Master Thesis

Design patterns in event-B and their tool support

Author(s):
Fürst, Andreas

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005779068

Rights / License:
In Copyright - Non-Commercial Use Permitted
Design Patterns in Event-B and Their Tool Support

Master Thesis

by

ANDREAS FÜRST

15th March 2009

Supervisor: Dr. Thai Son Hoang
Professor: Prof. Dr. David Basin

Information Security Group
Chair of Information Security
Department of Computer Science
Abstract

This thesis gives an overview of a new concept in Event-B called design patterns. It enables developers to reuse an existing development within the problem at hand. So far, the reuse of models in Event-B was restricted to merging models by hand. This is not only painful but also error prone.

One goal of the pattern approach as presented in this thesis, is to release the developer of this burden of manual copying. Such tasks are destined to be done by a tool.

A second and even more important goal is the reuse of proofs. A substantial part, when developing in Event-B, is proving the correctness of the models. Some of the proofs have to be done manually, which can be very time-consuming.

Therefore, we are not interested in doing proofs which were already done during the development of the pattern. With the approach evolved and described in this thesis, a new model within the development can be generated, that reuses the achievements of a pattern and is correct by construction.
Acknowledgments

My special thank goes to Son, who gave me the opportunity to contribute to this project. I enjoyed working with him very much. He always supported me during this thesis and was willing to answer questions or discuss problems.

I also thank Dr. Jean-Raymond Abrial. After all, it was he with his infective enthusiasm, who aspired me with the fascination for formal methods.

I am looking forward to continue my work on design patterns together with both of you.
Contents

List of Figures xi
List of Tables xiii
List of Listings xv

1 Introduction 1

2 The Event-B Modelling Method 3
  2.1 Machines 3
  2.2 Machine Refinement 6

3 Example -Question/Response Protocol 9
  3.1 Description of the Protocol 9
  3.2 Development of the Protocol 10
    3.2.1 Initial Model 10
    3.2.2 First Refinement 10
    3.2.3 Pattern Recognition 12
    3.2.4 Pattern Description 13
    3.2.5 Second Refinement 14
    3.2.6 Third Refinement 28
    3.2.7 Fourth Refinement 33
  3.3 Proof Statistics 36
  3.4 A Development Becomes a Pattern 37

4 Pattern Usage 39
  4.1 Overview 39
    4.1.1 Procedure 40
  4.2 Matching 40
    4.2.1 Matching of the Variables (M1) 41
    4.2.2 Matching of the Events (M2) 42
  4.3 Syntactical Check of the Matching 42
List of Figures

3.1 Question/response protocol ........................................ 9
3.2 Splitting of two-way communication ................................. 12
3.3 Involved models in the pattern usage ................................. 15
3.4 View of the local variables ........................................... 34

4.1 Involved models in the pattern usage ................................ 40
4.2 Pattern specification and refinement $n$ before matching ........ 41
4.3 Pattern specification and refinement $n$ after matching .......... 42
4.4 Guards and actions split ............................................. 44
4.5 Model of the pattern refinement ..................................... 45
4.6 Generated refinement $n+1$ of the development .................... 49
4.7 Most generic scenario when using patterns ......................... 54

5.1 State machine of phase 1 ........................................... 56
5.2 State machine of phase 2 ........................................... 57
5.3 State machine of the A2A protocol ................................. 61
5.4 Phase 1 folded ...................................................... 62
5.5 Phase 1 assigned to the parties ................................... 62
5.6 Phase 2 assigned to the parties ................................... 63

7.1 Starting the pattern wizard ........................................... 87
7.2 Matching within the wizard dialogue ............................... 87
LIST OF FIGURES
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Number of proofs in each refinement level</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Proved obligations used by meta-proofs</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of proofs in each refinement level</td>
<td>69</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview of all tasks</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Listings

3.1 Matching and checking ........................................... 20
3.2 Incorporation ...................................................... 28

4.1 Matching ............................................................. 52
4.2 Already proved obligations ....................................... 53

5.1 Example of a run of the A2A protocol ....................... 59
5.2 Another run of the A2A protocol ............................... 60
Chapter 1

Introduction

The idea of design patterns is by no means new. Nor is it restricted to the field of computer science or even object-oriented software. Design patterns were actually introduced by Christopher Alexander in the field of architecture.

In 1977 he spoke of patterns as, “each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”. [AIS+77]

Related to Event-B, by design patterns we understand former developments that can be reused in the current development. Of course not every development is a good choice for reuse. What we are interested in are generic solutions of common problems.

In our case studies, basically communication protocols, good solutions were needed for realising communication channels. As these were needed over and over, we developed proper solutions that were reused in the protocol developments as design patterns.

Although reusability is a good reason for having patterns, in Event-B, there is another important point. We aspire the reuse of proofs. A substantial part of the work when developing in Event-B is proving the correctness of the models. When reusing an already proved development, why should one do the same proofs again? The goal of design patterns in Event-B is therefore:

\[ \text{The reuse of already proved solutions, to refine the problem at hand, without doing any proofs.} \]

In the following chapters, the methodological approach of using design patterns in Event-B is explained. Furthermore the prerequisites for the usage of patterns are given, as they have been investigated during this master thesis. The theoretical aspects of the pattern approach are explained with examples from the case studies done so far.
CHAPTER 1. INTRODUCTION

The theory by itself, though, is not enough when a developer wants to use the approach of patterns. As we will see, there are a lot of small jobs that are quite painful to do by hand. One example would be to collect elements from different models and copy them into a new model to generate the refinement. Such stupid but error prone tasks are destined to be done by automated tools.

The thesis is organised as follows. Chapter 2 gives a brief introduction to the Event-B modeling method. Chapter 3 presents an example as the motivation for the idea of design patterns. Chapter 4 formalises the concept of patterns in Event-B together with its proof of correctness. Chapter 5 shows an application of the approach to a medium-size case study. Chapter 6 lists some patterns that were developed and used during this thesis. Chapter 7 discusses the requirements for tool support. Chapter 8 concludes and proposes direction for future work.
Chapter 2

The Event-B Modelling Method

Event-B [Abr09], unlike classical B [Abr96], does not have a fixed syntax. Instead, it is a collection of modelling elements that are stored in a repository. Still, we present the basic notation for Event-B using some syntax. We proceed like this to improve legibility and help the reader remembering the different constructs of Event-B. The syntax should be understood as a convention for presenting Event-B models in textual form rather than defining a language.

Event-B models are described in terms of the two basic constructs: contexts and machines. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain carrier sets, constants, axioms, where carrier sets are similar to types [AH06]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section 2.1 and machine refinement in Section 2.2.

2.1 Machines

Machines provide behavioural properties of Event-B models. Machines may contain variables, invariants and events. Variables \( v \) define the state of a machine. They are constrained by invariants \( I(v) \). Possible state changes are described by means of events. Each event is composed of a guard \( G(v) \) and an action \( S(v) \). The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve.

\(^1\)We do not take into account other elements such as theorems and variants

\(^2\)Within the scope of this dissertation, we do not consider events with parameters.
when the event occurs. An event can be represented by the following form

\[
\text{evt when} \ G(v) \ \text{then}\ S(v) \ \text{end} \tag{2.1}
\]

The short form

\[
\text{evt begin} \ S(v) \ \text{end} \tag{2.2}
\]

is used if the guard equals \textit{true}. A dedicated event of the form (2.2) is used for \textit{initialisation}.

The action of an event is composed of several \textit{assignments} of the form

\[
x := E(v) \tag{2.3}
\]
\[
x :\in E(v) \tag{2.4}
\]
\[
x :| Q(v, x') \tag{2.5}
\]

where \(x\) are some variables, \(E(v)\) expressions, and \(Q(v, x')\) a predicate. Assignment form (2.3) is \textit{deterministic}, the other two forms are \textit{nondeterministic}. Form (2.4) assigns \(x\) to an element of a set, and form (2.5) assigns to \(x\) a value satisfying a predicate. The effect of each assignment can also be described by a before-after predicate:

\[
\text{BAP}(x := E(v)) \equiv x' = E(v) \tag{2.6}
\]
\[
\text{BAP}(x :\in E(v)) \equiv x' \in E(v) \tag{2.7}
\]
\[
\text{BAP}(x :| Q(v, x')) \equiv Q(v, x') \tag{2.8}
\]

A before-after predicate describes the relationship between the state just before an assignment has occurred (represented by unprimed variable names \(x\)) and the state just after the assignment has occurred (represented by primed variable names \(x'\)). All assignments of an action \(S(v)\) occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate \(A(v, x')\). Variables \(y\) that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is
achieved by conjoining \( A(v, x') \) with \( y' = y \), yielding the before-after predicate of the action:

\[
BAP(S(v)) \equiv A(v, x') \land y' = y .
\] (2.9)

In proof obligations we represent the before-after predicate \( BAP(S(v)) \) of an action \( S(v) \) directly by the predicate

\[
S(v, v') .
\]

Proof obligations serve to verify certain properties of a machine. All proof obligations in this thesis are presented in the form of sequents: “hypotheses” \( \vdash \) “goal” (more information about sequents can be found in Appendix A).

For each event of a machine, feasibility must be proved.

> Invariant \( \vdash \) Guard \( \vdash \) \( \exists v' \cdot \) Before-after predicate \( \vdash \) FIS

By proving feasibility, we achieve that \( S(v, v') \) provides an after state whenever \( G(v) \) holds. This means that the guard indeed represents the enabling condition of the event.

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called invariant preservation:

> Invariant \( \vdash \) Guard \( \vdash \) Before-after predicate \( \vdash \) Modified invariant \( \vdash \) INV

Similar proof obligations are associated with the initialisation event of a machine. The only difference is that the invariant and guard do not appear in the antecedent of the proof obligations (FIS) and (INV).
2.2 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model \[AH06\]. For more on the well-known theory of refinement, we refer to the Action System formalism that has inspired the development of Event-B \[Bac89\]. We present some important proof obligations for machine refinement. As mentioned before, the user of Event-B is not presented with a behavioural model but only with proof obligations. The proof obligations describe the semantics of Event-B models.

A machine \(CM\) can refine at most one other machine \(AM\). We call \(AM\) the abstract machine and \(CM\) a concrete machine. The state of the abstract machine is related to the state of the concrete machine by a glueing invariant \(J(v, w)\), where \(v\) are the variables of the abstract machine and \(w\) the variables of the concrete machine.

Each event \(ea\) of the abstract machine is refined by one or more concrete events \(ec\). Let abstract event \(ea\) and concrete event \(ec\) be:

\[
\begin{align*}
\text{ea} & \quad \text{when} \quad G(v) \\
& \quad \text{then} \quad S(v) \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{ra} & \quad \text{refines} \quad ea \\
& \quad \text{when} \quad H(w) \\
& \quad \text{then} \quad T(w) \\
& \quad \text{end}
\end{align*}
\]

Somewhat simplified, we can say that \(ec\) refines \(ea\) if the following conditions hold.

