NTNU
Innovation and Creativity

A Pattern-Based Approach for the Consistent Design of Interaction Interfaces

Sebjørn Sæther Birkeland

Master of Science in Communication Technology
Submission date: June 2006
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Frank Krämer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

When we look at a number of service specifications, we observe that there often occur similar interactions between the components executing a service. While, for example, the data transferred may vary with the specific application, certain sequences of signal transmissions between two service components may occur in several applications. This makes it possible to improve the development of systems by storing such reoccurring interactions as patterns in a library. Instead of reinventing solutions for interaction problems again and again, interaction interfaces can then be combined by reusing these patterns.

In this work, an approach should be described that allows to compose interaction interfaces in an editor from interaction patterns. For that purpose, first of all a number of interaction patterns should be identified and described. Typical error situations or design flaws in interaction interfaces should be described based on the work on role validation by Jacqueline Floch and rules should be described how these situations can be detected, avoided, or resolved when interaction patterns are composed together. To prove the effectiveness of the approach, a editor should be implemented and integrated into the integrated tool suite Ramses.

Assignment given: 16. January 2006
Supervisor: Peter Herrmann, ITEM

...A hole had just appeared in the Galaxy. It was exactly a nothingth of a second long, a nothingth of an inch wide, and quite a lot of million light years from end to end. [...] Somewhere in the deeply remote past it seriously traumatized a small random group of atoms drifting through the empty sterility of space and made them cling together in the most extraordinarily unlikely patterns. These patterns quickly learnt to copy themselves (this was part of what was so extraordinary of the patterns) and went on to cause massive trouble on every planet they drifted on to. That was how life began in the Universe...

- Douglas Adams: The Hitch Hiker's Guide to the Galaxy (1979)

Abstract

We depend more and more on networked systems in our every day lives. The functionality and correctness of such systems are thus becoming increasingly important both for individuals and the society as a whole. At the same time, people expect new services to be developed more rapidly than ever. These two forces makes the design of these often complex services a difficult task.

Model-driven techniques are in widespread use for the design of distributed services, and software tools can support the designer through the development process. Our goal is to help the user of such modelling tools making it easier to develop correct services more quickly.

When we take a closer look at existing services, we find great similarities in the way two parts of the service interact. For instance, one part might request for some information from another part or perhaps notify another part of something. We identify them as so-called interaction patterns, which describe these generic interactions at a high level of abstraction. They are modelled using UML 2.0 collaboration templates.

We have defined an approach for using a number of interaction patterns to describe the interaction interface between two parts, modelled by a UML 2.0 collaboration. The user of the tool selects, customises and applies a number of patterns to the interaction interface. The applied patterns can then be composed, meaning that the user determines their sequence of activation. The composition technique defined is based on the use of UML 2.0 state machines and submachine states. We have designed an algorithm which generates the behaviour of the interaction interface based on the applied patterns and their composition. The behaviour is expressed using two state machines which describes the allowable sequence of signals for each participant in this interaction.

To ensure the correctness of the behaviour of the interaction interface, we have identified a number of common design flaws that might occur during the composition of pattern instances based on the work by Jacqueline Floch. The error situations are either prevented through reasonable constraints on the used model elements or resolved when the behaviour of the interaction interface is generated by the defined algorithm.

The approach has been implemented as an Eclipse plug-in and integrated with Ramses, a modelling tool suite developed at the Department of Telematics. This provides the possibility to get hands-on experience with using our approach for the design of reactive services. We believe that our work can make it easier for the user of such tools to design correct services more rapidly.

Preface

This master's thesis was written at the Department of Telematics at the Norwegian University of Science and Technology (NTNU) during the spring semester 2006.

The help from my supervisor PhD Student Frank Alexander Krämer has been essential in this work, and I thank him for always being available for assistance. My acknowledgements also go to Professor Peter Hermann, the academically responsible, for his comments and suggestions through the semester. I would also like to thank Professor Rolv Bræk for valuable input in the starting phase of my work. Finally, I appreciate the help from secretary Mona Nordaune in finding the necessary background material.

I have specialised in service engineering, and this thesis concerns tool development for modelling of telecommunication services. I hope the reader will find my work interesting, and that it can be of importance for future research within this area.

Trondheim, June 2006

Sebjørn Sæther Birkeland

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The task of developing advanced telecommunication services can be quite complex. A number of distributed parts work together to realise the complete service. The behaviour of the parts are described by state machines, which communicate asynchronously with messages through a buffered medium. How these parts work together in a cooperative matter makes up the total service.

The challenge for a designer of such systems is how to make sure that the parts interact correctly. An error in the interaction between two parts of the service will propagate and compromise the correctness of the entire service. It can be a difficult job to make certain, that any two parts interact consistently, as a real-life telecommunication service often consists of a fairly large number of interacting parts.

Our goal is to make it *easier* for a designer to develop *correct* services more *quickly*. But how can we do this? We start by searching in existing services.

When we take a closer look at present services we can see clear similarities in the way the parts of the system interact. Both the intention behind the interactions between two parts, as well as the message passing itself, is very often the same. For instance, one part might ask another part about something, and the latter responds with an answer or the requested information (later identified as the REQUEST-pattern). Another example is that one part tells another part that it wants to be notified every time a specific event occurs (later named the SUBSCRIBE-pattern).

This means that on a high level of abstraction, the same things happen over and over again. This is true even for services from completely different domains and with a great variety in the range of use. These re-occurring, abstract interactions are what we call *interaction patterns*. If the total interaction, that is all the messages exchanged between two parts of a service, can be put together from these generic interaction patterns, then we can use this knowledge to ease the job of making new services. We can collect and describe these common interactions and use them as a template when specifying new interactions.

Equipped with a number of interaction patterns, we can construct the complete interaction between two parts just by instantiating, customising and combining them to suit the particular situation. And if something can be constructed by putting together already defined and well-known pieces, it is quite unnecessary to construct it from scratch every time. This will make the job easier for the designer as he only needs to assemble pre-fabricated fragments into the complete interaction. Consequently, it will also allow a service to be developed more rapidly.

Some level of correctness can also be guaranteed, given that the interaction patterns are correct and that they are assembled correctly. The designer is less likely to make mistakes when all he needs to do is to combine a number of already defined interaction patterns, in contrast to beginning from scratch.

In this work, we would like to find out how this could be supported within a tool. A user should be able to select a number of pre-existing interaction patterns, customise them to fit the specific service and determine how they should be combined to realise the complete interaction between the two parts. We can help the designer towards a correct service by restricting the number of composition possibilities, and revolve common design flaws. We claim that such an approach will make service development easier and quicker, and help making the service correct.

1.1 Overview

Figure 1.1 shows the core of this work. We will go through the process in detail, but let us first start by looking at the bottom of the figure to see what we want as result.

When two parts of a system communicate (or interact) they will always do so through what we call an *interaction interface.* We have illustrated this as an ellipse in the figure. The interaction interface can be expressed by a two-way collaboration. The behaviour of the interaction interface can be described by two so-called association point state machines (APSMs), one for each participating part, which defines the allowable sequence of messages in the interaction. The aim of the whole approach is to be able to describe the behaviour of this interaction interface, that is the APSMs, only by using a set of pre-defined interaction patterns.

Pattern definition First of all we need a collection of interaction patterns. We gather them in a so-called pattern library, as illustrated at the top of Fig. 1.1. As explained earlier, the patterns are abstract pieces of interactions that emerge from knowledge and experience drawn from existing services. Each interaction pattern is described carefully, including a detailed explanation of what it does, the messages exchanged in the pattern, and the intention behind the pattern. The interaction patterns added to the library have to be well-formed, meaning

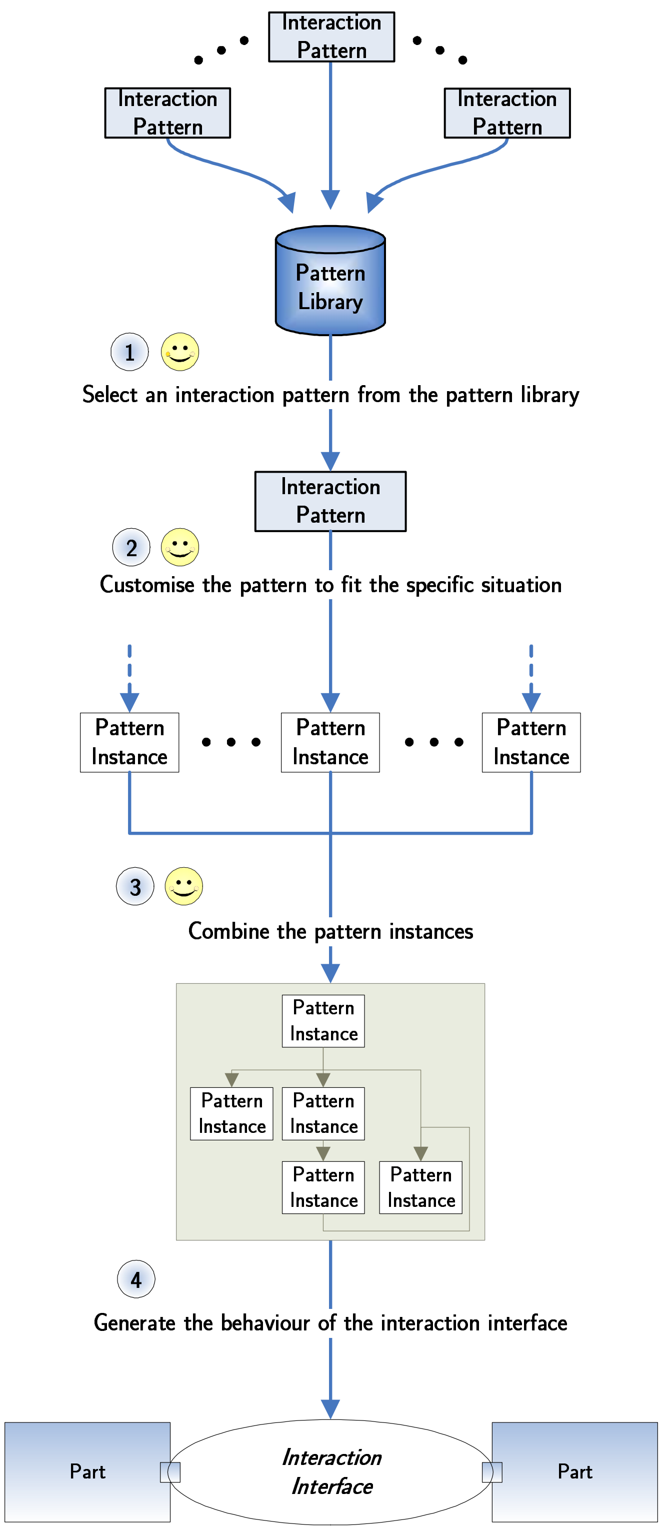


Figure 1.1: Overview of the approach. The behaviour of the interaction interface can be obtained by selecting, customising and composing a number of pre-defined interaction patterns from a library.

that they have to obey certain rules defined to ensure their correctness. This validation effort only needs to be done once, before the pattern is added into the library.

Pattern selection (step 1) The users first search through the pattern library, trying to find a suitable pattern. They should focus on what they wants the two collaborating parts to achieve, and not so much the messages needed to realise this. The users glance through the description of the interaction patterns in the library, investigate candidates more closely and select a suitable pattern.

Pattern customisation (step 2) Because the patterns are abstract and can be used in a wide variety of settings, they have to be tailored before they can be used in the concrete system. This means that the users specify the information that is to be exchanged and concretise the pattern, yielding in a so-called pattern instance. Some patterns might also have small variations which can be decided upon instantiation.

Pattern composition (step 3) After a number of patterns instances have been applied, the user can compose them. The pattern instances represent pieces of the interface behaviour we want to define. We now have to decide how these pieces are to be linked together to produce the desired overall behaviour of the interface. This means to decide what will be the next pattern instance to be activated after a pattern instance finalises with a certain result.

APSM generation (step 4) The tool can now automatically generate the behaviour of the interaction interface between the two parts based on the patterns applied and their composition. The behaviour is expressed using one state machine for each of the participants, called an APSM. The APSMs can be composed by an algorithm, without involving the users at all since the information needed has been specified by the users in the previous steps. Several error situations might occur during the composition of the APSMs, depending on how the users composed the pattern instances. These errors have to be identified and resolved to ensure correct APSMs, and thus a correct behaviour of the interaction interface.

This approach allows the users of the tool to create the behaviour of an interaction interface between two parts in a service by repeating the steps described above. Selection and customisation of new patterns can be done at any time. The composition of the pattern instances can be changed and new pattern instances can be linked together with existing ones. After a change has been made, the APSMs can be refreshed by telling the tool to generate them again. This means that if the resulting interaction interface behaviour does not

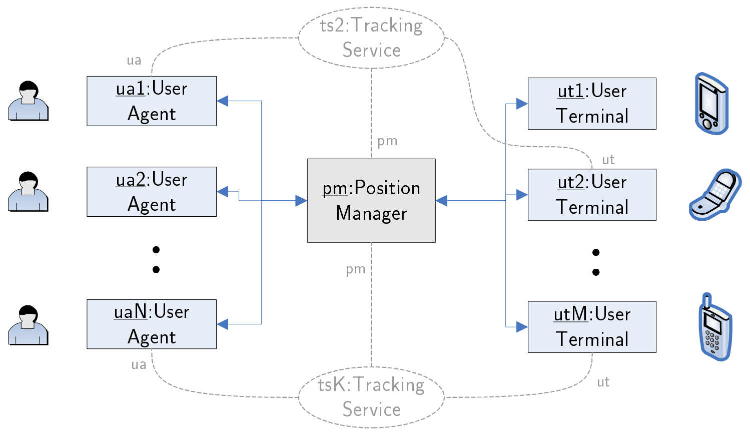


Figure 1.2: Overview of the example service. A central position manager allows user agents to track the movements of their friends' user terminals, through a service called *Tracking Service*.

reflect what the users had in mind, they can always re-compose or change the patterns applied and generate it once more.

1.2 Example service

Throughout this thesis we will illustrate our approach using a small, but still realistic, example service. Figure 1.2 shows its overall structure.

The service is named *Tracking Service* and its purpose is to enable users to track the location of their friends (buddies). This is done through a central server named the *Position Manager*. When a user wants to track the movements of a buddy, the position manager will start tracing the geographical position of the terminal belonging to that buddy. We can have many user agents and many terminal agents, but only one position manager. Note that the terminal agents are not representing the terminals of the users that are tracking (which are represented by the user agents), but they represent the terminals of the buddies that are being tracked. To keep the example small, we have deliberately omitted functionality such as managing buddy lists, relating users to terminals and authentication.

The main functionality of the system is:

- User agents can start using the service by logging on to the position manager.
- User agents can issue a request for tracking another user (a buddy).
- Provided that the request is accepted, the user agent can subscribe to the movements of a buddy and is notified every time this buddy moves.
- User agents can log off from the position manager and end the service.
- The position manager can issue a trace request to a terminal, asking if the terminal allows to be traced.

- The position manager periodically issues a request to the terminal about its whereabouts.
- The terminal can at any time during tracing ask the position manager to get a list of all the users that are currently tracing the terminal.
- The terminal can also choose to revoke the tracing allowance, and thus stop the tracing, by notifying the position manager of this.

Figure 1.3 shows the structure of this service using the notation of UML 2.0 collaborations. The notation and meaning of the collaborations will be explained in more details in the next chapter. In short, we use them to express the communication and interaction between two participants in a system, and their purpose can be understood rather intuitively. Part (a) of the figure shows that the complete tracking service actually is made up of two "sub-services". The user agent and the position manager are involved in a collaboration called *Buddy Tracking*, while the user terminals communicate with the position manager through a collaboration named *Terminal Trace*.

In part (b) and (c) of the figure, we can see that *Buddy Tracking* and *Terminal Trace* are in turn a combination of other, more fine-grained, collaborations. The tracker and the tracking server of the *Buddy Tracking*-collaboration cooperate to do the following tasks: logging in, requesting tracking, tracking a user and logging out. The trace server and the terminal in *Terminal Trace* collaborate in the following basic functions: requesting trace, location update, get trackers (users tracking this terminal) and revoking trace allowance.

The collaborations used in *Buddy Tracking* and *Terminal Trace* stem from interaction patterns, as shown in part (d) of the figure. We see that *LogIn* and *TrackRequest*, contained in *BuddyTracking*, stem from a pattern called REQUEST, *TrackUser* comes from a pattern SUBSCRIBE, while *LogOut* stems from the NOTIFY-pattern. Similar, the figure also shows which interaction patterns the collaborations used in *Terminal Trace* stem from.

The tracking service will be referred to and further elaborated throughout this thesis to illustrate our approach.

1.3 Main contributions

These are the most important contributions of our work:

- We define a way of modelling interaction patterns using UML 2.0. This is augmented with requirements for their correctness. We also identify a number of interaction patterns which comply with these requirements.
- We describe an approach for composing interaction interfaces from interaction patterns, relying on UML 2.0. This includes how to customise the interaction patterns and apply them in a concrete service. We also define a technique to compose the interaction patterns applied to an interaction interface.

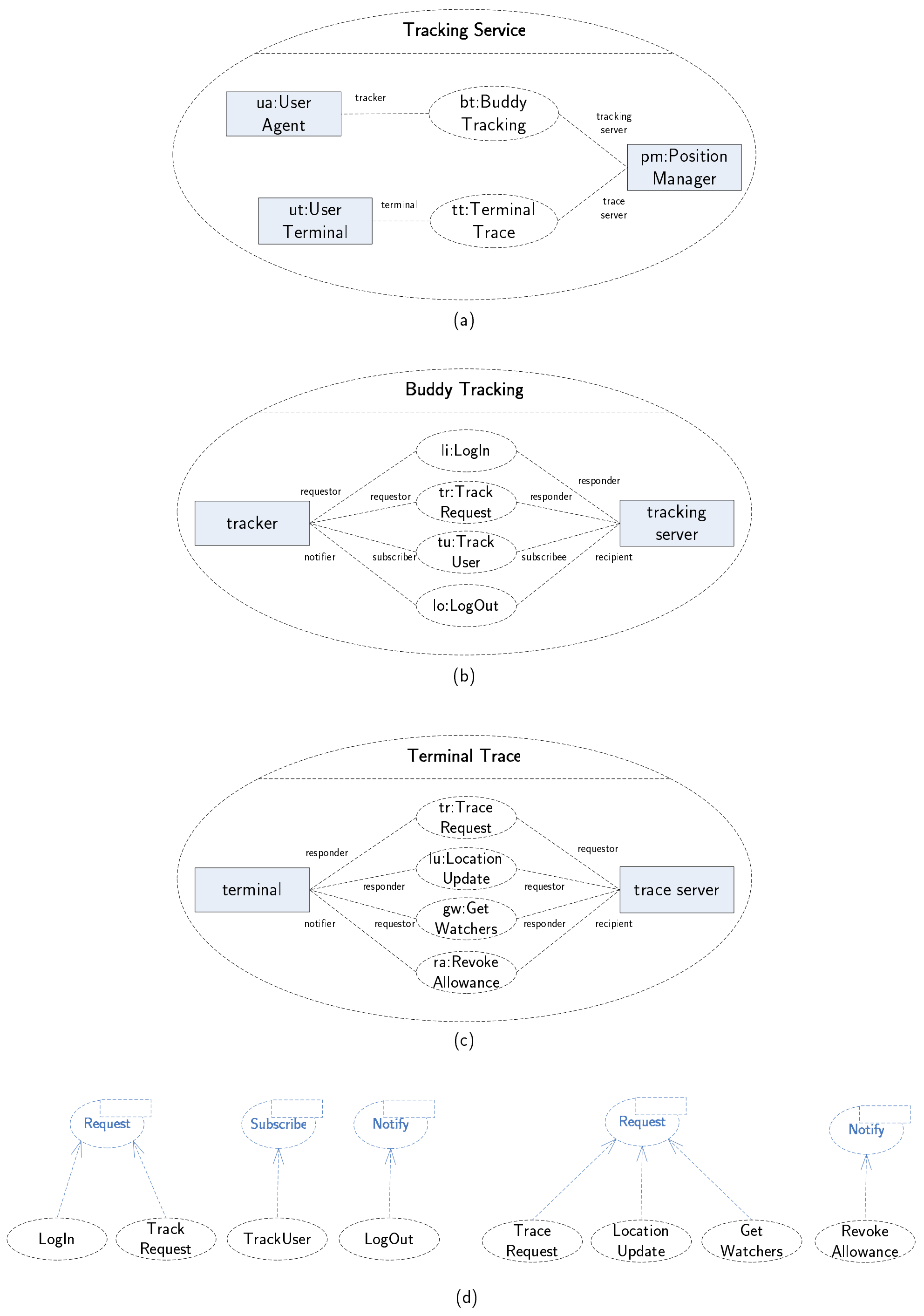


Figure 1.3: The example service expressed using collaborations. The service *Tracking Service* (a) is composed from a *Buddy Tracking* service (b) and a *Terminal Trace* service (c). These are in turn composed from more fine-grained collaborations stemming from interaction patterns (d).

- We design an algorithm which can generate the behaviour of the interaction interface based on the applied interaction patterns and their composition. During this, some typical error situations might occur, and we identify them and find reasonable ways of preventing or resolving them.
- We demonstrate the effectiveness of this approach by implementing and integrating it in the Ramses modelling tool suite. The implementation allows a user to apply and compose interaction patterns as well as generating the behaviour of the interaction interface using the algorithm defined.

1.4 Reader's guide

The background of our work will be presented in *Chapter 2*. Next, *Chapter 3* defines our interaction patterns, while *Chapter 4* explains how to apply and compose them. In *Chapter 5* we describe how to generate the APSMs of the interaction interface, and how to ensure their correctness. *Chapter 6* then explains the algorithm designed for composing these APSMs. *Chapter 7* describes our implementation of the approach. Finally, in *Chapter 8*, we discuss some aspects of our work and conclude our thesis.

Chapter 2

Background

This chapter will first present and describe the concept of patterns. Next, we will describe UML 2.0 collaborations and our underlying meta model. We will then elaborate the idea of interaction consistency, before we discuss two existing tools supporting software patterns. Finally, we give a brief introduction to the Ramses tool suite.

2.1 Patterns

There are many opinions and definitions on patterns, as the term has increased its popularity over the past decades. Before we try to define our own meaning of a pattern, let us look at some of the things others have said.

In the Oxford English Dictionary, a pattern is defined as *"[...] a model, design, or set of instructions for making something, [or] an example for others to follow. Origin from* patron *in the former sense something serving as a model, from the idea of a patron giving an example to be copied."* [41]. Alexander, which we will be acquainted with in the next section, thinks of a pattern as *"[...] something ≪in the world≫ [...] which repeats itself over and over again, in any given place, always appearing each time in a slightly different manifestation"* [2, p. 181].

Patterns help us to *"[...] write down good ideas that have solved real problems. The pattern form captures enough information about this solution so that when you read it, you can use the solution to solve your problems"* [45, p. 5]. In other words, a pattern never describes something new or innovative. As J.O. Coplien explains, a pattern *"[...] captures a solution with a track record, not theories or speculation"* [9]. Coplien is one of the leading persons in the pattern community, which has *"[...] a unique culture that values stating the obvious and recording known practices over inventing novel, possibly untested, solutions"* [35].

Patterns in software design *"[...] describes the core structure of a solution at* *a level high enough to generalize to many specific situations. [...] Patterns supplement general-purpose design by capturing expert solutions in a form that helps developers solve difficult, recurring problems"* [7]. We thus try to capture the best practices within a field using the pattern form because they *"[...] help create a shared language for communicating insight and experience about these problems and their solutions"* [24].

A pattern is not universally applicable. In the words of Alexander: *"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"* [2, p. 247]. This means that the pattern only provides a good solution (and is often only meaningful) within a specific context.

Now, trying to compress all this into one sentence, we end up with our explanation of what a pattern is: A pattern describes a useful and proven solution to a reoccurring problem within a specific context, and this solution is abstract and generic enough to be used over and over again. It focuses on solving "real life"-problems and is not theoretical or experimental. A pattern must have a certain structure making it easy to read and communicate to others. In short, and a bit superficial, one might say that a pattern is just a structured way of describing and storing best-practices within a domain.

2.1.1 The origin of patterns

The concept of patterns originates from the work of the architect and contractor Christopher Alexander. Alexander searched for, collected and described common solutions to the recurring problems architects were faced with when creating structures like walls, houses, town squares, parks, and neighbourhoods. This work resulted in two books, namely *A pattern language* [3] and *The timeless way of building* [2], published in 1977 and 1979 respectively. His idea was that there was a way, regardless of time, to build houses and other liveable structures so that people feel alive and home [2, p. 7]. By following this "timeless way", the houses and towns created will be appealing to the people living there. He talks about "a whole" and the "quality without a name" - that there is something you cannot name precisely, but what differs a good building from a bad one [2, p. 25]. There is a need for some heuristics for making great towns and beautiful houses. We can reach, or come nearer, this goal by documentation of the great existing buildings and the years of experience of professionals.

Alexander's explanation of a pattern is that it "*[...] describes a problem which occurs over and over again in our environment, and then describes a core solution to that problem, in such a way that you can reuse this solution a million times over, without ever doing it the same way twice*" [3, p. x]. He also stresses that patterns should have a clear structure to ease the job of looking through many patterns to find the one that solves your problem. A pattern must define the context and the problem it solves, contain an illustration or example, and explain and justify the solution provided.

Even though our domain is far from architecture and town planning, the basic concepts and ideas from Alexander's work are of use within software design as well. As with buildings, there is undoubtedly something that separates good software design from bad - and the more experienced software designers knows the difference.

2.1.2 Patterns in software

Cunningham and Beck were the first to use Alexander's thoughts on patterns in software when they, in 1987, presented a small pattern language for linear programming at the OOPSLA[1]-workshop [5]. This inspired others to collect the best-practices in their software domains, but it was a book published seven years later that still stand out as *the* book on software patterns.

The "Gang of Four"

The book *Design Patterns* [18] was published in 1994, and serves as both an introduction to and a catalogue of design patterns in object-oriented software design. This was the first time a large number of design patterns were collected and presented this extensively and the work by these authors, nicknamed the "Gang of Four", gave attention to the usage of patterns in software engineering.

The goal was to catalogue and document design solutions that was used by experienced object-oriented designers, which had evolved over time and proven to be the best way to solve a common design problem [18, p. 351]. The collection, with a total of 23 design patterns, provided reusable solutions to the recurring problems object-oriented designers often are faced with. Novice programmers can teach from the experiences of professionals and use the patterns to find flexible and good solutions to their design problems.

Design patterns are explained as "*[...] descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*"[18, p. 3]. This means that the patterns involve several objects and focuses on how to relate and arrange them to get the wanted result. General solutions are defined by utilising well-known object-oriented techniques like inheritance and polymorphism.

The patterns include careful descriptions of the solution. Class or sequence diagrams are important to understand the structure of the pattern, so is an explanation of how they collaborate. Example usage and code snips also amount to a large part of the pattern description. Once a suitable pattern has been selected, the solutions, of course, have to be implemented in software code

[1]Object-Oriented Programming: Systems, Languages and Applications [40]

by the designer. Detailed knowledge about the solution and implementation specific explanations is therefore a necessity.

Some other software patterns

After the great popularity of the design patterns from the "Gang of Four", the concept began spreading throughout the software community. The patterns and pattern languages existing are far too many to elaborate, nor mention, here. We will try to sketch a few examples to illustrate the diffusion of patterns to various software areas.

Buschmann and his colleagues described a large collection of patterns for software architecture in their book *Pattern-Oriented Software Architecture* (popularly called "POSA") from 1996 [8]. Patterns on different levels of abstraction are presented, all the way from high-level architectural patterns to design patterns and implementation-specific low-level patterns called idioms. The intention was to describe and document large-scale applications, for instance operating systems, through the use of these patterns with different abstraction and focus [8, p. xi].

Patterns have also found its way into the world of Java programming and Grand has captured design patterns for developing distributed and enterprise applications [22]. The work includes patterns for transaction (such as ACID and two-phase commit), distributed architecture and computing (such as object request broker, registry and proxy), and concurrence (such as threads and locking files). This yields in an extensive collection, where everything is exemplified thoroughly in Java code, making it easy to apply the patterns to concrete applications. Kuchana has done something similar, documenting 42 design patterns for Java, including "Java-versions" of the GoF-patterns [32]. Both authors use UML to model their patterns, which makes it easy to understand for readers acquainted with this notation.

Software patterns has become so widespread and popular that a non-profit community for software patterns, called the Hillside Group, has been established [24]. This community encourages the use of patterns for describing software best practices and also sponsors many conferences such as PLoP, EuroPLoP, ChiliPLoP, VikingPLoP and so on[2]. The conferences are workshops where people can learn about patterns and discuss and improve their ideas for new patterns.

Telecom patterns

At the ChiliPLoP in 1999, one of the topics was called TelePLoP, where experienced telecommunications software practitioners gathered to present and

[2] *PLoP* is of course an acronym and stands for Pattern Languages of Programming. More information about the different conferences can be found at the Hillside Group website [24].

discuss telecom patterns [11]. This resulted in a large collection of patterns related to communications software [45]. Some are distribution patterns that deal with the placement and replication of object throughout the distributed network. The pattern *Half Object Plus Protocol (HOPP)* [35, p. 28] is an example, which is used to replicate an object in each process of the network that requires real-time access to it, and making sure that these replicas are synchronised. The call object in a normal telephony call is replicated like this in every switch, and state changes in the call are propagated through the network to synchronise them. Other sets of patterns concerns the capacity [36, p. 63] and fault-tolerance [1, p. 81] of real-time reactive systems, the interface between the systems and humans [23, p. 95], as well as patterns on how to design distributed systems in general [10, p. 41]. As we understand, all proven solutions which we have become familiar with in the telecommunications domain can, of course, be documented as patterns. However, these patterns mainly capture issues regarding distribution and the network itself, and not the modelling of services as we are concerned.

SDL [30] is a modelling language for real-time systems, such as telecommunication services, and patterns have been presented to being reuse to SDL-based development by Geppert and Rössler [19]. Various SDL-patterns have been presented, from protocol related patterns (interactions and local management) to basic service related patterns (for instance error handling) and architectural patterns (for instance subsystems) [19]. The interaction patterns identified range from the simple "Asynchronous Notification" [21] to more complicated patterns including timers, such as "Watchdog' and "Heartbeat" [54]. The patterns can be used to construct the internal structure and behaviour of the components in an SDL-based system.

