern goes through the following steps:

1. The wiring's input links are satisfied.

2. The configured service methods of the concrete peer are called. Depending on the specific configuration, entries of the types *retry*, *ack_ready* and *act* may be emitted.

3. The *filter*-service containing the user-defined filter condition is called. It reads the corresponding *event* and consumes the available *retry*, *ack_ready* and *act* entries. Depending on the combination of the service methods, some entries may not be available, therefore ≥ 0 is used as query condition.

4. Based on the filter condition the following outputs are possible:

    a) Condition is fulfilled: The service emits only an *ack_ready* entry. Therefore, the forwarding of the event is stopped as there is no *retry* entry which activates the wiring of the *Send and Retry* pattern (for *Sensor* and *Network Node* peers). When deployed on *Level Crossing* peers, the non-existent *act* entry prevents the event from being processed by the *Process Event* pattern containing the user-defined logic. The emitted *ack_ready* is used as usual to acknowledge the sender of the event.

    b) Condition is not fulfilled: The items emitted of the previously executed services are moved back untouched to the wiring's entry collection. Therefore, the event can be forwarded or processed ordinarily.

In Akka such a filter can be added to the existing actors and is activatable by a flag in a configuration file or by creating specific filter based versions of the actors. As already mentioned such independent actors provide better readability and are easier to maintain and verify. The disadvantage following this strategy of creating specific actors for specific requirements is that a combination of features requires the creation of additional actors which means, as a consequence, increased development effort. This ends up in having a wide range of actors which, apart from these feature-driven deviations, provide almost the same core functionality. This common functionality can lead to massive refactoring work as all affected actors have to be modified if changes are made to central parts. Clearly, configuration files and libraries can help to mitigate these problems, but the sheer quantity of these actors increases the effort for maintenance and testing even though single actors may be maintained and tested with little effort. The other strategy is to incorporate all features into concrete actors with the ability to enable and disable single

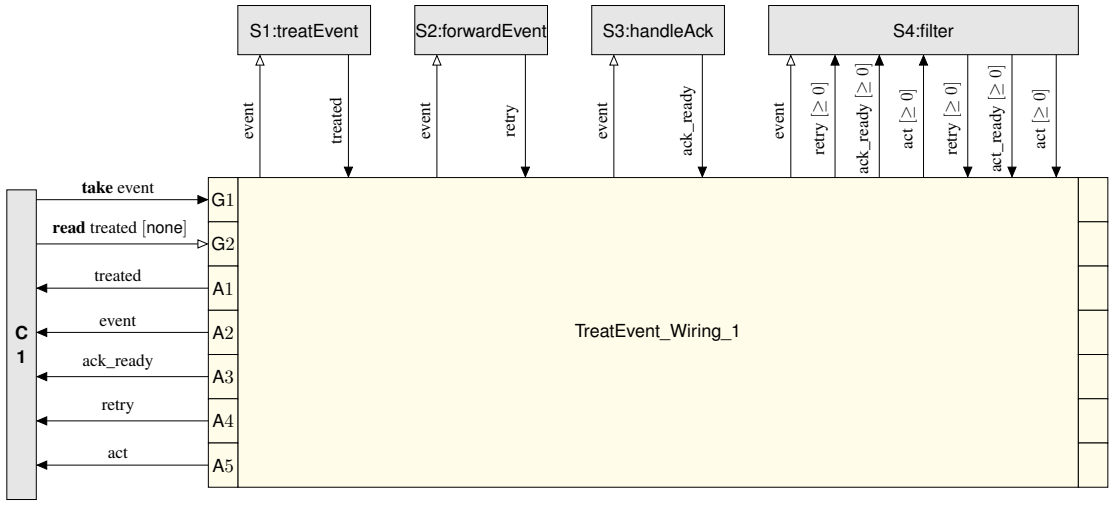

Figure 7.3: Treat Event Pattern extended with Filter

features. This strategy might be useful if there are only small deviations like in the end-to-end versus basic acknowledgement variants seen before, but when combining features like the acknowledgement and the filter mechanism a massive amount of conditional statements have to be included into the code and readability, testability and maintainability are significantly reduced. Further, such checks influence the performance of systems as well. However, these problems are not only related to the Actor Model, these are common software development problems. The *Peer Model* provides a more direct way, design decisions are delayed and the system's components are combined on a by need basis.

The flexibility of the pattern concept, which allows specifying different service methods as well as the chance to execute multiple service methods in a configurable order, is the key to keep the systems clean and readable. To demonstrate this elasticity, the previously presented end-to-end acknowledgement and the filter mechanism can be used both independently and in a combined fashion without interfering with each other and without additional configuration effort. As explained, the last peer in the network creates an *ack_ready* entry which is then sent upstream in a cascading way triggering the single *Send Acknowledgement* pattern wirings. If the filter condition is satisfied on an arbitrary peer, the coordination code of the filter service acts like the final node in the network and emits an *ack_ready* entry. Independently of whether the *handleAck* service method is used or not (i.e. whether the basic or the end-to-end variant is followed), the acknowledgement messaging is triggered correctly. Moreover, the presented filter mechanism can be used on all nodes and is not limited to *Network Node* peers. In Akka, this would have required the creation of several filter based actors for the different types. Further, in the *Peer Model* these two variants end-to-end acknowledgement and filtering can be used for both the basic upstream as well as the group-based upstream variant. The reason is that as already explained, the group-based version encapsulates the logic into a sub-peer which does not affect the group internal communication.

This two small examples have demonstrated the *Peer Model's* ability to react to changing business requirements without breaking the original design and architecture and therefore keeping the system clean and readable.

7.3 Conclusion

This section considers the evaluation criteria presented earlier in this work and investigates how the *Peer Model* combined with the proposed pattern concept fulfils the criterion catalog.

The first criterion was composition. The *Peer Model* offers two kinds of composition: first, peers can contain sub-peers, and thereby, arbitrary nesting of functionality is possible. Moreover, the proposed pattern concept introduces also a form of composition in which patterns can be formed out of other patterns. Chapter 5 described in detail how these two forms work together. As composition is an important factor in the *Peer Model* as well as a central point in this work, the corresponding criterion is fulfilled.

The second criterion was reuse. Basically, patterns are designed to be used more than once. This has been demonstrated extensively in chapter 6. The proposed work does indeed allow using multiple instances of the same pattern in a single peer — thus, the reuse of functionality is widely present in the *Peer Model* and the criterion is satisfied.

Patterns in the *Peer Model* are parametrizable to a great degree. Different service methods can be attached to wirings or replace existing ones to influence the system's behaviour. Furthermore, even the source or target containers of input and output links may be specified as well as the type of the entry to be transferred or the query condition. The configuration happens on the peer instance level meaning that every instance of a peer can be configured differently. The capability to specify default values (including services as well) during the pattern creation helps to mitigate configuration effort. In conclusion, the criterion concerning parametrization is also fulfilled.