1. The concrete event is feasible. This is formalised by the following proof obligation.

\[
\begin{align*}
\text{Abstract Invariant} & \\
\text{Concrete Invariant} & \\
\text{Concrete guard} & \vdash \\
\exists w' \cdot \text{Concrete before-after predicate}
\end{align*}
\]
2. The guard of \( ec \) is stronger than the guard of \( ea \). This is formalised by the following proof obligation.

3. The abstract event can always “simulate” the concrete event and preserve the glueing (concrete) invariant. This is formalised by the following proof obligation.
For the initialisation, the corresponding proof obligations are obvious.
The proofs of these above obligations guarantee the correctness of the refinement model with respect to the abstract model and the glueing invariant between them.
Chapter 3

Example - Question/Response Protocol

For a better understanding of what we mean by design patterns in Event-B, we first look at a simple communication protocol. We will discover the usage of patterns and see how powerful the concept of design pattern really is.

3.1 Description of the Protocol

The protocol is related to a common situation in communication scenarios. There are two parties A and B, of which A tries to get some information from B. Inspired by human communication it is intuitive to have party A asking a question and waiting for a response from B. We denote A as the questioner and B as the responder.

![Question/response protocol](image)

Figure 3.1: Question/response protocol

If the questioner chooses his questions wisely and the responder is honest and possesses the information asked for, the questioner will eventually get some new information.

In contrast to human beings, parties in a protocol can be designed to ask smart questions and behave honestly.
Another aspect we should think about is, at what time the parties are allowed to “speak”. Is it possible for the questioner to ask a new question even before he got the response to the last one? In this case we would call the communication asynchronous. However, we will look at the scenario, where the questioner has to wait with the new question until he received the pending response. We refer to this as a synchronous communication.

To sum up, the protocol is about two parties, a questioner and a responder, that communicate together in a synchronous way by exchanging questions and responses respectively. According to this, the protocol is called question/response protocol.

### 3.2 Development of the Protocol

#### 3.2.1 Initial Model

We start the development by defining the synchronisation of the protocol. This is done by splitting the protocol into rounds. Such a round will later consist of exactly one question transfer and one response transfer afterwards. First, the rounds are modeled as listed below.

```
variables: round

invariants: round ∈ \mathbb{N}
```

```
init
begin
round := 0
end

finishRound
begin
round := round + 1
end
```

Besides init, there exists only one other event. finishRound simply increments the number of rounds, meaning, another round of question and response is over. As a first refinement of the development, we describe what happens in such a round.

#### 3.2.2 First Refinement

As we already know, a round consists of exactly one question and one response. Additionally, we have to preserve the property, that the question appears first and afterward the response happens.
3.2. DEVELOPMENT OF THE PROTOCOL

We introduce two new variables to count the number of transferred questions and responses, called question and response.

| variables: | question, response |

Since each round consists of both, a question and a response, at the end of a round question must be the same as response.

Within a round, more precisely after the question and before the response, the numbers differ. Namely question is one greater than response. This is written down in the following invariants.

| invariants: |
| question = response \lor question = response + 1 |
| round = response |

We also see in the invariants, that round is connected to response. This is quite intuitive, since a round is finished as soon as the transfer of the response has taken place.

The variable round has been refined by question and response, and therefore we also have to refine event finishRound. It will be refined by respond, the event representative for the transfer of a response. It can only occur when question = response (at the beginning/end of a round).

For the transfer of a question there is a new event question. At this abstract level it only increments the number of transferred questions, as respond does for the responses.

Both variables, question and response, are initialised with 0.

| init |
| begin |
| question := 0 |
| response := 0 |
| end |
We have now a nice abstraction of the question/response protocol that is synchronous. But so far, no message is really transferred. Actually there are not even messages. We only increment the number of fictive messages.

We are going to change that in the next refinement.

3.2.3 Pattern Recognition

We now have to model the transfer of the questions and responses. Because the property of synchronisation is already defined, we can look at the exchange of the messages as independent operations. In other words, instead of a two-way communication we picture it as two one-way communications (Figure 3.2).

![Figure 3.2: Splitting of two-way communication](image)

This leads to two channels over which multiple messages can be transferred. One for the questions and one for the responses. But since the communication is synchronous there cannot be more than one message at a time.
Having a model of such a communication channel would be quite useful, because they appear a lot in communication protocols. Such a reusable model would be exactly what we call a pattern.

### 3.2.4 Pattern Description

Fortunately, we have a pattern that fits the required properties. Accordingly to its functionality it is called synchronous multiple-message communication pattern. Let us take an excursus and see what it looks like.

<table>
<thead>
<tr>
<th>variables: transferred</th>
<th>invariants: transferred ∈ ( \mathbb{N} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>( \text{transferred} := 0 )</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>transfer</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>( \text{transferred} := \text{transferred} + 1 )</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

This is the abstract model of the pattern. We refer to it as the pattern specification. We will see that this specification describes the usage interface of the pattern.

The model consists of a single variable \( \text{transferred} \), that represents the number of transferred messages. \( \text{transferred} \) is initially set to 0. In order to transfer a message (represented by event transfer) the message counter is incremented.

The solution of how to realise a communication channel is introduce by the refinement of the pattern. It looks as follows.

<table>
<thead>
<tr>
<th>variables: ( \text{sent}, \text{channel}, \text{received} )</th>
</tr>
</thead>
</table>
| invariants:
| \( \text{sent} = \text{transferred} \)
| \( \text{channel} = 0 \Leftrightarrow \text{sent} = \text{received} \)
| \( \text{channel} \neq 0 \Leftrightarrow \text{sent} = \text{received} + 1 \land \text{sent} = \text{channel} \)
| \( \text{received} \in \mathbb{N} \) |
CHAPTER 3. EXAMPLE - QUESTION/RESPONSE PROTOCOL

The refinement of the pattern implements a channel. The messages are represented by natural numbers. The bigger the number the newer the message. The number of the last message sent is stored in the variable \textit{sent}, the number of the latest received message in \textit{received}.

A message is sent by putting its number into the channel (done by \textbf{send}). To receive a message, its number is taken out of the channel (done by \textbf{receive}).

Note, that in this pattern \textit{transferred} is refined by \textit{sent}, meaning, a message counts as transferred as soon as it is sent.

The channel is limited to one single message at a time. Therefore the word synchronous in the pattern name. A new message cannot be sent before the previous one has been received.

3.2.5 Second Refinement

The pattern specification looks very familiar to the transfer of \textit{question} and \textit{response} in the question/response protocol. Since there is a solution in the pattern, how we could realise the communication channel, and the pattern specification is very similar to the problem at hand, we are going to reuse this idea in our development.

The solution of the problem (realising a synchronous multiple-message channel) is expressed by the pattern refinement. We therefore try to apply the pattern refinement to our protocol development as shown in Figure 3.3.
We actually have two positions where we can use the pattern, for the transfer of questions as well as of responses. We look first at the transfer of questions. Let us compare the pattern specification with our development.

Matching
We have to analyse to what extent our problem at hand (transfer of questions) fits into the pattern specification. Remember we want to apply the pattern refinement to our development. We say, we match the two models.

First we compare the variables. On the left side the pattern on the right the development.

| variables: transferred | variables: question, response |

As we use the pattern for the transfer of questions, transferred goes together with question. response is not matched and therefore ignored for the moment. It will not be changed when we apply the pattern refinement.

\[
\text{transferred} \sim \text{question} \\
\text{variable not matched: response}
\]
After having declared which variables go together, we have a look at the events that alter these variables. Again, left the pattern, right the development.

Both \texttt{init} events contain the same action for the matched variable. Of course in the development, \texttt{init} also contains an action for the not-matched variable \texttt{response}. We have to keep track of it, such that it does not get lost when we refine \texttt{init}.

We refer to such an action as an extra action since this is an additional element from the perspective of the pattern.

It is intuitive that the two \texttt{init} events are matched. But for the remaining events, we have to declare which events build a matching pair. In our development it is not hard to figure out that \texttt{transfer} should be matched with \texttt{question}. 
Checking

After we have matched two events, we always have to check whether they are really compatible. More precisely, we check if the contents of the events are the same. In this example we have two content groups, guards and actions. There can also be parameters. However, we are not looking at those in this thesis.

In terms of guards, we have to check that all guards of the events in the pattern specification are also in the corresponding events in the development. For the \textit{init} events this is always the case, as these events never have guards.

In our example, all guards of \textit{transfer} have also to be in \textit{question}. This is again trivial as \textit{transfer} has no guards either.

As we see, \textit{question} does have a guard concerning the not-matched variable \textit{response}. We have to keep track of this extra guard.
Not only the guards have to be checked. The actions need some attention too. We certainly want to match two events only if they do the same. With other words, the matched events must alter the matched variables in the same way.

In \texttt{init}, the matched variables \texttt{transferred} and \texttt{question} are both set to 0. The event matching \texttt{transfer} $\leadsto$ \texttt{question}, is also successfully checked, since \texttt{transferred} $:=$ \texttt{transferred} $+$ 1 is syntactically the same as \texttt{question} $:=$ \texttt{question} $+$ 1.

In the development, \texttt{init} has an extra action that alters the variable \texttt{response}. This is legitimate, since \texttt{response} is not matched. But we have to keep track of this extra action, as of the extra guard, when generating the refinement.
3.2. DEVELOPMENT OF THE PROTOCOL

init
...
response := 0
...

Note that also for question it would be legitimate to have extra actions for not-matched variables (but only for those).

Having all matched events checked, we check that no not-matched event alters a matched variable. This is required by the meta-proof described in Chapter 4 as well as by the fact that the functionality of the pattern should be congruent with that of the part in the development that uses the pattern.

respond
refines finishRound
when
...
then
response := response + 1
end

We convince ourselves that the remaining event respond does not alter the variable question.

The matching and checking is done. Before we continue, let us have a short overview of the tasks that were done so far in order to use the pattern.
M1 Matching of the variables
M2 Matching of the events

C1 Syntactical check of the guards of each matched event
C2 Check for extra guards
C3 Syntactical check of the actions of each matched event
C4 Check for extra actions
C5 Check that no not-matched event in the development alters a matched variable.

Listing 3.1: Matching and checking

The next step is to use the solution given by the pattern refinement and create a new refinement in our development.

Incorporation

The refinement, we are going to generate, will base on the first refinement of the development (seen in the Section 3.2.2) and the pattern refinement (Section 3.2.4). We call this copying the incorporation of the pattern.

The incorporation is subdivided into three parts, variables, invariants and events.