As we understand, software patterns are mainly about how to relate objects, and how they communicate or collaborate in order to do a desired task or provide a certain functionality. We will now present a way to describe this collaborative nature of software systems.

2.2 UML 2.0 collaborations

A UML 2.0 collaboration focuses on the interactions and relationships between parts of a system and describes how they work together to achieve some useful purpose or functionality [46, p. 227]. This fits well with our needs of expressing the co-operation and interaction between the parts in our services. A collaboration allows us to do exactly this without having to specify or worry about the internal behaviour of the parts themselves [39, p. 157]. The focus is not on what the parts can do in isolation, but what two or more parts can do together when they interact.

The participants in a collaboration are described using so-called collaboration

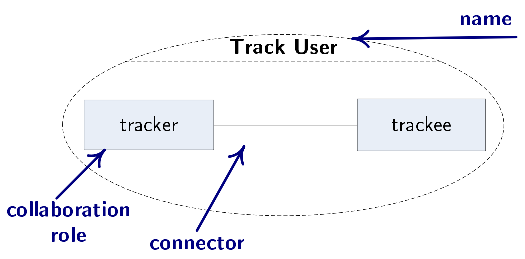


Figure 2.1: Graphical notation for a collaboration. The collaboration named *Track User* has two collaboration roles, the *tracker* and the *trackee*. They are related through a connector.

roles. In the end, the task described by the collaboration will be performed by parts in the system by exchanging signals. But, when designing the system we want to focus on the behaviour needed to realise the functionality of the collaboration, and not the behaviour of the actual parts. Using collaboration roles we dismantle the definition of a collaboration and its behaviour, from the task of determining which parts of the concrete system that are going to play each of the collaboration roles.

A collaboration is both a structural and a behavioural classifier. We can express how the collaboration roles relate to each other using so-called connectors. Two roles are connected by a connector if they communicate. This describes the structure of the collaboration and this structure is only valid within the context of the collaboration. In addition, the collaboration can contain behaviour specifications and one of these can describe the behaviour of the collaboration itself [39, p. 419]. The behaviours can be described by activities, interactions (shown as sequence diagrams or communication diagrams) or state machines [46, p. 190].

Figure 2.1 shows an example of a collaboration called *Track User*. This collaboration has two roles, namely the *tracker* and the *trackee*. They interact because there is a connector between them. We see that this notation describes the structural aspects good enough. However, it tells us nothing about the behaviour of the collaboration, which we will return to in section 2.3.2 on page 17.

2.2.1 Collaboration uses

A collaboration can be used by binding the collaboration roles to either concrete parts in the service or to other roles within the definition of a larger collaboration [46, p. 232]. The latter means that we can create a collaboration by using other collaborations. With this technique we can express collaborations on different layers of granularity and compose high-level collaborations from minor collaborations [46, p. 229].

The collaborations used are defined independently and only related to each

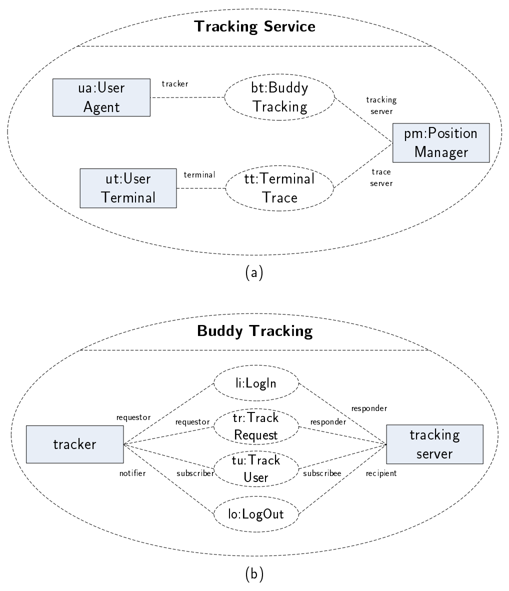


Figure 2.2: Using collaborations. The collaboration *Tracking Service* (a) uses two collaborations: *Buddy Tracking* and *Terminal Trace*. *Buddy Tracking* is in turn constructed by using collaborations with more basic functionality (b).

other within the context of the enclosing collaboration, the minute we decide to bind the roles. This property is of great importance for our work, as it provides a flexible way to compose and combine collaborations at different levels of abstraction and granularity. Figure 2.2 taken from our example service, illustrates this. In part (a) of the figure, the collaborations *Buddy Tracking* and *Terminal Trace* have been used in the larger collaboration *Tracking Service*. For instance, the role *tracker* of the collaboration *Buddy Tracking* is played by the role *User Agent* in *Tracking Service*. Part (b) of the figure shows that *Buddy Tracking* is in turn composed from several other collaboration uses. This illustrates how we can compose collaborations at different levels, from more basic to complex functionality.

Typically, the parameters are classifiers that represent attribute types, but they can also be integers or even operations

2.2.2 Collaboration templates

A collaboration template describes a collaboration with at least one unbound parameter [46, p. 638]. By specifying different values for these parameters, we can derive several collaborations from one collaboration template. The collaboration template is thus an abstract description of a collaboration, and we can reuse it by binding the parameters to different values. A collaboration template is a parametrized collaboration, and is what is called a *pattern* in UML 2.0 [39, p. 164, 615].

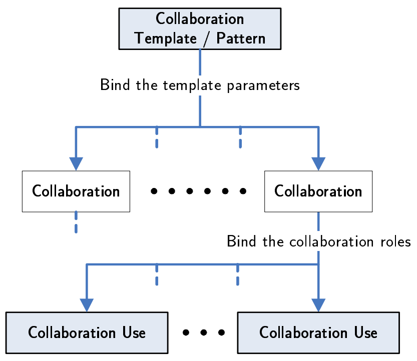

Collaboration Template / Pattern
Bind the template parameters
Collaboration
Collaboration
Bind the collaboration roles
Collaboration Use
Collaboration Use

Figure 2.3: From collaboration template to collaboration use. A collaboration template (a pattern) can produce many collaborations by binding its parameters, which in turn can be used many times by binding the collaboration roles.

Figure 2.3 shows how collaboration templates, collaborations and collaboration uses are related. One collaboration template (or pattern) can give many collaborations by binding parameters. One collaboration can be used many times by binding the roles.

2.3 The underlying meta model

In this work, we will make use of a restricted form of the UML 2.0 meta model which is used in the PATS-Lab[3] at NTNU. It is domain-specific for reactive systems in general and telecommunication services in particular. This meta model is described in a document called *PAX Ramses* [31], and the tool suite Ramses (see section 2.6 on page 28), in which we are to integrate our approach, comply with this model.

2.3.1 Collaborations

We introduce a terminology for collaborations. A *two-way collaboration* has exactly two roles, while a *multi-way collaboration* has three or more roles [31]. An *elementary collaboration* does not contain collaboration uses, while a *composite collaboration* is constructed from collaboration uses referring to other collaborations [31].

An elementary collaboration contains one state machine for each collaboration role as owned behaviours [31]. These are APSMs (explained below), and each of the collaboration roles are typed with one of them. This means that, for

[3]The Program for Advanced Telecom Services (PATS), also known as the “Tele-service Lab”, is a research agreement between NTNU and commercial partners, and provides an environment for experimenting with the development of advanced telecommunication services [42, 44]

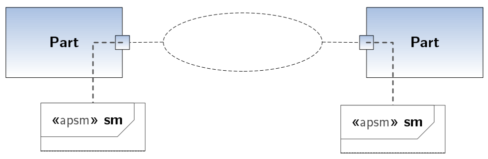


Figure 2.4: Describing the interaction interface using a collaboration. The interaction interface is expressed by a two-way collaboration with two valid APSMs.

instance, a two-way elementary collaboration always has two collaboration roles and one APSM for each of them.

2.3.2 Association point state machines

An association point state machine (APSM) is a UML 2.0 state machine with certain constraints, which describes the allowable sequence of signals to and from one state machine during the interaction with another state machine. Two communicating state machines will thus have one APSM each which describes the valid order of the messages exchanged. The APSM expresses the externally visible behaviour of a state machine in an interaction [6, p. 13]. They originate from the projected association roles (a-roles) used for SDL [30] by Floch [16]. An APSM has the following constraints [31]:

- It has exactly one initial state and the initial transition has neither a signal trigger or a send signal action.
- It can have one or more final states.
- All states, besides the initial and final states, are simple states.
- All transitions, besides the initial transition, have either a send signal action or a signal trigger.

2.3.3 Interaction interfaces

Interaction interfaces describe a valid interface between two parts of a service. The communication between the two parts is described using two compatible APSMs, one for each side of the interaction. We say the APSMs are compatible if they interact consistently, as defined by Floch [16]. This will be discussed shortly.

A two-way collaboration can be used to model an interaction interface. The participants in the interaction (the parts) are represented by the collaboration roles, and they each have an APSM describing their behaviour. Figure 2.4 illustrates this.

The interaction interface is related to the concept of *semantic interfaces* [47, 48]. They both describe the interaction between two parts of a service using a two-way collaboration. The semantic interfaces include the role behaviour of the

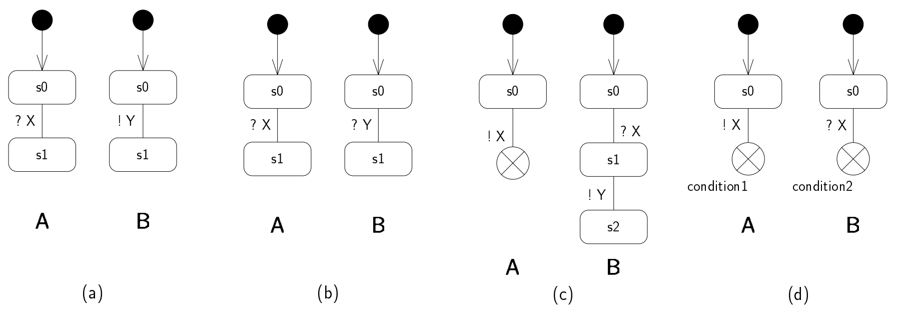


Figure 2.5: Violation of the safety properties. Two APSMs cannot interact consistently unless unspecified signal reception (a), deadlock (b) and improper termination (c)(d) are prevented.

participants in this interaction, just like the interaction interfaces have their APSMs. However, in contrast to the interaction interface, the semantic interface also incorporates the liveness of the interaction using progress goals [47, p. 88]. The interaction interface only encompasses safety properties.

2.4 Interaction consistency

Two APSMs interact consistently if unspecified signal reception, deadlock or improper termination never occurs when they interact [16, p. 78]. These are safety properties, and we can prevent "bad things" from happening if these situations are avoided. We can ensure this by applying the rules of containment and obligation [16, p. 211].

2.4.1 What can go wrong?

Let us take a quick look at the three situations which can hamper the consistency of an interaction between two parts of a service.

Unspecified signal reception Unspecified signal reception happens when an APSM receives a signal which it cannot handle in the current state [16, p. 77]. A simple example of this is shown in part (a) of Figure 2.5. A and B are interacting APSMs, both starting in the initial state. Reaching state *s0*, we see that A expects signal X to arrive, while B can only send signal Y. We thus say that A has an unspecified signal reception in state *s0*, and that the two APSMs do not interact consistently.

Deadlock Deadlock occurs when both APSMs are in a state where only signal reception is defined [16, p. 78]. That is, they both expect the other one to send a signal and cannot send a signal themselves. In part (b) of Fig. 2.5, in state *s0*, we see that A waits for B to send signal Y, while B waits for A to send X. Since none of the two APSMs can send a signal in the current state, the interaction can never go further.

Improper termination Improper termination happens when one of the two interacting APSMs have terminated and the other one has not [16, p. 78]. We do not want to have a situation where one APSM does not know if the APSM it interacts with has terminated or not. No signal should ever be sent to a terminated APSM. In part (c) of Fig. 2.5 we see that when A sends signal *X* in state *s0* it terminates. After receiving signal *X*, B proceeds to state *s1* and it will sometime later send signal *Y*. A and B does not interact consistently because A terminates while B still thinks it is alive and tries to send a signal to it.

A variant of improper termination can occur when we have conditions or labels attached to the termination of an APSM, and the two APSMs do not terminate with the same condition [16, p. 78]. Part (d) of Fig. 2.5 shows a quite trivial example of this. Both A and B terminate at the same time, but with different conditions. To be consistent, they should have terminate with equal conditions.

2.4.2 How can we prevent it?

As mentioned, the two APSMs can only interaction consistently if they are related by containment and obligation [16, p. 214]. If we can guarantee both, unspecified signal reception, deadlock or improper termination will never occur.

Containment A containment relation exists if any of the two APSMs, in interaction step, can receive all signals that can be sent from the other APSM [16, p. 212]. An APSM can be able to receive more signals, as long as it contains the reception of the signals specified by the other APSM. Containment ensures that unspecified signal reception never occurs. The containment requirement is exemplified in Figure 2.6. In state *s0*, A can receive both signal *X* and *Y*. B can send signal *X* in the corresponding state, and thus A can handle all signals B can send. Similar, in state *s1* in B, both the reception of signal *Z* and *W* is specified. At this stage in the interaction, A can send signal *Z*, and the input behaviour of B contains the output behaviour of A. We can therefore confirm that A and B are related through containment.

Obligation Obligation exists between two APSM when, in each interaction step, at least one of the two APSMs can send a signal [16, p. 213]. This means that at any time during the interaction, one of the APSMs will eventually send a signal to the other one, and this prevents deadlock. The example in Fig. 2.6 illustrates this property as well. We see that in each step in the interaction between A and B, one of them is able to send a signal to the other. In state *s0* B will eventually send signal *X* to A and they will both proceed to state *s1*.

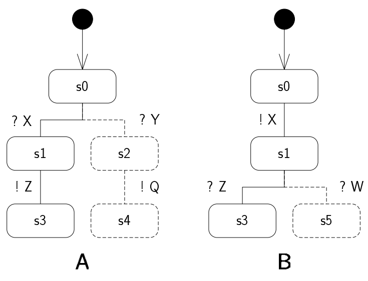


Figure 2.6: Containment and obligation. The APSMs A and B are related both through containment and obligation.

In this state, A will at some time send signal Z to B. We can conclude that A and B are related through obligation.

Containment prevents unspecified signal reception, obligation prevents deadlocks and together they prevent improper termination. To explain the latter, assume that one of the APSMs in an interaction terminates. Then the active APSM cannot send any signal, or else the containment requirement is not enforced because the terminated APSM cannot receive it. The active APSM can neither wait for a signal to arrive, as this would violate the obligation requirement because the terminated APSM cannot send a signal. The two APSMs must necessary terminate coordinated if both requirements are enforced.

2.4.3 Cases requiring special attention

The examples of inconsistent interaction so far have been of the more trivial kind. When so-called equivoque states or mixed initiatives comes into play, it becomes a bit more difficult. These are situations that have to be taken special care of to be able to ensure consistent interaction.

Equivoque states

A state is equivoque if two or more outgoing transitions are triggered by the same signal reception or signal sending and the transitions lead to non-equivalent states [16, p. 126]. This means that when an APSM is in a state and, for instance, a signal arrives, the further behaviour can be one of possibly many alternatives. This leads to ambiguous behaviour, since it not visible for the other APSM which behaviour that was executed. In all the cases in Figure 2.7 we can see that when signal X is received in state *s0*, it is not uniquely defined what will happen next. For instance, in part (a) of the figure, the APSM will proceed to either state *s1* or *s2*. The APSM it interacts with cannot know which of these alternatives that was executed, and will not know if the APSM in the figure expect a signal Z to arrive or if it is going to send signal Y.

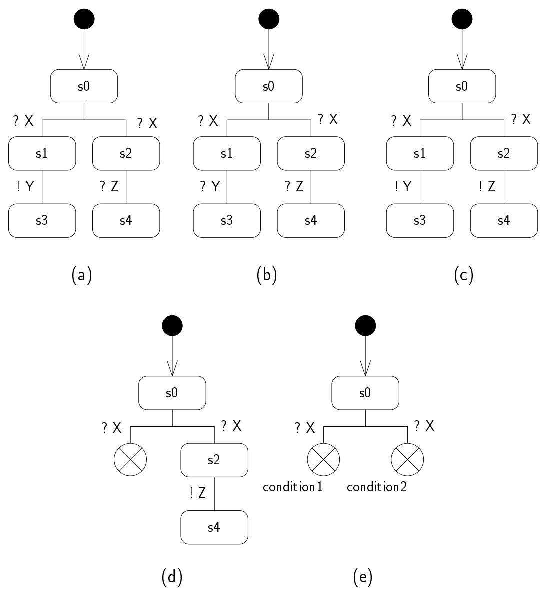


Figure 2.7: Equivoque states and resulting ambiguities. Equivoque states can result in mixed ambiguity (a), input ambiguity (b), output divergence (c), termination ambiguity (d) or termination condition ambiguity (e).

Ambiguities
The sort of ambiguity is characterised depending on the further behaviour succeeding the equivoque state. Related to Fig. 2.7 we see that in (a) we have a *mixed ambiguity*, since both signal sending and signal reception is possible after the equivoque state *s0*, but an external observer cannot know which signals the APSM expects to receive or send. Part (b) shows *input ambiguity*, since we know that the APSM will send a signal, but we do not know which one. In (c) we see a case of *output divergence*, as we know that only signal sending can occur, but we do not know which signal that will be sent. The situation illustrated in (d) shows *termination ambiguity*, where an external observer cannot know if this APSM has terminated or not as a response to the reception of signal X in state *s0*. Finally, in (e) we have *termination condition ambiguity*, because we know that the APSM has terminated, but we do not know the condition attached.

Mixed initiatives

Mixed initiative occurs when, at a certain step in the interaction between two APSMs, both can take the initiative to send a signal [16, p. 178]. The signals may cross, because we assume asynchronous signal transfer with a buffered transmission medium. Mixed initiatives might lead to unspecified signal reception or deadlock.

Mixed initiatives may describe concurrent behaviours or alternative orderings of input and output [16, p. 134]. We will describe each case in more detail.

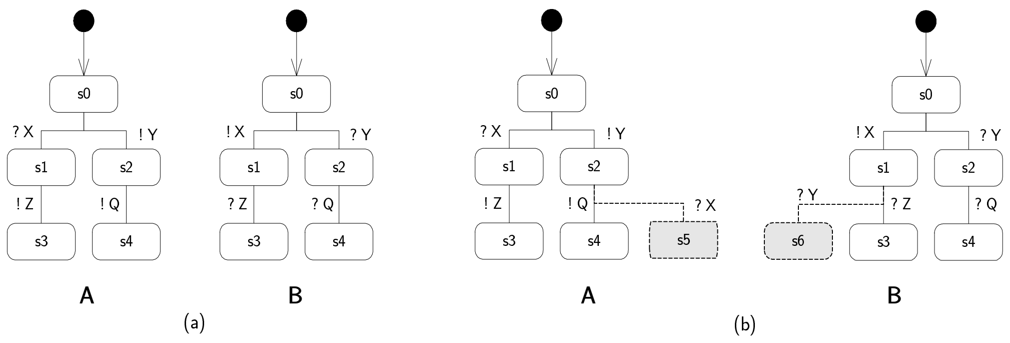


Figure 2.8: Mixed initiative describing concurrent behaviours: Input consistency. Concurrent behaviours can lead to unspecified signal reception (a) and must be handled to ensure input consistency (b).

Concurrent behaviours The mixed initiative describes concurrent behaviours when several "competing" behaviours might be started at some step in the interaction, and both APSMs are allowed to start at least one of these behaviours [16, p. 134].

Taking a look at part (a) of Figure 2.8, we see that the state *s0* is a mixed initiative state. It represents concurrent behaviours. The APSM A might decide to send Y to start one behaviour, while the APSM it communicates with, B, might send X to invoke another behaviour. If these two signals cross, A will go to state *s1* when it receives signal X, while B will proceed to state *s2* upon receiving signal Y. Consequently, A will sometime later send signal Z, but B resides in state *s2* and only waits for signal Q to arrive. This mixed initiative leads to unspecified signal reception.

Input inconsistency
To avoid unspecified signal reception in the case of concurrent behaviours, we must make sure that the signals that can be received in the mixed initiative state also can be received in all states following the sending of a signal in the mixed initiative state [16, p. 178]. If this is not the case, then we have input inconsistency. To make the interaction in part (a) of Fig. 2.8 input consistent, the reception of signal X should be added to state *s2* of A, and the reception of signal Y should be added to state *s1* in B. This is shown in part (b) of the figure. This action handles the input inconsistency caused by the mixed initiative in state *s0*.

Fixing input inconsistency may in fact also introduce new mixed initiative states. If we return to APSM A in part (b) of Fig. 2.8, we see that we introduced a new mixed initiative in state *s2*, when we added the reception of signal X to this state. Given that the states *s4* and *s5* are non-equivalent states, we are now faced with another input inconsistency. This "domino-effect" occurs because of the successive signal sendings defined in APSM A. After it has sent Y, it will go to state *s2* and later send signal Q. It thus sends two signals in sequence before it receives a signal from the communicating APSM. This can lead to quite complex APSMs, and it is advised to avoid this by ensuring

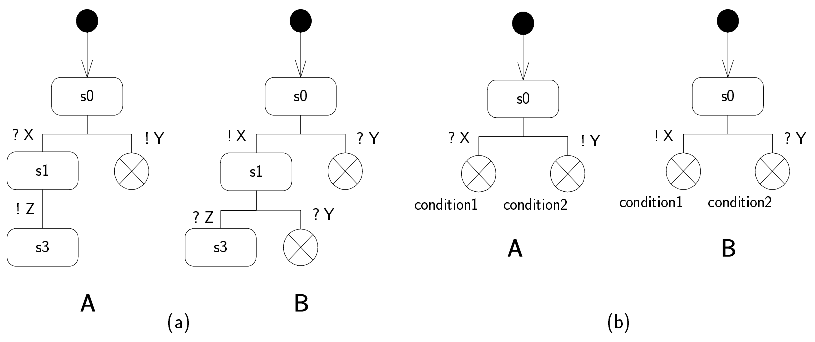


Figure 2.9: Mixed initiative describing concurrent behaviours: Termination. Concurrent behaviours can lead to unwanted termination (a) or improper termination (b) when the APSMs are allowed to terminate directly after a mixed initiative state.

that signal reception and sending occurs alternatively in the case of concurrent behaviours [16, p. 184]. If this is followed, such signal sending sequences will not happen, and we will never introduce any new mixed initiative states as in (b) in the figure.

Unwanted or improper termination
As discussed, the way to handle input consistency is to add signal reception to the states succeeding signal sending in the mixed initiative state. But, if the APSM terminates after sending a signal in a mixed initiative state, this “design rule” cannot be followed [16, p. 188].

Figure 2.9 shows, in part (a), a situation where APSM A will terminate after sending signal *Y* in state *s0*. We have added the reception of signal *Y* in state *s1* in B to avoid input inconsistency. APSM B should also terminate upon receiving *Y*, both in state *s0* and *s1*, to avoid improper termination. If the signals *X* and *Y* cross, signal *X* will never be received by A and thus lost. They are still consistent, as our definition of the concept does not take signal loss into account [16, p. 188]. Still, one should aim at designing the APSMs in such a way that signal loss does not happen because of mixed initiatives, and we can denote this as unwanted, but not improper, termination.

Improper termination can occur if mixed initiatives may lead to termination with different conditions [16, p. 189]. Part (b) of Fig. 2.9 illustrates this scenario. In state *s0* of the interaction between A and B, the signals *X* and *Y* can cross. A will terminate with *condition2* after sending *Y*, while B will terminate with *condition1* after sending *X*. As a consequence, the two APSMs do not interact consistently.

To avoid both unwanted and improper termination, we can impose a restriction saying that an APSM should not terminate directly after a mixed initiative [16, p. 190].

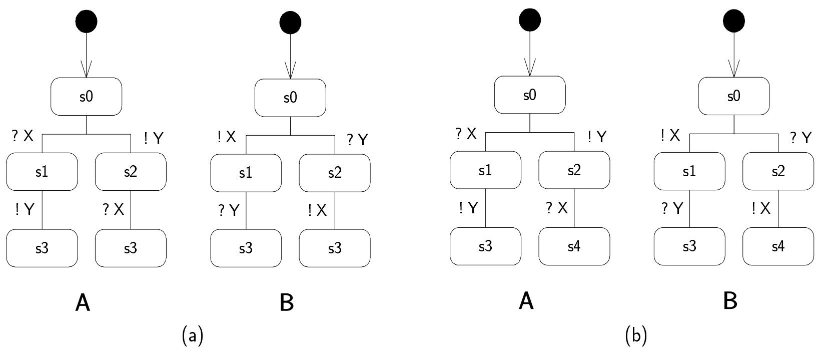


Figure 2.10: Mixed initiative describing alternative orderings. Alternative orderings have to end in equivalent states (a) or else they introduce ambiguities (b).

Alternative orderings If the mixed initiative represents an alternative ordering, it means that a signal may be sent indifferently before or after the reception of another signal [16, p. 134]. In part (a) of Figure 2.10 we see an example of this. In APSM A the sending of Y and the reception of X represent alternative orderings, and the other way around in APSM B. If A receives signal X in state *s0*, it will answer by sending Y. On the other hand, if it sends Y it will wait for the reception of X in state *s2*. It does not matter whether A and B perceives the input and output of the signals in the same sequence or not. Both sequences lead to a common state, *s3*, and the alternative ordering is handled properly.

Further behaviour ambiguity
The main task in the case of alternative orderings is to ensure that we eventually reach a common state. In part (b) of Fig. 2.10 we have an example of the contrary, where the alternative ordering leads to non-equivalent states. When APSM A sends signal Y, it is not able to know the ordering in B. If B sends X before it receives Y, A will receive X in *s2* and proceed to *s4*. B, on the other hand, continues to state *s1* after sending X, and upon receiving Y it proceeds to *s3*. A and B are now in different states. If the signals do not cross, the behaviour shown will not lead to an error, but since there is always the possibility of this happening, the further behaviour is not predictable. This thus represents an ambiguity.

Alternative orderings can involve more than two signals, but eventually they have to reach the same state [16, p. 186]. It is however advised not to use multiple alternative orderings, as this makes the APSMs more complex and it becomes more difficult to identify and handle the alternative orderings correctly [16, p. 186].

2.5 Related software pattern tools

We will now make a leap from the details of interaction consistency, and take a look at two existing software tools supporting the application of patterns. First we describe a commercial product from IBM, and then a research effort from a university supporting patterns for SDL.

2.5.1 IBM Rational Software Architect

IBM has released a collection of products called the IBM Rational Software Development Platform, targeted at helping developers with requirements analysis, architecture, design and deployment of software systems [26]. One of the products in this portfolio is Rational Software Architect (RSA), a construction tool for developers enabling model-driven development and code-generation [27]. RSA is build on top of the Eclipse platform and provides graphical modelling using UML 2.0. A more light-weight variant of RSA is the Rational Software Modeller (RSM), intended for users that do not need to generate or view the code. RSM thus supports the modelling and visualisation aspects and is intended to be used by designers to define and communicate their designs to the stakeholders [28]. Both RSM and RSA support the application of design patterns.

Design Patterns The design patterns included are the patterns by the "Gang of Four", as explained in section 2.1.2 on page 11. The patterns are presented in an own view called *Pattern Explorer*, and readers of the book will immediately recognize the structure, names and explanations provided. These patterns are concerned about classes and relationships between classes. Even though RSA supports modelling of many various UML 2.0 diagrams, the only patterns included are these design patterns which are restricted to class diagrams.

A pattern can be one of three types: a class, a package or a collaboration. In the latter case, the pattern is modelled as a collaboration between the classes involved in the pattern. All the design patterns are provided as collaborations. RSA supports another view, *Pattern Authoring*, which enables the users to create and customise their own patterns, and these can be of any of the three types.

Pattern application A pattern can be applied using a wizard consisting of two steps. The first step is to select a model element (normally a package) as the collaboration instance target. The second step is to specify the pattern parameter values. This means to choose or create the classes, interfaces and operations needed for applying each pattern. The multiplicity of and relationships between the classes can also be specified here. Applying the pattern will result in that the collaboration instance (that is the pattern instance) is added to the model, and it can then be viewed graphically in a diagram. The parameters of a pattern instance can also be changed afterwards. RSA can generate code (Java, C++ and EJB) from classes, interfaces and packages. Because the applied patterns are part of the class diagrams, the code generated will reflect the applied design patterns.

Pattern implementation Looking at how the patterns are implemented in the tool, we see that the patterns are structured as Eclipse plug-ins, packaged as so-called Reusable Asset Specifications (RAS). The RAS is a way to archive, search for, organise, document and share pattern assets [51, p. 81]. Each plug-in can be a collection of patterns, denoted as a pattern library. The library consists of a manifest file (in XML) and a Java class. For each pattern added to the pattern library a new manifest and Java class will be added to represent that pattern. The classes should not be edited, as they must be kept synchronised with the manifest files.

Difference to our interaction patterns Applying patterns to classes consists mainly of adding parameters and operations, as well as specifying the relationships between them by implementing interfaces, extending classes and instantiating parameters. Doing so will normally not affect the existing code (except when the chosen names collide with existing ones). A class only defines a list of operations, but it does not put any restrictions on the order they can or should be executed. On the contrary, state machines rigidly determine the sequence in which events can happen. This makes application of patterns to services based on state machines quite different from the object-oriented design patterns in RSA.

Embedding a pattern instance in a state diagram requires resolving more parameters than when applying to a class diagram. Any element added to a state diagram will influence the overall behaviour of the state machine. We must decide where in the state machine the elements of the pattern instance are to be embedded, identifying surrounding elements. The design patterns in RSA do not need this specification, as elements can be added to a without destroying the existing behaviour.

Concluding remarks Despite the somewhat more complex task of applying patterns to state machines, looking at the tool can still be beneficial for us. The way the pattern explorer is made, provides a intuitive way of learning about and selecting patterns. The wizard for applying a pattern is simple, but effective, and could be used as an inspiration when creating the user interface for application of our interaction patterns.

2.5.2 SDL Pattern Tool

The computer networks group at the University of Kaiserslautern is developing tool support for their SDL patterns[4], an effort called the SDL Pattern Tool (SPT) [19]. SPT is integrated with Telelogic Tau Developer Generation 2 (TTD G2), a developer tool suite mainly targeted at UML 2.0, but also with some support for SDL [52].

Patterns supported The SPT supports a sub-set of the SDL patterns defined. In 2005, when the latest paper on SPT was published, the structural pattern "Service Architecture", and the interaction patterns "Asynchronous Notification" and "Synchronous Request Response" were implemented [13].