The next criterion was "separation of concerns". As already described, the *Peer Model* follows the strategy of separating the coordination from the business logic, therefore this criterion is satisfied as well. The actual business logic is outsourced to service methods which call the corresponding code. The components of the *Peer Model*, namely peers, wirings, sub-peers, and service methods containing coordination code form the coordination logic.

In the *Peer Model*, peer instances can be added and removed at runtime. For example, it is possible to design a flexible version of variant B of the running use case. That would require that a new group member sends "hello" entries to its group members as well as to the downstream group to inform them about their new partner peer. Every group member requires maintaining a dynamic list of its members in form of an ordinary entry. In addition, another entry is required to store the addresses of the upstream partners. Both entries are then used whenever group acknowledgement or event forwarding tasks have to be performed. In the presented version of the group-based upstream example such dynamism is not provided as the addresses of the group members and upstream partners are specified in form of properties at configuration time. Nevertheless, the *Peer Model* provides the means to model such dynamic systems as explained and therefore the criterion regarding dynamics is satisfied.

In the *Peer Model* there exists communication within a single peer using wirings, as well as the possibility to communicate with remote peers using the system-defined property *DEST*. This allows creating such flexible systems as discussed in the previous paragraph where peer instances can be added and removed at runtime. Therefore, the connections between peers don't have to be "hardwired". Moreover, the concept of system-defined and user-defined properties allows one to also attach additional information to entries. For example, the sender of an entry delivered over the network to a remote peer can be stored, which gives the receiver the opportunity to reply. Thus, the requirements concerning the criterion "addressing" are fulfilled.

The next criterion that was discussed was scalability. In the *Peer Model*, a peer has to be designed only once. Afterwards it is possible to create an arbitrary amount of instances of this peer without the need for any adjustments to the original peer's design. The design stays the same; it is completely independent of the actual amount of runtime instances. As seen throughout the evaluation in chapter 3, some other approaches would have ended in explicitly modelling these additional connections and components and therefore the models would have blown up more and more and as a consequence they have become hard to read and understand. The *Peer Model* follows the strategy of distinguishing between the peer at the one hand (which abstracts the concrete number of runtime components) and the peer instances on the other hand (which correspond to these runtime components). This abstraction is the reason why the *Peer Model* satisfies the scalability criterion.

As seen in the *Send and Retry* pattern of variant A as well as in the series of the *Failover* patterns in variant B, the *Peer Model* allows the inclusion of time-functionality which influences the processing of entries. The system-defined property "TTS" allows specifying when an entry should be available for processing. In combination with a suitable wiring this could be used to realize a time triggered execution of a certain service method or even a complete execution path. Furthermore, the "TTL" property provides the means to control the amount of time before an entry is considered as expired and wrapped with an exception entry. The flow mechanism allows regulating these properties on an even higher level, affecting multiple entries. In conclusion, the *Peer Model* satisfies the "time" criterion.

Currently, there exists a Java-based implementation which supports the execution of systems modelled within the *Peer Model*. Moreover, a C# based version is already available [50]. It provides a core API, and on top of that an API with focus on usability. The extensible *Peer Model* domain specific language, code generation for embedded hardware in ANSI C and documentation generation is covered in [18]. Further, an interactive, visual monitoring tool is in development which supports developers in debugging models and understanding the flow of entries throughout them [14]. Currently, the *Peer Model* is realized for different platforms, however, the majority of these implementations and additional tools is not yet finished. The same holds for the documentation — thus, the criterion "toolchain & documentation" is partially fulfilled.

First, simple, models can be created easily. The most important step is to understand how wirings work and how they can be combined to realize the desired functionality. Wirings attached with services are then the typical way to include coordination logic (and later also business logic) and remote functionality. Therefore, users can start with simple models and extend them with their increasing knowledge to more complex systems.

Table 7.1 shows a copy of the results of the evaluation in chapter 3, table 3.1 — except those

from the Actor Model. Table 7.2 compares the results of the Actor Model with the previously presented findings of the *Peer Model*. As can be seen, the extended *Peer Model* provides the best results with respect to the set-up criterion catalog. This chapter has also shown the advantages of the proposed work in case of changing requirements. This is the reason why table 7.2 has an additional column containing *flexibility* as "new" criterion.

The results have to be read in the same fashion as in section 3.9: a '$+$' denotes that the related approach fulfils the considered criterion, whereas a '$\sim$' denotes that the criterion is partially fulfilled. A '$-$' indicates non-fulfilment.

	CPN	**Reo**	**Uppaal**	**BPMN**	**WS-BPEL**
Composition	$+$	$+$	$-$	$+$	$+$
Reuse	$+$	$+$	$+$	$+$	$+$
Parametrization	$\sim$	$+$	$+$	$+$	$+$
SoC	$-$	$+$	$-$	$+$	$+$
Dynamics	$-$	$+$	$-$	$\sim$	$+$
Addressing	$-$	$\sim$	$+$	$\sim$	$+$
Scalability	$-$	$-$	$+$	$+$	$+$
Time	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
Toolchain & Docs	$\sim$	$\sim$	$+$	$+$	$+$
Simplicity	$+$	$-$	$+$	$\sim$	$-$

Table 7.1: Final classification (part I) of the considered modelling concepts and tools

	Actor Model	**Peer Model**
Composition	$+$	$+$
Reuse	$+$	$+$
Parametrization	$+$	$+$
SoC	$+$	$+$
Dynamics	$+$	$+$
Addressing	$+$	$+$
Scalability	$+$	$+$
Time	$\sim$	$+$
Toolchain & Docs	$\sim$	$\sim$
Simplicity	$+$	$+$
Flexibility	$\sim$	$+$

Table 7.2: Final classification (part II) of the considered modelling concepts and tools

CHAPTER 8

Conclusion

8.1 Summary

Developing each component of a new product from scratch is costly, includes risks and has negative influence on the time to market. Therefore, the strategic reuse of software components is an important factor for the success of companies. It can leverage existing software investment, companies can build systems out of well-tested components of proven quality which have been used a couple of times and thus, both risks as well as costs for development and testing can be reduced. The software product line approach follows this aim by creating a platform of flexible components which can be selected and combined into different products tailored to stakeholder requirements. This approach enables cost efficient mass customization.

In this work, a pattern-based approach was presented which provides this variability known from software product lines for the *Peer Model*. Patterns were introduced as new components for the *Peer Model* and their interrelations with the original components were described. The pattern concept of this work allows defining generic patterns which depend on certain parameters or properties. Moreover, patterns can be combined to form more complex patterns — composition, in fact, is an essential factor in this work. At configuration time, immediately before the patterns are deployed, the user specifies the concrete behaviour. This influence on the behaviour is not limited to minor changes: the pattern concept allows for far-reaching modifications and thus for reuse on a large scale. Generic patterns can be designed and their concrete behaviour can be tailored to the particular requirements.