First, we merge the variables. We take the not-matched variable response and the three variables sent, channel and received from the pattern refinement.

\[
\begin{align*}
\text{variables:} & \quad \text{sent, channel, received} \\
\text{variables:} & \quad \text{question, response}
\end{align*}
\]
With regard to the second usage of the pattern, it is probably a good idea to rename the variables \textit{sent}, \textit{channel} and \textit{received} as \textit{sent\_question}, \textit{channel\_question} and \textit{received\_question} accordingly. This leads to the following new variables.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{variables:} & \textit{sent\_question, channel\_question, received\_question, response} \\
\hline
\end{tabular}
\end{center}

Next, we copy the invariants from the pattern refinement. By reason of the meta-proof, we also copy the invariants of the pattern specification.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{invariants:} & transferred \(\in\mathbb{N}\) \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{invariants:} & \\
\hline
sent = transferred \\
channel = 0 \(\Leftrightarrow\) sent = received \\
channel \neq 0 \(\Leftrightarrow\) sent = received + 1 \(\land\) sent = channel \\
received \(\in\mathbb{N}\) \\
\hline
\end{tabular}
\end{center}

Note that \textit{transferred} is unknown to the development. We therefore replace all occurrences of \textit{transferred} by \textit{question}, according to the matching. Also the renaming of the variables has to be taken into account.
invariants:

\[
\begin{align*}
\text{question} & \in \mathbb{N} \\
\text{sent}_{\text{question}} &= \text{question} \\
\text{channel}_{\text{question}} = 0 & \iff \text{sent}_{\text{question}} = \text{received}_{\text{question}} \\
\text{channel}_{\text{question}} \neq 0 & \iff \\
& \quad \text{sent}_{\text{question}} = \text{received}_{\text{question}} + 1 \land \\
& \quad \text{sent}_{\text{question}} = \text{channel}_{\text{question}} \\
\text{received}_{\text{question}} & \in \mathbb{N}
\end{align*}
\]

We now generate the events in the new refinement. There are three
different types of events.

- The first group contains all events from the development that were not
  matched. (respond)

- In the second group are the new events from the pattern refinement,
  which do not refine an event from the pattern specification. (receive)

- Group three contains of the events from the pattern refinement that
  refine a matched event. (init, send)

We start with the incorporation of the event from group one, the not-
matched event respond.

We have to copy respond from the previous refinement one-to-one to the
new refinement. It does refine the event where it was copied from.

\[
\begin{align*}
\text{respond} & \quad \text{refines} \quad \text{finishRound} \\
\text{when} & \quad \text{question} \neq \text{response} \\
\text{then} & \quad \text{response} := \text{response} + 1 \\
\text{end}
\end{align*}
\]

The variable question gets refined and therefore disappears in the new
refinement. Thus, question has to be replaced by an equivalent variable that
still exists.

We know from the matching that transferred = question holds. But
transferred does not exist in the refinement either. Looking at the invariants
3.2. DEVELOPMENT OF THE PROTOCOL

above, we recognise the glueing invariant \( \text{sent\_question} = \text{question} \). We now have a substitution rule for all appearances of \( \text{question} \).

\[
\text{question} = \text{sent\_question} \quad \text{I3}
\]

Replacing all appearances of \( \text{question} \) in the just copied event \( \text{respond} \) by \( \text{sent\_question} \), leads to the incorporation of group one as follows.

\[
\begin{align*}
\text{respond} & \quad \text{refines} \quad \text{respond} \\
\text{when} & \quad \text{sent\_question} \neq \text{response} \\
\text{then} & \quad \text{response} := \text{response} + 1 \\
\text{end}
\end{align*}
\quad \text{I4}
\]

For the incorporation of the events in group two (the new events from the pattern refinement), we copy the events one-to-one to the new refinement. We only have to remember the renaming of the variables.

It is reasonable to copy the event name from the pattern refinement too, since this model describes the functionality of the events best. However, to avoid confusion and because the pattern will be used a second time, we call the new event \( \text{receive\_question} \).
The events of group three are, in terms of incorporation, the most interesting ones. These events have to be build together with elements from the pattern refinement as well as the former refinement of the development. In this example there are two events in this group, init and send.

As basis for the new event init we take init from the pattern refinement and add the extra action from the former refinement in the development. The extra action was discovered in checking step C4.
Again, the renaming of the variables have to be taken into account.

For event `send`, we proceed in the same way. Instead of extra actions, this time it is an extra guard (discovered in checking step C2) that has to be added. The new event is renamed to `send_question` for the same reason as before. Note also that `send` refines `question`, instead of `transfer` like in the pattern refinement. We used the matching to find the right event.
As before, we have to rename the variables coming from the pattern refinement. In the added extra guard there appears the refined variable `question`. We have to replace this by `sent_question` according to the substitution rule in I3.

Here, an overview of the complete second refinement.

**variables:** `sent_question, channel_question, received_question, response`
### 3.2. DEVELOPMENT OF THE PROTOCOL

#### Invariants:

question ∈ N

sent_question = question

channel_question = 0 ⇔ sent_question = received_question

channel_question ≠ 0 ⇔ sent_question = received_question + 1 ∧ sent_question = channel_question

received_question ∈ N

#### Init

begin
  
  sent_question := 0
  channel_question := 0
  received_question := 0
  response := 0

end

#### Send_question

refines question

when
  channel_question = 0
  sent_question := response
then
  sent_question := sent_question + 1
  channel_question := sent_question + 1
end

#### Receive_question

when
  channel_question ≠ 0
then
  channel_question := 0
  received_question := channel_question
end
respond
defines respond
when
    sent_question ≠ response
then
    response := response + 1
end

Before we apply the pattern a second time for the transfer of responses, let us just summarise the several copying jobs.

I1 Merging the variables
I2 Copying of the invariants
I3 Extraction of the glueing invariants
I4 Copying of the not-matched events
I5 Copying of the new pattern events
I6 Merging the refined pattern events with the extra guards and extra actions

Listing 3.2: Incorporation

3.2.6 Third Refinement

For the transfer of responses, exactly the same steps are done to use the synchronous multiple-message pattern. Therefore only a brief overview of the matching, the checking and the resulting model after incorporation is given.
3.2. DEVELOPMENT OF THE PROTOCOL

Matching of the Variables (M1)

\[
\text{transferred} \leadsto \text{response} \\
\text{variables not matched: } \text{sent}_\text{question}, \text{channel}_\text{question}, \text{received}_\text{question} \quad \text{M1}
\]

Matching of the Events (M2)

\[
\text{init} \leadsto \text{init} \\
\text{transfer} \leadsto \text{respond} \quad \text{M2} \\
\text{events not matched: } \text{send}_\text{question}, \text{receive}_\text{question}
\]

The syntactical check results in the following.

Syntactical Check of the Guards of Each Matched Event (C1)

\[
\text{init} \quad \text{no guards} \\
\text{transfer} \quad \text{no guards} \quad \text{C1}
\]
CHAPTER 3. EXAMPLE - QUESTION/RESPONSE PROTOCOL

Check for Extra Guards (C2)

init
  no extra guards
question
  sent_question ≠ response

Syntactical Check of the Actions of Each Matched Event (C3)

init
  transferred := 0 ≡ response := 0

transfer
  transferred := transferred + 1 ≡ response := response + 1

Check for Extra Actions (C4)

init
  sent_question := 0
  channel_question := 0
  received_question := 0
respond
  no extra actions
3.2. DEVELOPMENT OF THE PROTOCOL

Check Not-matched Events (C5)

<table>
<thead>
<tr>
<th>send_question</th>
<th>no change of matched variables</th>
<th>C5</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive_question</td>
<td>no change of matched variables</td>
<td></td>
</tr>
</tbody>
</table>

The pattern can now be incorporated according to the matching and following the incorporation steps summarised in Listing 3.2. Before showing the result, note that we again renamed the new variables and events according to the following list.

<table>
<thead>
<tr>
<th>sent</th>
<th>sent_response</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>channel_response</td>
</tr>
<tr>
<td>received</td>
<td>received_response</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>send</th>
<th>send_response</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
<td>receive_response</td>
</tr>
</tbody>
</table>

Furthermore, the extraction of the glueing invariant led to the substitution of response by send_response. The replacements in send_question and send_response are underlined.

variables: send_question, channel_question, received_question, send_response, channel_response, received_response
invariants:
\[ \text{response} \in \mathbb{N} \]
\[ \text{sent}_{\text{response}} = \text{response} \]
\[ \text{channel}_{\text{response}} = 0 \iff \text{sent}_{\text{response}} = \text{received}_{\text{response}} \]
\[ \text{channel}_{\text{response}} \neq 0 \iff \]
\[ \text{sent}_{\text{response}} = \text{received}_{\text{response}} + 1 \land \]
\[ \text{sent}_{\text{response}} = \text{channel}_{\text{response}} \]
\[ \text{received}_{\text{response}} \in \mathbb{N} \]

init
begin
\[ \text{sent}_{\text{question}} := 0 \]
\[ \text{channel}_{\text{question}} := 0 \]
\[ \text{received}_{\text{question}} := 0 \]
\[ \text{sent}_{\text{response}} := 0 \]
\[ \text{channel}_{\text{response}} := 0 \]
\[ \text{received}_{\text{response}} := 0 \]
end

send_{question}
refines send_{question}
when
\[ \text{channel}_{\text{question}} = 0 \]
\[ \text{sent}_{\text{question}} = \text{sent}_{\text{response}} \]
then
\[ \text{sent}_{\text{question}} := \text{sent}_{\text{question}} + 1 \]
\[ \text{channel}_{\text{question}} := \text{sent}_{\text{question}} + 1 \]
end
3.2. DEVELOPMENT OF THE PROTOCOL

```
receive_question
  status  extended
  when
    channel_question ≠ 0
  then
    channel_question := 0
    received_question := channel_question
  end

send_response
  refines respond
  when
    channel_response = 0
    sent_question ≠ sent_response
  then
    sent_response := sent_response + 1
    channel_response := sent_response + 1
  end

receive_response
  when
    channel_response ≠ 0
  then
    channel_response := 0
    received_response := channel_response
  end
```

3.2.7 Fourth Refinement

After applying two instances of the synchronous multiple-message communication pattern, there has to be done some standard refinement. One such a refinement is the following.

As this is a model for a communication protocol, the only variable that both parties should have access to is the channel. Local variables like sent and received are only in the view of the corresponding party.
We know that the responder has to wait until a new question has been transferred, before he can transfer the response. He has therefore to watch the global variable \textit{question} (that denotes the number of transferred questions). By applying the patterns, however, this global transferred status was replaced by the local variable \textit{sent-question}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{local_variables}
\caption{View of the local variables}
\end{figure}

Through refinement, the responder would now have to watch \textit{sent-question} which is not in his view. When we add the invariant \textit{received-question} $\geq$ \textit{sent-response}, the guard in \textit{send-response} can be strengthened.

The guard \textit{sent-question} $\neq$ \textit{sent-response} is changed to the stronger guard \textit{received-question} $\neq$ \textit{sent-response}. Now, both of the concerned variables are in the view of the responder.

\begin{verbatim}
send_response
when
  ...
  sent_question $\neq$ sent_response
then
  ...
end

send_response (local view)
when
  ...
  received_question $\neq$ sent_response
then
  ...
end
\end{verbatim}
A corresponding refinement has to be done for the questioner. In event $\text{send}_{\text{question}}$ the guard $\text{sent}_{\text{question}} = \text{sent}_{\text{response}}$ is replaced by the stronger guard $\text{sent}_{\text{question}} = \text{received}_{\text{response}}$.