Pattern application Pattern application is a four-step process in SPT: pattern selection, identification of the design context, pattern adaptation and embedding into the design context [13, 12]. Pattern selection is done using a pull-down menu with all the supplied patterns. The user then has to identify the design context. This includes the components of the system the pattern is to be applied to, the ports the components are to communicate through and where in the state machines of the components the pattern is to be placed. The latter can, for instance, involve specifying states preceding and succeeding the pattern instance. This design context is identified through a series of simple dialogue windows. The third step is pattern adaptation, where the user gives names to the states and signals contained in the pattern. Finally, the pattern can be embedded automatically by the tool.

Pattern implementation TTD G2 provides the possibility to add new features through an application-specific API, and the patterns in the SPT are realised as pattern-specific scripts which can alter the model through this interface [13, p. 60]. This means that the SPT is bound to be used with TTD G2. The SDL patterns are, of course, defined for SDL, while the TTD G2 is mainly targeted at using UML 2.0. The SDL-patterns have thus actually been converted to UML 2.0 in order to implement them.

Difference to our interaction patterns The SDL-patterns describe message sequences, just like our interaction patterns. The main difference is that the SDL patterns focus on the internal behaviour of the state machines, while we only want to look at them from the outside. This makes the job of inserting, adapting and composing patterns quite different.

Concluding remarks We can use the SPT as an inspiration for our tool approach, even though the scope of the patterns differs and the patterns need

[4]We briefly discussed SDL patterns in section 2.1.2 on page 12.

to be embedded by the user in a different way. The SDL state machines and UML 2.0 state machines are very similar, and we the SDL patterns and the SPT as inspiration for our approach.

2.6 Ramses tool suite

Ramses is a service engineering suite developed at the Department of Telematics at NTNU. It is a prototype made to exemplify and demonstrate the ongoing research on different model-driven development techniques at the department. The aim of the Ramses tool suite is to make it easier to develop advanced telecommunication services [44].

Ramses is based on the Eclipse platform, which is an open source platform for tool integration [14]. Eclipse has a small run-time kernel, and everything else are contributions which each realise a certain functionality. The contributions are called plug-ins or extensions, and they are attached to so-called extension points. In the words of Gamma and Beck: *"Eclipse is a collection of places-to-plug-things-in (extension points) and things-plugged-in (extensions)."* [17, p. 5]. This enables everyone to contribute with their own plug-ins by just hooking on to the defined extension points.

A lot of different projects work on extending Eclipse with new functionality, and one of them is the UML2 project. This team implements a repository for the Eclipse platform based on the UML 2.0 meta model [15]. The UML2 repository stores the model elements using XML Metadata Interchange (XMI), which is a format for storing and exchanging models, including UML, standardized by OMG [38]. Using this repository, we have a way of creating, storing and accessing UML model elements. The UML2 repository is a part of the Eclipse Modelling Framework (EMF).

The Ramses tool suite is realised as a number of Eclipse plug-ins, and is constructed in such a way that it is easy to add new functionality by developing and integrating new plug-ins. Ramses makes use of the UML2 repository for managing model elements. Up until now, Ramses has mainly supported modelling of services using state machines, but this is currently being augmented with collaborations. It offers editing and visualisation of model elements, as well as validation and verification using so-called inspectors that check the model for unwanted properties. Ramses also includes code generation and trace visualisation of the system at run-time. The tool suite is constantly being added with new functionality to gain experience on different ways of modelling reactive, real-time systems.

Chapter 3

Interaction patterns

In this chapter, we introduce our interaction patterns. We define how to model them using UML 2.0, and determine the requirements of well-formed interaction patterns. We also present a way to describe the patterns textual and introduce the pattern library containing the interaction patterns identified in our work.

3.1 Introduction

An interaction pattern provides a proven solution to a re-occurring problem within the context of an interaction interface. At a high level of abstraction, we can observe that the sequences of messages exchanged between two parts of a service, and the intention behind them, have great similarities. The interaction patterns describe these re-occurring, abstract interactions and provide a solution that can be reused. It is worth emphasising that we want to find similarities in the purpose or motive behind an interaction between two parts. The signal exchanges just the way the distributed parts of our system exchange information.

Our interaction patterns can be regarded as basic building blocks for specifying an interaction interface. The complete interaction between two parts can be described and constructed from the generic interaction patterns. The abstract nature of the patterns provides the possibility to tailor them to fit the needs of a particular service. By using and combining many different interaction patterns (and of course perhaps using the same interaction pattern several times), we are able to define the interaction interface from a set of pre-defined interaction patterns which have proven to work in the past.

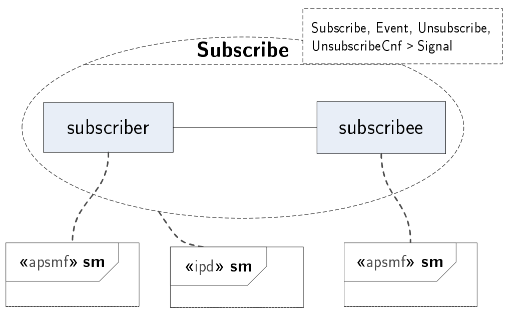


Figure 3.1: The interaction pattern SUBSCRIBE. The pattern has one APSMF describing the behaviour of each role, and an IPD describing its complete behaviour.

3.2 Modelling

The interaction patterns are modelled as UML 2.0 collaboration templates with two collaboration roles. They contain three state machines; two APSM fragments and one interaction pattern descriptor (IPD). The unbound parameters of the collaboration template are called pattern parameters. The interaction pattern SUBSCRIBE shown in Figure 3.1 will be used to exemplify the contents of the interaction patterns. We also refer to Appendix A on page 97 which summarises the constraints of the model elements described in this section.

3.2.1 Pattern parameters

The parameters of the interaction patterns are signals. The pattern will define the sequences of messages necessary to realise its purpose, but the information to be contained in each signal have to be specified by the user when the pattern is instantiated. The reason for this is that the information to be exchanged depends on each situation and can never be captured in a pattern. Concrete signals are assigned to the pattern parameters for each application of the pattern, and these signals can have signal parameters[1], which are used to transport the information. In Fig. 3.1 we see that the SUBSCRIBE-pattern has four parameters, which are the signals *Subscribe*, *Event*, *Unsubscribe* and *UnsubscribeCnf*.

3.2.2 Pattern roles

The roles describe the participants in the pattern and correspond to collaboration roles, as explained in section 2.2 on page 13. In the SUBSCRIBE-pattern, the two roles are named *subscriber* (the one who subscribes to a certain

[1]Signal parameters are properties of the signal, and have nothing to do with the pattern parameters.

event) and the *subscribee* (who pushes the event updates).

3.2.3 APSM fragments

Each pattern role is typed with a so-called APSM fragment (APSMF), which is a state machine with certain constraints. It defines the behaviour of the participant necessary to realise the intention of the pattern. We call them fragments to indicate that they will be assembled into APSMs when the pattern instances are composed to form the behaviour of the interaction interface.

An APSM fragment is quite similar to an APSM, but there are some differences:

1. The state succeeding the initial transition in an APSM fragment has exactly one outgoing transition[2]. This is not a requirement for an APSM.
2. An APSM fragment can not have any final states.
3. An APSM fragment must have least one uniquely labelled exit point.

Figure 3.2 shows the two APSM fragments defined for the Subscribe-pattern. Part (a) shows the fragment attached to the role named *subscriber* and part (b) the fragment for the *subscribee.* The two APSMF's obey to the constraints given above. We might think that because the two APSMF's describe the behaviour of the participants on each side of an interaction, they would be mirror images of one another. But, as we also can see from the figure, this is not always the case. This has to do with ensuring interaction consistency between the fragments, and will be discussed later in section 3.3 on the following page when we lay out the requirements for well-formed patterns.

3.2.4 Interaction pattern descriptor

The interaction pattern descriptor (IPD) is a state machine which defines how an instantiated interaction pattern can be related to other pattern instances. The IPD is the classifier behaviour of an interaction pattern. It reflects the exit points of the APSM fragments, and restrictions for this is given in section 3.3 on the next page. The IPD thus represents how an instantiated pattern can be activated and deactivated and is to be used as a building block when composing pattern instances. Section 4.3.4 on page 45 will discuss how the composition is done and why we have chosen to describe the behaviour of the interaction pattern as a state machine.

The constraints of an interaction pattern descriptor are:

1. It contains one or more uniquely labelled exit points.
2. It never contains any transitions or other vertices.

The interaction pattern descriptor of the Subscribe-pattern is shown in part (c) of Fig. 3.2. It reflects the exit points of the APSM fragments in (a) and (b).

[2]There is always one unique signal that starts the communication in a collaboration.

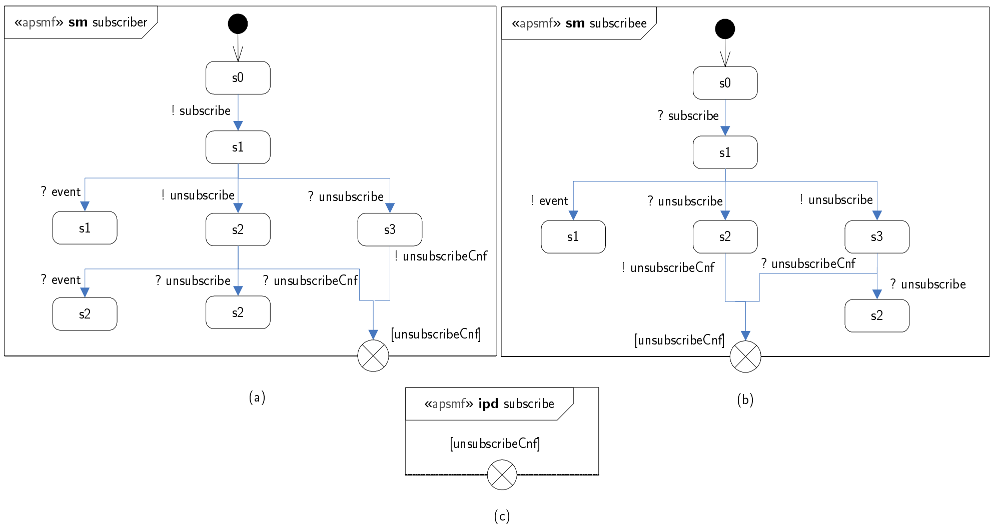


Figure 3.2: The state machines attached to the SUBSCRIBE **interaction pattern.** The two APSMFs describe the behaviour of the *subscriber* (a) and the *subscribee* (b) role. The IPD reflects the exit points of the APSMFs (c).

3.3 Well-formed patterns

An interaction pattern is well-formed when it enforces a set of rules ensuring its correctness. We must always make sure that the patterns contained in the pattern library are correct, or else they can result to errors during composition of APSMs and possibly ambiguous behaviour in the resulting APSMs. For an interaction pattern to be well-formed, we impose two requirements:

1. The pair of APSM fragments in the interaction pattern need to interact consistently.
2. The interaction pattern descriptor of an interaction pattern must be valid.

3.3.1 Consistent APSM fragments

To ensure that two APSM fragments interact in a consistent matter, we must apply the principles of containment and obligation to prevent unspecified signal reception, deadlock or improper termination when they interact (as explained in section 2.4 on page 18). In particular, we must make sure to handle equivoque states and mixed initiatives properly. All errors and ambiguities that might appear in an APSM, can also occur in an APSM fragment. When the two APSM fragments are found to be consistent, this will also ensure that they terminate in a coordinated matter.

If we return to Fig. 3.2, we see that the two APSM fragments actually contains a mixed initiative state, namely state *s1*. In this step in the interaction, the subscriber can send *unsubscribe* and the subscribee can send an *event* or *unsubscribe*. This is an example of concurrent behaviours (see section 2.4.3 on page 21).

In the APSMF of the subscriber role in part (a) of the figure, we see that the reception of *event* and the reception of *unsubscribe* are in conflict with the sending of *unsubscribe*. We handle this by specifying the reception of the two signals in the state we reach after sending *unsubscribe*. The action chosen is to ignore these receptions after we have sent *unsubscribe*, and we remain in state *s2*.

Part (b) of the same figure shows how the APSMF of the subscribee role is defined to handle this conflict. Here, we might receive *unsubscribe* both after we have sent *event* and after we have sent *unsubscribe*. Since we return to the same state, *s1*, after sending the event signal, we do not have to add anything in this case. However, when we have sent *unsubscribe*, we must add the reception of *unsubscribe* to state *s3*. Following the choice made for the subscriber role, we then have to proceed to state *s2* or else the two APSMFs will not interact consistently. Both APSMFs are now input consistent.

It is not important for us how one chooses to secure that the two APSMFs are consistent. Any suitable approach can be chosen to avoid ambiguities and conflicts. As long as the two APSMFs are related through both containment and obligation, they interact consistently, and are thus well-formed.

3.3.2 Valid interaction pattern descriptor

An interaction pattern descriptor is considered to be valid if is reflects the exit points of the APSM fragments in its interaction pattern. Assuming that the APSM fragments interact consistently, they will have the same exit points. The interaction pattern descriptor will then have one exit point for each exit point in the fragment, resulting in a one-to-one relationship between an exit point in the descriptor and each of the fragments.

In Fig. 3.2 the descriptor in part (c) has one exit point labelled *unsubscribeCnf*. This reflects the exit points of the fragments in part (a) and (b) correctly, and the descriptor is therefore valid.

3.4 Pattern description

We have so far only discussed the modelling aspects of an interaction pattern. To become a pattern, in the true sense of the term, the UML 2.0 pattern definition has to be supplied with a textual description to explain other properties of the pattern. The pattern description is important for understanding what the pattern is for and when to use it.

The pattern should be described in such a way that it is easy to communicate it to others. It is therefore important to include enough information to understand the pattern, but not to swamp the pattern description with a lot of unnecessary information [34]. There is no single format suited to describe all types of patterns, and we have chosen to include the properties we feel are sufficient for understanding what our interaction patterns are about. The emphasis of our pattern description is to help understand the details of the interaction pattern and ease the pattern selection process. Each interaction pattern is described using the following elements:

Name First of all, each pattern must have a good name which makes it easy to refer to and communicate the pattern to others. The name should capture the essence of the pattern and the result it creates [34]. Without a descriptive pattern name, the pattern can never become a part of the daily vocabulary [18, p. 6].

Problem The specific problem to be addressed by the interaction pattern. It gives a short explanation to the user about when the pattern can be of interest.

Solution A short explanation on how the interaction pattern solves the problem identified. It will thus be a brief explanation of what the pattern does. (The succeeding sections will explain the solution in more detail.)

Example usage A concrete example to make it easier to relate the abstract pattern to real-life services, illustrated with a sequence diagram.

Pattern roles A short description of the participants in the interaction pattern.

Pattern alternatives Some patterns might have variation possibilities. The alternatives are described in this section.

Pattern parameters A list of the parameters of the pattern together with a short explanation of each of them.

APSM fragments A graphical illustration of the APSM fragments of the two pattern roles.

Known uses Examples of the pattern being used in existing services.

Related patterns A list of patterns which are closely related to the current pattern. Included to help the user to find possible alternatives and/or complementary patterns.

We have omitted defining the context for each pattern. This will always be the services we are concerned about, namely telecommunication services where parts communicate asynchronously with messages through a buffered medium.

3.5 Pattern library

The pattern library is, not surprisingly, the place where we store our interaction patterns. By gathering all patterns in a single library, we ease the job of finding and selecting patterns. The library thus acts as a repository where users can search for a suitable interaction pattern to solve their design problem. The interaction patterns in the library have to be well-formed.

The library currently contains three generic interaction patterns:

- NOTIFY - Single message from A to B.
- REQUEST - Request from A to B, followed by a response in the opposite direction.
- SUBSCRIBE - A receives notifications from B each time a specific event occurs.

We have manually ensured the well-formedness of these three patterns. They are further described in Appendix B on page 107.

The three interaction patterns identified in this work are quite basic and generic, and they can be found in nearly all telecommunication services. The library can also be augmented with new interaction patterns in the future, including service and domain specific patterns.

Chapter 4

Applying and composing interaction patterns

This chapter first describes how we express the interaction interface. The process of applying an interaction pattern is then described, before we explain how to compose the applied interaction pattern instances of the interaction interface.

4.1 Creating the interaction interface collaboration

We describe the interaction interface using a two-way collaboration. It is simply called an *interaction interface collaboration*, and is constructed in its entirety from a number of pattern instances.

An interaction interface collaboration has the following constraints:

1. An interaction interface collaboration has one state machine as classifier behaviour that is a so-called applied interaction pattern instances composer (AIPIC).
2. An interaction interface collaboration has two state machines as owned behaviours that are association point state machines (APSMs). Each of the collaboration roles are typed with one of the APSMs.
3. All collaboration uses of an interaction interface collaboration must be typed with an interaction pattern instance.

The AIPIC is a special state machine where we define the execution order of the applied pattern instances, and we will describe this element further in section 4.3.2 on page 43.

In Figure 4.1 we have defined the interaction interface collaboration *Buddy Tracking* of our example service. We see that each role has an APSM attached to it. The collaboration also has an AIPIC, where we define the sequence of the interaction patterns we are to apply.

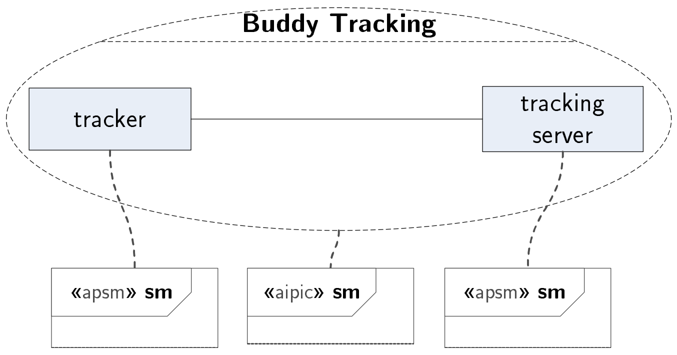


Figure 4.1: The interaction interface collaboration *Buddy Tracking.* The collaboration has one APSM attached to each collaboration role, as well as an AIPIC as classifier behaviour.

4.2 Applying an interaction pattern

To apply an interaction pattern, we select a pattern from the library and customise it so it suits our needs. We recognise this as step one and two in Fig. 1.1 on page 3. More precisely, we first have to select a pattern, and then decide on any pattern alternatives, before we can bind the pattern parameters and the pattern roles. This will be explained in detail in this section.

Let us exemplify this process by illustrating how we can apply a pattern to our interaction interface collaboration *Buddy Tracking* from Fig. 4.1. One of the things we want to happen in the interaction between the *tracker* and the *tracking server*, is that the *tracker* is notified by the *tracking server* every time a buddy moves[1]. Throughout this section we will explain how we can apply a pattern to solve this.

4.2.1 Select an interaction pattern

First, we select the appropriate interaction pattern by looking through the pattern library. Based on the pattern description, we can find a pattern that solves our problem and results in the interaction we want to happen.

Searching through the pattern library, we find that the suitable pattern to apply to *Buddy Tracking* is the pattern named SUBSCRIBE. This pattern allows us to subscribe to a certain event, and receive update messages every time this event happens. We can use this interaction pattern to subscribe to the position changes of the buddy we want to track.

[1]The requirements of the service was elaborated in 1.2 on page 5.

4.2.2 Decide between pattern alternatives

When we have found the right interaction pattern, we have to decide on the pattern alternatives, if the selected pattern has any. As explained in section 3.4 on page 33, these alternatives are small variations of an interaction pattern and we have to choose which alternative we want to use in the particular situation.

The Subscribe-pattern has in fact a pattern alternative, which concerns which of the two participants that can end the subscription. It is possible that either the *subscriber* or the *subscribee* can end the subscription, or both. We decide that we want both participants to be able to end the subscription. The *tracker* can stop receiving the position updates, and the *tracking server* can stop sending them, for instance if the buddy revokes the allowance to trace him, or if the buddy goes off-line.

4.2.3 Bind pattern parameters

To be able to use an interaction pattern in a concrete system, we have to instantiate it by binding its parameters to specific values. As we know from section 3.2.1 on page 30, the parameters are signals.

Interaction pattern instance

When all parameters are bound to concrete signals, we obtain an interaction pattern instance, which is a two-way elementary collaboration.

In Figure 4.2 we have illustrated how the Subscribe-pattern is instantiated, obtaining the interaction pattern instance *Track User* of our example service. We see that the pattern parameters have been bound to signals. For instance, a signal *StartTracking* is assigned to the parameter *Subscribe*.

The interaction pattern instance contains three state machines; two APSM fragments and an interaction pattern instance descriptor. Appendix A on page 97 defines these model element, and we will also quickly go through them here.

APSM fragments Just like the interaction pattern, the interaction pattern instance contains two APSM fragments, one for each pattern role. They now make use of the concrete signals assigned to the parameters and are adjusted according to what the user decided regarding the pattern alternatives.

In Figure 4.3 we see the APSM fragments of the pattern instance *Track User*. Part (a) shows the APSM fragment of the role *subscriber* and part (b) of the *subscribee*. Compared to the APSM fragments of the Subscribe-pattern, as shown in Fig. 3.2 on page 32, we see that they are almost the same, only the signals used are different.

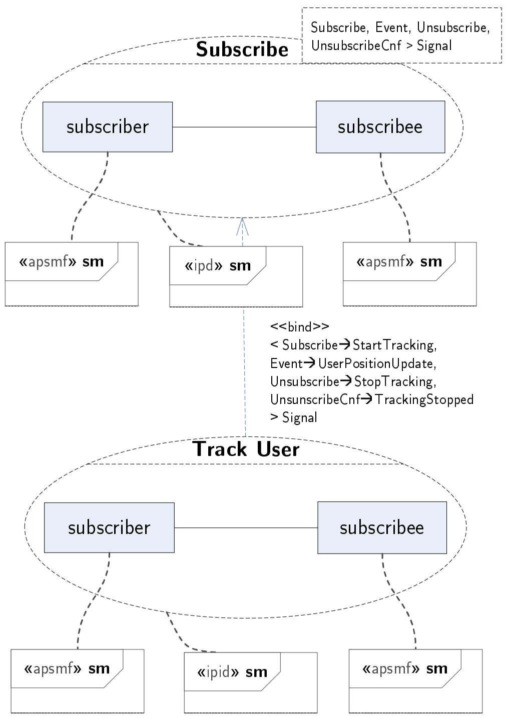


Figure 4.2: Obtaining an interaction pattern instance. The interaction pattern instance *Track User* is obtained by instantiating the SUBSCRIBER pattern.

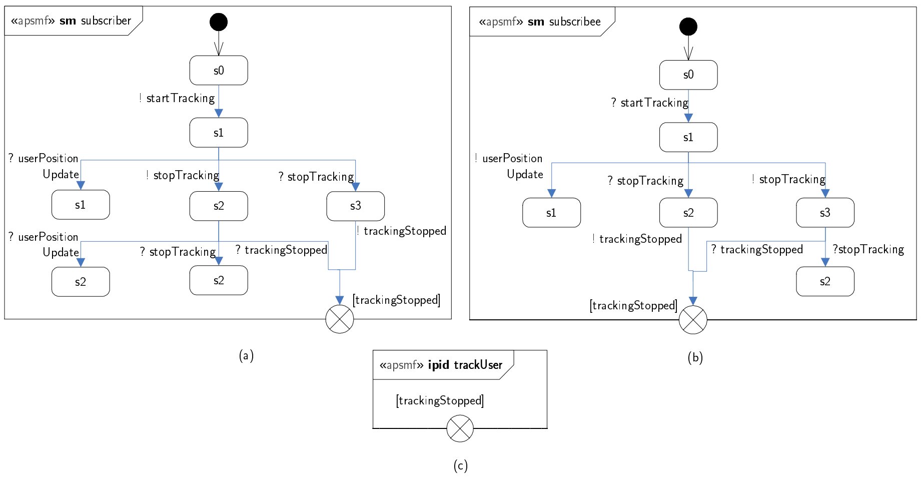


Figure 4.3: The state machines attached to the *TrackUser* interaction pattern instance. The two APSMFs describe the behaviour of the *subscriber* (a) and the *subscribee* (b) role. The IPID reflects the exit points of the APSMFs (c).

Interaction pattern instance descriptor The interaction pattern instance descriptor (IPID) is the same as the interaction pattern descriptor from section 3.2.4 on page 31, only that it is attached to an instantiated interaction pattern. It reflects the exit points of the APSM fragments of the interaction pattern instance, and describes how this interaction pattern instance can be composed with other interaction pattern instances. The IPID is used during composition of interaction pattern instances, which will be described in section 4.3.3 on page 44.

Fig. 4.3, part (c), shows the interaction pattern instance descriptor of *Track User*. We see that it has one exit point, *Tracking Stopped*, which correctly reflects the exit points of the APSM fragments shown in part (a) and (b) of the same figure.

The interaction pattern instance has almost the same elements as an interaction pattern. We can thus state that the constraints of an interaction pattern instance are:

1. An interaction pattern instance has one state machine as classifier behaviour that is an interaction pattern instance descriptor (IPID).
2. An interaction pattern instance has two state machine as owned behaviour that are association point state machine fragments (APSM fragments).
3. An interaction pattern instance has exactly two collaboration roles.
4. Each of the two collaboration roles in an interaction pattern instance is typed with one of the APSM fragments.

The interaction pattern instances must have distinct signal sets, meaning that signals used by one pattern instance cannot be used by another one. This restriction can be found in the meta model we obey to, which defines that a signal can be assigned to only one elementary collaboration [31]. This means that we have to create new signals, or use existing signals that are not used in other pattern instances, for each pattern parameter.

4.2.4 Use the interaction pattern instance in the interaction interface collaboration

The newly obtained interaction pattern instance must now be used in the interaction interface collaboration. We have to decide which of our collaboration roles in the interaction interface collaboration that should play the roles of the pattern instance. Because we deal with two-way collaborations, this is quite a simple decision, as we only have two possibilities.

An interaction pattern instance can only be used once. We reuse the interaction patterns, and not the interaction pattern instances. Since the pattern instances have distinct signal sets, this implies that all pattern instances applied to an interaction interface collaboration will make use of different signals. This makes

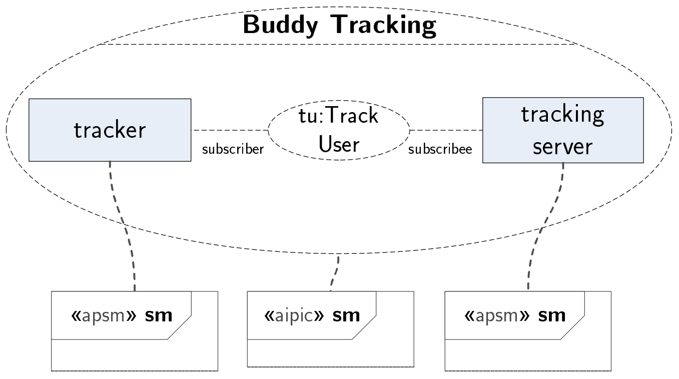


Figure 4.4: Using the interaction pattern instance *TrackUser* in the interaction interface collaboration *BuddyTracking.* *Track User* is used by binding its pattern roles to the collaboration roles of *Buddy Tracking.*

it easy to separate the applied pattern instances and make sure that they cannot interfere with each other.

Returning to our example, we must determine which of the collaboration roles in *Buddy Tracking* that are to play which roles of the instantiation of the SUBSCRIBE-pattern we just have created, named *Track User.* Figure 4.4 shows how the *subscriber* role of *Track User* is bound to the *tracker* role of *Buddy Tracking*, and similar for the *subscribee* role.

4.3 Composing interaction pattern instances

After applying a number of interaction pattern instances to the interaction interface collaboration, we can start composing them. In other words, it is time to decide how they relate to one another and when they can be activated.

4.3.1 Composition possibilities

The pattern instances can only be activated sequentially, and can never interleave. We assume that only one elementary collaboration can be active at an association point at a time, which also is assumed in our underlying meta model [31]. This means that only one pattern instance will be active at a time and that they will be activated in sequence. If two parts need to communicate different things at the same time, they can do so through an extra pair of association points. This will yield in a new interaction interface, completely independent of the first one.

A pattern instance always terminates with a description (a label) indicating the result of the interaction of the pattern instance. After the interaction of a pattern instance has ended with a certain result, one of three things can happen:

No succeeding interaction The interaction of the interaction interface collaboration finishes. This means that after the pattern instance has terminated with a certain result, nothing else can happen in the interaction between the two parts. No other pattern instance can ever be activated.

One succeeding interaction Another pattern instance is to follow the one that has just terminated, and we know exactly which one. There is one, and only one, possible pattern instance to be activated.

Two or more possible succeeding interactions One out of two or more candidate pattern instances can be activated. Internal decisions in the parts determine the succeeding interaction. As our focus is on the interaction interface between the parts, we only know that there will be a decision, but we do not know how the choice is made. From our viewpoint, we thus have a number of candidates for the succeeding interaction, and we know that one of them will happen.

4.3.2 The applied interaction pattern instances composer

The composition is done inside the applied interaction pattern instances composer (AIPIC) of the interaction interface collaboration. This is a state machine with certain constraints which is solely used to specify how the pattern instances applied to the interaction interface collaboration are related to each another. Appendix A on page 97 defines the model element. The constraints of the AIPIC are:

1. An AIPIC has exactly one initial state, and exactly one initial transition.
2. All states in an AIPIC are submachine states.
   (a) For each applied interaction pattern instance, a submachine state exists referring to the interaction pattern instance descriptor (IPID) as its submachine.
   (b) The submachine state has one exit connection point reference for each exit point in its submachine. This exit connection point reference has the same name (that is, the same label) as the exit point.
3. A transition in an AIPIC has no send signal actions or signal triggers.
4. The source of a transition in an AIPIC is always either the initial state or an exit connection point reference of a submachine state.
5. The target of a transition in an AIPIC is always a submachine state.

The vertices in the AIPIC are thus submachine states pointing to the interaction pattern instance descriptor (IPID) of the pattern instance if represents. Figure 4.5 illustrates this. As we know, the IPID describes how a pattern instance can relate to other pattern instances. The exit points of the submachine, in our case the IPID, are reflected as exit connection point references of the submachine state [46, p. 628].

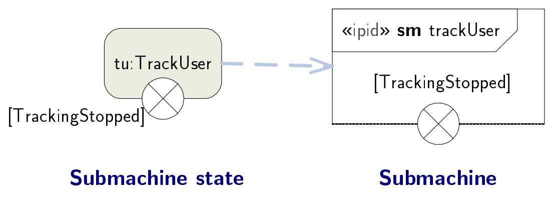


Figure 4.5: A submachine state and its submachine. A submachine state in the AIPIC refers to an interaction pattern instance descriptor (IPID) as its submachine. The exit points of the IPID are reflected as exit connection point references in the submachine state.