The advantages of this approach were demonstrated by an example use case from a real world domain, the train traffic telematics domain. Signals of approaching trains need to be transferred from a sensor over multiple network nodes along the track to the level crossing unit. The use case was split up in two variants: a basic and a more complex variant including the clustering of single nodes into groups to form a more fault-tolerant system. Various modelling concepts and tools ranging from more low-level to highly abstract approaches were selected and with each of them the example use case was realized. In order to create a meaningful evaluation, a set of criteria was developed emphasizing elements which are important for the design of

highly distributed and concurrent systems and essential for the aspired-to variability. The final evaluation has shown that the *Peer Model*, extended with the pattern concept, stands out from the other approaches in the domain of distributed and highly concurrent systems with respect to the developed criteria. Systems designed with the *Peer Model* can be adapted to changing requirements without modifying the underlying architectural design and thus, cost- and time-intensive remodelling and refactoring work can be avoided.

8.2 Future Work

Following the work presented in this thesis, there is possible further work which could not be addressed sufficiently:

- Implementation: A version of the *Peer Model* which provides the features of this pattern concept is subject of future work. Currently, different variants of the *Peer Model* are in development but none support patterns as "first-class citizens". Though the parameter mechanism can be realized using simple programming language constructs, patterns as components and the adding of input links, output links and services at configuration time are not yet supported. In current versions, a pattern is added to a model either by putting its components into the particular peer or by creating a sub-peer.

- Extending the *Peer Model's* domain specific language: A prerequisite for the above mentioned implementation is the extension of the domain specific language. The existing domain specific language has to be analyzed and extended such that patterns and their properties can be supported.

- Distributed Patterns: The pattern concept proposed in this work defined patterns as location-free, virtual. This allows deploying different parts of a pattern on different nodes in the network. However, the concrete mechanism including a lookup-alike approach to discover the parts of a pattern in the network, as well as the effects of this distribution on the flow of entries throughout the pattern is subject to future work.

- Pattern Catalog: This work has provided a proper basis for the creation of a so called "pattern catalog", which is a collection of frequently used and well-documented patterns. The idea of this catalog is to support the development of systems where the user/designer selects particular patterns and combines them as needed. The flexibility provided in this work allows that arbitrary patterns can be combined and adjusted to the concrete stakeholder requirements. Such an approach would enforce the reuse of components and as a consequence also decrease development time and risks, and increase product quality.

- Limiting a pattern's modifiability: In the proposed pattern concept, the parametrization allows one to influence a pattern's behaviour to a great extent. The question is whether this capability should be restricted in such a way that the designed behaviour can only be slightly modified — thus, patterns must retain their original purpose.

APPENDIX A

WS-BPEL Example Processes

A.1 Variant A

A.1.1 WS-BPEL Sensor Process

```
<bpel:process name="SensorProcess"
              targetNamespace="http://sensorprocess.localhost"
              suppressJoinFailure="yes"
              xmlns:tns="http://sensorprocess.localhost"
              xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
                  executable"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:np="http://nodeprocess.localhost">

  <bpel:import namespace="http://nodeprocess.localhost"
               location="NodeProcessArtifacts.wsdl"
               importType="http://schemas.xmlsoap.org/wsdl/" />
  <bpel:import location="SensorProcessArtifacts.wsdl"
               namespace="http://sensorprocess.localhost"
               importType="http://schemas.xmlsoap.org/wsdl/" />

  <!-- ============================================================ -->
  <!-- PARTNERLINKS -->
  <!-- ============================================================ -->
  <bpel:partnerLinks>
    <!-- The 'client' role represents the requester of this service. -->
    <!-- In this case the sensor unit. -->
    <bpel:partnerLink name="client"
                      partnerLinkType="tns:SensorProcess"
                      myRole="SensorProcessProvider" />

    <bpel:partnerLink name="ForwardEventPL"
                      partnerLinkType="np:NodeProcess"
                      partnerRole="NodeProcessProvider"
```

```
                    myRole="NodeProcessRequester" />

  </bpel:partnerLinks>

  <!-- ============================================================ -->
  <!-- VARIABLES -->
  <!-- ============================================================ -->
  <bpel:variables>
    <bpel:variable name="input"
                   messageType="tns:SensorProcessRequestMessage" />
    <bpel:variable name="output"
                   messageType="tns:SensorProcessResponseMessage" />
    <bpel:variable name="ForwardEventMessage"
                   messageType="np:NodeProcessRequestMessage" />
    <bpel:variable name="ReceiveAckMessage"
                   messageType="np:NodeProcessResponseMessage" />
    <bpel:variable name="terminate"
                   type="xs:boolean"/>
    <bpel:variable name="counter"
                   type="xs:integer"/>
  </bpel:variables>

  <!-- ============================================================ -->
  <!-- ORCHESTRATION LOGIC -->
  <!-- Set of activities coordinating the flow of messages across
       the services integrated within this business process        -->
  <!-- ============================================================ -->
  <bpel:sequence name="main">
    <bpel:receive name="receiveEvent"
                  partnerLink="client"
                  portType="tns:SensorProcess"
                  operation="process"
                  variable="input"
                  createInstance="yes" />

    <bpel:assign validate="no"
                 name="AssignEvent">
      <bpel:copy>
        <bpel:from>
          <bpel:literal>
            <tns:NodeProcessRequest
               xmlns:tns="http://nodeprocess.localhost"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
              <tns:input>tns:input</tns:input>
            </tns:NodeProcessRequest>
          </bpel:literal>
        </bpel:from>
        <bpel:to variable="ForwardEventMessage"
                 part="payload"/>
      </bpel:copy>
      <bpel:copy>
        <bpel:from part="payload"
                   variable="input">
```

```
      <bpel:query
        queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
        <![CDATA[tns:input]]>
      </bpel:query>
    </bpel:from>
    <bpel:to part="payload"
             variable="ForwardEventMessage">
      <bpel:query
        queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
        <![CDATA[np:input]]>
      </bpel:query>
    </bpel:to>
  </bpel:copy>
</bpel:assign>

<bpel:assign validate="no"
             name="InitCounter">
  <bpel:copy>
    <bpel:from>
      <bpel:literal xml:space="preserve">5</bpel:literal>
    </bpel:from>
    <bpel:to variable="counter"></bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:assign validate="no"
             name="InitFlag">
  <bpel:copy>
    <bpel:from>
      <bpel:literal xml:space="preserve">false</bpel:literal>
    </bpel:from>
    <bpel:to variable="terminate"></bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:repeatUntil name="RepeatUntil">
  <bpel:sequence>
    <bpel:invoke name="ForwardEvent"
                 partnerLink="ForwardEventPL"
                 operation="initiate"
                 portType="np:NodeProcess"
                 inputVariable="ForwardEventMessage"></bpel:invoke>

    <bpel:pick name="Pick">
      <bpel:onMessage partnerLink="ForwardEventPL"
                      operation="onAcknowledgement"
                      portType="np:NodeProcessCallback"
                      variable="ReceiveAckMessage">

        <bpel:assign validate="no"
                     name="endLoop">
          <bpel:copy>
            <bpel:from>
              <bpel:literal xml:space="preserve">true</bpel:literal>
            </bpel:from>
```

```
            <bpel:to variable="terminate"></bpel:to>
          </bpel:copy>
        </bpel:assign>
      </bpel:onMessage>

      <bpel:onAlarm>
        <bpel:for>'PT5S'</bpel:for>
        <bpel:assign name="decreaseCounter">
          <bpel:copy>
            <bpel:from
              expressionLanguage="urn:oasis:names:tc:wsbpel:2.0
                  :sublang:xpath1.0">
              <![CDATA[$counter - 1]]>
            </bpel:from>
            <bpel:to variable="counter"></bpel:to>
          </bpel:copy>
        </bpel:assign>
      </bpel:onAlarm>
    </bpel:pick>

  </bpel:sequence>
  <bpel:condition
    expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
    <![CDATA[$terminate or $counter = 0]]>
  </bpel:condition>
</bpel:repeatUntil>

<bpel:if name="acknowledgementReceived">
  <bpel:condition
    expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
    <![CDATA[$terminate]]>
  </bpel:condition>

  <bpel:assign validate="no"
               name="AssignOK">
    <bpel:copy>
      <bpel:from>
        <bpel:literal>
          <tns:SensorProcessResponse
            xmlns:tns="http://sensorprocess.localhost"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <tns:result>tns:result</tns:result>
          </tns:SensorProcessResponse>
        </bpel:literal>
      </bpel:from>
      <bpel:to variable="output"
               part="payload"></bpel:to>
    </bpel:copy>
    <bpel:copy>
      <bpel:from part="payload"
                 variable="ReceiveAckMessage">
        <bpel:query
          queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
```

```
            <![CDATA[ np:result ]]>
          </bpel:query>
        </bpel:from>
        <bpel:to part="payload"
                 variable="output">
          <bpel:query
            queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
            <![CDATA[ tns:result ]]>
          </bpel:query>
        </bpel:to>
      </bpel:copy>
    </bpel:assign>
    <bpel:else>
      <bpel:assign validate="no"
                   name="AssignErrorMessage">
        <bpel:copy>
          <bpel:from>
            <bpel:literal>
              <tns:SensorProcessResponse
                xmlns:tns="http://sensorprocess.localhost"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <tns:result>tns:result</tns:result>
              </tns:SensorProcessResponse>
            </bpel:literal>
          </bpel:from>
          <bpel:to variable="output"
                   part="payload"></bpel:to>
        </bpel:copy>
        <bpel:copy>
          <bpel:from>
            <bpel:literal>
              <tns:SensorProcessResponse
                xmlns:tns="http://sensorprocess.localhost"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <tns:result>Event couldn't be forwarded!</tns:result>
              </tns:SensorProcessResponse>
            </bpel:literal>
          </bpel:from>
          <bpel:to part="payload"
                   variable="output"></bpel:to>
        </bpel:copy>
      </bpel:assign>
    </bpel:else>
  </bpel:if>
  <bpel:reply name="replyOutput"
              partnerLink="client"
              portType="tns:SensorProcess"
              operation="process"
              variable="output" />
 </bpel:sequence>
</bpel:process>
```