An additional invariant is needed tightening the synchronisation property, such that the questioner cannot send a new question before the pending response was received.

$$\text{sent}_{\text{question}} = \text{received}_{\text{response}} \lor \text{sent}_{\text{question}} = \text{received}_{\text{response}} + 1$$

The final refinement of the question/response protocol looks like this.

```plaintext
variables: \text{sent}_{\text{question}}, \text{channel}_{\text{question}}, \text{received}_{\text{question}}, \\
                 \text{sent}_{\text{response}}, \text{channel}_{\text{response}}, \text{received}_{\text{response}}

invariants: \\
\text{received}_{\text{question}} \geq \text{sent}_{\text{response}} \\
\text{sent}_{\text{question}} = \text{received}_{\text{response}} \lor \\
\text{sent}_{\text{question}} = \text{received}_{\text{response}} + 1

init \\
begin \\
\text{sent}_{\text{question}} := 0 \\
\text{channel}_{\text{question}} := 0 \\
\text{received}_{\text{question}} := 0 \\
\text{sent}_{\text{response}} := 0 \\
\text{channel}_{\text{response}} := 0 \\
\text{received}_{\text{response}} := 0 \\
end

\text{send}_{\text{question}} \\
\text{refines} \text{send}_{\text{question}} \\
\text{when} \\
\text{sent}_{\text{question}} = \text{received}_{\text{response}} \\
\text{then} \\
\text{sent}_{\text{question}} := \text{sent}_{\text{question}} + 1 \\
\text{channel}_{\text{question}} := \text{sent}_{\text{question}} + 1 \\
\text{end}
```
3.3 Proof Statistics

As we saw, using a pattern is nothing that happens by clicking on a magic button. Of course there are certain steps, which can be automated to disburden the developer, for example annoying copying and pasting jobs. But there are still parts in the pattern usage approach, that have to be done by the developer.

Therefore using patterns should have a clear advantage that justifies its appliance. As already mentioned in the beginning, besides the reusability of

```plaintext
receive_question
status extended
when
  channel_question ≠ 0
then
  channel_question := 0
  received_question := channel_question
end

send_response
refines send_response
when
  received_question ≠ sent_response
then
  sent_response := sent_response + 1
  channel_response := sent_response + 1
end

receive_response
status extended
when
  channel_response ≠ 0
then
  channel_response := 0
  received_response := channel_response
end
```
former developments, using patterns saves proofs.

To demonstrate this effect, Table 3.1 lists the number of proofs for each model of the question/response pattern development without using patterns. The numbers of proofs that can be saved are highlighted.

<table>
<thead>
<tr>
<th>model number</th>
<th>number of proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>question_response_0</td>
<td>2</td>
</tr>
<tr>
<td>question_response_1</td>
<td>5</td>
</tr>
<tr>
<td>question_response_2</td>
<td>15</td>
</tr>
<tr>
<td>question_response_3</td>
<td>14</td>
</tr>
<tr>
<td>question_response_4</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td><strong>48</strong></td>
</tr>
</tbody>
</table>

**savings**  

<table>
<thead>
<tr>
<th></th>
<th>29</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%</td>
<td>(60.4%)</td>
<td>(75.0%)</td>
</tr>
</tbody>
</table>

Table 3.1: Number of proofs in each refinement level

As we see, using patterns in the development of the question/response protocol results in saving about 60% of the proofs. Even 75% were saved of the manual proofs.

### 3.4 A Development Becomes a Pattern

The very fascinating thing now is, that this question/response protocol can be used as a pattern itself. This is actually done in Chapter 5 as we will see. In this case the first refinement in Section 3.2.2 acts as the pattern specification and the just finished fourth refinement is the pattern refinement.

You may already imagine how powerful this pattern approach is with its recursive applicability.

With this example we got a good impression how to use patterns. Although the reader may be convinced of the correctness of this approach already, we are going to see its formalisation in the next chapter.
Chapter 4

Pattern Usage

In this chapter we will see exactly how the concept of patterns in Event-B works. The theoretical basis, including the proofs about the correctness of the pattern usage approach, is explained. Thus, the reader can convince himself of the correctness.

In contrast to the example in Chapter 3, the models do not consist of specific variables and events. Instead, we use more abstract models containing letters that represent a group of guards or a group of actions, for example.

Therefore, also the obligations are going to be proved on this abstract level. This results in proving the correctness of the construction rather than of a certain model. We call this kind of proof a meta-proof.

4.1 Overview

What we want to offer a developer with this pattern usage approach is the possibility to reuse solutions from former developments in the current project. More precisely, to give the developer a methodology to generate a new refinement of the problem at hand where a certain part of the model is replaced accordingly to a pattern that already exists.

This will lead to a correct refinement in the chain of models in the development, without arising proof obligations. Since the correctness of the pattern has been proved during its development, nothing is to prove again when using this pattern.

This statement is quite bold. We will see that for the correctness of the refinement, we first have to match the pattern with the current development. The connection of the involved models can be seen in Figure 4.1.
4.1.1 Procedure

The development has arrived at refinement \( n \). At this point the developer has noticed that he could make use of a pattern. In order to use the pattern, it somehow has to be compared with the development, to see if this pattern is even applicable. This is done by matching and subsequent checking, that is explained in Section 4.2 and Section 4.3 respectively.

After linking the development together with the pattern through matching, the refinement of the pattern (the pattern refinement) has to be entered in the development. This step is called incorporation and is explained in more detail in Section 4.3.

Sometimes it is necessary to rename variables if there is a clash (a variable name in the refinement of the pattern already exists in the development). Even if there is no clash, a renaming could be desired by the developer to give the variables, coming from the pattern, a more meaningful name within the development. Renaming is a straightforward operation.

4.2 Matching

The first task in the process of using a pattern is defining the linking, or matching as we name it, between pattern specification (the abstract initial model of the pattern) and refinement \( n \) of the development. The steps of the matching are as follows.

M1 Define which variable of the pattern specification goes together with which variable of refinement \( n \).
4.2. MATCHING

M2 Define which event of the pattern specification goes together with which event of refinement n.

4.2.1 Matching of the Variables (M1)

As we can see in Figure 4.2, the pattern specification has variables $v$ and refinement n has variables $a$ that have to be matched together. Usually refinement n consists of much more variables than the pattern, thus, not all of the variables from $a$ are matched.

<table>
<thead>
<tr>
<th>pattern specification</th>
<th>refinement n</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables:</td>
<td>variables:</td>
</tr>
<tr>
<td>$v$</td>
<td>$a$</td>
</tr>
<tr>
<td>invariants:</td>
<td>invariants:</td>
</tr>
<tr>
<td>$J(v)$</td>
<td>$I(a)$</td>
</tr>
<tr>
<td>events:</td>
<td>events:</td>
</tr>
<tr>
<td>$p$</td>
<td>$e$</td>
</tr>
<tr>
<td>when $L(v)$</td>
<td>when $G(a)$</td>
</tr>
<tr>
<td>then $T(v, v')$</td>
<td>then $Q(a, a')$</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

Figure 4.2: Pattern specification and refinement n before matching

Therefore variables $a$ are split into $b$ and $c$, where $b$ consists of all the variables that are matched with $v$ and $c$ consists of the remaining variables of refinement n. $c$ are the variables that will not be refined by the usage of the pattern.

This splitting of the variables leads to a new illustration of the models in Figure 4.3 where also the guards and actions are expressed with the new variables.
4.2.2 Matching of the Events (M2)

In the same way as the variables were matched, we have to define which events in refinement $n$ are the counterparts of the events in the pattern specification. In this general model, we only have one event $e$ in refinement $n$ and one event $p$ in the pattern specification as representatives for all matched events.

Here again, there are probably more events in refinement $n$ than can be matched with the pattern. The remaining events that are not matched will not be touched by the pattern usage (unless renaming), like the variables $c$. Therefore, these events are not shown in our general model.

Nevertheless, we have to keep track of those when we later incorporate the pattern and generate the new refinement $n+1$ of the development.

4.3 Syntactical Check of the Matching

Although we have event $p$ matched with event $e$, there is no guarantee for them being really compatible. But without this guarantee, the pattern approach is not valid. We therefore have to check that the chosen matching holds.
4.3. **SYNTACTICAL CHECK OF THE MATCHING**

The checking is subdivided into the following five steps.

- **C1** Check that the guards of each event of the *pattern specification* are syntactically the same as the guards of the corresponding event in *refinement n*.

- **C2** Check if there are extra guards in the matched events of *refinement n*.

- **C3** Check that the actions of each event of the *pattern specification* are syntactically the same as the actions of the corresponding event in *refinement n*.

- **C4** Check if there are extra actions in the matched events of *refinement n*.

- **C5** Check that no event in *refinement n* that is not matched alters a matched variable.

### 4.3.1 Checking of the Guards (C1,C2)

In terms of guards, we often speak of being stronger or weaker. We do not want the guards of $p$ to be stronger than those of $e$. Otherwise, there are variable constellations where $e$ is enabled while $p$ is not. Using the solution from the pattern (applying the pattern refinement) would then unintentionally change the behavior of the model.

To avoid this change of behavior, the guards of $e$, i.e. $G(b, c)$ have to be at least as strong as the guards of $p$, i.e. $L(v)$. But what happens when $G(b, c)$ is stronger than $L(v)$? When we carelessly apply the pattern refinement to the development, the behavior would also change.

In contrast to the situation before, this time, the model would be changed towards a too casual enabling of the event. This defect can be repaired by strengthening the guards of the affected event in the refinement.

The drawback of comparing the strength of guards is that this cannot be checked, it has to be proved and we are not interested in new proofs. Instead of $G(b, c)$ being at least as strong as $L(v)$, we want $e$ to contain the syntactically same guards as $p$ does, plus some possibly additional guards. Those additional guards are called extra guards. This is equivalent to the previous statement about the strength but has the advantage that it can be syntactically checked.

The group of guards that are syntactically the same as $L(v)$ are denoted as $H(b)$, whereas the extra guards are represented by $N(b, c)$. Note that the guards in $H(b)$ can only depend on $b$, since variables $c$ are not matched. The extra guards, however, can depend on all variables.
In Figure 4.4 the former guards $G(a)$ are now split into two groups of guards, $H(b)$ and $N(b, c)$

### 4.3.2 Checking of the Actions (C3,C4)

As for the guards we proceed for the comparison of the actions. The actions are specified in the form of before-after predicates (BAP). The group of BAPs in $p$, i.e. $T(v, v')$ has to be syntactically the same as a subgroup of the BAPs in $e$, i.e. $R(b, b')$, namely those BAPs that change the matched variables ($b$).

Event $e$ can have extra actions that alter the variables $c$, i.e. $S(b, c, c')$. Again, the extra guards and extra actions are not touched by the pattern and therefore we have to keep track of them when generating the new refinement.