Exit connection point reference	Combination case
LogIn.LogInDenied	No succeeding interaction
LogIn.LogInOK	One succeeding interaction
TrackRequest.TrackingAccepted	One succeeding interaction
TrackRequest.TrackingDenied	Two or more possible succeeding interactions
TrackUser.TrackingStopped	Two or more possible succeeding interactions
LogOut.LogOut	No succeeding interaction

Table 4.1: The combination cases in *BuddyTracking*. The combination cases of the exit connection point references in the AIPIC in Fig. 4.6 on the facing page

4.3.3 How do we compose?

We compose by linking the exit connection point references of the submachine states in the AIPIC to other submachine states. Let us return to our example service to exemplify this.

Figure 4.6 illustrates how we compose the pattern instances of the interaction interface collaboration *Buddy Tracking*. The figure shows four submachine states in the AIPIC, one for each applied pattern instance, and the links defined between the exit connection point references and the submachine states. The AIPIC shows all three combination cases explained in 4.3.1 on page 42. We see that the first pattern instance to be activated is *LogIn*. If this results in a *LogInDenied*, no succeeding interaction is defined. This will result in the termination of *BuddyTracking*. If the log in is accepted, then *Track Request* is the only possible succeeding interaction. If *Track Request* terminates with a *TrackingDenied*, the succeeding interaction can be both a new track request (possibly to track another user), or we log out. One of these two things can happen, and we have two possibilities for further interaction. Table 4.1 summarises the composition cases.

A note on the syntax used
Figure 4.6 on the next page shows that we choose to model a decision without any decision symbol (choice). The reason for this is, as stated earlier, that we have no knowledge of what the decision is based on. This is part of the internal behaviour of the parts. To keep it simple we therefore omit the choice that otherwise would have been included to describe that a decision has to be made about the further interaction. Figure 4.7 shows how the a link would have

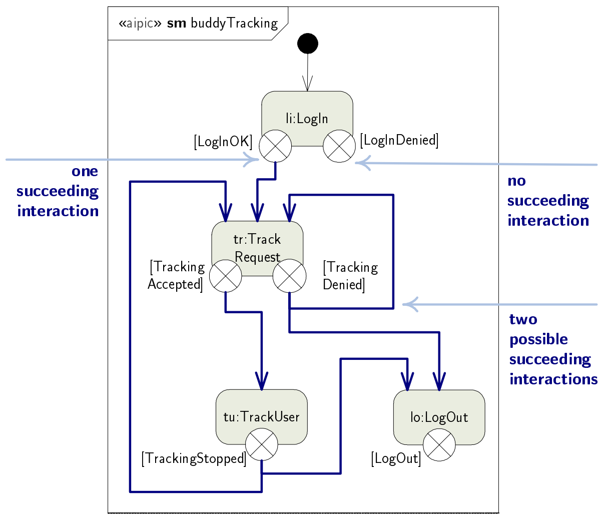


Figure 4.6: The AIPIC of *BuddyTracking.* We compose the applied interaction pattern instances of *BuddyTracking* by linking the submachine states of its AIPIC.

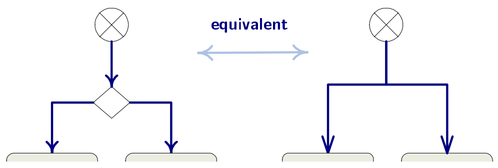


Figure 4.7: Syntax for modelling decision. Because the actual decision is hidden from the interface behaviour, a decision is modelled without a choice state. The meaning is equivalent.

looked like with a decision symbol (at the left), in contrast to how we model a decision (at the right). Only the syntax differs, and they are semantically equivalent.

4.3.4 On the usage of state machines for composition

The behaviour of the pattern instances could be modelled as interactions, and composed using an interaction overview diagram [39, p. 499], which is used to get an overview of the control flow between interactions [46, p. 410]. This resembles what we want to do: to describe the control flow between the pattern instances. We need to be able to describe different termination possibilities of a pattern instance, meaning specifying different outcomes of the interactions in the interaction overview diagram. The UML 2.0 specification [39] does not show this possibility in an interaction overview diagram explicitly, but continuations [39, p. 459] could perhaps be used for this purpose.

Composition of the pattern instances (which are collaboration uses of the interaction interface collaboration) could also be modelled using UML 2.0 activities [39, p. 285]. The behaviour of a collaboration use is represented by an activity. The actual composition is done inside another activity, referring to these former activities through so-called call behaviour actions [39, p. 337]. These activities and call behaviour actions can have so-called output pins [39, p. 383], making it possible to describe different termination possibilities of a collaboration use. This could be used to model the different termination possibilities of a pattern instance. Activities also allow us to describe parallel execution of collaboration uses, but this is not necessary in our case as we only consider sequential composition.

State machines were chosen mainly due to the experience we have with modelling and understanding them. In addition, the behaviour of the participants in a pattern instance are described with APSM fragments, which are state machines. Together they make up the total behaviour of the pattern instance, and is it therefore convenient and natural to use a state machine as well for representing the whole pattern instance. Because we only consider sequential composition, it is easy to express this inside a state machine by drawing transitions between submachine states referring to the pattern instances. With all model elements involved in the composition being state machines, it is fairly easy to understand how they relate to each other and this results in a quite coherent approach.

Chapter 5

Generating APSMs

After a brief introduction, this chapter will present the semantics of the APSM composition. We then explain how to ensure correct APSMs by handling error situations in a reasonable way.

5.1 Introduction

The APSMs are generated by assembling APSM fragments from the pattern instances. One APSM fragment from each pattern instance is used to create the APSM of one of the collaboration roles of the interaction interface collaboration. There are two main questions when doing so: how to select which of the APSM fragments of a pattern instance to use when creating which APSM, and how these APSM fragments should be combined.

Which APSM fragment? The role bindings determine which of the two APSM fragments of a pattern instance to use in what APSM. Once more, we return to the interaction interface collaboration *Buddy Tracking* in Figure 5.1. We know that the collaboration role *tracker* plays the following roles: the *requestor* of the *LogIn* pattern instance, the *requestor* of *TrackRequest*, the *subscriber* of *TrackUser* and the *notifier* of *LogOut*. This, of course, means that the APSM we are going to create for the *tracker*-role, will be composed from the APSM fragments of these four pattern roles. Correspondingly, the APSM of *tracking server* will be put together from the APSM fragments of the roles it plays in each pattern instance.

Which sequence? How the APSM fragments are to be combined is already defined by the composition done inside the applied interaction pattern instances composer (AIPIC). The composition of *Buddy Tracking* was shown in Fig. 4.6 on page 45. From this composition we know which order the pattern instances are to be activated, and we thus know in what order to assemble the APSM

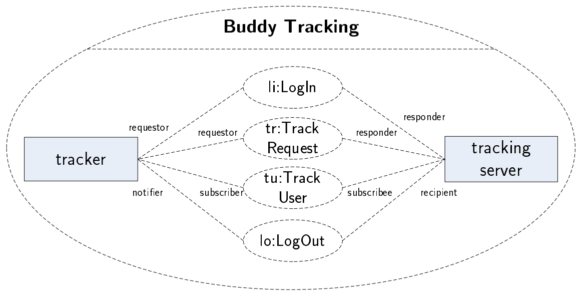


Figure 5.1: Selecting the correct APSMF using the knowledge from the role binding. To compose the APSM of a collaboration role of *Buddy Tracker*, we retrieve the APSM fragments of the pattern roles it plays in each applied interaction pattern instance.

fragments. The AIPIC is our road map for composing the APSM fragments into one APSM.

5.2 Semantics

We will now define the semantics of combining several APSM fragments into one APSM. That is, how the APSM fragments should be assembled in each of the combination cases identified in section 4.3.1 on page 42. Extracts from the composition of the pattern instances in *BuddyTracker*, as previously shown in Fig. 4.6 on page 45, will be used to exemplify this.

5.2.1 Case 1: No succeeding interaction

When no succeeding interaction is specified in the AIPIC, the incoming transitions of the exit point in the fragments are targeted in a final state when constructing the APSMs. As we remember from section 4.3.1 on page 42, the interaction between the two parts is finished. This implies that the APSMs of the interaction interface collaboration should terminate, which is done through a final state.

Part (a) of Figure 5.2 shows a portion of the AIPIC of *BuddyTracking*. The exit connection point reference *LogInDenied* has no succeeding interaction. Part (b) shows extracts from the APSM fragments of the pattern instance in question. In part (c) we show the result when this is spelled out in the APSMs. After the exchange of the signal *LogInDenied*, the APSMs terminate in a final state.

5.2.2 Case 2: One succeeding interaction

In the case of only one succeeding interaction, we have to combine the APSM fragments of the two pattern instances. The incoming transitions of the exit

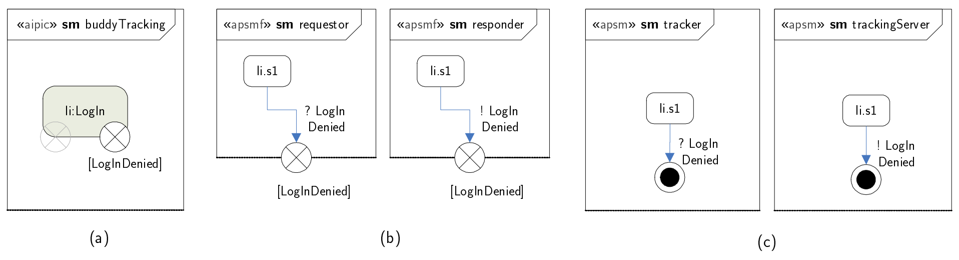


Figure 5.2: APSM composition semantics: no succeeding interaction. When no succeeding interaction is specified in the AIPIC (a), the exit points of the APSM fragments (b) are replaced by final states when inserted into the APSMs (c).

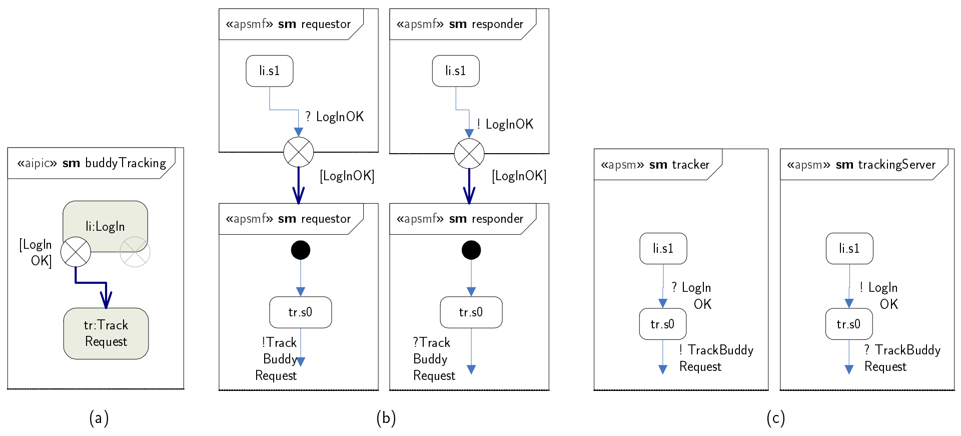


Figure 5.3: APSM composition semantics: one succeeding interaction. In the case of one succeeding interaction in the AIPIC (a), the APSM fragments (b) are combined as shown in (c).

point in the preceding fragment are attached to the state succeeding the initial state in the next fragment.

Figure 5.3 illustrates how we compose APSMs in this situation. Part (a) shows a part of the *BuddyTracking* AIPIC, where the exit connection point reference *LogInOK* is connected to *TrackRequest*. In part (b) we have shown the relevant APSM fragments that are to be combined and part (c) shows the resulting APSMs. After the exchange of *LogInOK*, the APSMs proceed to a state *tr.s0*, awaiting the exchange of *TrackBuddyRequest*.

5.2.3 Case 3: Two or more possible succeeding interactions

When two or more pattern instances are candidates for the further interaction, each transition preceding the exit point of the preceding fragment has to be connected to the state succeeding the initial transition of each of the succeeding fragments.

Part (a) of Figure 5.4 shows an extract from the AIPIC of *BuddyTracking*

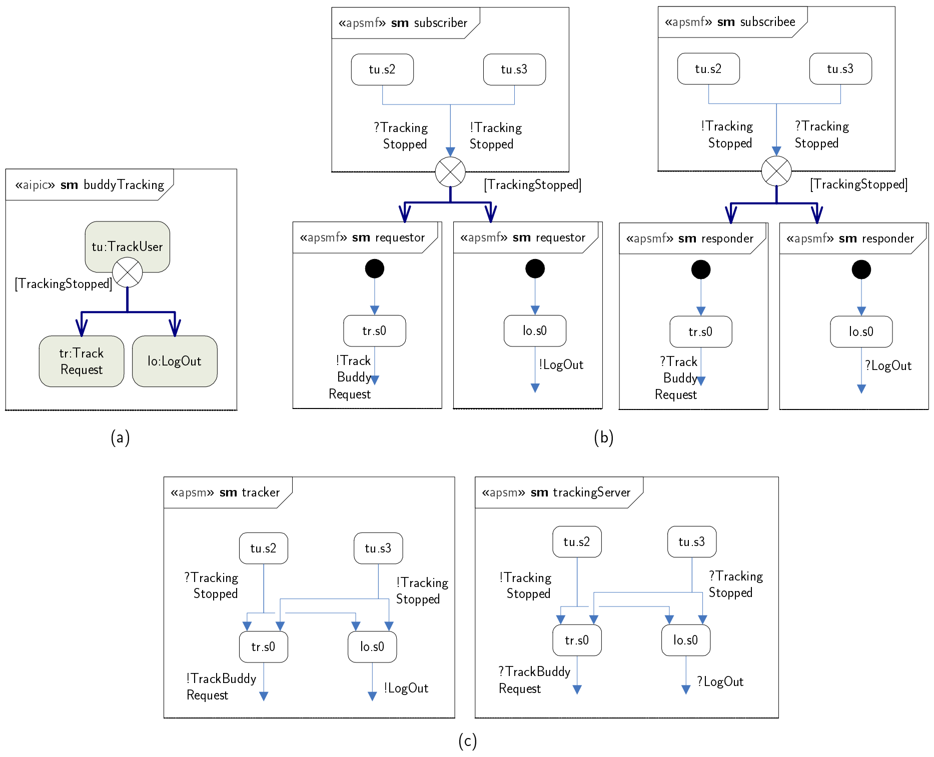


Figure 5.4: APSM composition semantics: two or more possible succeeding interactions. When two or more succeeding interactions are specified in the AIPIC (a), the APSM fragments (b) are combined as shown in (c).

illustrating this scenario. In part (b) we see the different APSM fragments involved in this example. We see that the signals *TrackBuddyRequest* and *LogOut* are candidates for exchange after the *TrackUser* pattern instance has terminated. The APSMs are shown in part (c). If we look at the APSM for the *tracker* role, we see that after receiving *TrackingStopped* in *tu.s2*, we can either proceed to *tr.s0* to send *TrackBuddyRequest* or to state *lo.s0* and send *LogOut*. The similar happens for the sending of *TrackingStopped* in *tu.s3*.

This example also illustrates that an exit point of an fragment may have more than only one incoming transition. Here, the signal *TrackingStopped* can go in any direction. We have to take this into account in all three combination cases.

Two or more possible succeeding interactions will always yield in equivoque states and thus ambiguities. In the APSM of the *tracker* in part (c) of Fig. 5.4 we see that when receiving *TrackingStopped* in *tu.s2* we can either proceed to *tr.s0* or to *lo.s0*. In *tu.s3*, the same ambiguity arises upon sending *TrackingStopped*. The situation is reverse for the APSM of the *trackingServer*. Generally speaking, any state preceding the exit point of a fragment will become an equivoque state when two or more pattern instances are candidates for the succeeding interaction.

5.3 Ensuring correct APSMs

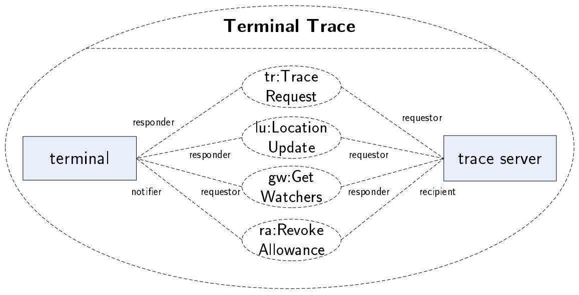
We now know how to combine the APSM fragments into one APSM. Before we can set off with generating APSMs, however, we need to find out how to ensure that the APSMs we create interact consistently. There are three prerequisites for correct APSMs:

- Well-formed patterns
- Correct composition of applied pattern instances
- Correct composition of APSMs from APSM fragments

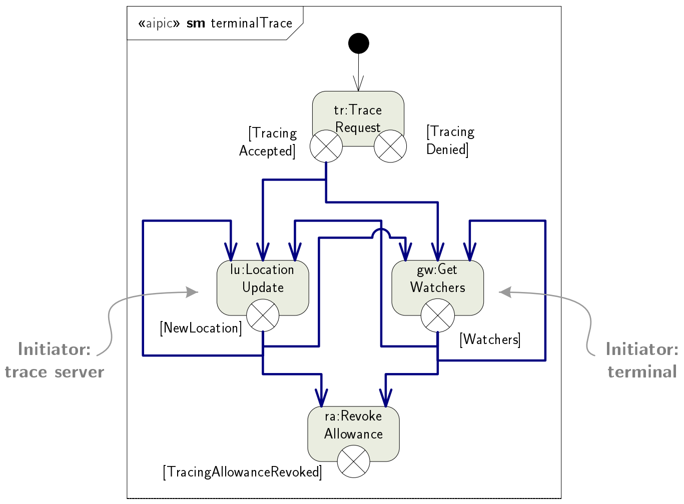
The first two requirements are assumed to be fulfilled at this stage. The patterns in the pattern library are expected to be well-formed, as discussed in section 3.3 on page 32. The pattern instances have also been composed correctly if the AIPIC of the interaction interface collaboration is in accordance with the constraints defined in section 4.3.2 on page 43. Now, we have to explore and identify the errors and ambiguities that might occur when the APSMs are composed.

The situations that might jeopardise the consistency of the APSMs are the ones involving equivoque states and mixed initiative states, as identified and described in section 2.4 on page 18. We will go through each of the errors they might produce, identify whether it can occur during the APSM composition or not, and find a way to handle them so we can produce correct APSMs.

We will use our *Tracking Service* to exemplify the different situations that might arise. Figure 5.5, part (a), shows the interaction interface collaboration



(a)



(b)

Figure 5.5: The interaction interface collaboration *Terminal Trace* and its AIPIC. The structure of *Terminal Trace* (a) and the AIPIC (b). Note that the interaction pattern instances *Location Update* and *Get Watchers* can be activated at the same time, but have different initiators.

Terminal Trace, a part of our example service. Part (b) shows its AIPIC which has been constructed based on the requirements given in section 1.2 on page 5.

5.3.1 Equivoque states

Equivoque states will always appear then two or more pattern instances are candidates for the succeeding interaction. The sort of ambiguity introduced is based on the contents of the succeeding APSM fragments.

Input and mixed ambiguity

Both input and mixed ambiguity are removed in the same manner. We will thus illustrate with the situation of mixed ambiguity, but the technique applies to both situations.

Consider Figure 5.6, where part (a) shows a portion of the AIPIC of *TerminalTrace*. Both *LocationUpdate* and *GetWatchers* can be activated, and they have different initiators. Part (b) shows the APSM fragments from these pattern instances which we are going to combine and insert into the APSMs of the *terminal* and *traceServer* roles. We use the semantic defined in section 5.2 on page 48, yielding in APSMs shown in part (c) of the figure. The *tr.s1*-states of both APSMs can quickly be identified as equivoque states. Because both signal sending and signal reception is possible in the equivoque state, it represents a mixed ambiguity.

Ambiguities can be removed by a transformation called merging [16, p. 164]. More precisely, we merge (that is combine) the states succeeding the outgoing transitions which define equal signal reception or signal sending. We return to Fig. 5.6, and see that in part (d) the states *lu.s0* and *gw.s0* have been merged in both APSMs. This is done by creating a new state which contains all the outgoing transitions of the two states, and define this to be the next state after the exchange of *TracingAccepted*. We have now removed the ambiguity and *tr.s1* is no longer an equivoque state.

Removing input ambiguity is done the same way. The only difference to the case of mixed ambiguity is that the same collaboration role initiates all the possible succeeding interactions. Output divergence will always come in pair with input ambiguity. If we experience input ambiguity in one of the APSMs, we will always have output divergence in the other one. This follows directly from the fact that every signal sent from one APSM is received in the other APSM.

If we take a closer look at the APSMs in part (d) of Fig. 5.6, we see that the new merged state, *lu.s0/gw.s0*, actually is a mixed initiative state. In fact, the only time mixed initiative states can be introduced in the APSMs are when we remove mixed ambiguities. How to handle this will be explored in section 5.3.2.

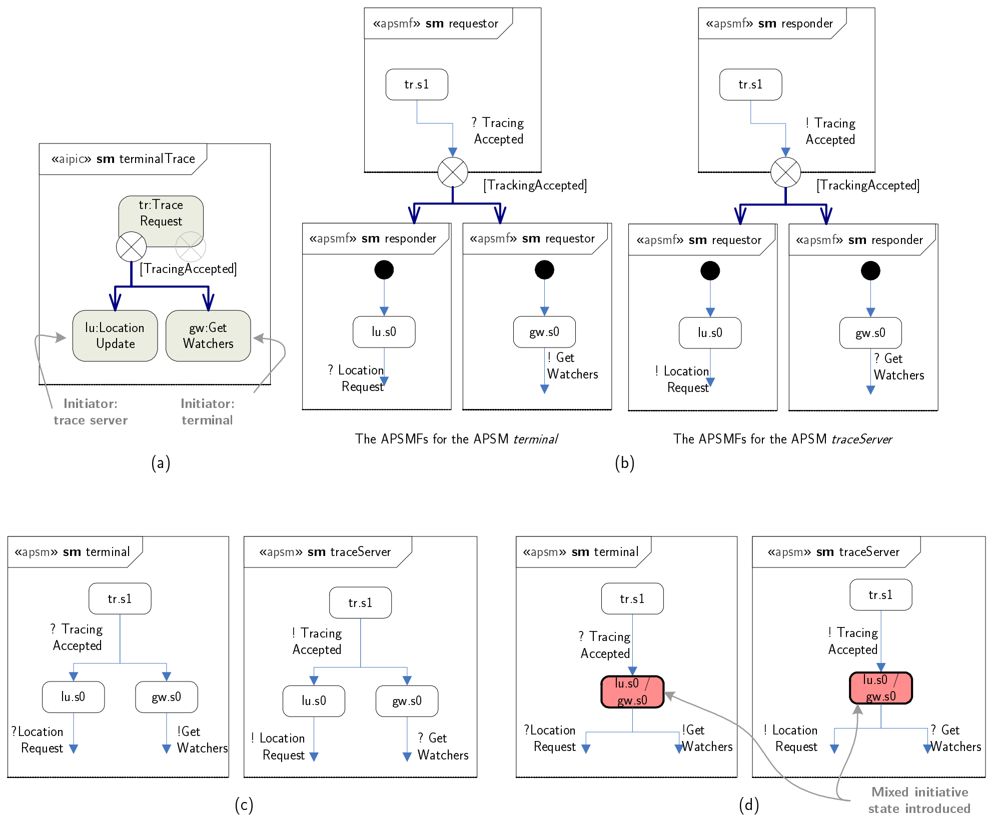


Figure 5.6: Removing mixed ambiguity by merging. Two pattern instances can be activated at the same time (a), with the APSM fragments shown in (b). The resulting APSMs (c) will contain an ambiguity, which can be removed by merging (d). Note that, in the case of mixed ambiguity, the merged state will have a mixed initiative.

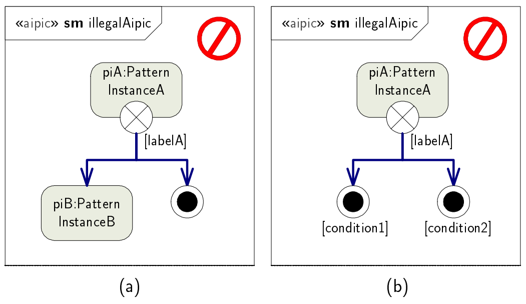


Figure 5.7: Preventing termination ambiguity and termination condition ambiguity. Due to the constraints defined for the AIPIC, the user is never allowed to introduce termination ambiguity (a) or termination condition ambiguity (b).

Termination ambiguity and termination condition ambiguity

In section 2.4.3 on page 20, we explained that termination ambiguity occurs when an external observer is unable to decide whether an APSM has terminated or not. This means that sending (or receiving) the same signal can result in both a termination of the APSM or that it proceeds to another state and stays active. Termination condition ambiguity occurs when the APSM can terminate with different conditions when the same signal is sent or received.

Termination ambiguity can never occur during composition of APSMs. This is prevented by restricting the composition possibilities for the user inside the AIPIC: To indicate no succeeding interaction, that is termination, the exit connection point reference is not linked to any submachine state. To indicate succeeding interaction, the exit connection point reference is linked to one or more submachine states. It is obvious that both these things cannot be present at the same time. There is thus no way of specifying both termination and further interaction when a pattern instance finalises with a certain result, and we can never experience termination ambiguity.

Termination condition ambiguity can neither happen during composition, also due to the constraints of the AIPIC. Termination of the APSMs after a pattern instance has finished with a certain result can only be defined by no links from the exit connection points reference in the AIPIC. Thus, in the case of no succeeding interaction, the APSMs terminate with the condition attached to the current exit connection point reference. It is impossible to introduce different conditions, and this ambiguity is prevented.

Let us briefly discuss how these ambiguities could have appeared, if the constraints of the AIPIC were different. Assume that final states were allowed in the AIPIC, and that the way to specify no succeeding interaction was to draw a link from an exit connection point reference to a final state. This seems quite intuitive: if the interaction between the two parts should terminate after a pattern instance is finished, then connect it to a final state. However, this could easily introduce ambiguities if a user decides to draw links both to a final state and a submachine state from the same exit connection point reference. This is illustrated in part (a) of Figure 5.7. The user has defined that after *PatternInstanceA* terminates through *labelA*, the interaction can both terminate or it can continue with *PatternInstanceB*. This seems quite unnatural, and would introduce termination ambiguities. Similarly, part (b) of the same figure shows an illegal AIPIC where termination condition ambiguity would emerge during composition. This is the same as saying that after *PatternInstanceA* terminates with *labelA*, we either terminate with *condition1* or *condition2*, which is pretty illogical. It is best to prevent the user from defining such erroneous situations, which is done through the restrictions on the composition possibilities.

5.3.2 Mixed initiatives

During composition of APSMs, mixed initiatives occur when two or more succeeding interactions are possible and these have different initiators. This is what we called *concurrent behaviours* in section 2.4.3 on page 21, which can lead to conflicts if the signals cross. But, as we remember from the same section, mixed initiatives can also occur due to alternative orderings.

Alternative orderings

Alternative orderings can actually never happen as a result of the composition. This is because we have defined that each pattern instance has its own distinct set of signals (section 4.2.3 on page 39) and that a pattern instance can be used inside a interaction interface collaboration exactly once (section 4.2.4 on page 41).

Consider Figure 5.8, where part (a) shows an example AIPIC. Part (b) shows the APSM fragments to be inserted into the APSM of one of the two collaboration roles in part (c). We see that the mixed initiative state *piB.s0/piC.s0* in the APSM of *role1* emerges due to alternative orderings. However, the APSM fragments in part (b) are not valid. They cannot both make use of the signals *B* and *C*. Consequently, alternative orderings can never occur during composition of APSMs, and we do not have to consider the possible further behaviour ambiguities emerging from this situation.

We still have to take care of the concurrent behaviours. When we assemble APSM fragments, mixed ambiguity is removed by merging, and the merged state introduced will always be a mixed initiative state, as described in 5.3.1 on page 53. We have to ensure input consistency to avoid a mixed initiative conflict. In some special cases, termination might prevent us from doing so, and our composition has to take this into consideration as well. We will examine these two possible errors in turn.

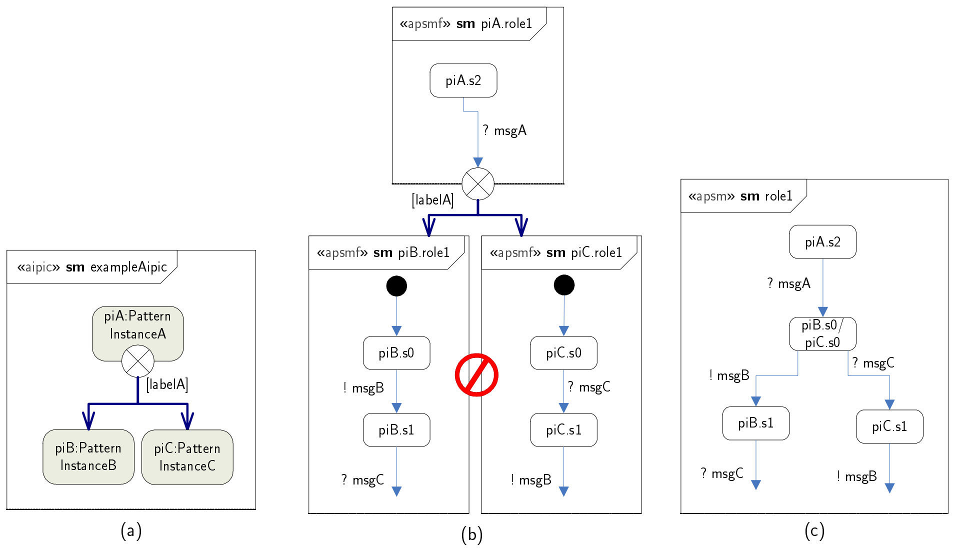


Figure 5.8: Alternative orderings during composition. A composition (a) with pattern instances using the same set of signals (b) would have introduced alternative orderings in the APSM (c). However, this is not allowed.

Input inconsistency

Mixed initiative states emerging due to concurrent behaviours need some sort of conflict resolution. The two APSMs have each initiated a new behaviour, and we need a way of determining which of these behaviours that should be allowed to continue. Both APSMs can detect this conflict when they, in a state following signal sending in the mixed initiative state, receives a signal that is also defined as an input in the mixed initiative state [16, p. 181]. We need a way of determining which of the behaviours initiated that should survive and which should be suppressed.