Listing A.1: WS-BPEL - Sensor Process

A.1.2 WS-BPEL Sensor Process WSDL

```
<?xml version="1.0"?>
<definitions name="SensorProcess"
             targetNamespace="http://sensorprocess.localhost"
             xmlns:tns="http://sensorprocess.localhost"
             xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <!-- ================================================================= -->
  <!-- TYPE DEFINITION - List of types participating in this process -->
  <!-- ================================================================= -->
  <types>
    <schema attributeFormDefault="unqualified"
            elementFormDefault="qualified"
            targetNamespace="http://sensorprocess.localhost"
            xmlns="http://www.w3.org/2001/XMLSchema">

      <element name="SensorProcessRequest">
        <complexType>
          <sequence>
            <element name="input"
                     type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="SensorProcessResponse">
        <complexType>
          <sequence>
            <element name="result"
                     type="string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>

  <!-- ================================================================= -->
  <!-- MESSAGE TYPE DEFINITION - Definition of the message types
       used as part of the port type defintions -->
  <!-- ================================================================= -->
  <message name="SensorProcessRequestMessage">
    <part name="payload"
          element="tns:SensorProcessRequest"/>
  </message>
  <message name="SensorProcessResponseMessage">
    <part name="payload"
          element="tns:SensorProcessResponse"/>
  </message>

  <!-- ================================================================= -->
  <!-- PORT TYPE DEFINITION - A port type groups a set of operations
```

```
      into a logical service unit. -->
  <!-- ================================================================= -->

  <!-- portType implemented by the SensorProcess BPEL process -->
  <portType name="SensorProcess">
    <operation name="process">
      <input  message="tns:SensorProcessRequestMessage" />
      <output message="tns:SensorProcessResponseMessage"/>
    </operation>
  </portType>

  <!-- ================================================================= -->
  <!-- PARTNER LINK TYPE DEFINITION -->
  <!-- ================================================================= -->
  <plnk:partnerLinkType name="SensorProcess">
    <plnk:role name="SensorProcessProvider"
               portType="tns:SensorProcess"/>
  </plnk:partnerLinkType>

  <!-- ================================================================= -->
  <!-- BINDING DEFINITION - Defines the message format and protocol
       details for a web service. -->
  <!-- ================================================================= -->
  <binding name="SensorProcessBinding"
           type="tns:SensorProcess">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="process">
      <soap:operation
        soapAction="http://sensorprocess.localhost/process" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>

  <!-- ================================================================= -->
  <!-- SERVICE DEFINITION - A service groups a set of ports into
       a service unit. -->
  <!-- ================================================================= -->
  <service name="SensorProcessService">
    <port name="SensorProcessPort"
          binding="tns:SensorProcessBinding">
      <soap:address
        location="http://localhost:8080/ode/processes/SensorProcess" />
    </port>
  </service>
</definitions>
```