---

\[\begin{align*}
\text{pattern specification} & \\
\text{variables:} & \quad v \\
\text{invariants:} & \quad J(v) \\
\text{events:} & \\
\quad & \begin{cases}
    \text{when} & L(v) \\
    \text{then} & T(v, v') \\
    \text{end}
\end{cases}
\end{align*}\]

\[\begin{align*}
\text{refinement n} & \\
\text{variables:} & \quad b, c \\
\text{invariants:} & \quad I(b, c) \\
\text{events:} & \\
\quad & \begin{cases}
    \text{when} & H(b) \\
    \text{then} & N(b, c) \\
    \text{end}
\end{cases}
\end{align*}\]

\[\begin{align*}
\quad & \begin{cases}
    \text{when} & R(b, b') \\
    \text{then} & S(b, c, c') \\
    \text{end}
\end{cases}
\end{align*}\]

---

**Figure 4.4: Guards and actions split**

### 4.3.3 Checking of the Remaining Events (C5)

The later described meta-proofs base on the fact that no event other than the matched ones alters the matched variables. So far, this restriction has not limited the usage of patterns in the case studies. In the future work it has
to be evaluated if this restriction could have an impact on the applicability of patterns.

To preserve this restriction, we have to check all not-matched events in addition. This check fails if there is an event that is not matched but consists of an action changing a matched variable from $b$.

4.4 Incorporation

After matching the models, we should have a look at the refinement of the pattern. The variables $v$ have changed to $w$. There is a group of invariants $K(v, w)$ that also includes glueing invariants connecting the pattern specification with the pattern refinement (see Figure 4.5).

![Pattern Refinement Diagram](image)

The new event $q$ refines former event $p$ with guards $M(w)$ and BAP $U(w, w')$. This, together with the elements in refinement $n$, that are not affected by the pattern usage, build the basis for the incorporation of the pattern.

The generation of the new refinement $n+1$ will include the following parts.
I1 Merge the variables from the pattern refinement with the not-matched variables from refinement n.

I2 Copy the invariants from the pattern specification and the pattern refinement.

I3 Extract the glueing invariants out of the invariants from the pattern refinement.

I4 Copy all not-matched events from refinement n.

I5 Copy all new events from the pattern refinement.

I6 Merge all events from the pattern refinement that refine a matched event with the related extra guards and extra actions from refinement n.

4.4.1 Merging the Variables (I1)

The variables in the new refinement $n+1$ consist of the not-matched variables $c$ from the former refinement $n$ and the new variables $w$ evolved in the pattern refinement.

Note that variables $c$ are renamed to $d$. This is made to keep the proofs as general as possible. Thus we will add the invariant $c = d$, which is only necessary in this abstract model for the meta-proof.

4.4.2 Copying of the Invariants (I2)

For the correctness of the new refinement $n+1$ we also need to copy the invariants from the pattern. Namely the invariants from the pattern specification $J(v)$ and the invariants from the pattern refinement $K(v, w)$.

Since $v$ does not exist in refinement $n$ but is defined to be the same as $b$ (through matching), $J(v)$ and $K(v, w)$ can be stated as $J(b)$ and $K(b, w)$ respectively.
The invariant \( c = d \) is added for the reason described before.

\[
\begin{align*}
\text{invariants:} & \quad J(v) \\
invariants: & \quad K(v, w) \\
c = d
\end{align*}
\]

### 4.4.3 Extraction of the Gluing Invariants (I3)

In the extra guards and extra actions as well as in the events not matched, there might be variables from \( b \) used as reference. In refinement \( n+1 \) though, \( b \) does not exist anymore. Through matching, \( b \) is the same as \( v \), but \( v \) does not exist either. Hence we need to extract a relation between \( b \) and \( w \).

As the pattern specification and the pattern refinement are connected, there must exist a glueing invariant in \( K(v, w) \) that expresses their relation. By extracting this glueing invariant, we get the connection between these variables \( v \) and \( w \) in form of the expression \( v = X(w) \).

We already replaced \( v \) by \( b \) and can therefore directly extract the expression \( b = X(w) \) out of \( K(b, w) \).

\[
\begin{align*}
\text{b = X(w)}
\end{align*}
\]

As we discovered so far in our research about patterns, this glueing invariants are usually simple equalities \( b = w \).

### 4.4.4 Copying of the Not-matched Events (I4)

All remaining events from the former refinement \( n \) are copied one-to-one to the new refinement \( n+1 \). All appearances of \( b \) in these events are replaced by \( X(w) \).
4.4.5 Copying of the New Pattern Events (I5)

Through refinement of the pattern there probably evolved new events. These events are copied one-to-one from the pattern refinement to the new refinement \( n+1 \).

4.4.6 Merging the Refined Pattern Events with the extra guards and extra actions (I6)

In this last step all the events of the pattern refinement are copied, that refine an event of the pattern specification. In the new refinement \( n+1 \), these events will refine a matched event from refinement \( n \).

In our model, event \( r \) that refines event \( p \) is copied to the new refinement \( n+1 \) as event \( f \) refining event \( e \).

The extra guards and extra actions of the matched events, that we had to keep track of, are added to the events just copied.

In our model, event \( f \) gets the additional guards \( N(b, c) \) and additional actions \( S(b, c, c') \). Again, all appearances of \( b \) in these events are replaced by \( X(w) \).
The new refinement \( n+1 \) of the development, generated following the incorporation steps stated above, yields to the model in Figure 4.6.

**refinement \( n+1 \)**

**variables:**
- \( w, d \)

**invariants:**
- \( J(b) \)
- \( K(b, w) \)
- \( c = d \)

**events:**

\[
\begin{align*}
f & \text{ refines } e \\
& \text{ when} \\
& M(w) \\
& N(X(w), d) \\
& \text{ then} \\
& U(w, w') \\
& S(X(w), d, d') \\
& \text{ end}
\end{align*}
\]

Figure 4.6: Generated refinement \( n+1 \) of the development

### 4.5 Correctness

The correctness of this approach is proved by means of meta-proofs. In other words, the proof obligations that normally arise when developing a refinement are proved on a theoretical level. We make thereby use of the proof obligations from the pattern that has been proved to be correct.

Rather than proving the specific refinement, we prove the construction of the refinement, namely the incorporation of the pattern after a correct matching and successful checking. This results in a method for the usage of patterns that is correct by construction when following the matching rules and all the syntactical checks turn out to be true.
The proof obligations generated when doing a normal refinement are, accordingly to Section 2.2, the following. \texttt{FIS\_REF}, \texttt{GRD\_REF} and \texttt{INV\_REF} have to be proved for every event $f_i$ in the new refinement $n+1$.

\[
\begin{array}{|c|c|}
\hline
\text{Abstract invariants} & \text{FIS\_REF} \\
\text{Concrete invariants} & \\
\text{Concrete guards} & \\
\hline
\vdash & \exists w'. \text{Concrete before-after predicate} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Abstract invariants} & \text{GRD\_REF} \\
\text{Concrete invariants} & \\
\text{Concrete guards} & \\
\hline
\vdash & \text{Abstract guards} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Abstract invariants} & \text{INV\_REF} \\
\text{Concrete invariants} & \\
\text{Concrete guards} & \\
\text{Concrete before-after predicate} & \\
\hline
\vdash & \exists v'. \left( \text{Abstract before-after predicate} \land \text{Modified concrete invariants} \right) \\
\hline
\end{array}
\]

Applied to the models presented in Figure 4.2-4.6 we obtain the following proof obligations. These are the meta-proofs that, if proved to be correct, guarantee the correctness of the pattern usage approach.
4.5. CORRECTNESS

4.5.1 Meta-proof

In order to prove these obligations, we need information about the matching and the proofs already done during the development of the pattern. First, we notice that there are still different labels used for the variables that are
matched and therefore designated to be the same.

As a first step, we should normalize the naming of the variables accordingly to the matching. For example, the matching \( b = v \) can be taken as a hypothesis and with the EQ\_LR rule\(^1\) all free appearances of \( b \) are simply replaced by \( v \).

After substitution of the names, the different names of the matched guards and before-after predicates can be substituted accordingly to the matching. For example, \( H_i(v) \) becomes \( L_i(v) \). This is correct since \( b \) has been defined to be \( v \) and after this matching we checked \( H_i(b) \) and \( L_i(v) \) to be syntactically the same.

In Listing 4.1 all the matches are summarised as an overview. They are applied with the rule SUBST, whereas all occurrences of the left parts of the listed matching equations are substituted with the corresponding right side (multiple appliance of the EQ\_LR rule). After disposing duplicate namings we can start doing the proof.

\[
\begin{align*}
  b &= v \\
  X(w) &= v \\
  H_i &\equiv L_i \\
  R_{INIT} &\equiv T_{INIT} \\
  R_i &\equiv T_i
\end{align*}
\]

Listing 4.1: Matching

At some point we need the knowledge of the pattern proofs to get on. As the obligations that appeared during development of the pattern are proved, we can use them now as hypotheses. Listing 4.2 summarises all the proved obligations that are needed to do the meta-proofs. They originate from both, the pattern specification and the pattern refinement as well as the current development itself.

\(^1\)see Appendix A for more information about rules of inference in sequent calculus
obligations from the pattern specification

INV
\[ \forall v, v' \cdot J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \]  
(1)

obligations from the development

FIS
\[ I(b, c) \land H_i(b) \land N_i(b, c) \Rightarrow (\exists b', c' \cdot R_i(b, b') \land S_i(b, c, c')) \]  
(2)

obligations from the pattern refinement

GRD_REF
\[ J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \]  
(3)

INV_REF
\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow (\exists v' \cdot T_i(v, v') \land K(v', w')) \]  
(4)

FIS_REF
\[ J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w' \cdot U_i(w, w') \]  
(5)

Listing 4.2: Already proved obligations

Not every proved obligation of the pattern is used in every obligation of the meta-proof. Table 4.1 shows which obligation is needed for which proof. The detailed proofs are listed in Appendix B.

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRD_REF</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INV_REF</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIS_REF</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 4.1: Proved obligations used by meta-proofs
4.6 Conclusion

At this stage of the pattern usage concept we look at a configuration where the pattern consists of only two models, the *pattern specification* and the *pattern refinement*. It is quite imaginable to think of patterns with many refinement levels and therefore many positions where the matching could happen.

Depending on the matching position, the *pattern specification* may be already a refinement within the pattern. There could also be multiple refinement steps up to the refinement we would like to incorporate.

This does not violate the pattern approach since a refinement can always be seen as an abstract model. For the case, having several refinement steps, we can for each of them generate a new refinement in the development as shown in Figure 4.7.

![Pattern Development Diagram](image)

**Figure 4.7:** Most generic scenario when using patterns
Chapter 5

A2A Protocol

After we have seen an example of a pattern and the theoretical aspects, we are going to look at a larger development. We will identify places in the development where we can use patterns. Hopefully, we can also make use of the developed pattern example from Chapter §. The development we are going to look at, is a business communication protocol called Ordering/Supply Chain A2A Communication. We further refine to it as the A2A protocol.