Floch presents a conflict resolution scheme where one of the two APSMs acts as a coordinator to resolve the conflict [16, p. 180]. The coordinator can either be selected at design time or run-time, in the latter possibly after a negotiation phase [16, p. 180f]. The further behaviour of the interaction is decided by the exchange of an extra signal from the coordinator to the other APSM. We choose to solve this sort of conflict a bit different.

Conflict resolution using priorities To resolve a mixed initiative conflict, the further behaviour can be decided by giving the pattern instances involved in the conflict a unique priority. As stated earlier, the mixed initiative conflict during composition emerges because two or more pattern instances, with different initiators, are candidates for activation. The goal of conflict resolution will in this case be to decide which of these pattern instances that should be allowed to continue, and which one to abort. Every pattern instance can be assigned a priority, and we can decide on the further behaviour after a mixed

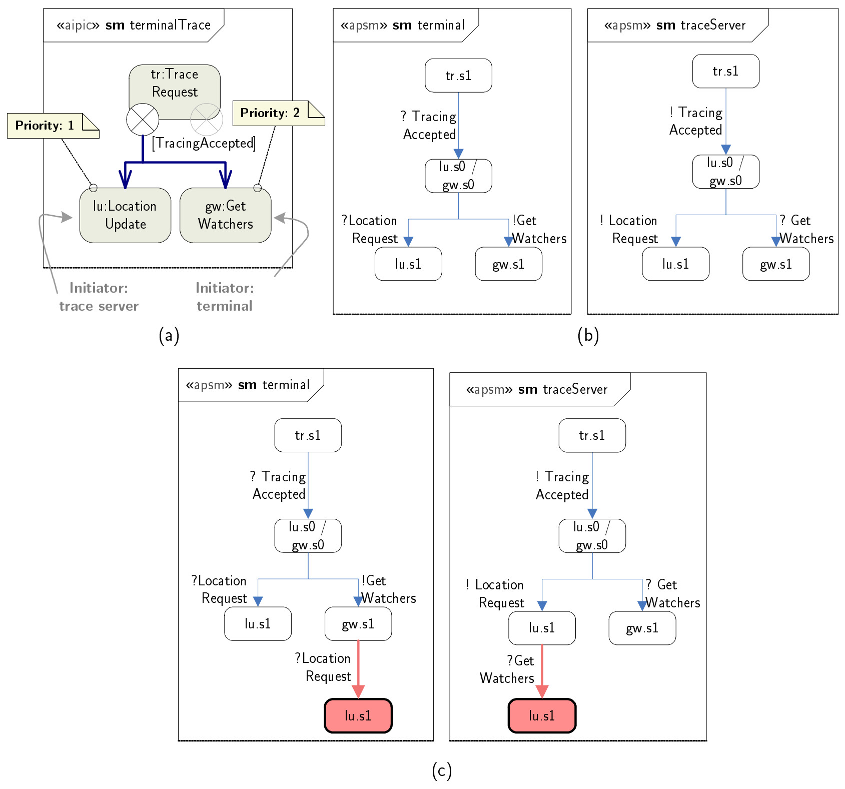


Figure 5.9: Conflict resolution during composition using priority. The combination defined in the AIPIC (a) will result in mixed initiative states in the APSMs after merging (b). This can be resolved through enforcing input consistency (c) and deciding the further behaviour based on the priority of the pattern instances.

initiative conflict based on this priority. The pattern instance with the highest priority "wins". This way we avoid having a coordinator and agreeing on the further behaviour at run-time, because this can be determined based on a fixed priority stated by the user at design-time.

Figure 5.9 is a continuation of Fig. 5.6 on page 54. In part (a) we have repeated the extract from the AIPIC, and part (b) shows the APSMs of the *terminal* and *traceServer* roles as we left them after the merging transformation. In the AIPIC we have indicated the priority given to the two pattern instances, and we have chosen to give *LocationUpdate* the highest priority. The following situation will cause a mixed initiative conflict: Consider that both APSMs reside in the mixed initiative state *lu.s0/gw.s0*. Then, the *terminal* decides to ask for which users it is currently being traced by, and starts the *GetWatchers* pattern instance by sending the signal (which also happens to be called) *GetWatchers*. However, before the *tracing server* receives this signal, it decides to request a location update from the terminal, and sends *LocationRequest*. As a result, the APSM of the *terminal* continues to state *gw.s1*, while the APSM of the *trace server* proceeds to *lu.s1*.

Part (c) of the figure shows the APSMs after the mixed initiative conflict has been resolved. Input consistency is enforced in the APSM of the *terminal* role by adding the reception of *LocationUpdate* to state *gw.s1*. We proceed to state *lu.s1* because we want to keep the effect of this received signal. In the APSM of *traceServer*, the reception of *GetWatchers* is added to state *lu.s1*. Because this signal belongs to the pattern instance with the lower priority, we ignore it and remain in the same state. The two APSMs will now interact consistently, as they reach the same state, *lu.s1*, after they have received the crossed signals.

The described solution to conflict resolution using priority is in fact quite satisfying. Due to the distributed nature of the services in our domain, it is impossible to synchronise the parts of the system and situations can occur where concurrent behaviours are initiated. During the design-phase of these services, we are very much aware of this characteristic. If, at a certain step in the interaction, two or more different things can be started in two different parts of the service, it stands to reason that the one who designs the service also prioritises them. There is no reason to develop sophisticated negotiation mechanisms to resolve the conflict during run-time, when the designer himself knows (or at least should know) the intention behind every interaction. Thus, we feel that this resolution scheme is both reasonable and sufficient.

Note that even though we only have two pattern instances involved in the conflict in this example, the number of pattern instances can be greater in general. As long as they all are prioritised, the resolving can be done similarly. If any of the pattern instances involved are assigned equal priority (or any of them lack a priority), it is considered to be a design flaw committed by the user. This can be identified and the user can be warned and requested to do the prioritising correctly.

Propagating input inconsistency As pointed out in section 2.4.3 on page 21, when we add signal reception to ensure input consistency after a mixed initiative state, we may actually introduce a new mixed initiative state. This can also happen during the APSM composition, and have consequences for the composition done in succeeding exit connection point references in the AIPIC.

Figure 5.10 shows an example of a situation where we have to repeatedly add new transitions to ensure input consistency. We illustrate a general concept, and thus only show the composition of one APSM. Part (a) shows an example AIPIC, while part (b) shows the APSM fragments of the pattern instances involved. We see that *PatternInstanceB* and *PatternInstanceC* has different initiators. Part (c) shows the composed APSM for the role named *role1*. We have removed ambiguities as previously explained, resulting in the merged state *piB.s0/piC.s0*. In *piB.s1* we have added the reception of signal *X* to ensure input consistency. The target state of this transition is not of importance. However, now we see that state *piB.s1* is a mixed initiative state. We added a signal reception, and it already defined two signal sendings. Because the interaction is to terminate after the sending of signal *H*, this will lead to

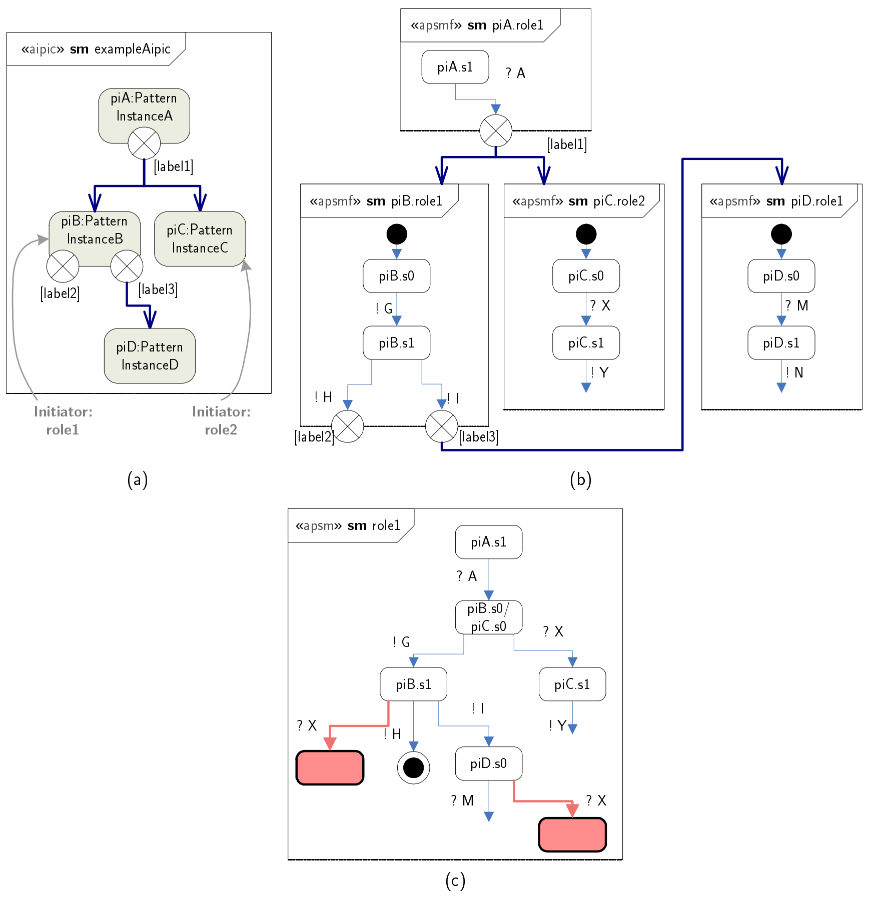


Figure 5.10: Propagating input inconsistency. The combination shown in (a) with the APSM fragments shown in (b) will result in propagating input inconsistency in the APSM (c). This can effect the composition in succeeding exit connection point references of the AIPIC or in an unwanted termination.

improper termination, which will be discussed shortly. After sending signal *I* in *piB.s1* we reach the state *piD.s0* where the reception of signal *M* from the fragment of *PatternInstanceD* is defined. We have to add the reception of signal *X* also here, because it is possible that *role1* sends both signal *G* and signal *I* before it receives signal *X*. In this latter situation, we see that the handling of the mixed initiative between *PatternInstanceB* and *PatternInstanceC* actually affects the insertion of the fragment of *PatternInstanceD*.

This phenomenon, which we call propagating input inconsistency, has to be taken into consideration when composing the APSMs from APSM fragments. When we add a new fragment, like *piD.role1* in the example above, we have to check if there are unresolved input inconsistencies in the preceding state. Since we require well-formed pattern instances, we know that any unresolved input inconsistency in the APSM has been introduced during composition because of this propagating phenomenon.

In section 2.4.3 on page 21 it was advised not to define successive signal sendings, but it is not illegal. It just makes the conflict resolution a bit more complicated and the resulting APSMs uglier and harder to understand. Especially when the input inconsistency propagates to succeeding APSM fragments, the final APSM will become less intuitive. We thus recommend avoiding this situation, but we do not forbid it.

Eventually, the input inconsistency will stop propagating. Most of the time this will happen when we reach a state where only signal reception is defined, like state *piD.s0* in Fig. 5.10. If not, it will cause an unwanted termination, like in state *piB.s1* in the same figure.

Unwanted or improper termination

Unwanted or improper termination occurs when we cannot refine the APSM to ensure input consistency after a mixed initiative, because the APSM terminates. This was elaborated in section 2.4.3 on page 21. During composition, we would like to handle this in such a way that both APSMs agree on terminating at the same time, and with the same termination condition.

Figure 5.11 shows an example of unwanted termination. Part (a) shows a portion of the AIPIC of *TerminalTrace*, where both *RevokeAllowance* and *LocationUpdate* can be started after *GetWatchers* has finished. The relevant APSM fragments are shown in part (b). In part (c) the two APSMs have been put together from these fragments. The state *ra.s0/lu.s0* is a mixed initiative state in both APSMs, but we cannot enforce input consistency to fix it. After receiving *RevokeAllowance* in the APSM of the *terminal* role, we cannot specify the reception of *LocationRequest* because the APSM terminates.

We solve the problem of unwanted termination by adding the exchange of another signal which will notify the other APSM of the termination. Floch uses such a notification signal to solve termination ambiguity after equivoque

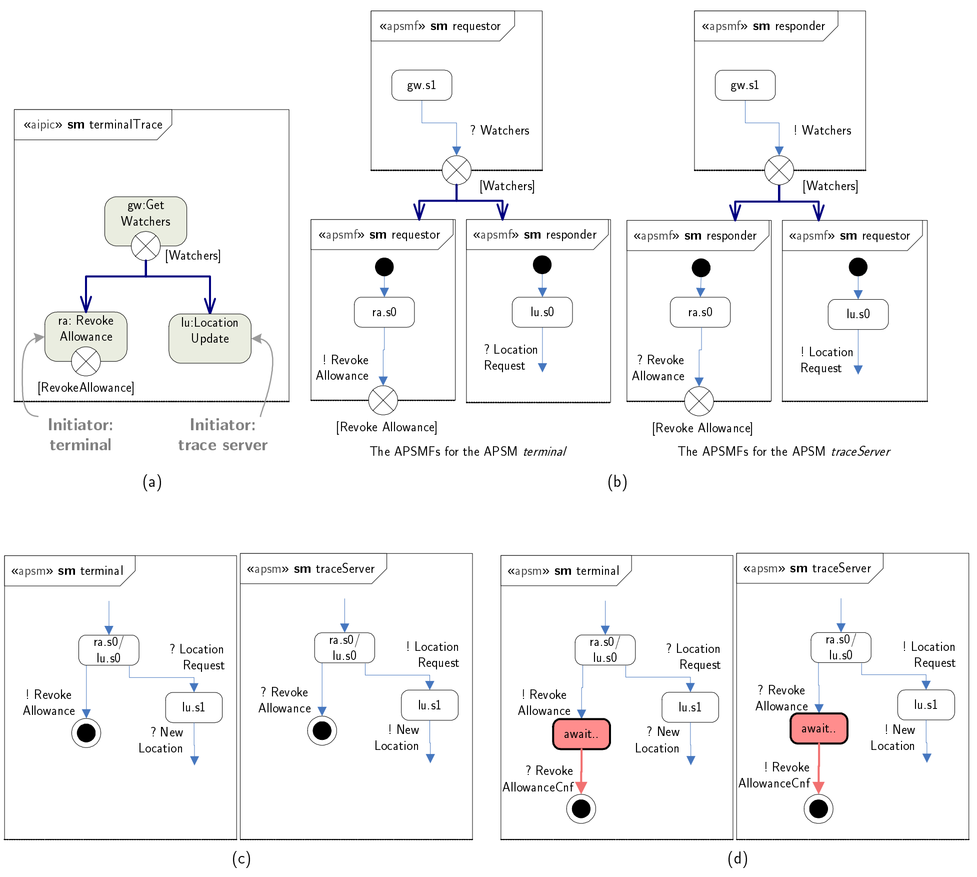


Figure 5.11: Removal of unwanted termination during composition. The combination in the AIPIC in (a), with the APSM fragments shown in (b), will result in an unwanted termination when composing the APSMs (c). This can be resolved by adding the exchange of a new signal to ensure that both APSMs terminate coordinated (d). Input consistency can now be enforced as normal.

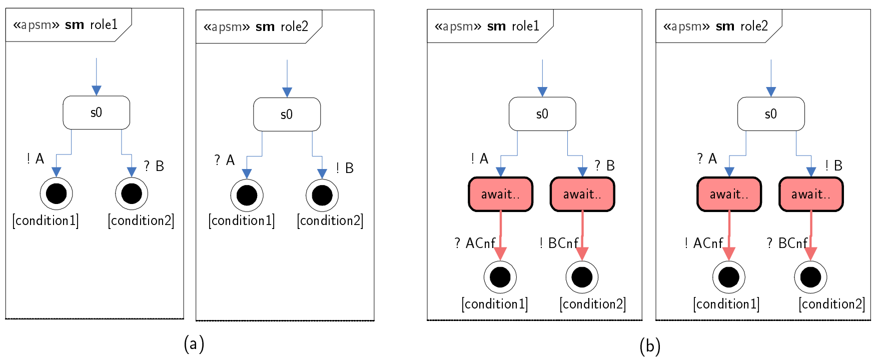


Figure 5.12: Removal of improper termination during composition. The case of improper termination in the APSMs (a) can be removed by adding the exchange of an extra signal for each of the termination possibilities (b). Input consistency can now be enforced as normal, and the APSMs will terminate with the same condition.

states [16, p. 176], but we use it to handle unwanted termination after mixed initiatives. Part (d) of Fig. 5.11 shows how we solve the unwanted termination in the APSMs of the two roles. A new state and the sending of a new signal have been added. The APSM of the *terminal* role will await a confirmation for *RevokeAllowance* before it terminates. The APSM of the *traceServer* will consequently send a new *RevokeAllowanceCnf*-signal to indicate that it terminates. The unwanted termination is now removed, because the APSMs no longer terminate directly after the mixed initiative. Input consistency can now be enforced and the mixed initiative conflict resolved.

Improper termination can be resolved similarly to the case of unwanted termination. The APSMs can terminate in more than one way, meaning with different conditions, after a mixed initiative. This can only happen if more than one of the pattern instances contains the exchange of one exactly one signal (like *RevokeAllowance* in Fig. 5.11). Figure 5.12 shows a general example of how improper termination is removed. In part (a) we see two example APSMs where the exchange of signals A and B lead to different termination conditions. If these two signals cross, then the APSMs will terminate with different conditions, resulting in an improper termination. To resolve this, we add the exchange of an additional signal for each of the termination possibilities, as shown in part (b) of the figure. This is quite similar to what we did to resolve unwanted termination, just "repeated" for each termination possibility succeeding the mixed initiative state. Input consistency can now be enforced to remove the mixed initiative conflict in state *s0*.

5.3.3 Summary

We have now completed a detailed study of the possible ambiguities and conflicts that we have to take into consideration when composing APSMs from APSM fragments. Table 5.1 sums up the main points of this discussion. The

Error	**Error handling**	**References**
Equivoque states		
Input ambiguity	RESOLVED DURING COMPOSITION Removed by the merging transformation.	Theory: 2.4.3 on page 20 Handling: 5.3.1 on page 53
Output divergence	RESOLVED DURING COMPOSITION Removed by the merging transformation.	Theory: 2.4.3 on page 20 Handling: 5.3.1 on page 53
Mixed ambiguity	RESOLVED DURING COMPOSITION Removed by the merging transformation.	Theory: 2.4.3 on page 20 Handling: 5.3.1 on page 53
Termination ambiguity	PREVENTED Avoided by restricting the composition possibilities of the pattern instances.	Theory: 2.4.3 on page 20 Handling: 5.3.1 on page 55
Termination condition ambiguity	PREVENTED Avoided by restricting the composition possibilities of the pattern instances.	Theory: 2.4.3 on page 20 Handling: 5.3.1 on page 55
Mixed initiatives: Concurrent behaviours		
Input inconsistency	RESOLVED DURING COMPOSITION Removed by enforcing input consistency and resolving conflicts based on prioritised pattern instances.	Theory: 2.4.3 on page 22 Handling: 5.3.2 on page 57
Unwanted or improper termination	RESOLVED DURING COMPOSITION Removed by adding the exchange of an additional confirmation signal. Then enforcing input consistency through prioritised pattern instances.	Theory: 2.4.3 on page 23 Handling: 5.3.2 on page 61
Mixed initiatives: Alternative orderings		
Further behaviour ambiguity	PREVENTED Avoided due to the requirement of distinct signal sets in the used pattern instances.	Theory: 2.4.3 on page 23 Handling: 5.3.2 on page 56

Table 5.1: Summary of error handling during composition. The table summarizes how the different errors are prevented or resolved, including references to more detailed discussions.

table shows which errors that are prevented, and which that are resolved during composition. In addition, it includes a reference to the theory of the error case, as well as to where the handling has been elaborated.

Chapter 6

APSM composition algorithm

This chapter will describe how we have designed the algorithm composing the APSMs, based on the discussion in the previous chapter. After a brief introduction, we will give a high-level overview of how the algorithm works, before we describe some details of the algorithm concerning how the error situations are resolved.

6.1 Introduction

The input of the composition algorithm is the interaction interface collaboration. It constructs the APSMs based on the applied interaction patterns and the desired composition of them as described in the AIPIC.

The APSM composition algorithm has the following preconditions to guarantee correct APSMs:

- The algorithm must be invoked with a valid interaction interface collaboration as input, as defined in section 4.1 on page 37.
- The AIPIC of the interaction interface collaboration is constructed based on the constraints defined in section 4.3.2 on page 43.
- The pattern instances applied stem from well-formed patterns, as defined in section 3.3 on page 32.
- The pattern instances make use of distinct sets of signals, as defined in section 4.2.3 on page 39.

These assumptions and constraints have been discussed earlier in this thesis, and the composition algorithm relies on that these have been obeyed.

6.2 High-level explanation

The composition algorithm traverses the AIPIC of the interaction interface collaboration recursively and builds the two APSMs. In its basic idea, the algorithm is quite simple. What makes it difficult is to be able to handle all the possible combinations and conflicts that may arise, as explained in 5.3 on page 51. In this high-level explanation we go through the overall flow of the algorithm without concerning special cases or errors. In the next section we will go deeper into some selected details of the algorithm.

We will call the initial state and the exit connection point references of the AIPIC *composition nodes.* Each such composition node is processed in turn and the APSM fragments related to the submachine states linked to this composition node are inserted into the APSMs. Let us go through how the algorithm works, referring to the numbering in Figure 6.1.

1. We start traversing the AIPIC from its initial state.
2. *Process composition node in AIPIC* is executed for each composition node in the AIPIC, which in the algorithm is named `aipicvertex`. The first time it is called with the initial state as `aipicvertex`, the remaining times the `aipicvertex` is an exit connection point reference.
3. We first get all submachine states attached to `aipicvertex`.
4. The number of submachine states determine the further action. If no submachine states are attached to the `aipicvertex`, then we handle the case of no succeeding interaction (see 5). Else, we handle succeeding interaction (see 6).
5. In the case of no succeeding interaction, we are finished with handling this `aipicvertex`.
6. If submachine states are found, we handle them one by one. An arbitrary of them is picked and called `sms`.
7. We find the pattern instance the submachine state `sms` represents, and retrieve its APSM fragments.
8. These fragments are inserted into the APSMs, while ensuring that error situations are identified and resolved.
9. This pattern instance has now been visited, and the exit connection point references of `sms` are fetched to start the recursive processing of them.
10. An arbitrary `ecpr` is picked from the list of exit connection point references of the `sms`.
11. We now call *Process composition node in AIPIC* with `ecpr` being the `aipicvertex`. This will trigger the processing of that `ecpr` (see 3).
12. When the processing of the `ecpr` is finished (either through 5 or 14), we check to see if there are more exit connection point references of this `sms` that have not been processed. If so, we return to 10.
13. When all exit connection point references of an `sms` are processed, we check to see if there still are unvisited submachine states attached to the `aipicvertex`. If yes, we return to 6.
14. When all submachine states attached to an `aipicvertex` are visited, the

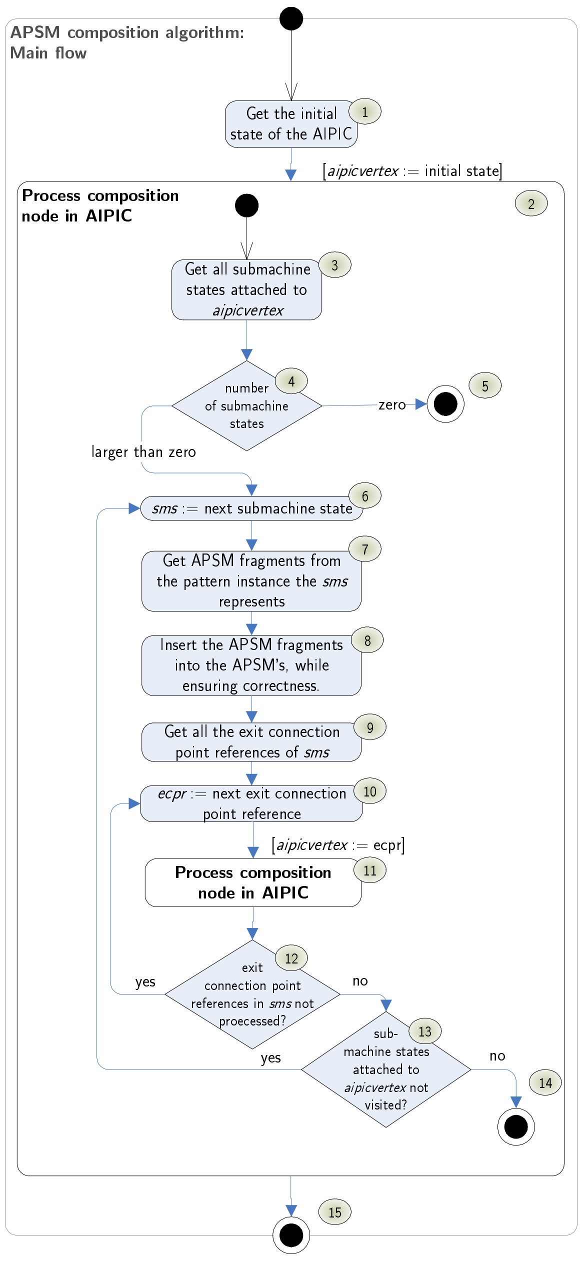

APSM composition algorithm: Main flow
Get the initial state of the AIPIC
1
[*aipicvertex* := initial state]
Process composition node in AIPIC
2
Get all submachine states attached to *aipicvertex*
3
number of submachine states
4
zero
5
larger than zero
sms := next submachine state
6
Get APSM fragments from the pattern instance the *sms* represents
7
Insert the APSM fragments into the APSM's, while ensuring correctness.
8
Get all the exit connection point references of *sms*
9
ecpr := next exit connection point reference
10
[*aipicvertex* := ecpr]
Process composition node in AIPIC
11
exit connection point references in *sms* not proecessed?
12
yes
no
sub-machine states attached to *aipicvertex* not visited?
13
yes
no
14
15

Figure 6.1: Composition algorithm: high-level flowchart. The recursive composition algorithm traverses the AIPIC and gradually builds the APSMs of the interaction interface collaboration.

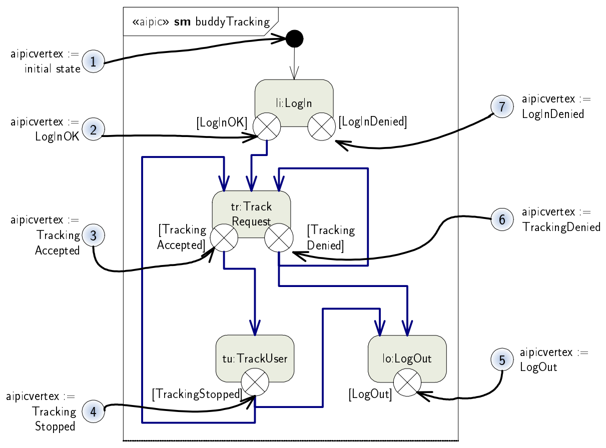


Figure 6.2: Composition algorithm: example traversal of an AIPIC. The AIPIC of *Buddy Tracker* will be traversed in a depth-first manner.

processing of this `aipicvertex` is done.

15. The whole algorithm stops when all the composition nodes in the AIPIC have been processed, and thus all the submachine states have been visited.

The composition algorithm is a depth-first search (DFS) algorithm [33, p. 165]. We also remember which submachine states we have visited in order to handle loops in the AIPIC. Figure 6.2 illustrates how the composition algorithm could traverse the AIPIC of *Buddy Tracking*. This is only one of many possible traces of the algorithm, since we arbitrary select which submachine state to visit first, and which exit connection point reference to process first. The numbers in the figure shows a possible sequence of processing the composition nodes[1]. We see that the sequence reflects the depth-first-nature of the algorithm.

During the traversal of the AIPIC, and insertion of the APSM fragments, we always keep track of where in the APSMs the fragments of the succeeding interactions are to be attached. This is done by keeping a reference to a state in each APSM where the first signal exchanges in the succeeding fragments are to be started. This, and more, will be elaborated next.

6.3 Selected details concerning error handling

In the following we will explain some details of the composition algorithm based on the overview presented in the previous section. We will extract portions from the complete algorithm to explain how we resolve the ambiguities and conflicts we have seen can occur during the composition of the APSMs. First, we explain how we insert the actual APSM fragments into the APSMs, as this will influence how the error situations are identified and resolved. Then, we

[1]Please note that the numbering in this figure has nothing to do with the numbering in Fig. 6.1.

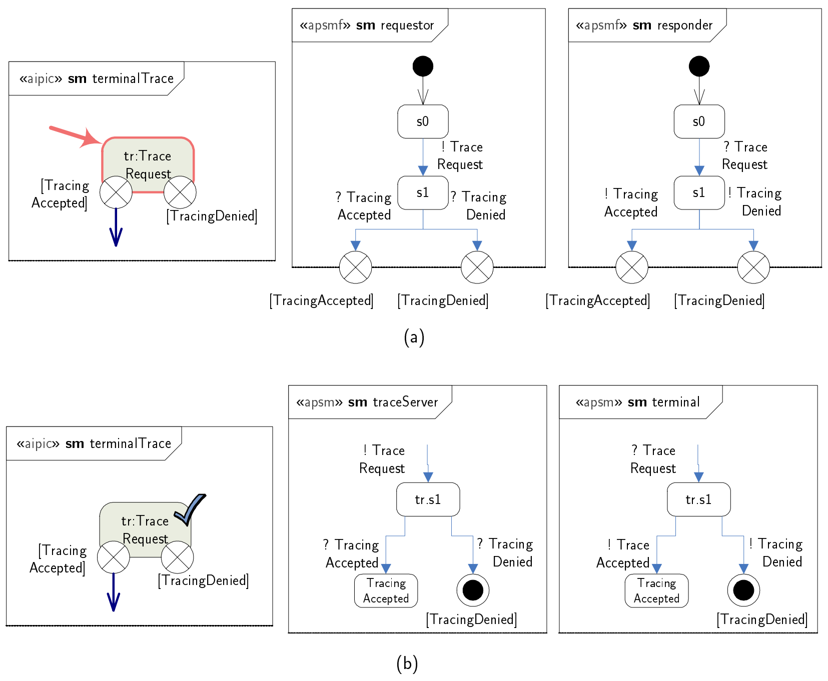


Figure 6.3: Composition algorithm: insertion of APSM fragments. The APSM fragments of *TraceRequest* are inserted into the APSMs depending on the composition in the exit connection point references in the AIPIC (a). The *TracingAccepted* exit point is inserted as a simple state due to further interaction in the AIPIC, while *TracingDenied* is inserted as a final state because no succeeding interaction is defined in the AIPIC (b).

will explore how the algorithm resolves ambiguities and how we identify and resolve mixed initiative conflicts.