Listing A.2: WS-BPEL - Sensor Process WSDL

A.1.3 WS-BPEL Node Process WSDL

```
<?xml version="1.0"?>
<definitions name="NodeProcess"
             targetNamespace="http://nodeprocess.localhost"
             xmlns:tns="http://nodeprocess.localhost"
             xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        >
  <types>
    <schema attributeFormDefault="unqualified"
            elementFormDefault="qualified"
            targetNamespace="http://nodeprocess.localhost"
            xmlns="http://www.w3.org/2001/XMLSchema">

      <element name="NodeProcessRequest">
        <complexType>
          <sequence>
            <element name="input"
                     type="string" />
          </sequence>
        </complexType>
      </element>

      <element name="NodeProcessResponse">
        <complexType>
          <sequence>
            <element name="result"
                     type="string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>

  <!-- ================================================================= -->
  <!-- MESSAGE TYPE DEFINITION - Definition of the message types used
       as part of the port type defintions -->
  <!-- ================================================================= -->

  <message name="NodeProcessRequestMessage">
    <part name="payload"
          element="tns:NodeProcessRequest"/>
  </message>

  <message name="NodeProcessResponseMessage">
    <part name="payload"
          element="tns:NodeProcessResponse"/>
  </message>

  <!-- ================================================================= -->
  <!-- PORT TYPE DEFINITION - A port type groups a set of operations
```

```
     into a logical service unit. -->
<!-- ================================================================= -->

<!-- portType implemented by the NodeProcess BPEL process -->
<portType name="NodeProcess">
  <operation name="initiate">
    <input message="tns:NodeProcessRequestMessage"/>
  </operation>
</portType>

<!-- portType implemented by the requester of NodeProcess BPEL
       process for asynchronous callback purposes
       -->
<portType name="NodeProcessCallback">
  <operation name="onAcknowledgement">
    <input message="tns:NodeProcessResponseMessage"/>
  </operation>
</portType>

<!-- ================================================================= -->
<!--   PARTNER LINK TYPE DEFINITION -->
<!-- ================================================================= -->
<plnk:partnerLinkType name="NodeProcess">
  <plnk:role name="NodeProcessProvider"
             portType="tns:NodeProcess"/>
  <plnk:role name="NodeProcessRequester"
             portType="tns:NodeProcessCallback"/>
</plnk:partnerLinkType>

<!-- ================================================================= -->
<!-- BINDING DEFINITION -->
<!-- ================================================================= -->
<binding name="NodeProcessBinding"
         type="tns:NodeProcess">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="initiate">
    <soap:operation
      soapAction="http://nodeprocess.localhost/initiate"/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>

<binding name="NodeProcessCallbackBinding"
         type="tns:NodeProcessCallback">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="onAcknowledgement">
    <soap:operation
      soapAction="http://nodeprocess.localhost/onResult"/>
    <input>
```

```
        <soap:body use="literal"/>
      </input>
    </operation>
  </binding>

  <!-- ================================================================= -->
  <!-- SERVICE DEFINITION -->
  <!-- ================================================================= -->

  <service name="NodeProcessService">
    <port name="NodeProcessPort"
          binding="tns:NodeProcessBinding">
      <soap:address
        location="http://localhost:8080/ode/processes/NodeProcess"/>
    </port>
  </service>
  <service name="NodeProcessServiceCallback">
    <port name="NodeProcessPortCallbackPort"
          binding="tns:NodeProcessCallbackBinding">
      <soap:address
        location="http://localhost:8080/ode/processes/NodeProcessCallback"/>
    </port>
  </service>
</definitions>
```