The A2A protocol is a mini-pilot of SAP Research\textsuperscript{1} within the DEPLOY-project\textsuperscript{2}. It describes a protocol that is needed for the communication between components in the ordering process of goods. Although it is not the original protocol used by SAP, it fits their demands on a communication protocol.

By developing this protocol in Event-B in the traditional way, we were able to show where patterns could have been used and what we gain when doing so. Furthermore, while developing and looking for possible usage of patterns we got a better understanding of where a tool can support the developer and what the prerequisites are for using patterns.

5.1 The Protocol in Detail

The A2A protocol describes the communication between an ordering component and a supply chain component. The ordering component (OC) collects orders from buying organizations and sends this in form of requirements to the supply chain component (SCC). In return, SCC provides OC with information on the availability of the required goods.

\textsuperscript{1}SAP Research Center, CEC Darmstadt, Germany. http://www.sap.com

\textsuperscript{2}Deploy is an European Commission Information and Communication Technologies FP7 project. http://www.deploy-project.eu
The protocol goes on until either the required goods are available or \( OC \) aborts the protocol by deleting the requirement. The protocol can be seen as a two phase protocol where phase 1 can take forever and phase 2 finally terminates. In Event-B one would say phase 2 converges whereas phase 1 does not.

### 5.1.1 Phase 1

Phase 1 is kind of a negotiation phase, where \( OC \) asks for the availability of certain goods. In return \( SCC \) informs \( OC \) about the availability of this goods and provisionally reserves them. Although provisionally reserved, \( OC \) can still change the requirements by sending a new query, or simply abort the protocol when not interested in this goods anymore. This negotiation is stated as a state machine in Figure 5.1.

![Figure 5.1: State machine of phase 1](image)

The states are a combination of two separate statuses: \( RequirementStatus \) and \( CancellationStatus \). The \( RequirementStatus \) represents the flow of the ordering process whereas the \( CancellationStatus \) simply states whether the protocol is still running or not. Every transition between the states is linked with a message that is transferred.

In phase 1, the \( CancellationStatus \) is always set to \( NOT\_CANCELLED \). The \( RequirementStatus \) starts with \( ReqInit \), the initial status. This is also the starting state of the entire state machine of the A2A protocol (marked with the start arrow).

When \( OC \) transfers a \( AvailabilityCheckRequest \) (\( ACR \)), \( RequirementStatus \) changes to \( ReqQueried \). Now it is at \( SCC \) to transfer an \( AvailabilityConfirmation \) (\( AC \)) to \( OC \). When the goods are available \( RequirementStatus \) will change to \( ReqProv \) stating that the goods are provisionally reserved. The duration until the goods are available is smaller than infinity (\( d < \text{infinity} \)). Otherwise, if the goods are not available at all (\( d = \text{infinity} \)) \( RequirementStatus \) is set back to \( ReqInit \).
Important to note here is the synchronisation of this message exchange. After transferring an ACR message, OC has to wait (means is blocked) until it gets an AC message back. Accordingly, SCC has to wait for an ACR in order to transfer an AC as a response.

This question/response game can take forever as already mentioned. Once OC is satisfied with the information it got from SCC concerning the availability of the required goods, OC can enter phase 2 of the protocol.

### 5.1.2 Phase 2

In phase 2, the protocol will finally terminate except for one single case that we discuss later. The termination property means, in terms of state machines, that no loop exists. In Figure 5.2 one can see that almost all transitions are pointing towards the accepted states CANCELLED and ReqFulfilled/NOT CANCELLED.

The only loop is the transfer of FulfillmentUpdates (FU). But this transition can only occur within a certain time, namely until the reserved goods are available. This time was limited during the negotiation phase by the promised duration until the goods are available. Therefore the state machine cannot be caught within this loop forever.

It might seem a bit odd, that even the accepting states of the state machine have outgoing transitions. One could doubt the terminating property of phase 2, seeing this transitions. As we will see later on, this transitions can only occur once and only within a certain time slot caused by the communication latency.
The only exception that can violate the termination property is the case where OC directly reserves the goods without a negotiation in advance. If the goods are not available at all, the protocol may stick in \textit{ReqResv} (Reserved Status) until OC decides to abort the protocol.

After having discussed the termination property we should focus on the intention of phase 2. After negotiation and being either in \textit{ReqInit} or \textit{ReqProv}, OC can decide to trigger the reservation by transferring a \textit{RequirementFulfillmentRequest} (RFR). The \textit{RequirementStatus} is now reserved (\textit{ReqResv}) and OC has to wait until SCC has the goods available.

Until fulfillment, SCC has the option to transfer \textit{FulfillmentUpdates}. This updates are for information only. They, for example, inform OC when the goods are available earlier than promised.

As soon as the goods are available, SCC will transfer the \textit{FulfillmentConfirmation} (FC) and the protocol ends in the accepted state \textit{ReqFulfilled}.

During phase 2, OC has the possibility to abort the protocol by transferring a \textit{RequirementDeletion} (RD) that leads the protocol to the state \textit{CANCELLED}. In this state the protocol does not take the \textit{RequirementStatus} into account as it does not matter.

Now is the moment to explain why there are transitions going out of the accepted states. When SCC transfers FC, the protocol changes its state to \textit{ReqFulfilled}, even though the message has not been noticed yet by OC. Unaware of the fulfillment of the requirement, OC decides to abort the protocol and the state changes from \textit{ReqFulfilled} to \textit{CANCELLED}.

The same story happens in the other direction. OC aborts the protocol and the state changes to \textit{CANCELLED}. Before SCC gets aware of the transferred RD it transfers FC as a direct consequence of the just available goods. Unlike before the state will not change anymore, it remains in \textit{CANCELLED}; CANCELLED once - CANCELLED forever.

In Figure 5.3 on page 61 the entire state machine of the A2A protocol is shown.

Let us have two scenarios, as they could happen.
5.1. THE PROTOCOL IN DETAIL

<table>
<thead>
<tr>
<th>state</th>
<th>OC</th>
<th>SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqInit</td>
<td>transfer ACR “10 blue cars are required”</td>
<td>→</td>
</tr>
<tr>
<td>ReqQueried</td>
<td>transfer AC “not available”</td>
<td></td>
</tr>
<tr>
<td>ReqInit</td>
<td>transfer ACR “what about 10 <strong>silver</strong> cars”</td>
<td>→</td>
</tr>
<tr>
<td>ReqQueried</td>
<td>transfer AC “available in 4 weeks”</td>
<td></td>
</tr>
<tr>
<td>ReqProv</td>
<td>transfer ACR “how long take 5 silver cars”</td>
<td>→</td>
</tr>
<tr>
<td>ReqQueried</td>
<td>transfer AC “available in 1 weeks”</td>
<td></td>
</tr>
<tr>
<td>ReqProv</td>
<td>transfer RFR “reserve 5 silver cars”</td>
<td>→</td>
</tr>
<tr>
<td>ReqResv</td>
<td>transfer FU “already available in 4 days”</td>
<td></td>
</tr>
<tr>
<td>ReqResv</td>
<td>transfer FC “5 silver cars available”</td>
<td></td>
</tr>
</tbody>
</table>

Listing 5.1: Example of a run of the A2A protocol

In the first example of a run in the A2A protocol, OC tries to reserve 10 blue cars. As they are not available, it tries to reserve 10 silver cars instead. Although the 10 silver cars are available, the duration until they are available seems to be too long for OC.

It therefore tries another request for 5 silver cars. This time, OC is satisfied and reserves the cars. While OC is waiting for the cars, SCC updates OC, that the cars are available earlier.

As soon as the cars are available, SCC informs OC and the protocol ends.
CHAPTER 5. A2A PROTOCOL

<table>
<thead>
<tr>
<th>State</th>
<th>OC</th>
<th>SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqInit</td>
<td>transfer RFR “1 pallet of toilet paper” →</td>
<td></td>
</tr>
<tr>
<td>ReqResv</td>
<td>← transfer FC “toilet paper available”</td>
<td></td>
</tr>
<tr>
<td>ReqFulfilled</td>
<td>transfer RD “toilet paper not needed anymore” →</td>
<td></td>
</tr>
<tr>
<td>CANCELLED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 5.2: Another run of the A2A protocol

In this second example, OC does not even bother to negotiate with SCC to get information about the availability. It directly reserves a pallet of toilet paper as this is usually available.

As OC rightly assumed, the toilet paper is available and SCC informs OC about that. Before OC gets the information, it decides to cancel the reservation and therefore the protocol gets aborted.

5.2 Recognition of Patterns in the Protocol

After this overview of the A2A protocol, we are going to recognise spots in the protocol where pattern can be used.

5.2.1 Recognition of Patterns in Phase 1

Coming back to phase 1, the state diagram in Figure 5.1 looks quite symmetric. Also, the transferred messages are the same. Neither coming from left (ReqInit) or right (ReqProv) to the center (ReqQueried) nor going from the center to the left or right makes any difference in terms of the transferred messages.

Being aware of this fact, we fold the state machine and therefore get an abstraction of phase 1 (Figure 5.4). The states ReqInit and ReqProv are merged together to a combined state called ReqInit/ReqProv.
Figure 5.3: State machine of the A2A protocol
Analysing the issuer of the messages, the new state \textit{ReqInit/ReqProv} and the remaining state \textit{ReqQueried} can be associated with \textit{OC} and \textit{SCC} respectively (Figure 5.5). It turns out that this perfectly fits the question/response pattern developed in the beginning.

\begin{figure}[h]
  \centering
  \includegraphics[width=\textwidth]{figure5_4}
  \caption{Phase 1 folded}
\end{figure}

\begin{figure}[h]
  \centering
  \includegraphics[width=\textwidth]{figure5_5}
  \caption{Phase 1 assigned to the parties}
\end{figure}

\subsection{5.2.2 Recognition of Patterns in Phase 2}

Phase 2 consists basically of transitions where single messages are transferred. For this task there exists a pattern called single-message communication pattern that just fits this property. This pattern is described in more detail in Chapter 6.

The only transition that can happen more than once is the transfer of the \textit{FulfillmentUpdate} message. For this we use a multiple-message communication pattern. As the name indicates, several messages can be sent over the channel, even though the chronological order of the messages is not guaranteed. Figure 5.5 shows an overview of the recognised patterns in phase 2.
5.3 Development of the Protocol

The first model of the development (a2a_00) is a very abstract machine. It includes the two statuses RequirementStatus and CancellationStatus whose possible values are defined in the first context component (ctx_0).

The transitions are abstracted by two events that non-deterministically change the statuses, one for phase 1 (progress_phase1) and one for phase 2 (progress_phase2). According to the termination property, progress_phase1 has the convergent status ordinary whereas the status of progress_phase2 is anticipated.