6.3.1 Insertion of APSM fragments into the APSMs

Each APSM fragment will consist of a number of simple states and transitions, and eventually end with one or more exit points. As we know, these exit points are reflected as exit connection point references in the AIPIC. Adding transitions and simple states to the APSMs is a rather simple task. However, what we choose to do when we reach the exit point(s) of the APSM fragments is very important for the way we resolve the ambiguities and conflicts in our algorithm. This is because it is where the pattern instances, and thus the fragments, terminate that the composition will be done and the composition error situations occur.

When we reach an exit point in the APSM fragment, we determine what kind of vertex to insert into the APSMs based on the information contained in the AIPIC. This depends on the composition case of the exit connection point reference corresponding to the exit point in question. Let us illustrate this with an example. Figure 6.3 shows how we insert the APSM fragments of the

TraceRequest pattern instance into the APSMs of the *TerminalTrace* interaction interface collaboration. In part (a) show an extract from the AIPIC showing the submachine state of this pattern instance. The exit connection point reference *TracingAccepted* leads to succeeding interaction, while *TracingDenied* has no succeeding interaction. The APSM fragments of the pattern instance are also shown. In part (b), we see the APSMs of the two collaboration roles of *TerminalTrace* after the fragments in (a) has been inserted. The exit point *TracingAccepted* from the APSM fragments of (a) have been inserted as a simple state in the APSMs. This is because we know from the AIPIC that we sometime later are going to attach additional fragments at this point in the APSM. We thus create a simple state with the name of the exit point, and the fragments of succeeding pattern instances can later be attached to this state. The exit point *TracingDenied* is inserted into the APSMs as a final state. We do so because we know that the APSMs are to terminate due to the lack of succeeding interactions, and no other fragments are ever going to be inserted here.

The method implies scowling at the succeeding interaction of the pattern instance during the insertion of its APSM fragments. A more coarse approach of inserting all exit points as either final states or simple states would mean that we have to replace them when additional fragments are to be added later, or we are to terminate the APSMs, respectively. In the following, we will see that the chosen method will make things easier when we are faced with error situations during the APSM composition.

6.3.2 Resolving ambiguities

Ambiguities are in fact removed as a consequence of the chosen way of adding the exit points of the APSM fragments. Figure 6.4 exemplifies this. In part (a) we continue where we left off in Fig. 6.3. The fragments of *TraceRequest* have been added to the APSMs *traceServer* and *terminal*. The composition algorithm is currently in the exit connection point reference *TracingAccepted* in the AIPIC, and the states named *TracingAccepted* in the APSMs are where the succeeding fragments are to be attached. This follows from the discussion above. We also note that the initiators of *LocationUpdate* and *GetWatchers* are different, which means that the ambiguity we deal with is mixed ambiguity. In part (b), the algorithm inserts the fragments of the *LocationUpdate* pattern instance into the APSMs. They are attached to the *TracingAccepted*-states. In part (c), we add the fragments of *GetWatchers*. They are also attached to the same state in the APSMs. We see that we have avoided the ambiguity. Our algorithm actually merges implicitly due to the way we insert the APSM fragments and traverse the AIPIC.

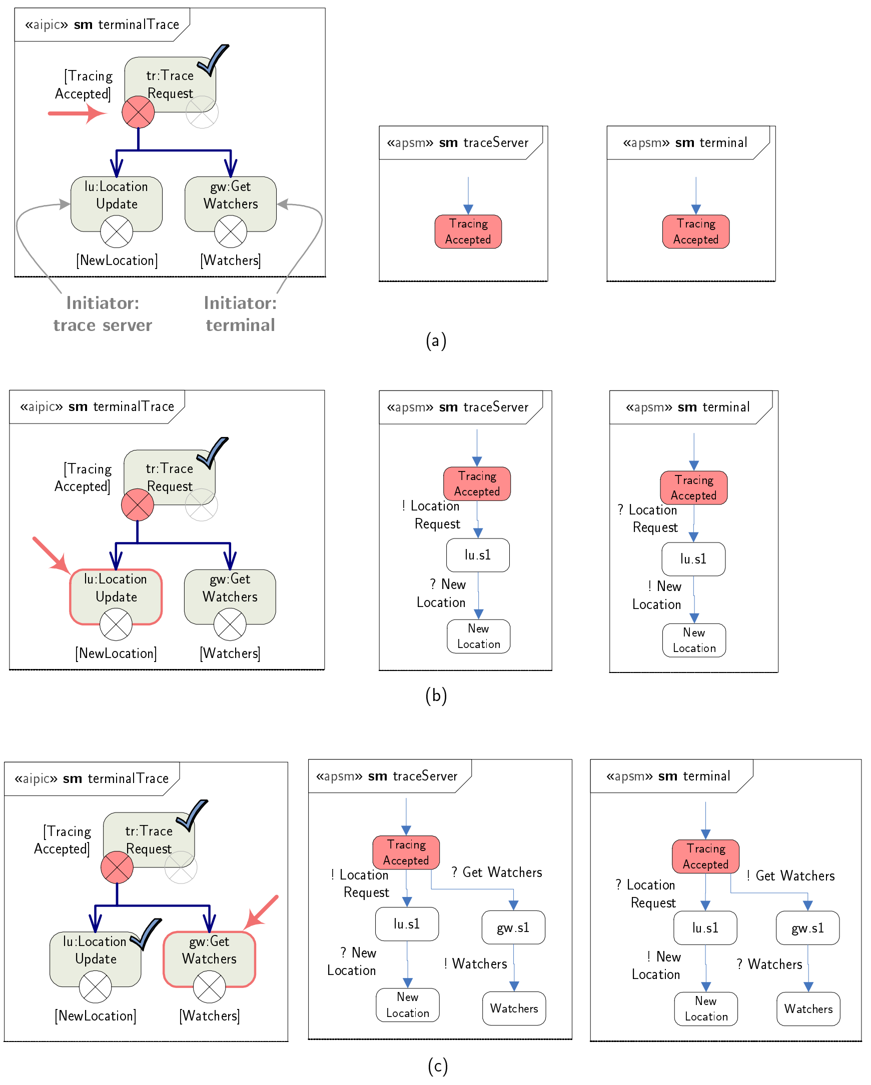


Figure 6.4: Composition algorithm: handling ambiguities. The figure shows how the algorithm handles ambiguities during the insertion of the fragments into the APSMs. In (a) the algorithm has processed *TracingAccepted* in the AIPIC and the further interaction is to be attached to states *TracingAccepted* in the APSMs. In (b) we add the fragments of *LocationUpdate*, attached to these states. The fragments of *GetWatchers* are then inserted in (c), attached to the same state. No ambiguity is present in the APSMs due to the way the algorithm is defined.

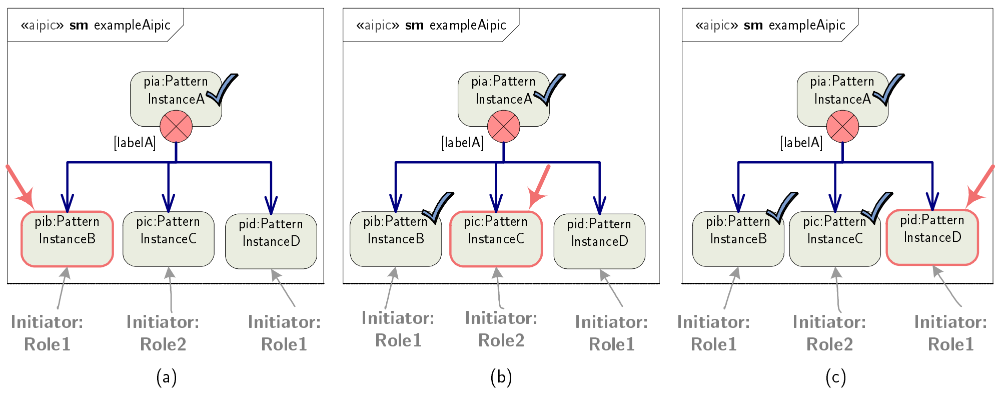


Figure 6.5: Composition algorithm: determining when to resolve mixed initiatives. When visiting *PatternInstanceB* (a), the algorithm will identify the mixed initiative but not handle it. When *PatternInstanceC* is inserted (b), the mixed initiative with the visited *PatternInstanceB* will be resolved, but not the conflict with *PatternInstanceD*. The latter will be resolved later when inserting the fragments of *PatternInstanceD* (c).

6.3.3 Resolving mixed initiatives

To resolve mixed initiatives in our algorithm, we first have to identify when they occur and when we can do something about it. Every time we insert the APSM fragments of a pattern instance into the APSMs, we check for mixed initiatives between that pattern instance and each of the other pattern instances that might be activated at the same time. But, we can only resolve the mixed initiative if the submachine state in the AIPIC representing the other pattern instance has been visited. The reason for this is that to resolve the mixed initiative conflict between two pattern instances, the fragments of both of them have to be inserted into the APSMs. If the other pattern instance has not been visited yet, we delay the handling of the mixed initiative conflict until then.

In Figure 6.5 we have a situation where three pattern instances can be activated at the same time. The algorithm is processing the exit connection point reference *labelA* in the AIPIC. In part (a) we have inserted the fragments of *PatternInstanceB*. Now, we check for a mixed initiative conflict. *PatternInstanceC* is initiated by *Role2*, and we therefore have a mixed initiative between these two. But, *PatternInstanceC* has not been visited yet and we ignore it. In part (b) the time has come to *PatternInstanceC*. It is in conflict with both *PatternInstanceB* and *PatternInstanceD*. Since *PatternInstanceB* has been visited, we resolve the conflict now. Finally, in part (c), we have inserted the fragments of *PatternInstanceD*, and discover that it has a mixed initiative conflict with *PatternInstanceC*. This can be resolved now, because the latter pattern instance has been visited.

One might find it a bit strange that we speak of "delaying the handling of the mixed initiative conflict". Let us return to Fig. 6.4 to explain why this really is quite natural. When we visit *LocationUpdate* (part (b)), and have inserted its fragments into the APSMs, we can identify a mixed initiative conflict with *GetWatchers*. Following the procedure just described, we do nothing about this. We see from the APSMs in the figure that this is completely legitimate, as we have not introduced any mixed initiative in the APSMs yet. This does not happen before we visit *GetWatchers* in part (c) of the same figure. When we insert the fragments of the second of the two pattern instances involved in a conflict, the mixed initiative is introduced in the APSMs. This is therefore when we need to resolve the conflict.

We have now identified the mixed initiatives and also determined if we can handle the conflict at this point in the algorithm. Now we have to check if we can enforce input consistency, or if termination is involved and prevents us from doing so. If a final state succeeds directly after the mixed initiative state in the APSMs, the exchange of an extra signal is inserted as described in section 5.3.2 on page 61.

Based on the priority of the pattern instances involved in the conflict, we insert additional transitions to ensure input consistency and thus resolve the mixed initiative. This is done according to the procedure previously elaborated, and we refer to section 5.3.2 on page 57 for an explanation of the procedure.

The case of so-called propagating input inconsistency is also taken into account in the algorithm. After the fragments have been inserted, and any mixed initiative caused by the pattern instance composition have been resolved, we check for propagating input inconsistency and handle this according to the procedure presented earlier in section 5.3.2 on page 59.

This concludes the discussion of the designed composition algorithm. If further details are desirable, we refer to the actual implementation of the algorithm, which includes explanatory comments to ease the understanding of its mode of operation[2]. The implementation done in our work is also the next stop in this thesis.

[2]See appendix C on page 115 for information on how to get hold of the source code.

Chapter 7

Implementation

In this chapter, we describe the implementation done in our work, starting with a brief overview. Next, we explain the realisation of the interaction patterns, before the tasks of applying and composing interaction patterns in the tool are described. The composition of the APSMs is then described, before we say some words about the usability of the implemented approach.

7.1 Overview

The implementation is realised as an Eclipse plug-in named *no.ntnu.item.-ramses.interactionpattern.* It extends the functionality of the Ramses tool suite with the pattern-based approach for creating interaction interfaces. Appendix C on page 115 lists its requirements.

The plug-in is organised into sub-packages. These are:

- **Algorithms** - the algorithm defined for APSM composition
- **Exceptions** - exceptions that can be thrown by the plug-in
- **Genericpatterns** - the interaction patterns identified and described in our work
- **Helpers** - helper classes for accessing and creating UML elements
- **Internal** - further divided into:
  - **Actions** - menu actions for invoking the wizards and the composition algorithm
  - **Plugin** - plug-in information needed by Eclipse
  - **Wizards** - even further divided into:
    - **Collaboration** - creating a new interaction interface collaboration
    - **Link** - composing pattern instances applied to the interaction interface collaboration
    - **Patterninstance** - applying patterns to the interaction interface collaboration
- **Patternframework** - the pattern library, the interface for interaction patterns and other classes related to patterns in general

The rest of this chapter will explain the plug-in in more detail. We will go through the implementation in roughly the same order as with the modelling in the previous chapters. This means that we first start with the definition of interaction patterns, then continue with how to apply and compose them, before we describe the APSM composition.

7.2 Interaction patterns

This section will describe the implementation of the interaction patterns. First, we discuss how the interaction patterns are represented, before we describe the interface and extension point we have defined.

7.2.1 Representing interaction patterns

The interaction patterns can either be implemented as model resources or Java code. The first alternative is totally in line with the modelling of the interaction patterns as presented in chapter 3 on page 29. The patterns are represented in XMI and thus stored as UML model elements. Each interaction pattern will consist of an XMI-file defining it as a collaboration template. Using Java code, the pattern itself is not stored as a model element, but we create the pattern instances element for element using the UML2 repository of Eclipse. This means that we have a piece of code that constructs a pattern instance based on the user input using the factory methods of the repository. The interaction patterns are thus only a recipe for constructing the pattern instances of that particular interaction pattern.

We have chosen to implement the interaction patterns in Java code. This makes us independent of the actual representation of the model elements. Using the UML2 repository we can create and change model elements through Java methods, and do not need to know that they are actually stored in XMI. If the patterns were defined as model resources, they would be vulnerable to any changes affecting the XMI format which we have no control over. Accessing and managing model resources are also a complicated task, and it is therefore better to utilise the UML2 repository which can do exactly this for us. Customisation of the pattern instances based on the user input is also easier with the chosen solution, because the input can be taken into account when building a pattern instance element for element. If the patterns were stored as model resources we would have to read them and use them as a foundation for creating the tailored pattern instances. Because we cannot simply copy a pattern and use it in any situation, some Java code will always be necessary for customisation. With the pattern described in Java code as well, we only need one Java file for each pattern.

```
public interface InteractionPattern{

        public PatternDescription getPatternDescription();
        public List getPatternParameters(String instanceName);
        public Collaboration getPatternInstance(Package inThisPackage);

        public Property getRole1();
        public Property getRole2();

        public String getName();
        public String getNameOfRole1();
        public String getNameOfRole2();
}
```

Listing 7.1: The *InteractionPattern* interface. The listing shows the methods of the interface all interaction patterns have to implement

7.2.2 Interface for patterns

We have defined the interface *InteractionPattern*, which all interaction patterns must implement. Listing 7.1 shows the methods declared for this interface. The method `getPatternDescription` retrieves the description of the interaction pattern. An own class called *PatternDescription* has been defined to contain the elements of this description. Next, `getPatternParameters` will return a list with the pattern parameters. The input parameter *instanceName* is the name of the pattern instance the user is about to create, and is stored until the actual pattern instance is created. Through the method `getPatternInstance`, the pattern instance is retrieved, which is a *Collaboration*. The package which the pattern instance is to be contained in has to be included as an input parameter. This follows from the way the UML2 repository creates model elements, requiring them to be associated with a package when they are constructed. The remaining five methods in the interface are rather self-explanatory, and are plain getter methods. In section 7.3.2 on page 83 we will see how we use the interface when selecting, customising and instantiating interaction patterns.

An abstract class *AbstractInteractionPattern* has also been defined, which implements *InteractionPattern*. When defining the Java code for new interaction patterns this class can be extended, instead of implementing the interface directly. This will make it easier to construct the code, as a lot of variables and methods relevant for all interaction patterns have been implemented already. This allows one to concentrate on making the code specific for each interaction pattern.

7.2.3 Extension point for patterns

To provide flexibility and extensibility, we have defined a so-called extension point in Eclipse for interaction patterns. This means that additional interaction patterns can be defined in own plug-ins. They just have to be attached to the extension point, and implement the interface *InteractionPattern*. This gives a

```
<extension-point id="pattern" name="Pattern" schema="schema/pattern.exsd"/>
```

Listing 7.2: Declaration of the *pattern* extension point. Extract from the `plugin.xml` defining the extension point *Pattern*, which is further defined in the schema in Figure 7.1.

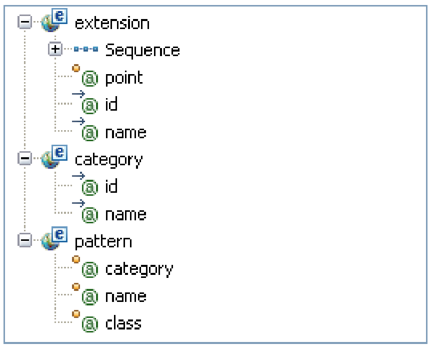


Figure 7.1: The schema for the *pattern* extension point. Two elements are defined in the extension point, namely *category* and *pattern*.

loose coupling between the pattern-based approach, and the actual interaction patterns. We thus allow other people to add their own, possibly domain or service specific, interaction patterns to the pattern library.

Listing 7.2 shows how the extension point is defined in the `plugin.xml`-file (the manifest file) of our plug-in. We see that the extension point is given the id `pattern`, and it is further defined in a so-called extension point schema named `pattern.exsd`. A schema specifies the elements and attributes of an extension point, and thus prevents users from making errors when they attach extensions to the extension point [17, p. 269]. Figure 7.1 gives an overview of our schema. We have defined two elements, a *category* and a *pattern*, each described below.

It is possible to contribute categories for the interaction patterns. This element of the extension point has the attributes *id* and *name*. Listing 7.3 shows how to use this element of the extension point. First, in line 1, we declare the extension point we are going to attach to. Then, we specify a new category with the ID `generic` and named `Generic Patterns`. The category is now added.

```
1  <extension id="pattern" point="no.ntnu.item.ramses.interactionpattern.pattern">
2     <category id="generic" name="Generic patterns"/>
3  </extension>
```

Listing 7.3: Using the *pattern* extension point to add a category. A new category `Generic Patterns` is added to the library through the defined extension point.

```
<extension id="pattern" point="no.ntnu.item.ramses.interactionpattern.pattern">
    <pattern category = "generic"
          class="no.ntnu.item.ramses.interactionpattern.genericpatterns.SubscribePattern"
          name="Subscribe"/>
 </extension>
```

Listing 7.4: Using the *pattern* extension point to add a pattern. A new interaction pattern `Subscribe` is added to the library through the extension point.

Returning to Fig. 7.1, we see that the *pattern* element has three attributes. First we have the category and the name of the pattern. The *class* attribute is used to specify the actual Java class with the code that makes up the interaction pattern. Listing 7.4 shows how we use the extension point to add the SUBSCRIBER-pattern to the library. Again, first we have to declare the extension point in question. Then we say that this pattern is to be included in the generic category (referring to its ID). The value of the `class`-attribute is the path of the Java class of the pattern. Finally, we name it `Subscribe`. As we can see from these examples, it is quite easy to contribute with new interaction patterns to the pattern library through this extension point.

The extensions (that is the patterns and the categories) are loaded the first time a user wants to apply a pattern. This way we delay the loading as much as possible, and we avoid loading them at all if the user is not going to create any pattern instances.

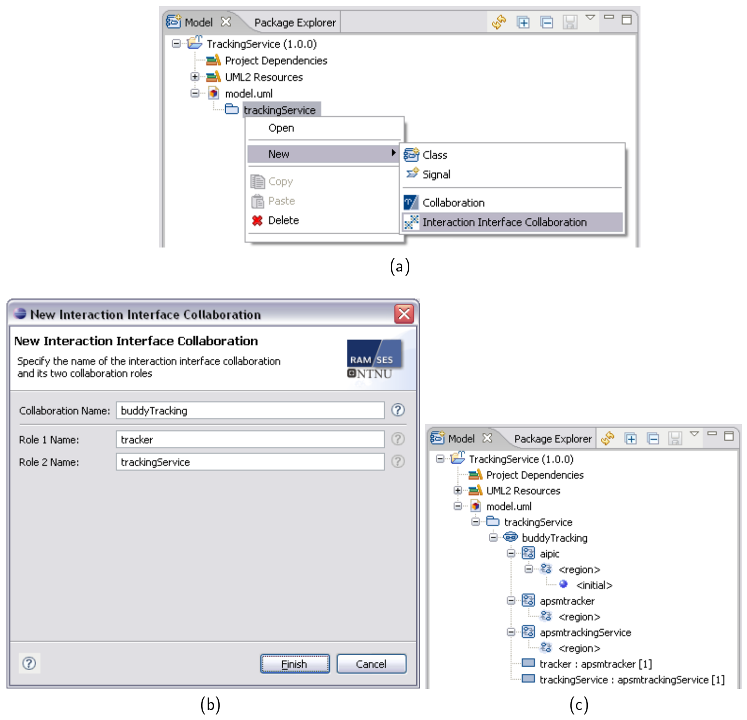
7.3 Applying and composing interaction patterns

Now that we know how the interaction patterns are implemented, we can start explaining how we have implemented the process of applying and composing them. The last step of the approach, generating the APSMs, will be described in the next section.

7.3.1 Defining the interaction interface

A user can define the interaction interface using a small wizard page. The wizard is started from a menu action, and will produce an interaction interface collaboration with all required elements based on the user input. This wizard, and all other wizards we have made, is implemented using a standard framework provided by Eclipse. The menu actions are integrated through an extension point *org.eclipse.ui.popupMenus* of Eclipse.

Figure 7.2 shows several screenshots from the implemented tool support for creating interaction interfaces. Part (a) shows the menu item that triggers the wizard. It is enabled for elements in the model view of the type *Package*, and



New Interaction Interface Collaboration

Specify the name of the interaction interface collaboration and its two collaboration roles

Collaboration Name: buddyTracking

Role 1 Name: tracker

Role 2 Name: trackingService

Figure 7.2: Creating an interaction interface collaboration. The user can make a new interaction interface collaboration through the pop-up menu when selecting a package in the model view (a). The name of the collaboration and its roles are entered in a small wizard (b), and the collaboration is created with all the correct elements (c).

added to the sub menu *New*. The package selected defines where the interaction interface collaboration is to be added. Part (b) shows the wizard window. The user has to provide the name of the collaboration, as well as the names of the two roles. Once this is finished, a new interaction interface collaboration will be created inside the package, as shown in part (c). The wizard has created the collaboration *buddyTracking* and its three state machines. The AIPIC is just called *aipic*, while the APSMs are given the same name as the roles they are tied to, prefixed with *apsm*. *Tracker* and *trackingServer* are the collaboration roles of the interaction interface collaboration and are typed with each of the APSMs.

7.3.2 Applying an interaction pattern

An interaction pattern is applied using a three-page wizard:

- **Page 1 - Selecting pattern:** The user selects the desired interaction pattern from a categorised list of installed patterns.
- **Page 2 - Customising pattern:** The user provides the name of the pattern instance and binds the pattern roles to the collaboration roles of the interaction interface collaboration. If the selected pattern has any

pattern alternatives, the user has to decide on these.

- **Page 3 - Binding pattern parameters:** The pattern parameters of the selected pattern are assigned values. Signals are created using a small pop-up window.

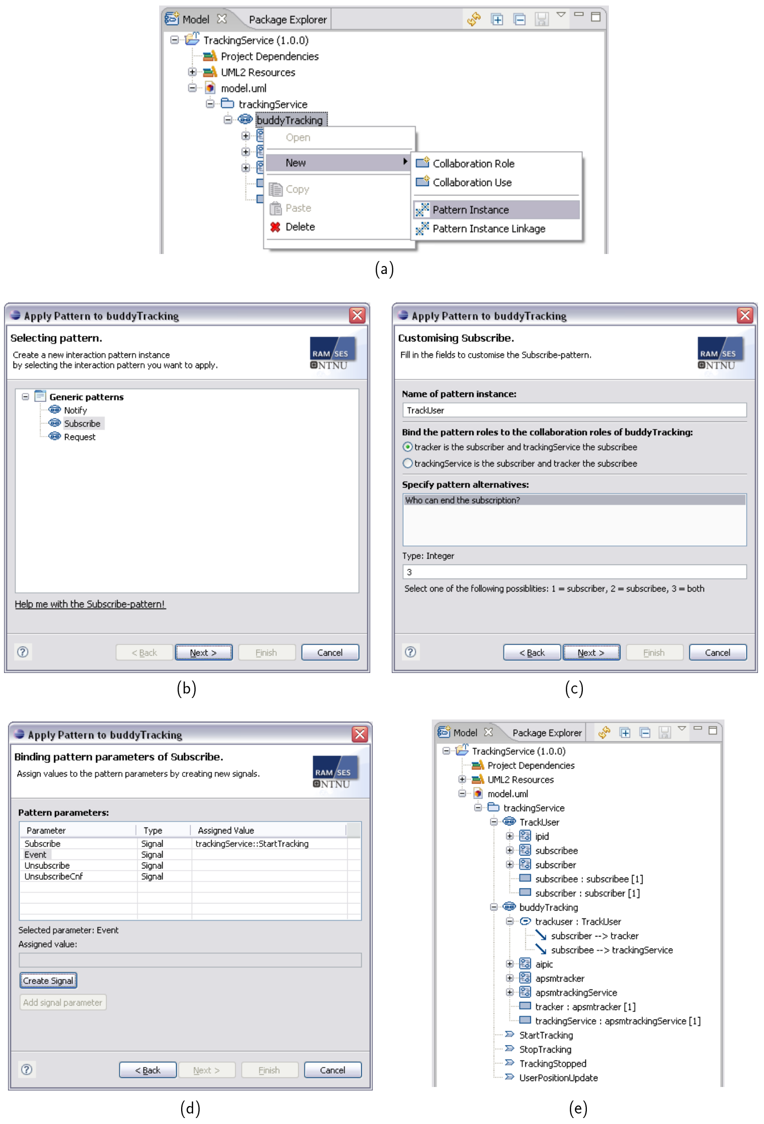
Figure 7.3 shows a series of screenshots from the process of applying an interaction pattern instance in the tool. In part (a) we see that a new pattern instance can be added by selecting an interaction interface collaboration and choosing from the pop-up menu: *New → Pattern Instance*. This identifies the context and the new pattern instance we are to create will be applied to this collaboration.

Page 1 - Selecting interaction pattern. The first page of the wizard displays all installed interaction patterns, as shown in part (b) of the figure. This is done in a simple tree-structure where the patterns are sorted according to their category. If more information about a pattern is desired, the link at the bottom of the wizard can be used to retrieve an explanation (for instance the complete or parts of the pattern description) of the currently selected pattern[1]. The user chooses the interaction pattern to apply proceeds to the next page.

Page 2 - Customising pattern. The customisation page is shown in part (c) of the figure. The user is first asked to provide a name for this pattern instance. As the user types, we check if the name is in line with Java name conventions and that the name is not equal to an already existing pattern instance. Second, the pattern roles are to be bound to the roles of the interaction interface collaboration. We have chosen to do this rather simple using radio buttons, since we only have two possibilities. In this case, the collaboration role *tracker* can either be the *subscriber* or the *subscribee* of the pattern, and the tracking server has to be the opposite. The last thing we have to do on this page is to specify the pattern alternatives. Since the SUBSCRIBE-pattern has one alternative, it is displayed in the list. If a pattern has many alternatives they are added to the list and can be selected and specified one at a time. For a pattern with no alternatives, this page ends with the role binding section. In the SUBSCRIBE-pattern we have defined the alternative *"Who can end the subscription"* as a number, which can be assigned the values 1, 2 or 3 as described at the bottom of the wizard page. The user inputs the desired alternative, before pressing *Next* to enter the last page of the wizard.

Page 3 - Binding pattern parameters. Page three of the wizard is shown in part (d) of the figure and concerns assigning values to the pattern parameters. All parameters are displayed in a table, and can be selected one-by-one to assign values. Pressing the *Create Signal*-button causes a simple pop-up wizard to appear, where the name of the signal can be entered (not shown in the figure).

[1]We have only prepared for this functionality, and not implemented it due to time constraints.



TrackingService (1.0.0)
Project Dependencies
UML2 Resources
model.uml
trackingService
buddyTracking

Apply Pattern to buddyTracking

Selecting pattern.
Create a new interaction pattern instance
by selecting the interaction pattern you want to apply.

Generic patterns
Notify
Subscribe
Request

Apply Pattern to buddyTracking

Customising Subscribe.
Fill in the fields to customise the Subscribe-pattern.

Name of pattern instance:
TrackUser

Bind the pattern roles to the collaboration roles of buddyTracking:
[✓ tracker is the subscriber and trackingService the subscribee]
trackingService is the subscriber and tracker the subscribee

Specify pattern alternatives:
Who can end the subscription?

Type: Integer
3
Select one of the following possiblities: 1 = subscriber, 2 = subscribee, 3 = both

Apply Pattern to buddyTracking

Binding pattern parameters of Subscribe.
Assign values to the pattern parameters by creating new signals.

Pattern parameters:

Parameter	Type	Assigned Value
Subscribe	Signal	trackingService::StartTracking
Event	Signal	
Unsubscribe	Signal	
UnsubscribeCnf	Signal	

Selected parameter: Event
Assigned value:

TrackingService (1.0.0)
Project Dependencies
UML2 Resources
model.uml
trackingService
TrackUser
ipid
subscribee
subscriber
subscribee : subscribee [1]
subscriber : subscriber [1]
buddyTracking
trackuser : TrackUser
subscriber --> tracker
subscribee --> trackingService
aipic
apsmtracker
apsmtrackingService
tracker : apsmtracker [1]
trackingService : apsmtrackingService [1]
StartTracking
StopTracking
TrackingStopped
UserPositionUpdate

Figure 7.3: Applying an interaction pattern. The wizard for adding a new pattern instance can be invoked from the pop-up menu of the interaction interface collaboration (a). On the first page of the wizard (b), the desired interaction pattern is selected. Then the pattern instance is customised (c), before the pattern parameters are bound to newly created signals (d). The pattern instance is now created and used properly in the interaction interface collaboration (e).