Listing A.3: WS-BPEL - Node Process WSDL

A.2 Variant B

A.2.1 WS-BPEL Group Member Process

```
<bpel:process name="BPELForward"
              targetNamespace="http://node.train.demo"
              suppressJoinFailure="yes"
              xmlns:tns="http://node.train.demo"
              xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
                  executable"
              xmlns:ns="http://group.train.demo/"
              xmlns:fw="http://forward.train.demo/"
              xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <bpel:import namespace="http://forward.train.demo/"
               location="ForwardSignalWS.wsdl"
               importType="http://schemas.xmlsoap.org/wsdl/"></bpel:import>
  <bpel:import namespace="http://group.train.demo/"
               location="GroupCoordinationWS.wsdl"
               importType="http://schemas.xmlsoap.org/wsdl/"></bpel:import>
  <bpel:import location="BPELForwardArtifacts.wsdl"
               namespace="http://node.train.demo"
               importType="http://schemas.xmlsoap.org/wsdl/" />
```

```
<!-- ================================================================ -->
<!-- PARTNERLINKS -->
<!-- ================================================================ -->
<bpel:partnerLinks>
  <bpel:partnerLink name="downstream"
                    partnerLinkType="tns:BPELForward"
                    myRole="BPELForwardProvider"
                    partnerRole="BPELForwardRequester" />

  <bpel:partnerLink name="group"
                    partnerLinkType="tns:GroupCoordinationWS"
                    partnerRole="groupOps"></bpel:partnerLink>

  <bpel:partnerLink name="forward"
                    partnerLinkType="tns:ForwardSignalWS"
                    partnerRole="forwardSignal"></bpel:partnerLink>
</bpel:partnerLinks>

<!-- ================================================================ -->
<!-- VARIABLES -->
<!-- ================================================================ -->
<bpel:variables>
  <bpel:variable name="input"
                 messageType="tns:BPELForwardRequestMessage" />
  <bpel:variable name="ackMsg"
                 messageType="tns:BPELForwardResponseMessage" />
  <bpel:variable name="isLeaderRsp"
                 messageType="ns:isGroupLeaderResponse"/>
  <bpel:variable name="nodeIDMsg"
                 messageType="ns:isGroupLeader"/>
  <bpel:variable name="groupAckMsg"
                 messageType="ns:groupAck"/>
  <bpel:variable name="groupAckRsp"
                 messageType="ns:groupAckResponse"/>
  <bpel:variable name="forwardMsg"
                 messageType="fw:forwardSignal"/>
  <bpel:variable name="recGroupAckMsg"
                 messageType="tns:GroupAcknowledgementMessage"/>
  <bpel:variable name="claimLeadMsg"
                 messageType="ns:claimLeadership"/>
  <bpel:variable name="claimLeadRsp"
                 messageType="ns:claimLeadershipResponse"/>
  <bpel:variable name="nodeID"
                 type="xs:int"/>
</bpel:variables>

<!-- ================================================================ -->
<!-- ORCHESTRATION LOGIC -->
<!-- Set of activities coordinating the flow of messages across the
     services integrated within this business process -->
<!-- ================================================================ -->
<bpel:sequence name="main">
  <bpel:receive name="receiveInput"
```

```
                partnerLink="downstream"
                portType="tns:BPELForward"
                operation="initiate"
                variable="input"
                createInstance="yes" />

<bpel:assign validate="no"
             name="AssignNodeID">
  <bpel:copy>
    <bpel:from>
      <bpel:literal xml:space="preserve">1</bpel:literal>
    </bpel:from>
    <bpel:to variable="nodeID"></bpel:to>
  </bpel:copy>
</bpel:assign>

<bpel:assign validate="no"
             name="AssignLeadershipMsg">
  <bpel:copy>
    <bpel:from>
      <bpel:literal>
        <tns:isGroupLeader xmlns:tns="http://group.train.demo/"
                           xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                               instance">
          <nodeID>0</nodeID>
        </tns:isGroupLeader>
      </bpel:literal>
    </bpel:from>
    <bpel:to variable="nodeIDMsg"
             part="parameters"></bpel:to>
  </bpel:copy>
  <bpel:copy>
    <bpel:from variable="nodeID"></bpel:from>
    <bpel:to part="parameters"
             variable="nodeIDMsg">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[nodeID]]></bpel:query>
    </bpel:to>
  </bpel:copy>
</bpel:assign>

<bpel:invoke name="checkLeadership"
             partnerLink="group"
             operation="isGroupLeader"
             portType="ns:GroupCoordinationWS"
             inputVariable="nodeIDMsg"
             outputVariable="isLeaderRsp"></bpel:invoke>

<bpel:if name="isLeader">
  <bpel:condition>
    $isLeaderRsp.parameters//return = "true"
  </bpel:condition>
  <bpel:sequence>
```

```
<bpel:assign validate="no"
             name="AssignGroupAckMsg">
  <bpel:copy>
    <bpel:from>
      <bpel:literal>
        <tns:groupAck xmlns:tns="http://group.train.demo/"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                          instance">
          <signal>signal</signal>
        </tns:groupAck>
      </bpel:literal>
    </bpel:from>
    <bpel:to variable="groupAckMsg"
             part="parameters"></bpel:to>
  </bpel:copy>
  <bpel:copy>
    <bpel:from part="payload"
               variable="input">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[tns:signal]]></bpel:query>
    </bpel:from>
    <bpel:to part="parameters"
             variable="groupAckMsg">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[signal]]></bpel:query>
    </bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:invoke name="groupAck"
             partnerLink="group"
             operation="groupAck"
             portType="ns:GroupCoordinationWS"
             inputVariable="groupAckMsg"
             outputVariable="groupAckRsp"></bpel:invoke>
<bpel:assign validate="no"
             name="AssignForwardMsg">
  <bpel:copy>
    <bpel:from>
      <bpel:literal>
        <tns:forwardSignal xmlns:tns="http://forward.train.demo/"
                           xmlns:xsi="http://www.w3.org/2001/
                               XMLSchema-instance">
          <signal>signal</signal>
        </tns:forwardSignal>
      </bpel:literal>
    </bpel:from>
    <bpel:to variable="forwardMsg"
             part="parameters"></bpel:to>
  </bpel:copy>
  <bpel:copy>
    <bpel:from part="payload"
               variable="input">
```

```
          <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
              :sublang:xpath1.0"><![CDATA[tns:signal]]></bpel:query>
        </bpel:from>
        <bpel:to part="parameters"
                 variable="forwardMsg">
          <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
              :sublang:xpath1.0"><![CDATA[signal]]></bpel:query>
        </bpel:to>
      </bpel:copy>
    </bpel:assign>
    <bpel:invoke name="EventForwarder"
                 partnerLink="forward"
                 operation="forwardSignal"
                 portType="fw:ForwardSignalWS"
                 inputVariable="forwardMsg"></bpel:invoke>
    <bpel:assign validate="no"
                 name="AssignAckMsg">
      <bpel:copy>
        <bpel:from>
          <bpel:literal xml:space="preserve"><tns:BPELForwardResponse
              xmlns:tns="http://node.train.demo"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                        <tns:result>is Leader</tns:result>
                        </tns:BPELForwardResponse></bpel:literal>
        </bpel:from>
        <bpel:to part="payload"
                 variable="ackMsg" />
      </bpel:copy>

    </bpel:assign>
  </bpel:sequence>
  <bpel:else>
    <bpel:pick name="Pick">
      <bpel:onMessage partnerLink="downstream"
                      operation="groupAck"
                      portType="tns:BPELForward"
                      variable="recGroupAckMsg">
        <bpel:exit />
      </bpel:onMessage>

      <bpel:onAlarm>
        <bpel:for>'PT10S'</bpel:for>
        <bpel:sequence>
          <bpel:assign validate="no"
                       name="AssignLeadMsg">
            <bpel:copy>
              <bpel:from>
                <bpel:literal>
                  <tns:claimLeadership xmlns:tns="http://group.train.demo
                      /"
                                       xmlns:xsi="http://www.w3.org/2001/
                                           XMLSchema-instance">
                    <nodeId>0</nodeId>
```

```
        </tns:claimLeadership>
      </bpel:literal>
    </bpel:from>
    <bpel:to variable="claimLeadMsg"
             part="parameters"></bpel:to>
  </bpel:copy>
  <bpel:copy>
    <bpel:from variable="nodeID"></bpel:from>
    <bpel:to part="parameters"
             variable="claimLeadMsg">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[nodeId]]></bpel:query>
    </bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:invoke name="claimLeadership"
             partnerLink="group"
             operation="claimLeadership"
             portType="ns:GroupCoordinationWS"
             inputVariable="claimLeadMsg"
             outputVariable="claimLeadRsp"></bpel:invoke>

<bpel:assign validate="no"
             name="AssignForwardMsg">
  <bpel:copy>
    <bpel:from>
      <bpel:literal>
        <tns:forwardSignal xmlns:tns="http://forward.train.demo
            /"
                           xmlns:xsi="http://www.w3.org/2001/
                               XMLSchema-instance">
          <signal>signal</signal>
        </tns:forwardSignal>
      </bpel:literal>
    </bpel:from>
    <bpel:to variable="forwardMsg"
             part="parameters"></bpel:to>
  </bpel:copy>
  <bpel:copy>
    <bpel:from part="payload"
               variable="input">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[tns:signal]]></bpel:query
          >
    </bpel:from>
    <bpel:to part="parameters"
             variable="forwardMsg">
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
          :sublang:xpath1.0"><![CDATA[signal]]></bpel:query>
    </bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:invoke name="EventForwarder"
```

```
                        partnerLink="forward"
                        operation="forwardSignal"
                        portType="fw:ForwardSignalWS"
                        inputVariable="forwardMsg"></bpel:invoke>

            <bpel:assign validate="no"
                         name="AssignAckMsg">
              <bpel:copy>
                <bpel:from>
                  <bpel:literal xml:space="preserve"><
                      tns:BPELForwardResponse
                       xmlns:tns="http://node.train.demo"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                           >
                                    <tns:result>Time elapsed - take group
                                         lead</tns:result>
                                    </tns:BPELForwardResponse></
                                        bpel:literal>
                </bpel:from>
                <bpel:to part="payload"
                         variable="ackMsg" />
              </bpel:copy>
            </bpel:assign>
          </bpel:sequence>
        </bpel:onAlarm>
      </bpel:pick>
    </bpel:else>
  </bpel:if>

  <bpel:invoke name="sendAck"
               partnerLink="downstream"
               portType="tns:BPELForwardCallback"
               operation="onResult"
               inputVariable="ackMsg" />
 </bpel:sequence>
</bpel:process>
```