There is an additional event termination that can happen when RequirementStatus is set to ReqFulfilled or CancellationStatus is set to CANCELLED.

variables: reqStatus, cancellationStatus

invariants:

reqStatus ∈ REQ_STATUS
 cancellationStatus ∈ CANCELLATION_STATUS
CHAPTER 5. A2A PROTOCOL

init
begin
  reqStatus := ReqInit
  cancellationStatus := NOT_CANCELLED
end

progress_phase1
when
  reqStatus ∈ {ReqInit, ReqQueried, ReqProv}
  cancellationStatus = NOT_CANCELLED
then
  reqStatus :∈ {ReqInit, ReqQueried, ReqProv}
end

progress_phase2
when
  reqStatus ≠ ReqQueried
then
  reqStatus :∈ REQ_STATUS
  cancellationStatus :∈ CANCELLATION_STATUS
end

termination
when
  reqStatus = ReqFulfilled ∨ cancellationStatus = CANCELLED
then
  SKIP
end

In the further refinement models, the abstract progress events are refined step-by-step by expressing the transitions. This transition events on the other hand are then refined by using patterns as they were recognised in Section 5.2.

When inserting the transition events, the progress events have to be restricted in order not to interfere with the new properties coming from the transitions. For example in phase 2, when inserting the RequirementFulfillmentRequest (RFR), we also insert the property that after arriving at Re-
qResv we cannot go back to the upper states ReqInit and ReqProv. Leaving progress\_phase2 unchanged would violate this property.

### 5.3.1 Refinement Strategy

As an overview of the A2A development, here is a list of all the models within the development, from the very abstract one to the last refinement. For each of them, it is stated what exactly happens in this refinement step. Of special interest are of course the refinement models where patterns were used.

**context ctx\_0**

- the types of the statuses RequirementStatus and CancellationStatus are defined as sets containing the possible values

**context ctx\_1**

- \textit{infinity} is defined as a certain natural number

**context ctx\_2**

- \(g\) and \(f\) are defined, two functions mapping the statuses to natural numbers (this is needed to specify the variant in phase 2)

**machine a2a\_00**

- variables RequirementStatus and CancellationStatus are defined

- events progress\_phase1 and progress\_phase2, and their possible status changes are defined

**machine a2a\_01**

- phase 1 is modeled as there are a \texttt{question} and a \texttt{response} event changing the status

- progress\_phase1 is refined and therefore disappears, since all possible transitions in phase 1 are stated

**machine a2a\_02**
• message exchange is modeled with transferred variables and connected to event question and response

• the exchange of the messages and therefore event question and response are defined to be synchronous

machine a2a_03

• question/response pattern is applied for question and response

machine a2a_04

• guard refinement (instead of checking the status, the message numbers are compared)

• event send_AC and receive_ACR are split into two events each (separated status ReqInit and ReqProv)

machine a2a_05

• variable duration is introduced to distinguish whether the status changes to ReqInit or ReqProv when receiving an AC

machine a2a_06

• the RD message transfer is modeled (three different events for ReqProv, ReqResv and ReqFulfilled)

• progress_phase2 adapted

machine a2a_07

• 3 single communication patterns are applied for transfer of RD

• receive_RD events are collected to one

machine a2a_07.5

• CancellationStatus refined and therefore disappears (guards concerning CancellationStatus are replaced by OC_sent_RD)
5.3. DEVELOPMENT OF THE PROTOCOL

- progress_phase2 adapted and split into two separate events (progress_phase2_fromOC and progress_phase2_fromSCC)

machine a2a_08

- the RFR message transfer is modeled (two different events for ReqInit and ReqProv)
- progress events adapted

machine a2a_09

- 2 single communication patterns are applied for transfer of RFR
- receive_RFR events are collected to one
- progress_phase2_fromOC is refined and therefore disappears

machine a2a_10

- the FC message transfer is modeled
- progress_phase2_fromSCC is refined and therefore disappears

machine a2a_11

- a single communication pattern is applied for transfer of FC

machine a2a_12

- RequirementStatus refined and therefore disappears (guards concerning RequirementStatus are replaced by local variables)

machine a2a_13

- OC_send_RD_fromResv and OC_send_RD_fromFulfilled are merged together, in order to remove the guard about SCC_sent_FC, which is not in the view of OC
- guard added to the merged event OC_send_RD following the requirements (OC_received_FC = FALSE)
machine a2a_14

- the $FU$ message transfer is modeled

machine a2a_15

- a multiple-message communication pattern is applied for transfer of $FU$

machine a2a_16

- channel introduced to send the remaining duration when $FU$ is transferred
- $SCC_{\text{send}}_{\text{FU}}$ is adapted to also feed the new duration channel

5.3.2 Proof Statistics

As in the development of the question/response pattern in Chapter 3 we calculate the number of proofs that were saved using patterns.

Table 5.1 lists the number of proofs for each model of the A2A development without using patterns. The numbers of proofs that can be saved are highlighted.

Using patterns in the development of the A2A protocol results in saving about 35% of the proofs. About the same ratio was saved at the manual proofs.

There are already ideas, how the way as the patterns were used could be optimised, in order to increase the savings. Therefore these 34.5% of savings in the A2A protocol can be taken as a lower bound. More details about the optimisation can be found in Chapter 8.
### Table 5.1: Number of proofs in each refinement level

<table>
<thead>
<tr>
<th>model number</th>
<th>number of proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>contexts</td>
<td></td>
</tr>
<tr>
<td>ctx_0</td>
<td>0</td>
</tr>
<tr>
<td>ctx_1</td>
<td>0</td>
</tr>
<tr>
<td>ctx_2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>machines</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a2a_00</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>a2a_01</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>a2a_02</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>a2a_03</strong></td>
<td><strong>34</strong></td>
<td><strong>9</strong></td>
</tr>
<tr>
<td>a2a_04</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>a2a_05</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>a2a_06</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td><strong>a2a_07</strong></td>
<td><strong>19</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td>a2a_07.5</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>a2a_08</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td><strong>a2a_09</strong></td>
<td><strong>16</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td>a2a_10</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td><strong>a2a_11</strong></td>
<td><strong>9</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td>a2a_12</td>
<td>38</td>
<td>2</td>
</tr>
<tr>
<td>a2a_13</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a2a_14</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>a2a_15</strong></td>
<td><strong>19</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td>a2a_16</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

|              |        |        |
| total        | 281    | 35     |

<table>
<thead>
<tr>
<th>savings</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>97</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>(34.5%)</td>
<td>(34.3%)</td>
</tr>
</tbody>
</table>

Table 5.1: Number of proofs in each refinement level
Chapter 6

Patterns Description

This chapter lists some patterns that were developed and used during the evaluation phase of the concept of using patterns in Event-B. They are presented in a form that could be useful for a developer interested in using patterns. The presentation style is a mix of API\(^1\) and pattern description.

**Description**

Short description of what the pattern is about.

**Illustration**

Drawing or diagram of the main functionality of the pattern.

**Matching points**

- **variables:**
  
  List of the variables in the pattern specification that have to be matched.

- **events:**
  
  List of the events in the pattern specification that have to be matched.

**Notes for the developer**

Additional notes for the developer concerning the implementation of the pattern.

**Used patterns**

List of patterns that have been used for the development of the described pattern.

\(^1\)API = application programming interface
6.1 Single-Message Communication

Description

The single-message communication pattern realises the transfer of a single message over a channel. The message itself is only a Boolean. Data can be attached to the message in a refinement, as it is not part of the communication behavior.

Illustration

Matching points

variables:

\[ transferred \ (BOOL) \]

events:

\[
\begin{align*}
\text{init} \\
\begin{align*}
\text{begin} \quad transferred := \text{FALSE} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{transfers} \\
\text{when} \quad transferred = \text{FALSE} \\
\text{then} \quad transferred := \text{TRUE} \\
\text{end}
\end{align*}
\]

Notes for the developer

\[ transferred \] is refined by \[ sent \].
A variant of this pattern would be, \[ transferred \] being refined by \[ received \].
6.2 Request/Confirm

Description

The request/confirm pattern realises a communication protocol between two parties A and B. First, A transfers a single request message to B. Afterwards B transfers a single confirmation message back to A and the protocol ends.

Illustration

Matching points

variables:

\[
\begin{align*}
\text{requested} (\text{BOOL}) \\
\text{confirmed} (\text{BOOL})
\end{align*}
\]

events:

```plaintext
init
begin
    requested := FALSE
    confirmed := FALSE
end

requests
when
    requested = FALSE
then
    requested := TRUE
end
```
CHAPTER 6. PATTERNS DESCRIPTION

```plaintext
confirms
  when
    requested = TRUE
    confirmed = FALSE
  then
    confirmed := TRUE
  end
```

Used patterns

2 single-message communication patterns (Section 6.1)
6.3 Request/Confirm/Reject

Description

The request/confirm/reject pattern is very similar to the request/confirm pattern in Section 6.2. The difference is the possibility to return a positive or a negative answer. Additionally to the confirmation, there is a rejection that can be transferred. There are two parties A and B. First, A transfers a single request message to B. Afterwards B transfers either a single confirmation message or a single rejection message back to A and the protocol ends.

Illustration

Matching points

variables:

-_requested (BOOL)
-_confirmed (BOOL)
-Rejected (BOOL)
events:

```
init
begin
  requested := FALSE
  confirmed := FALSE
  rejected := FALSE
end
```

```
requests
when
  requested = FALSE
then
  requested := TRUE
end
```

```
confirms
when
  requested = TRUE
  confirmed = FALSE
  rejected = FALSE
then
  confirmed := TRUE
end
```

```
rejects
when
  requested = TRUE
  confirmed = FALSE
  rejected = FALSE
then
  rejected := TRUE
end
```

Used patterns

3 single-message communication patterns (Section 5.1)
6.4 Multiple-Message Communication

Description

The multiple-message communication pattern realises the transfer of more than one message over the same channel. The transfer is asynchronous as the messages can be received in any order. The messages are represented by natural numbers, whereas the greater the number the newer the message. Data can be attached to the messages in a refinement.

Illustration

Matching points

variables:

\begin{align*}
\text{transferred} & (\mathbb{P}(\mathbb{N})) \\
\text{last} & (\mathbb{N})
\end{align*}

events:

\begin{align*}
\text{init} & \\
\text{begin} & \\
\text{transferred} & := \emptyset \\
\text{last} & := 0 \\
\text{end} & \\
\text{transfers} & \\
\text{any} & n \text{ where} \\
& n > \text{last} \\
\text{then} & \\
\text{transferred} & := \text{transferred} \cup \{n\} \\
\text{last} & := n \\
\text{end} &
\end{align*}
6.5 Multiple-Message Communication with Repetition

Description

The multiple-message communication with repetition pattern is very similar to the multiple-message communication pattern in Section 6.4. The only difference is, that the last (and only the last) message can be resent.