The signals can also be given signal parameters, using the *Add signal parameter*-button, but this can also be specified any time later. After creating a signal for each of the parameters, the *Finish*-button will be enabled and the customised pattern instance can be applied.

The last piece of the figure, part (e), shows the model view of after this pattern instance has been applied. As we can see, the pattern instance *TrackUser* has been created with its three state machines. The IPID is simply called *ipid*, while the APSM fragments are given the same name as the roles they type. The contents of these state machines is not shown in the figure. In *buddyTracking*, we see that a collaboration use *trackUser*, pointing to the pattern instance *TrackUser*, is added, and it contains the binding of the roles as we specified in the wizard. In the AIPIC, a submachine state *trackuser* has been added, containing an exit connection point reference *TrackingStopped*. This reflects the IPID of *trackuser*, as we argued in section 4.2.3 on page 41. Finally, we see that the signals are added to the same package as the interaction interface collaboration. If wanted, signal parameters can now be added to these signals, as with any other signal in Ramses.

The wizard for applying a pattern is shared for all interaction patterns, and contents of the pages of this wizard can change according to each pattern. The opposite solution is to equip every pattern with its own little pattern-specific wizard, which was done for instance in the SPT (section 2.5.2 on page 27). However, we do not find this satisfying, as this means that when we define a new interaction pattern, we also have to create a new wizard. This is not necessary when we have a standard format for the interaction patterns, and a wizard smart enough to prepare the pages according to the information contained in the description of each pattern. This is in line with our goal of separating the pattern-based approach from the actual interaction patterns.

How the wizard works

Up to now, we have shown the process of applying a pattern from the user's point of view. We will now describe some main points concerning how this wizard is implemented. Figure 7.4 tries to illustrate how the wizard retrieves the information it displays to the user. It also shows how we use the *InteractionPattern* interface described in section 7.2.2 on page 77. To the left we show the user actions triggering the method calls. The *PatternLibrary* is a singleton containing all the installed patterns, organised into categories. The interaction pattern object *selectedPattern* is used to illustrate the selected pattern.

Page 1 - Selecting interaction pattern. When the wizard is invoked, the pattern library is asked for all its categories. Then, we retrieve the

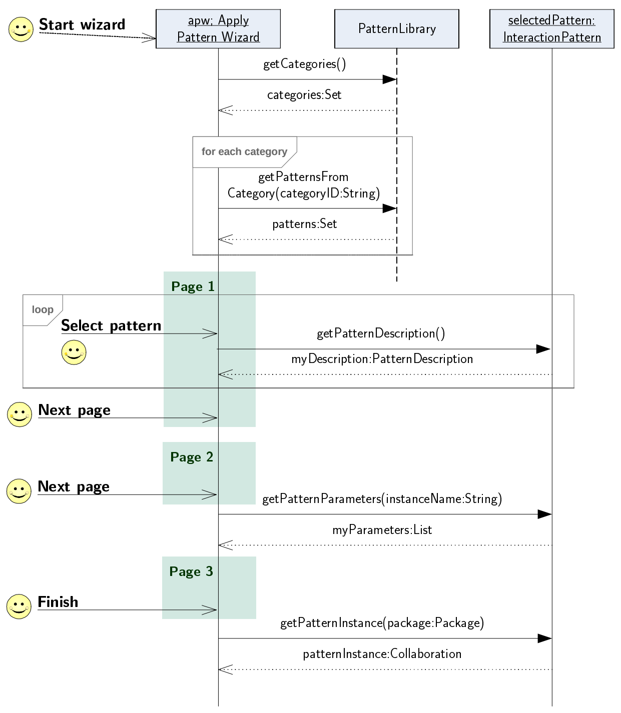

Start wizard
apw: Apply Pattern Wizard
PatternLibrary
selectedPattern: InteractionPattern
getCategories()
categories:Set
for each category
getPatternsFrom Category(categoryID:String)
patterns:Set
Page 1
loop
Select pattern
getPatternDescription()
myDescription:PatternDescription
Next page
Page 2
Next page
getPatternParameters(instanceName:String)
myParameters:List
Page 3
Finish
getPatternInstance(package:Package)
patternInstance:Collaboration

Figure 7.4: How the *apply pattern wizard* works. The wizard prepares and shows its pages based on the information received from the *Pattern library* and the *selectedPattern* through a series of method calls.

set of patterns for each of the categories, which we display in the tree structure in the wizard window. When the user selects a pattern, the pattern description is retrieved through the method `getPatternDescription` of the *InteractionPattern* interface. The *PatternDescription*-object in return will contain any pattern alternatives and other textual information about the pattern. This enables viewing the description to understand the pattern better before applying it. When the desired pattern is found, the user flips to the next page.

Page 2 - Customising pattern. Page two can now be displayed, and the user inputs the desired values. The information in the pattern description of the selected pattern, which is already retrieved, is used to construct the wizard page. Because the pattern alternatives might affect the pattern parameters, the user first have to decide on the alternatives, before the parameters can be bound in the third page of the wizard. Upon pressing *Next*, we call the method `getPatternParameters` which returns a list with the pattern parameters according to the alternative chosen.

Page 3 - Binding pattern parameters. When all parameters have been assigned a value, the method `getPatternInstance` will be invoked upon pressing *Finish*. This causes the code of the selected interaction pattern to create a new pattern instance with the correct signals and elements, based on the user input. This collaboration is then returned. Finally, even though not shown in the figure, the wizard uses the pattern instance in the interaction interface collaboration and binds the roles correctly, resulting in what was shown in part (e) of Fig. 7.3.

7.3.3 Composing interaction pattern instances

Yet another wizard is used for the composition of the pattern instances. To keep it simple, the user defines the links of the AIPIC one at a time.

Figure 7.5 shows three screen shots from the implemented functionality for this. A menu action invokes a simple one-page wizard. The interaction interface collaboration, here *buddyTracker*, is selected and *New → Pattern Instance Link* is chosen from the pop-up menu, as shown in part (a) of the figure. The wizard in part (b) will then appear. Here, the source and the target for the link are selected. The choices available have been filtered according to the constraints described in section 4.3.2 on page 43. This means that as source vertex, the user can select either the initial state or an exit connection point reference. The latter is displayed in the format: ⟨nameOfPatternInstance⟩.⟨nameOfExitPoint⟩. The target vertex is always one of the submachine states. In the figure, we create a link from the exit connection point reference *TrackingAccepted* of *TrackRequest* to *TrackUser*. The link is just a normal transition without any signal trigger or send signal action, but we have chosen to call it a link to separate it from the wizard of adding transitions. Upon pressing *Finish*, the link is stored as a transition in the AIPIC. The wizard is used repeatedly to declare all links wanted between the applied pattern instances.

Part (c) of the figure shows the AIPIC of *buddyTracker* after we have composed all its interaction patterns. This is how it looks in the model view of Ramses, but the AIPIC can also be viewed in the state machine editor. The composition of the applied interaction pattern instances of this interaction interface is now completed. The last step is then to let the tool generate the APSMs.

7.4 Generating APSMs

Composing the APSMs of the interaction interface collaboration is a completely automatic process. Our tool will thus generate them without demanding any user involvement.

Part (a) of Figure 7.6 shows that the composition of the APSMs is invoked by selecting the menu item named *Generate APSMs* from the pop-up menu of the interaction interface collaboration. This triggers the implementation of the composition algorithm which was described in chapter 6 on page 65. The algorithm is defined through the class *ApsmGenerator.java* in the *algorithms*-package of the plug-in.

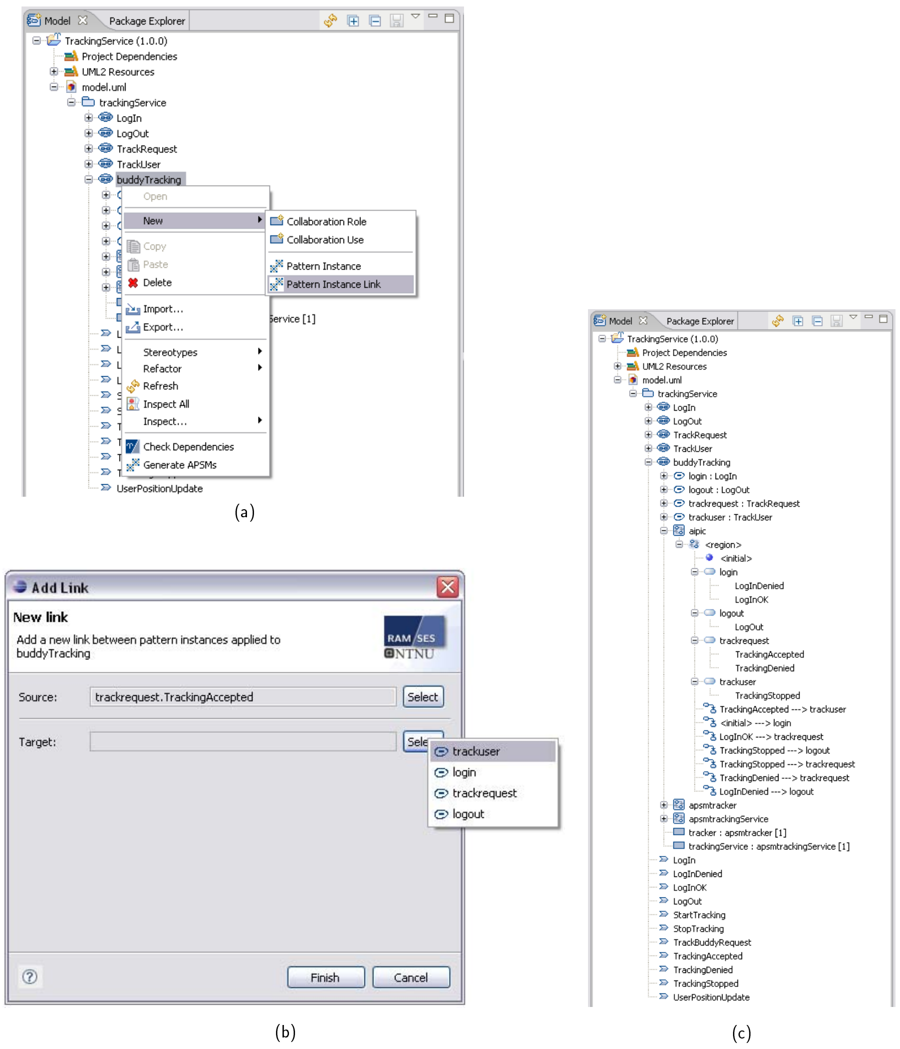
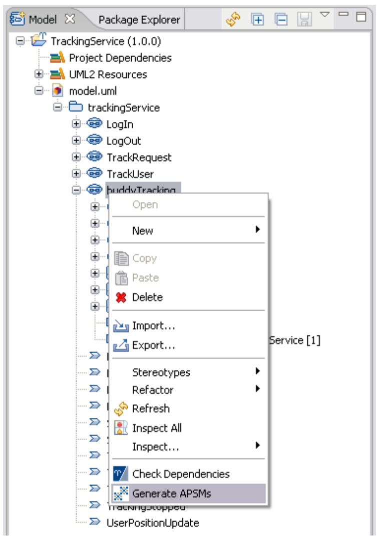
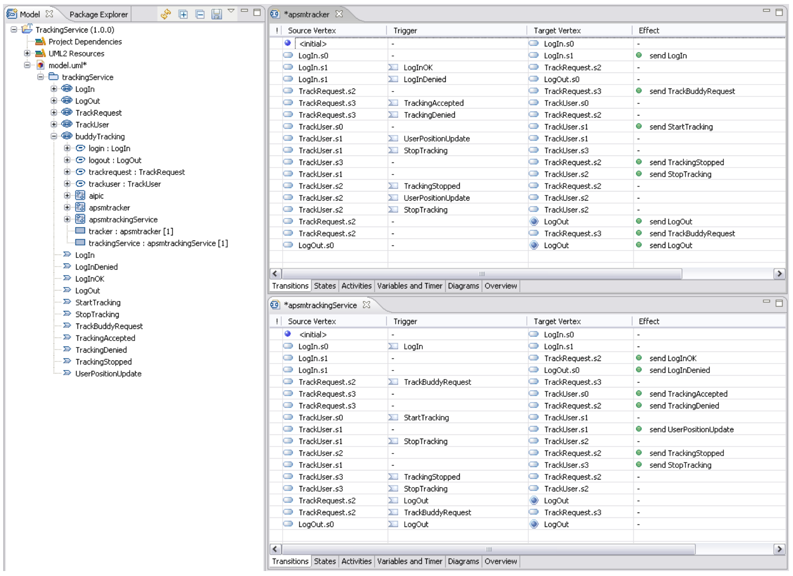


Figure 7.5: Composing the applied pattern instances. The wizard is invoked from the pop-up menu of the interaction interface collaboration (a). The source and target of the link are selected from a list of possible alternatives (b). Part (c) shows the links added to the AIPIC of the *Buddy Tracker*.



(a)



*apsmtracker

!	Source Vertex	Trigger	Target Vertex	Effect
	<initial>	-	LogIn.s0	-
	LogIn.s0	-	LogIn.s1	send LogIn
	LogIn.s1	LogInOK	TrackRequest.s2	-
	LogIn.s1	LogInDenied	LogOut.s0	-
	TrackRequest.s2	-	TrackRequest.s3	send TrackBuddyRequest
	TrackRequest.s3	TrackingAccepted	TrackUser.s0	-
	TrackRequest.s3	TrackingDenied	TrackRequest.s2	-
	TrackUser.s0	-	TrackUser.s1	send StartTracking
	TrackUser.s1	UserPositionUpdate	TrackUser.s1	-
	TrackUser.s1	StopTracking	TrackUser.s3	-
	TrackUser.s3	-	TrackRequest.s2	send TrackingStopped
	TrackUser.s1	-	TrackUser.s2	send StopTracking
	TrackUser.s2	TrackingStopped	TrackRequest.s2	-
	TrackUser.s2	UserPositionUpdate	TrackUser.s2	-
	TrackUser.s2	StopTracking	TrackUser.s2	-
	TrackRequest.s2	-	LogOut	send LogOut
	TrackRequest.s2	-	TrackRequest.s3	send TrackBuddyRequest
	LogOut.s0	-	LogOut	send LogOut

Transitions | States | Activities | Variables and Timer | Diagrams | Overview

*apsmtrackingService

!	Source Vertex	Trigger	Target Vertex	Effect
	<initial>	-	LogIn.s0	-
	LogIn.s0	LogIn	LogIn.s1	-
	LogIn.s1	-	TrackRequest.s2	send LogInOK
	LogIn.s1	-	LogOut.s0	send LogInDenied
	TrackRequest.s2	TrackBuddyRequest	TrackRequest.s3	-
	TrackRequest.s3	-	TrackUser.s0	send TrackingAccepted
	TrackRequest.s3	-	TrackRequest.s2	send TrackingDenied
	TrackUser.s0	StartTracking	TrackUser.s1	-
	TrackUser.s1	-	TrackUser.s1	send UserPositionUpdate
	TrackUser.s1	StopTracking	TrackUser.s2	-
	TrackUser.s2	-	TrackRequest.s2	send TrackingStopped
	TrackUser.s1	-	TrackUser.s3	send StopTracking
	TrackUser.s3	TrackingStopped	TrackRequest.s2	-
	TrackUser.s3	StopTracking	TrackUser.s2	-
	TrackRequest.s2	LogOut	LogOut	-
	TrackRequest.s2	TrackBuddyRequest	TrackRequest.s3	-
	LogOut.s0	LogOut	LogOut	-

Transitions | States | Activities | Variables and Timer | Diagrams | Overview

(b)

Figure 7.6: Generate APSMs. The algorithm for generating the APSMs of the interaction interface collaboration is invoked from the pop-up menu (a). The resulting APSMs for *Buddy Tracker* is shown in (b).

The result of the algorithm can be viewed by expanding the APSMs in the model view or by opening them in the state machine editor, as shown in part (b) of the same figure. This is just one way of viewing the APSMs, another might be using the APSM editor of Ramses (not shown). The APSMs are stored as any other state machine in Ramses.

Tool support for assigning priority to pattern instances, which is used by the algorithm to resolve mixed initiative conflicts, is currently not implemented due to time constraints. The algorithm implemented solves mixed initiatives according to the priority scheme, but the priority is currently decided alphabetically. However, this can easily be implemented by defining a stereotype for interaction pattern instances. Eclipse provides a simple way of assigning values to stereotypes, and this makes it no problem adding functionality for this.

7.5 Usability

In the implementation we have tried to hide as much details of the approach as possible. For instance, we never ask the user to compose submachine states in the AIPIC using transitions from an exit connection point reference to another submachine state. Instead, we ask to link the applied pattern instances together. Actually, nothing in the wizard unveil how we actually compose the pattern instances. We have emphasised implementing the approach at this level to make it easy to use without understanding all the little things in the approach we have created. This also means that the actual composition technique can be replaced without having to change the interface towards the user.

The approach can be carried out just concentrating on the interaction interface collaboration. All wizards are invoked from menu items by selecting this collaboration in the model view of Ramses. We do add several elements both inside the collaboration (such as the AIPIC and the dependencies determining the role bindings) and next to it in its package (such as the pattern instances). But, the user does not need to look inside these elements, unless he wants to. Just focus on the interaction interface collaboration and use the rather self-explanatory menu choices for adding new pattern instances and links, in addition to invoking the APSM composition.

The tasks of adding new pattern instances, composing applied patterns and generating APSMs can be invoked at any time and in any order. For instance, after adding and composing a number of pattern instances, one can always sometime later add additional instances and compose them. The APSMs can be generated over and over again, as the algorithm each time will remove any existing behaviour in the APSMs and construct new behaviour based on the currently composed pattern instances. This results in a flexible approach for the user of the tool, as he can try and fail when constructing the interaction interface collaboration.

Chapter 8

Discussion and conclusion

This chapter starts with a discussion of the assumption on sequential composition, before we say some words about the number of interaction patterns identified. Next, we summarise the achievements of our work and look at the approach in a wider context. Finally, we give some suggestions regarding future work.

8.1 Discussion

We have already discussed some modelling choices in its respective chapters. Sequential composition within an interaction interface is a prerequisite for our work, and we now will briefly discuss this assumption. We will also discuss why we only managed to identify three interaction patterns while searching through existing services.

8.1.1 Assumption of sequential composition

Our work is founded on the assumption of sequential composition of pattern instances, meaning that only one pattern instance can be active in an interaction interface at a time. If we allow parallel execution or interleaving of pattern instances, it would yield in a much more complicated composition. For instance, assume that one instantiation of the Subscribe-pattern (sub1), and two instantiations of the Request-pattern (req1, req2) can be active at the same time. To be able to define consistent APSMs, we probably have to be able to handle all the messages exchanged in *req1* and *req2* in all the states of *sub1*, all messages exchanged in *req2* and *sub1* in all states of *req1*, and so on. The signals of the different pattern instances can be sent and received in any order and this leads to a very APSMs. Mixed initiatives and ambiguities become even more difficult to both identify and resolve.

By restricting the composition to sequential, we can keep the composition semantics and the resulting APSMs quite comprehensive. If two parts of the system wants to do two things together at the same time, they can communicate through an additional pair of association points and an additional interaction interface. We think this is a satisfying solution, and also believe that two parts usually will communicate about one thing at a time.

8.1.2 Number of identified interaction patterns

We managed to identify three two-way interaction patterns, two of them with small variations. Originally we expected to find more patterns. Surprisingly, the interactions in all the services of the PATS-lab we analysed could be expressed using the three identified patterns[1]. The services mainly build on notifications and request-response-pairs, and a lot of them also contain some sort of subscription.

Our interaction patterns only deal with the external behaviour of the parts. If one also consider the internal behaviour of the parts, more patterns emerge. These will be more service-specific as they are closer to the actual functionality of the particular service. The interactions between the parts, however, can only involve message sequences. This makes them very generic, but also restricts the number of possible patterns.

Instead of defining a lot of closely related interaction patterns, we allowed them to have alternatives to represent the small variations in their behaviour. For instance, the SUBSCRIBE-pattern has three alternatives regarding the termination of the subscription. We could easily have created three patterns instead, one for each alternative. We felt that this was a bit unnecessary though, as the intention behind a subscription is the same, regardless of which participant that is allowed to unsubscribe. It is just a variation of the same thing.

It is also of importance that we have focused on defining a generic approach. Additional interaction patterns can easily be added in the future if desired. New interaction patterns that obey the restrictions defined can be included in the library and used in the approach. The number of patterns can thus grow in the future.

8.2 Achievements

Our motivation was to make it easier to design correct services more rapidly. We think that our pattern-based approach helps doing exactly this.

We have modelled interaction patterns as two-way collaboration templates, with signals as unbound parameters that need to be specified upon instantiation.

[1]Of course, the NOTIFY-pattern is truly fundamental and we can build any interaction by applying it repeatedly.

The interaction patterns also contains two APSM fragments which describe the behaviour of each of the participants in the interaction, and in a well-formed interaction pattern, the APSM fragments interact consistently. An additional state machine describes the termination possibilities of the interaction as a whole. We have also identified three fundamental interaction patterns: NOTIFY, REQUEST and SUBSCRIBE.

The interaction interface between two parts is expressed as a two-way collaboration with one APSM for each participant. The approach starts with applying interaction patterns to this collaboration. The composition of the applied interaction patterns relies on the use of state machines and is performed by creating transitions between submachine states referring to each applied pattern instance inside the AIPIC state machine.

A composition algorithm generates the APSMs of the interaction interface collaboration based on the applied patterns and their composition. The algorithm builds the APSMs gradually from the APSM fragments of each applied pattern. Error situations that might occur concerning equivoque states and mixed initiatives are taken special care of. Termination ambiguities and termination condition ambiguities are prevented by composition constraints, and input and mixed ambiguities are resolved by the algorithm. Mixed initiatives due to alternative orderings can never occur due to restrictions imposed, while conflicts due to concurrent behaviours are resolved by prioritising the pattern instances involved. Unwanted or improper termination is removed through the exchange of an additional signal between the APSMs.

We have implemented this approach as an Eclipse plug-in and integrated it with the Ramses tool suite. The interaction patterns are implemented in Java code, because this makes it easier to create customised pattern instances using the UML2 repository from Eclipse. An extension point has also been defined to allow easy adding of additional interaction patterns. The user of the tool performs the approach through menu actions and simple wizards and the details of the actual approach are hidden. The interaction interface is constructed from interaction patterns through a series of simple, comprehensive tasks.

We feel that this approach has the potential to increase the speed of developing correct, advanced telecommunication services and it will be interesting to see how our approach will be welcomed by service designers and users of the tool.

8.3 The approach in a greater perspective

The pattern-approach we have presented can be used within a more extensive approach for the design a large service. The interaction interfaces constructed through our approach represent the basic interactions in a complex service, namely the interactions between two and two parts. These can be used as building blocks for more complicated interactions, involving several parts. Through the nesting property of collaborations, the interaction interfaces can be used in larger multi-way collaborations. In fact, it is possible to describe a complete service from interaction interfaces using our pattern-based approach. We just have to combine them in a hierarchy of collaborations. However, this requires that we compose the interaction interfaces inside a larger collaboration, and these relationships are often not sequential. Another composition technique than the one we have used is therefore needed, and this is being addressed by ongoing work at the department.

8.4 Future work

Experience with using this approach in the tool is required. It should be made available to the users of Ramses to gain knowledge about how they grasp and utilise it.

Support for prioritising pattern instances should be implemented. A stereotype with a value for this can easily be integrated with the approach and the composition algorithm. One should also check if priority is assigned to all pattern instances involved in a combination which can result in a mixed initiative conflict. If not, the user should be warned. The so-called inspectors in Ramses can be used for this.

The approach presented and its model elements have constraints and restrictions attached to them. These should be checked for in the tool, using the inspectors in Ramses. This is a fairy easy task, and would result in a quite large number of inspectors, each checking one property or restriction. We might also need to implement stereotypes for the various elements in the approach, so that the inspectors can separate the state machines and collaborations used in our approach from ordinary model elements. These checks will ensure that the elements of the approach are valid at any time, and warn the user if he does something breaking with the approach.

For making the selection of patterns easier for the user, and especially if more complicated patterns are included in the future, the selection process should be supported with more information about each pattern. Parts of, or the complete, pattern description could be embedded in the tool. This can for instance be provided through the so-called help labels in Ramses.

Bibliography

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-tolerant telecommunication system patterns. In Rising [45], pages 81–94.

[2] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[3] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[4] J. Askgaard. Using positioning technology to guide the blind and visually impaired. Project assignment, Department of Telematics, NTNU, January 2006.

[5] K. Beck and W. Cunningham. Using Pattern Languages for Object-Oriented Programs. Submitted to the OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming., September 17 1987. Available at: `http://c2.com/doc/oopsla87.html`.

[6] S. S. Birkeland. Behavioural Projections and Validation from UML 2.0 State Machines. Project Thesis, NTNU, December 2005.

[7] J. A. van den Broecke and J. O. Coplien. Using design patterns to build a framework for multimedia networking. In Rising [45], pages 259–292.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns.* Wiley, 1996.

[9] J. O. Coplien. A pattern definition - software patterns. Website: `http://hillside.net/`.

[10] D. L. DeBruler. A generative pattern language for distributed processing. In Rising [45], pages 41–61.

[11] J. Doble. TelePLoP Summary, 1998. Available at: `http://hillside.net/chiliplop/1998/98_teleplop.htm`.

[12] J. Dorsch. SPT - The SDL Pattern Tool. Presentation, June 2004. Presented at System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004.

[13] J. Dorsch, A. Ek, and R. Gotzhein. SPT - The SDL Pattern Tool. In D. Amyot and A.W. Williams, editors, *SAM*, volume 3319 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2004.

[14] The Eclipse Foundation. Website: `http://www.eclipse.org`.

[15] The Eclipse Foundation. UML2 Project. Website: `http://www.eclipse.org/uml2/`.

[16] J. Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, NTNU, 2003.

[17] E. Gamma and K. Beck. *Contibuting to Eclipse - Principles, Patterns and Plug-ins*. Addison-Wesley, 2004.

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

[19] B. Geppert and F. Rössler. The SDL pattern approach - a reuse-driven SDL design methodology. *Computer Networks*, 35(6):627–645, May 2001.

[20] D. N. Goins. SOA: The Subscriber-Publisher Model, Introduction and Implementation, 2005. Available at: `http://www.codeproject.com/soap/SOA_PublisherSubscriber.asp`.

[21] R. Gotzhein. The SDL Pattern Pool. Technical report, Computer Networks Group, University of Kaiserslautern, 2002.

[22] M. Grand. *Java enterprise design patterns*. Wiley, 2002.

[23] R. Hanmer and G. Stymfal. An input and output pattern language: Lessions from telecommunications. In Rising [45], pages 95–129.

[24] The Hillside Group. Website: `http://hillside.net/`.

[25] K. E. Husa, R. Braek, and G. Melby. ServiceFrame and ActorFrame, September 2002. Presentation given to the course TTM4160 at NTNU, autumn 2004. Available at: `http://www.item.ntnu.no/fag/ttm4160/ServicePlatforms/ServiceFrame2002.pdf`.

[26] IBM. IBM Rational Software Development Platform. Available at: `http://www-128.ibm.com/developerworks/platform/products.html`.

[27] IBM. IBM Rational Software Architect. Datasheet. Published at IBM Website: `http://www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheets/rsa.pdf`, 2004.

[28] IBM. IBM Rational Software Modeler. Datasheet. Published at IBM Website: `http://www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheets/rsm.pdf`, 2004.

[29] Ø. Isaksen. Designing a group communiation service to be used by nurses in a hospital ward. Project assignment, Department of Telematics, NTNU, November 2004.

[30] ITU-T. Rec. Z.100 (08/2002) Specification and Description Language (SDL). Standard. Available at: `http://www.itu.int/rec/T-REC-z/en`.

[31] F. A. Kraemer. Pax Ramses – Constraints on UML 2.0 Models for the Use with Ramses. Internal Note, March 2006.

[32] P. Kuchana. *Software architecture design patterns in Java.* Auerbach, 2004.

[33] A. Levitin. *Introduction to The Design and Analysis of Algorithms.* Addison-Wesley, 2nd edition, 2007.

[34] G. Meszaros and J. Doble. Metapatterns: A pattern language for pattern writing, September 1996. The 3rd Pattern Languages of Programming conference, Monticello, Illinois.

[35] G. Mezaros. Design patterns in telecommunications system architecture. In Rising [45], pages 21–37.

[36] G. Mezaros. Improving the capacity of reactive systems. In Rising [45], pages 63–80.

[37] F. Ødegaard. Location-based services using WLAN. Project assignment, Department of Telematics, NTNU, December 2005.

[38] OMG. MOF 2.0 / XMI Mapping Specification, v2.1. Available at: `http://www.omg.org/`.

[39] OMG. Unified Modeling Language: Superstructure. Version 2.0. Available at: `http://www.omg.org/docs/formal/05-07-04.pdf`.

[40] OOPSLA. Object-Oriented Programming: Systems, Languages and Applications. Website: `http://www.oopsla.org/`.

[41] Oxford University Press. Compact Oxford English Dictionary. Website: `http://www.askoxford.com/`.

[42] PATS. Program for Advanced Telecom Services. Website: `http://www.pats.no/`.

[43] S. F. Pedersen. Micro Positioning. Master's Thesis, Department of Telematics, NTNU, June 2004.

[44] Ramses Online Documentation. Website: `http://www.item.ntnu.no/lab/pats`.

[45] L. Rising, editor. *Design patterns in communications software.* Cambridge University Press, 2001.

[46] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2. edition, 2005.

[47] R. Sanders, R. Bræk, G. von Bochmann, and D. Amyot. Service Discovery and Component Reuse with Semantic Interfaces. In A. Prinz, R. Reed, and J. Reed, editors, *12th International SDL Forum, Grimstad, Norway*, volume 3530 of *Lecture Notes in Computer Science*, pages 85–102. Springer, June 2005.

[48] R. T. Sanders, H. N. Castejón, F. A. Kraemer, and R. Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In L. Briand and C. Williams, editors, *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.

[49] T. Senneset. WLAN positioning service. Project assignment, Department of Telematics, NTNU, December 2005.

[50] RougeWave Software. Web Services Development Guide: Section 5.2 Message Patterns in WSDL. Available at: `http://www.roguewave.com/support/docs/leif/leif/html/webservicesug/5-2.html`.