Listing A.4: WS-BPEL - Group Member Process

A.2.2 WS-BPEL Group Member Process WSDL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://node.train.demo"
             xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
             xmlns:wsdl="http://group.train.demo/"
             xmlns:wsdl1="http://forward.train.demo/"
             name="BPELForward"
             targetNamespace="http://node.train.demo">
```

```
<import location="GroupCoordinationWS.wsdl"
        namespace="http://group.train.demo/" />
<import location="ForwardSignalWS.wsdl"
        namespace="http://forward.train.demo/" />

<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
          attributeFormDefault="unqualified"
          elementFormDefault="qualified"
          targetNamespace="http://node.train.demo">

    <element name="BPELForwardRequest">
      <complexType>
        <sequence>
          <element name="signal"
                   type="string" />
        </sequence>
      </complexType>
    </element>

    <element name="BPELForwardResponse">
      <complexType>
        <sequence>
          <element name="result"
                   type="string" />
        </sequence>
      </complexType>
    </element>

    <element name="GroupAcknowledgement">
      <complexType>
        <sequence>
          <element name="acknowledgement"
                   type="string" />
        </sequence>
      </complexType>
    </element>

  </schema>
</types>

<!-- ================================================================= -->
<!-- MESSAGE TYPE DEFINITION -->
<!-- ================================================================= -->

<message name="BPELForwardRequestMessage">
  <part element="tns:BPELForwardRequest"
        name="payload" />
</message>

<message name="BPELForwardResponseMessage">
  <part element="tns:BPELForwardResponse"
        name="payload" />
```

```
</message>

<message name="GroupAcknowledgementMessage">
  <part element="tns:GroupAcknowledgement"
        name="payload" />
</message>

<!-- ================================================================= -->
<!-- PORT TYPE DEFINITION -->
<!-- ================================================================= -->

<!-- portType implemented by the BPELForward BPEL process -->
<portType name="BPELForward">
  <operation name="initiate">
    <input message="tns:BPELForwardRequestMessage" />
  </operation>
  <operation name="groupAck">
    <input message="tns:GroupAcknowledgementMessage" />
  </operation>
</portType>

<!-- portType implemented by the requester of BPELForward BPEL process for
             asynchronous callback purposes -->
<portType name="BPELForwardCallback">
  <operation name="onResult">
    <input message="tns:BPELForwardResponseMessage" />
  </operation>
</portType>

<!-- ================================================================= -->
<!-- PARTNER LINK TYPE DEFINITION -->
<!-- ================================================================= -->

<plnk:partnerLinkType name="BPELForward">
  <plnk:role name="BPELForwardProvider"
             portType="tns:BPELForward" />
  <plnk:role name="BPELForwardRequester"
             portType="tns:BPELForwardCallback" />
</plnk:partnerLinkType>

<plnk:partnerLinkType name="GroupCoordinationWS">
  <plnk:role name="groupOps"
             portType="wsdl:GroupCoordinationWS" />
</plnk:partnerLinkType>

<plnk:partnerLinkType name="ForwardSignalWS">
  <plnk:role name="forwardSignal"
             portType="wsdl1:ForwardSignalWS" />
</plnk:partnerLinkType>

<!-- ================================================================= -->
<!-- BINDING DEFINITION -->
<!-- ================================================================= -->
```

```
  <binding name="BPELForwardBinding"
           type="tns:BPELForward">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="initiate">
      <soap:operation soapAction="http://node.train.demo/initiate" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
    <operation name="groupAck">
      <soap:operation soapAction="http://node.train.demo/groupAck" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
  </binding>

  <binding name="BPELForwardCallbackBinding"
           type="tns:BPELForwardCallback">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="onResult">
      <soap:operation soapAction="http://node.train.demo/onResult" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
  </binding>

  <!-- ================================================================= -->
  <!-- SERVICE DEFINITION - A service groups a set of ports into a
       service unit -->
  <!-- ================================================================= -->

  <service name="BPELForwardService">
    <port binding="tns:BPELForwardBinding"
          name="BPELForwardPort">
      <soap:address location="http://localhost:8080/ode/processes/BPELForward
          " />
    </port>
  </service>
  <service name="BPELForwardServiceCallback">
    <port binding="tns:BPELForwardCallbackBinding"
          name="BPELForwardPortCallbackPort">
      <soap:address
        location="http://localhost:8080/ode/processes/BPELForwardCallback" />
    </port>
  </service>
</definitions>
```

Listing A.5: WS-BPEL - Group Member Process WSDL

APPENDIX B

Akka Examples

B.1 Network Node Actor with End-to-End Acknowledgement

```
public class NetworkNode extends UntypedActor {
   private LoggingAdapter log =
      Logging.getLogger(getContext().system(), this);
   private TreeSet<Integer> processed;
   private final String upstream;
   private final boolean end2end;

   public NetworkNode(String upstream, boolean end2end) {
      this.upstream = upstream;
      this.end2end = end2end;
      this.processed = new TreeSet<>();
   }

   public static Props makeProps(String upstream, boolean end2end) {
      return Props.create(NetworkNode.class, upstream, end2end);
   }

   @Override
   public void onReceive(Object o) throws Exception {
      if(o instanceof EventMsg) {
         EventMsg m = (EventMsg) o;
         log.info(m.toString());

         if(!end2end) {
            //send ack
            this.getSender().tell(new AckMsg(m.getID()), this.getSelf());
         }

         if(!this.processed.contains(m.getID())) {
            this.processed.add(m.getID());
            this.getContext().actorOf(Forwarder.makeProps(
```

```
                    upstream, this.getSender(), m, end2end));
            }
        }else
            unhandled(o);
    }
}
```

Listing B.1: Akka - End-to-End Network Node Actor

B.2 Forwarder Actor with End-to-End Acknowledgement

```
public class Forwarder extends UntypedActor {
    private LoggingAdapter log =
        Logging.getLogger(getContext().system(), this);
    private final String upstream;
    private final EventMsg event;
    private final boolean end2end;
    private Cancellable handler;
    private ActorRef downstream;

    public Forwarder(String upstream, ActorRef downstream, EventMsg event,
        boolean end2end) {
        this.upstream = upstream;
        this.downstream = downstream;
        this.event = event;
        this.end2end = end2end;
        this.handler = null;
    }

    public static Props makeProps(String upstream, ActorRef downstream,
        EventMsg event, boolean end2end) {
        return Props.create(Forwarder.class, upstream, downstream, event,
            end2end);
    }

    @Override
    public void preStart() throws Exception {
        super.preStart();

        final ActorSelection selection = getContext().actorSelection(upstream);
        this.handler = this.getContext().system().scheduler().schedule(
            FiniteDuration.Zero(), Duration.create(5L, TimeUnit.SECONDS),
            new Runnable(){
                @Override
                public void run() {
                    selection.tell(event, getSelf());
                }

            }, this.getContext().dispatcher());
        this.getContext().setReceiveTimeout(Duration.create(25L, TimeUnit.
            SECONDS));
```

```
    }

    @Override
    public void onReceive(Object o) throws Exception {
        if (o instanceof AckMsg) {
            log.info(o.toString());
            this.handler.cancel();

            if(this.end2end && this.downstream!=null) {
                this.downstream.tell(o, getContext().parent());
            }
            this.getContext().stop(getSelf());
        } else if (o instanceof ReceiveTimeout) {
            log.info("Timeout occured, could not forward event!");
            this.handler.cancel();
            this.getContext().stop(getSelf());
        } else {
            this.unhandled(o);
        }
    }
}
```

Listing B.2: Akka - End-to-End Forwarder Actor

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Akka.
`http://akka.io/`. Accessed: 27.01.2014.

[3] Apache ODE - Orchestration Director Engine.
`http://ode.apache.org/`. Accessed: 27.11.2013.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin/Heidelberg, 2013. 308 pages, ISBN 978-3-642-37520-0.

[5] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.

[6] Farhad Arbab. Puff, The Magic Protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

[7] Farhad Arbab, Christel Baier, Frank De Boer, and Jan Rutten. Models and temporal logics for timed component connectors. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 198–207. IEEE, 2004.

[8] Christel Baier. Probabilistic models for reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, oct 2005. `http://www.jucs.org/jucs_11_10/probabilistic_models_for_reo`, Accessed: 22.11.2013.

[9] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[10] Egon Börger. Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl. *Software and System Modeling*, 11(3):305–318, 2012.

[11] Søren Christensen and Niels Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf*, pages 159–178. Springer.

[12] Stefan Craß. A formal model of the Extensible Virtual Shared Memory (XVSM) and its implementation in Haskell. Master's thesis, Technische Universität Wien, Institut für Computersprachen, 2010.

[13] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, pages 301–306, New York, NY, USA, 2009. ACM.

[14] Maximilian Csuk. Developing an Interactive, Visual Monitoring Software for the Peer Model Approach. Master's thesis, Technische Universität Wien, Institut für Computersprachen, 2014. in preparation.

[15] Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In *Proceedings of the 3rd international conference on Agent-oriented software engineering III*, AOSE'02, pages 59–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[18] Thomas Hamböck. Towards a Toolchain for Asynchronous Embedded Programming based on the Peer-Model. Master's thesis, Technische Universität Wien, Institut für Computersprachen, 2014. in preparation.

[19] Carl Hewitt. Actor model of computation: Scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[21] Carl Hewitt, Erik Meijer, and Clemens Szyperski. The actor model. `http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask`. Accessed: 24.01.2014.

[22] Paul Istoan. Defining Composition Operators for BPMN. In Thomas Gschwind, Flavio Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 17–34. Springer Berlin Heidelberg, 2012.

[23] Kurt Jensen. A brief introduction to coloured petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 203–208. Springer, 1997.

[24] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL Extensions for Subprocesses (BPEL-SPE). IBM Corporation and SAP AG, 2005.

[25] Christian Koehler, Farhad Arbab, and Erik de Vink. Reconfiguring distributed reo connectors. In *Recent Trends in Algebraic Development Techniques*, pages 221–235. Springer, 2009.

[26] Christian Koehler, David Costa, José Proença, and Farhad Arbab. Reconfiguration of reo connectors triggered by dataflow. *Electronic Communications of the EASST*, 10, 2008.

[27] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011.

[28] Lars M. Kristensen, Søren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[29] eva Kühn. Peer Model Tutorial. Technical report, Technische Universität Wien, E185/1, February 2012.

[30] eva Kühn. A Group based Upstream Notification Protocol. Technical report, Technische Universität Wien, E185/1, January 2013.

[31] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In *COORDINATION*, pages 121–135, 2013.

[32] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible Modeling of Policy-Driven Upstream Notification Strategies. In *29th Symposium On Applied Computing (SAC)*, Gyeongju, Korea, March 24-28 2014. ACM.

[33] eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 625–632, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[34] eva Kühn, Richard Mordinyi, and Christian Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 634–648. Springer, 2009.

[35] eva Kühn, Johannes Riemer, and Gerson Joskowicz. XVSM (eXtensible Virtual Shared Memory) Architecture and Application. Technical report, Technische Universität Wien.

[36] Olaf Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.

[37] K.G. Larsen, P. Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 76–87, 1995.

[38] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[39] Daniel Moldt and Heiko Rölke. Pattern based workflow design using reference nets. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pages 246–260, Berlin, Heidelberg, 2003. Springer-Verlag.

[40] Richard Mordinyi, eva Kühn, and Alexander Schatten. Space-based architectures as abstraction layer for distributed business applications. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 47–53. IEEE, 2010.

[41] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[42] Linda Northrop. Software product lines essentials. *Software Engineering Institute, Carnegie Mellon University*, 2008.

[43] Linda Northrop, Paul Clements, et al. A Framework for Software Product Line Practice, Version 5.0. *Software Engineering Institute, Carnegie Mellon University,* `http://www.sei.cmu.edu/productlines/frame_report/index.html`. Accessed: 07.03.2014.

[44] OASIS. UDDI Specification Version 3.0.2. `http://uddi.org/pubs/uddi_v3.htm`, October 2004. Accessed: 27.11.2013.

[45] OASIS. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, April 2007. Accessed: 25.11.2013.

[46] Object Management Group (OMG). Business Process Model and Notation (BPMN) Version 2.0.
`http://www.omg.org/spec/BPMN/2.0/`, January 2011. Accessed: 22.11.2013.

[47] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.

[48] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering*, volume 10. Springer, 2005.

[49] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[50] Dominik Rauch. PeerSpace.NET - Implementing and Evaluating the Peer Model with Focus on API Usability. Master's thesis, Technische Universität Wien, Institut für Computersprachen, 2014. in preparation.

[51] Reo - Extensible Coordination Tools. `http://reo.project.cwi.nl/reo/wiki/Tools`. Accessed: 27.11.2013.

[52] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[53] Uppaal Language Reference. `http://www.uppaal.com/index.php?sida=217&rubrik=101`. Accessed: 04.10.2013.

[54] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, ICATPN '98, pages 1–25, London, UK, 1998. Springer-Verlag.

[55] World Wide Web Consortium (W3C). SOAP Version 1.2. `http://www.w3.org/TR/soap/`, April 2007. Accessed: 27.11.2013.

[56] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0. `http://www.w3.org/TR/2007/REC-wsdl20-20070626`, June 2007. Accessed: 27.11.2013.

[57] World Wide Web Consortium (W3C). XML Schema Version 1.1. `http://www.w3.org/XML/Schema`, April 2012. Accessed: 27.11.2013.