[51] P. Swithinbank, M. Chessel, Dr. T. Gardner, C. Griffin, J. Man, H. Wylie, and L. Yusuf. *Patterns: Model-Driven Development Using IBM Rational Software Architect.* IBM, 2005. IBM Redbook.

[52] Telelogic. TAU Generation2 2.2. Website: `http://www.telelogic.com/campaigns/2003/global/taug22_launch/index.cfm`.

[53] W3R. SOAP Version 1.2 Part 2: Adjuncts - W3C Recommendation 24 June 2003. Available at: `http://www.w3.org/TR/soap12-part2/`.

[54] C. Webel and I. Fliege. SDL Design Patterns and Components - Watchdog and Heartbeat. Technical report, Computer Science Department, University of Kaiserslautern, 2004.

Appendix A

Metamodel

This appendix defines our meta model, which is based on, and a subset of, the UML 2.0 meta model [39]. The document describes the elements used in our approach, including constraints and illustration of the element structures.

A.1 Interaction Pattern

An interaction pattern (IP) is a parametrized two-way elementary collaboration, also called a collaboration template. The template parameters are always signals.

Constraints

1. An interaction pattern has one state machine as classifier behaviour that is an interaction pattern descriptor.
2. An interaction pattern has two state machine as owned behaviours that are association point state machine fragments.
3. An interaction pattern has exactly two collaboration roles.
4. Each of the two collaboration roles in an interaction pattern is typed with one of the association point state machine fragments.

Stereotype

≪ ip ≫

Repository model

Figure A.1 on the next page shows the repository model of this element.

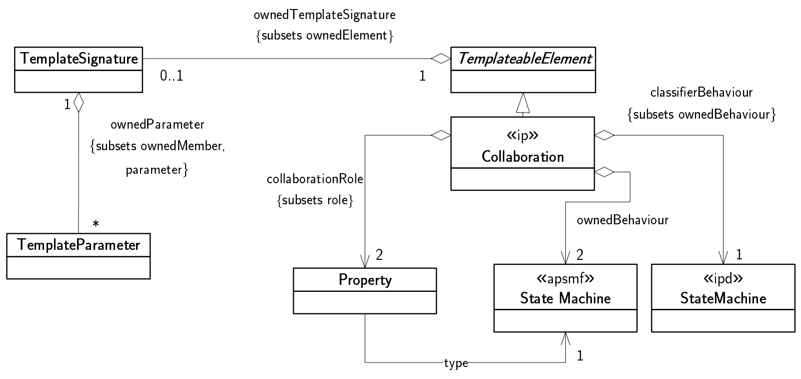


Figure A.1: Element model for the Interaction Pattern (IP)

A.2 Association Point State Machine Fragment

An association point state machine fragment (APSM fragment / APSMF) defines the behaviour of one of the collaboration roles in an interaction pattern or an interaction pattern instance. The APSMFs are attached to the interaction pattern or the interaction pattern instance as owned behaviours.

Constraints

1. An APSM fragment has exactly one initial state and one initial transition.
2. The initial transition of an APSM fragment has neither a trigger nor effect.
3. The state succeeding the initial transition in an APSM fragment has exactly one outgoing transition.
4. An APSM fragment has at least one labelled exit point.
5. All transitions in an APSM fragment, besides the initial transition, have either a send signal action or a signal trigger.
6. All states in an APSM fragment, besides the initial state and exit points, are simple states.

Stereotype

≪ apsmf ≫

Repository model

Figure A.2 on the facing page shows the repository model of this element.

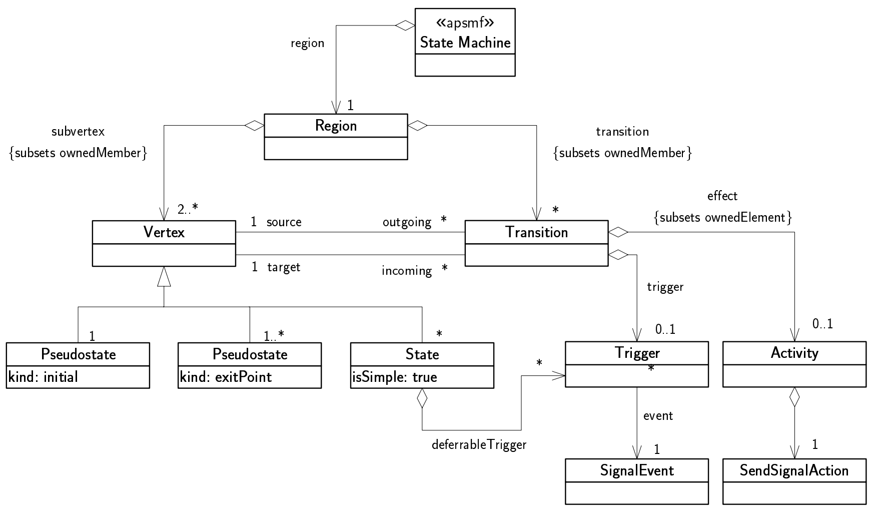


Figure A.2: Element model for the Association Point State Machine Fragment (APSMF)

A.3 Interaction Pattern Descriptor

The interaction pattern instance descriptor (IPD) specifies how the interaction pattern, when instantiated, can be interconnected to other interaction pattern instances. It is a state machine which is attached to the interaction pattern as the classifier behaviour. It does not have any actual behaviour, but simply defines the activation and deactivation possibilities of the interaction pattern.

Constraints

1. An interaction pattern descriptor contains one or more uniquely labelled exit points.
2. An interaction pattern descriptor never contains any transitions or vertices besides exit points.

Stereotype

≪ ipd ≫

Repository model

Figure A.3 on the next page shows the repository model of this element.

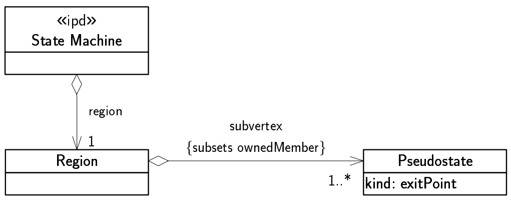


Figure A.3: Element model for the Interaction Pattern Descriptor (IPD)

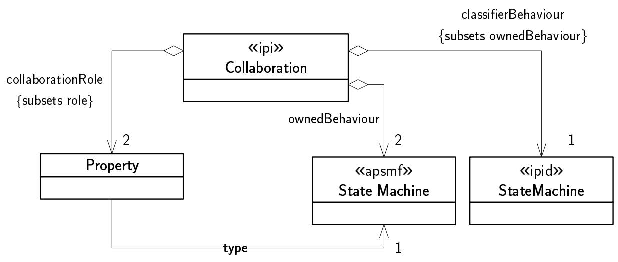


Figure A.4: Element model for the Interaction Pattern Instance (IPI)

A.4 Interaction Pattern Instance

An interaction pattern is instantiated by binding its parameters. An interaction pattern instance (IPI) is a two-way elementary collaboration. The constraints of an interaction pattern instance are similar to those of an interaction pattern.

Constraints

1. An interaction pattern instance has one state machine as classifier behaviour that is an interaction pattern instance descriptor.
2. An interaction pattern instance has two state machine as owned behaviours that are association point state machine fragments.
3. An interaction pattern instance has exactly two collaboration roles.
4. Each of the two collaboration roles in an interaction pattern instance is typed with one of the association point state machine fragments.

Stereotype

$\ll$ ipi $\gg$

Repository model

Figure A.4 shows the repository model of this element.

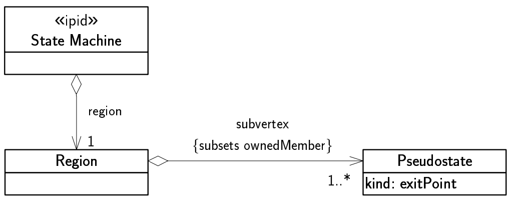


Figure A.5: Element model for the Interaction Pattern Instance Descriptor (IPID)

A.5 Interaction Pattern Instance Descriptor

The interaction pattern instance descriptor (IPID) is the same as an interaction pattern descriptor (IPD), only that it is attached to an interaction pattern instance. They both share the same constraints.

Constraints

The same constraints as for the interaction pattern descriptor (IPD), see A.3 on page 99.

Stereotype

$\ll$ ipid $\gg$

Repository model

Figure A.5 shows the repository model of this element.

A.6 Interaction Interface Collaboration

An interaction interface collaboration (IIC) is a two-way composite collaboration. It is constructed entirely from interaction pattern instances.

Constraints

1. An interaction interface collaboration has one state machine as classifier behaviour that is an applied interaction pattern instances composer.
2. An interaction interface collaboration has two state machine as owned behaviour that are association point state machines.
3. An interaction interface collaboration has exactly two collaboration roles, each typed with one of the association point state machines.

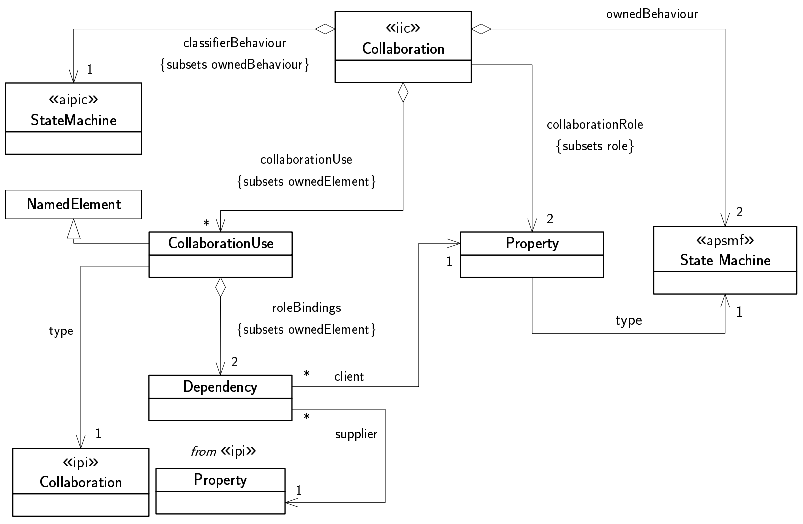


Figure A.6: Element model for the Interaction Interface Collaboration (IIC)

4. An interaction interface collaboration has at least one collaboration use, and all collaboration uses must be typed with an interaction pattern instance.
5. An interaction interface collaboration has exactly two dependencies for each used interaction pattern instance, where each dependency specifies how one of the collaboration roles of the used interaction pattern instance are bound to a collaboration role of the interaction pattern based collaboration.

Stereotype

≪ iic ≫

Repository model

Figure A.6 shows the repository model of this element.

A.7 Applied Interaction Pattern Instances Composer

An applied interaction pattern instances composer (AIPIC) is used to define how the applied interaction pattern instances relate to one another, meaning in what sequence they should be combined. It is a state machine owned by the interaction interface collaboration.

Constraints

1. An applied interaction pattern instances composer has exactly one initial state, and exactly one initial transition.
2. All vertices in an applied interaction pattern instances composer are submachine states.
   (a) For each applied interaction pattern instance, a submachine state exists referring to the interaction pattern instance descriptor as its submachine.
   (b) The submachine state has one exit connection point reference for each exit point in its submachine. This exit connection point reference has the same name (that is, the same label) as the exit point.
3. A transition in an applied interaction pattern instances composer has no trigger or effect.
4. The source of a transition in an applied interaction pattern instances composer is always either the initial state or an exit connection point reference of a submachine state.
5. The target of a transition in an applied interaction pattern instances composer is always a submachine state.

Stereotype

≪ aipic ≫

Repository model

Figure A.7 on the following page shows the repository model of this element.

A.8 Association Point State Machine

In this context, an association point state machine (APSM) defines the behaviour of one of the collaboration roles in an interaction pattern based collaboration. The APSM is modelled as a state machine, and generated from the association point state machine fragments of the interaction pattern instances used in the collaboration.

Constraints

1. An association point state machine has exactly one initial state and one empty initial transition.
2. An association point state machine can have one of more final states.
3. All states in an association point state machine, besides the initial and final state, are simple states.

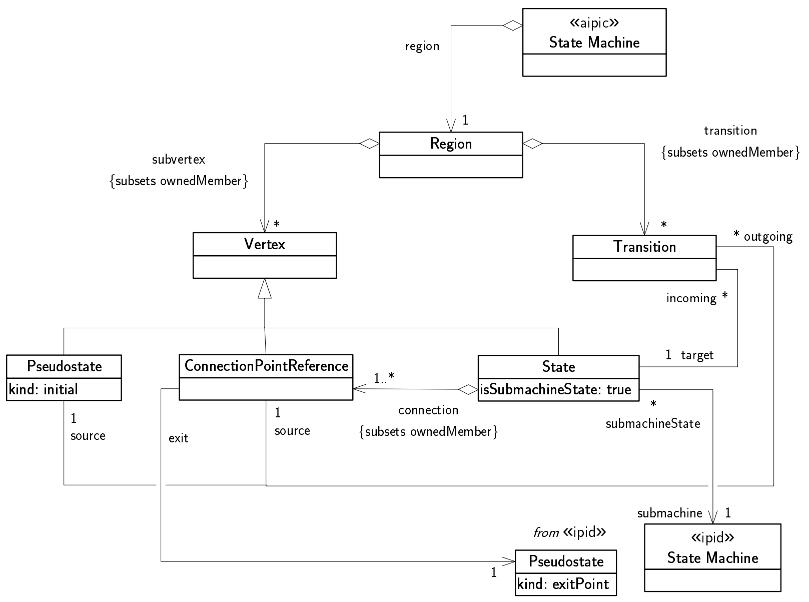


Figure A.7: Element model for Applied Interaction Pattern Instances Composer (AIPIC)

4. The transitions of an association point state machine, besides the initial transition, contain either a send signal action or a signal trigger.

Stereotype

≪ apsm ≫

Repository model

Figure A.8 on the next page shows the repository model of this element.

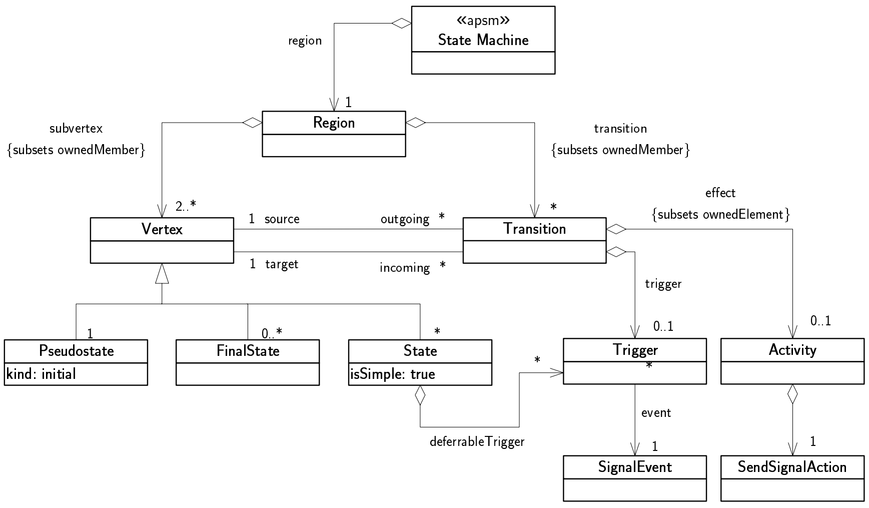


Figure A.8: Element model for the Association Point State Machine (APSM)

Appendix B

Pattern Library

B.1 Notify

Notify

Problem: Consider two parts, A and B, of a distributed system. A needs to inform B of something. This can either be (a) to indicate that something has happened that B should know about, or that (b) it has some data (or information) that should be passed to B.

Solution: A sends a single message to Notify B of the thing that happened, and includes any necessary data in this message.

Example usage: Two users want to send messages to each other through their user terminals. This is, for instance, the case in a service called the "Hospital Ward Group Service" [29, p. 44], where nurses in a hospital are equipped with portable terminals. Information messages can be sent between them to communicate. This is shown in Figure B.1 on the following page, where terminal *A* sends an *InformationMsg* to terminal *B*.

Pattern roles: The *notifier* notifies the *recipient*. The notifier is the initiator of this pattern.

Pattern alternatives: No alternatives are defined for this pattern.

Pattern parameters: The pattern parameter is:

- *Notification:* The message sent from the notifier to the recipient.

APSM fragments: Figure B.2 on the next page shows the APSM fragments for the notifier and the recipient in this pattern.

Known uses: This pattern is truly fundamental. These are just a few examples, and the pattern reader can probably name a handful of additional examples.

- *Using positioning technology to guide the blind and visually impaired* [4, p. 48]: The user can be guided through a number of routes. From a list of possible routes at the current location, the user sends a *RouteChoice(routeID)*-message to the system to indicate which route it would like to be helped with.
- *WLAN Positioning Service* [49, p. 37]: Users update their contact lists by sending *SaveNewContact* or *DeleteContact* messages to the central system.
- *Micro Positioning* [43, p. 77]: Objects in a hospital environment can trigger off alarms by sending a notification message to a central server indicating the reason for the alarm.

Related patterns: Some related patterns are:

- The SDL-pattern *AsynchronousNotification* [21].
- Message Exchange Patterns *One-Way* and *Notification* from Web-services [50].

If you need a confirmation of the reception of the notification message, use the REQUEST-pattern.

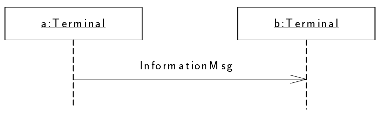


Figure B.1: NOTIFY: Sequence diagram for the example

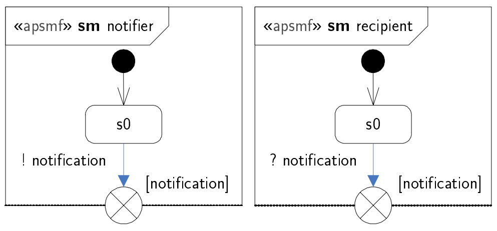


Figure B.2: NOTIFY: The APSM fragments for the *notifier* and the *recipient*.

B.2 Request

Request

Problem: Part A of a distributed service needs to get some information from another part, B. A either (a) wants to get some data or information, or (b) needs to ask permission to do something.

Solution: A sends a message to Request the permission or wanted data. B will respond with an answer and possibly the data requested.

Example usage: In "Location-based services using WLAN" [37], a *UserPosition* server receives the position updates from a tracking service when the user moves, but has to ask a *LocationRegister* to translate this into physical locations such as rooms or buildings [37, p. 54]. This is done by issuing a request with the position coordinates (which so-called zone the user is in) to the location register, which maps this to the registered locations in its database and returns the actual location.

Pattern roles: A *requestor* sends the request, which a *responder* replies to it. The requestor is the initiator of this pattern.

Pattern alternatives: This pattern has an alternative. The number of possible reply's to the request can be changed. The default value is one reply.

Pattern parameters: The pattern parameters are:

- *Request:* The message sent from the requestor to the responder with the question.
- *Reply1:* The answer, sent from the responder back to the requestor.

Note that, because the number of possible reply messages is a pattern alternative, the number of pattern parameters can vary. One pattern parameter will exist for each possible reply signal, and they are named *Reply2*, *Reply3*, and so on.

APSM fragments: Figure B.4 on the following page shows the APSM fragment for each of the pattern roles, with the default value of the pattern alternative.

Known uses: The pattern can be found in a lot of distributed services. It is one of the basic building blocks for such systems. The following list provides a taste of some of the services where the pattern can be found:

- *ActorFrame* [25, p. 12]: The so-called *role request protocol* allows objects in the system to request certain functionality from other objects, which will be either replied with a confirmation or denial.

- *Hospital Ward Group Service* [29, p. 41]: Nurses in a hospital can request other nurses to help them. The reply will be either “confirmed” or “not confirmed”.
- *Using positioning technology to guide the blind and visually impaired* [4, p. 45]: In this service, the users can start guidance sessions to help them find their way in unfamiliar places. To select a route, a request is issued to a route agent, which will return all available routes at the present location.

Related patterns: Some related patterns are:

- The SDL-pattern *SynchronousInquiry* [21].
- Message Exchange Pattern *Request/Response (Remote Procedure Call)* from Webservices [53].

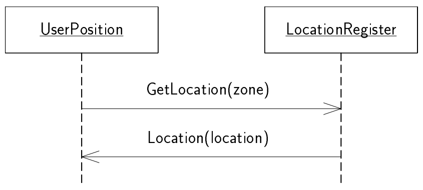


Figure B.3: REQUEST: Sequence diagram for the example.

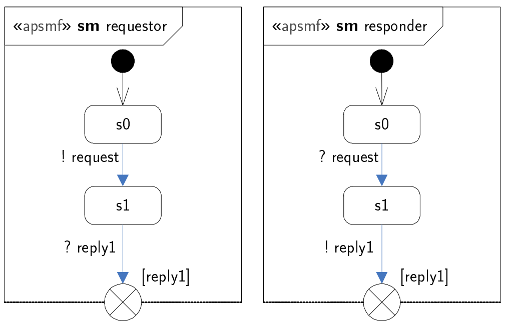


Figure B.4: REQUEST: APSM fragments for the *requestor* and the *responder*.

B.3 Subscribe

Subscribe

Problem: Part A of a service wants to be notified every time some data changes in part B. There is no way to know exactly when this change of information occurs in B. If A requests the information periodically, it might result in loss of updates or unnecessary requests.

Solution: Part A should Subscribe to this event. Whenever a change occurs in B, a message will be sent with the updated information, until the subscription is ended.

Example usage: In "Using positioning technology to guide the blind and visually impaired" [4], one can commence guidance sessions to help the visually impaired navigate in unknown places. Doing so, we must always know the position of the user and use this to tell him where to go next. During the establishment of a guidance session, the terminal, through its *TerminalAgent*, asks a *PositionAgent* to start tracking the terminal [4, p. 46]. Figure B.5 on page 113 shows a sequence diagram for this. The *PositionAgent* polls another object for the position of the user. A *PositionUpdate*-message is sent to the *TerminalAgent* with the new location coordinates [4, p. 47]. When the user reaches the wanted destination, the guidance session ends, and the position updates are stopped.

Pattern roles: The *subscriber* subscribes to the event, while the *subscribee* notifies the subscriber when this event occurs. The subscriber is the initiator of the pattern.

Pattern alternatives: This pattern has an alternative which deals with who that should have the power to stop the subscription. Either the subscriber, subscribee or both can be allowed to unsubscribe. The default is that both parts can unsubscribe.

Pattern parameters: The pattern has the following parameters:

- *Subscribe:* The message starting the subscription, sent from the subscriber to the subscribee.
- *Event:* Sent from the subscribee to the subscriber each time the specified event occurs.
- *Unsubscribe:* Sent when one of the parts wants to end the subscription.
- *UnsubscribeCnf:* A confirmation to the unsubscription from the other part.

Note that, due to the pattern alternative, the messages *unsubscribe* and *unsubscribeCnf* messages can be sent in any direction.

APSM fragments: Figure B.6 on the facing page shows the APSM fragments for each of the pattern roles, with the default value of the pattern alternative.

Known uses: The pattern can be found in the following designs or systems:

- *WLAN Positioning Service* [49, p. 38f]: In this service users can track other users in their contact list. A *TrackContact*-message triggers the receptions of position updates whenever the other user moves. This continues until the user being tracked does not want to be tracked any more and stops the updates.
- *Micro Positioning* [43, p. 78f]: Users can monitor the location of moveable objects (for instance doctors, patients, equipment) inside a hospital. A *TerminalAgent* sends a message to a *LocationAgent* with a list of the objects its user wants to monitor. When one of these objects move, the *LocationAgent*, with the help of a map system, draws the new location of the objects in a map and returns this to the user's terminal. The user can stop the monitoring when it no longer is of interest.
- *Location-based services using WLAN* [37, p. 54]: Users can store reminders which can be triggered by changes in their geographical location. A *UserAgent* subscribes to these changes in location from a positioning device. When the user reaches a location which he has registered a reminder for, for instance when he reaches the office, the reminder will be pushed to his terminal. When a user has no registered reminders, it can unsubscribe to the location changes of the user.

Related patterns: Some related patterns are:

- The design pattern *Observer* from the "Gang of Four" [18, p. 293].
- The Java enterprise design pattern *Publish-Subscribe* [22, p. 175].
- Message Exchange Pattern *Publisher-Subscriber* from Webservices [20].
- *Publisher-Subscriber*-pattern from POSA [8, p. 339].

If permission needs to be given before subscription can commence, use the REQUEST-pattern and only allow to SUBSCRIBE if the permission is granted.

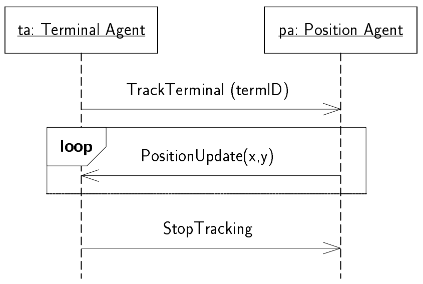


Figure B.5: SUBSCRIBE: Sequence diagram for the example

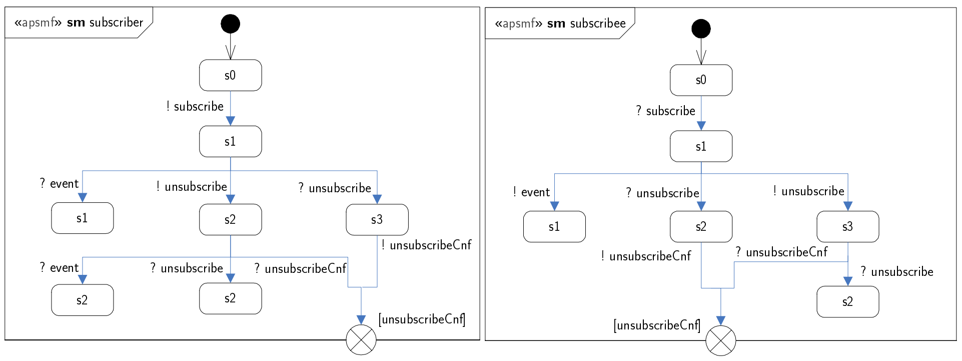


Figure B.6: SUBSCRIBE: APSM fragments for the *subscriber* and the *subscribee*

Appendix C

The plug-in

Our plug-in *no.ntnu.item.ramses.interactionpattern* depends on the following configuration:

- Eclipse 3.2.RC1
- GEF 3.2 M6
- EMF 2.2.0 M6
- UML2 2.0 M4
- Ramses III

Eclipse, GEF, EMF and UML2 can be found at the Eclipse website: `http://www.eclipse.org`. Ramses III is under development, and we refer to `http://www.item.ntnu.no/lab/pats` for further information about the Ramses tool suite.

The plug-in is stored on the CVS-server of Ramses.

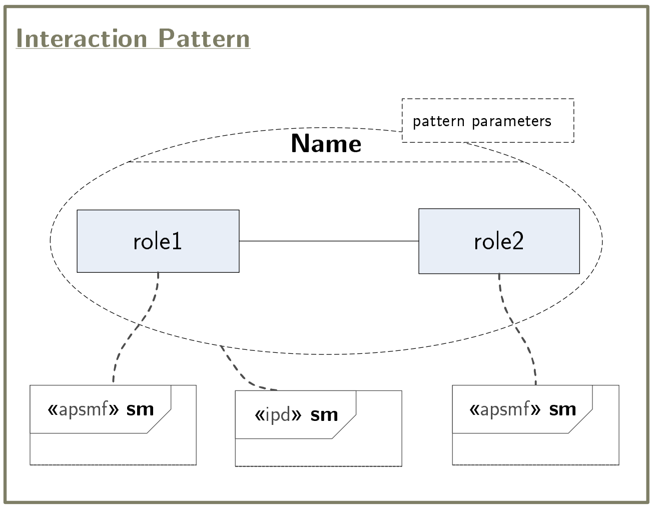
Appendix D

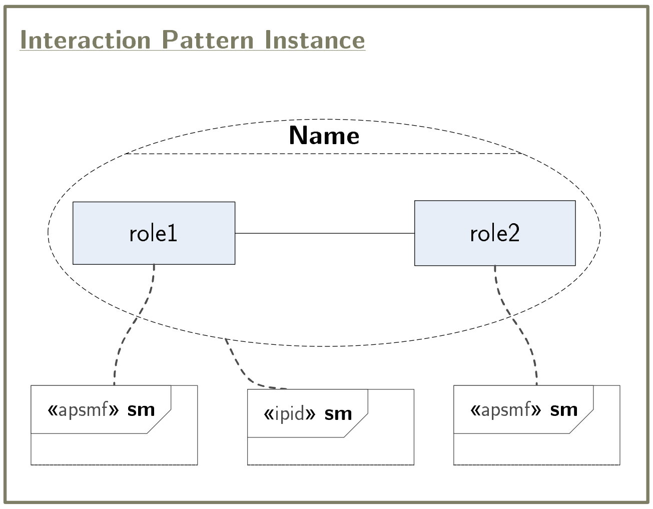
Bookmark

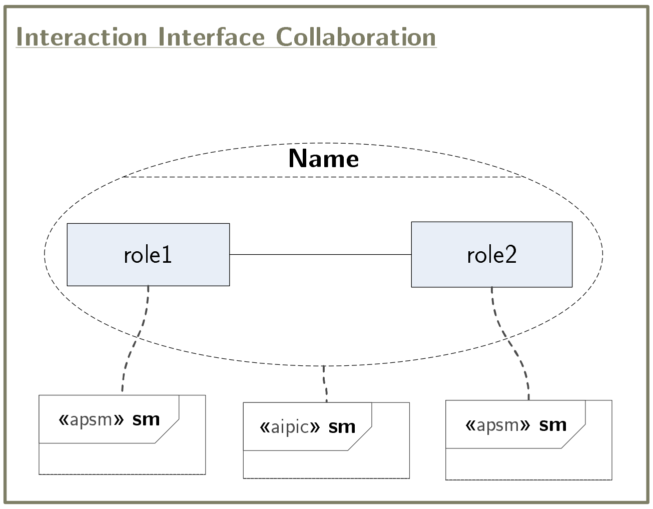
We have summarised the model elements introduced in our approach in the shape of a bookmark. It can be used while reading the thesis, making it easier to remember the different model elements we have presented in our work.

cut out and use as bookmark

The essential model elements of the pattern-based approach


Interaction Pattern
pattern parameters
Name
role1
role2
«apsmf» sm
«ipd» sm
«apsmf» sm


Interaction Pattern Instance
Name
role1
role2
«apsmf» sm
«ipid» sm
«apsmf» sm


Interaction Interface Collaboration
Name
role1
role2
«apsm» sm
«aipic» sm
«apsm» sm