Illustration

Matching points

variables:

\[
\begin{align*}
\text{transferred} & \colon \mathbb{N} \rightarrow \mathbb{N} \\
\text{last} & \colon \mathbb{N}
\end{align*}
\]

events:

\[
\begin{align*}
\text{init} & \\
\text{begin} & \\
\text{transferred} & : = \{0 \mapsto 0\} \\
\text{last} & : = 0 \\
\text{end} & \\
\text{transfers} & \\
\text{any} & \ n \ \text{where} \\
\text{\ \ } & n > \text{last} \\
\text{then} & \\
\text{transferred}(n) & : = 1 \\
\text{last} & : = n \\
\text{end}
\end{align*}
\]
6.5. MULTIPLE-MESSAGE COMMUNICATION WITH REPETITION

```
retransfers
begin
  transferred(last) := transferred(last) + 1
end
```

Notes for the developer

The set `transferred` is not initialised as an empty set. It has to include the mapping $0 \mapsto 0$ for implementation reasons.
6.6 Synchronous Multiple-Message Communication

Description

The synchronous multiple-message communication pattern is, as the name says, the synchronous correspondent of the multiple-message communication pattern in Section 6.4. As only one message can be in the channel at a time this leads to a simpler implementation.

Illustration

Matching points

variables:

\[ \text{transferred}(N) \]

events:

\begin{verbatim}
init
  begin
    transferred := 0
  end

transfer
  begin
    transferred := transferred + 1
  end
\end{verbatim}
6.7 Question/Response

Description

The question/response pattern realises a communication protocol between two parties, questioner and responder. The protocol consists of an unbounded number of rounds. In each round the questioner transfers a question to the responder and in return the responder transfers a response to the questioner. This communication protocol is synchronous.

Illustration

Matching points

variables:

| question (N) |
| response (N) |

events:

```
init
begin
  question := 0
  response := 0
end

question
when
  question = response
then
  question := question + 1
end
```
response
  when
    question \neq response
  then
    response := response + 1
end

Used patterns

2 synchronous multiple-message communication patterns (Section 6.6)
Chapter 7

Tool Support

This chapter describes the requirements for a tool that supports the usage of patterns in Event-B. For developing in Event-B there exists Rodin, an Integrated Development Environment (IDE) that bases on Eclipse.

Additional features for Rodin are implemented with plug-ins, as it is the core idea of Eclipse to extend a platform by plug-ins.

Plug-in development is specially supported within Eclipse, because Eclipse has been developed with plug-ins itself.

Since all sources of Eclipse, as well as of Rodin, are public, there is no problem to build an own plug-in that extends the functionality of Rodin.

A pattern tool therefore can be implemented by developing a plug-in for Rodin. Besides the implementation itself it is important to declare first, what functionality the tool should have.

To figure this out, we should have a look at the tasks to be done when using a pattern and see which of them could be automated. This leads to the requirements of the tool.

7.1 Requirements

The following list contains the different steps needed when applying a pattern.

P Choice of the pattern and the problem at hand

M1 Matching of the variables

M2 Matching of the events

C1 Syntactical check of the guards of each matched event

C2 Check for extra guards
C3 Syntactical check of the actions of each matched event

C4 Check for extra actions

C5 Check that no not-matched event in the development alters a matched variable.

I1 Merging the variables

I2 Copying of the invariants

I3 Extraction of the glueing invariants

I4 Copying of the not-matched events

I5 Copying of the new pattern events

I6 Merging the refined pattern events with the extra guards and extra actions

Having this list of tasks, we can determine which task can be done automatically by a tool and which still have to be done by the developer. If developer interaction is required we can think of how the developer could be supported by dialogues.

There are also some tasks that can be done automatically by a tool, if the models are not too complex. Otherwise, developer interaction will be needed.

7.1.1 Fully Automated Tasks

Most of the incorporation steps are simple copy and paste jobs. Except for I3, all tasks are copying task that follow strict instructions obtained through matching and checking.

These most painful and error prone tasks can be fully automated in the tool.

Checking of the not-matched events (C5) can also be automated. It has only to be checked, if on the left-hand side of the actions a matched variable occurs.

7.1.2 Manual Tasks

Certainly, the choice of the pattern (P) as well as the matching (M1+M2) need full interaction of the developer. These tasks are definitions and cannot be automated.
7.1.3 Partially Automated Tasks

The difficulty of the extraction of the glueing invariant (I3) depends on the complexity of the invariants. If the glueing invariant is a simple equality, as they were in the case studies so far, the tool can automatically extract it. For more complex invariants, the tool may ask for developer interaction.

The checks C1-C4 become more complex the more guards and actions are in the events. It could be hard for a tool to determine which of the guards have to be syntactically the same. We therefore decided to delegate the matching of the guards and actions as well as the detection of the extra guards and extra actions to the developer, in a first version of the tool.

The check of the syntactical sameness can be done by the tool as a checkup for the developer.

In Table 7.1 for every task it is listed again, whether it can be done automatically.

<table>
<thead>
<tr>
<th>task</th>
<th>manually</th>
<th>partially automated</th>
<th>fully automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Overview of all tasks
7.2 Implementation

To support the developer, the tool provides graphical dialogues for all situations where the developer’s interaction is needed. In these dialogues, the developer can choose from possibilities (matching), enter text (renaming) and trigger actions (generating the new refinement).

The reader can get an impression of these dialogues from the following figures. They show the current look of the tool.
Figure 7.1: Starting the pattern wizard

Figure 7.2: Matching within the wizard dialogue
Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis gives a deeper view inside the concept of reusing models in Event-B in the form of design patterns. It was shown that this approach has the ability to save proofs in contrast to a simple composition of two models, where its purpose is to refine sub-models separately. It was also presented, how the concept exactly works and what the prerequisites are. Instead of proving specific models, the correctness of the pattern usage approach itself was proved and led to a methodology that is correct by construction. Furthermore, the concept was demonstrated on an example to give an easy introduction for developers.

8.2 Future Work

Saving more proofs

As already mentioned, the way as we use patterns right now is matching the pattern specification with the problem at hand. Therefore the functionality of the pattern specification has already to be in the development.

A developer might want to use a pattern right from the beginning. Instead of using a pattern because it just fits into the development, the development is designed in such a way that it can make use of the pattern.

In this case the developer would simply copy the pattern specification into the development and then apply the pattern usage approach. This looks like an unnecessary detour.

The idea is to incorporate the pattern directly. The prerequisites of this shortcut have to be evaluated and the correctness has to be proved. Another
motivation for incorporating the whole pattern is to save even more proofs.

**Multiple levels of refinement**

Patterns can have more than one refinement between pattern specification and the pattern refinement that we want to use. This should be no problem as we simply can generate a refinement in our development for each refinement of the pattern. This leads to a correct refinement at the end, as each of this incorporations follows the pattern approach.

What we want to find is a way to directly incorporate the last refinement without the intermediate steps. Probably more checks are needed for this.

**Tool implementation**

The described tool in Chapter 7 is not fully implemented. The graphic user interface as well as the functionality has to be completed to a first version of the tool.

There are ideas how to improve the functionality of the tool, such as using a static file to describe the matching instead of a on-the-fly input into the dialogue.

**A collection of patterns**

At the moment, there are only the few design patterns that were developed for the case studies. It is desirable to have like a library of patterns for common problems. Furthermore, we want to have also patterns in other areas than in business protocols.

**Generalised meta-proof**

The meta-proof is tuned to models with events that do not include parameters. Practical tests led to the assumption, that also parametrised events does not demonstrate a problem to the presented approach. This has of course to be proofed by an adapted meta-proof.
Checking versus proving

The presented approach restricts the usage of patterns as the guards have to be syntactically the same. A developer should have the possibility to choose whether he wants to do the check or instead prove an obligation stating the strength of the guards. In some cases, if the guards are not too complex, the proof would be very easy and could be done automatically anyway.
Bibliography


Appendix A

Sequent Calculus

This appendix gives a short introduction to sequent calculus.

A.1 Sequents

A sequent composes of a finite set of predicates (called hypotheses) and a single goal predicate. A sequent with the hypotheses $H$ and the goal $G$ is written as follows:

$$H \vdash G$$

Informally, a sequent stands for some statement that we want to prove: Under the assumption of the hypotheses $H$, prove the goal $G$.

A.2 Proof rules

A proof rule is a device for constructing proofs of sequents. A proof rule contains a finite set of antecedents and a consequent. Each antecedent is a sequent. Similarly the consequent is also a sequent. Moreover, we give a name to each proof rule. The proof rule with name $r$, the set of antecedents $A_1, \ldots, A_n$, and the consequent $C$ is presented as follows.

$$
\begin{array}{c}
A_1 \quad \ldots \quad A_n \\
\hline
C
\end{array}
$$

\footnote{We will not going into details about defining predicates. More information can be found in [Abr09]}
APPENDIX A. SEQUENT CALCULUS

The interpretation of the above proof rule is as follows: the proofs of the antecedents $A_1, \ldots, A_n$ together gives us the proof of the consequent $C$.

However, the proof rules are usually applied in the back-ward fashion: in order to prove the consequent $C$, we will prove the antecedents $A_1, \ldots, A_n$.

A.3 Some Proof Rules

This section presents some proof rules that will be used for the proofs in Appendix B. First, here is the used taxonomy.

- $H$ is the set of predicates.
- $G$ is a predicate.
- $P$ is a predicate.
- $Q$ is a predicate.
- $[x := E]P$: Replace all the free occurrences of $x$ in $P$ by $E$.
- $[x := E]H$: Replace all the free occurrences of $x$ in $H$ by $E$.

The set of proof rules is as follows.

- Monotonicity

\[
\begin{array}{c}
H \vdash G \\
H, P \vdash G
\end{array}
\]

MON

- Hypothesis

\[
\begin{array}{c}
G \vdash G
\end{array}
\]

HYP

- Conjunction on the left (in the hypothesis).

\[
\begin{array}{c}
H, P, Q \vdash G \\
H, P \land Q \vdash G
\end{array}
\]

AND_L
A.3. SOME PROOF RULES

(Note that this rule can be applied in both direction).

- Conjunction on the right (in the goal).

\[
\begin{array}{c}
H \vdash P \\
\hline
H \vdash Q \\
\hline
H \vdash P \land Q
\end{array}
\]  

\text{AND}_R

- Implication on the left (in the hypothesis).

\[
\begin{array}{c}
H, P, Q \vdash G \\
\hline
H, P, P \Rightarrow Q \vdash G
\end{array}
\]  

\text{IMP}_L

- Implication on the right (in the goal).

\[
\begin{array}{c}
H, P \vdash Q \\
\hline
H \vdash P \Rightarrow Q
\end{array}
\]  

\text{IMP}_R

- Universal quantifier on the left (in the hypothesis). Instantiate \( x \) with \( E \).

\[
\begin{array}{c}
H, \forall x \cdot P, [x := E]P \vdash G \\
\hline
H, \forall x \cdot P \vdash G
\end{array}
\]  

\text{ALL}_L

- Existential quantifier on the left (in the hypothesis). Here there is a side condition that \( x \) is a fresh variable.

\[
\begin{array}{c}
H, P \vdash G \\
\hline
H, \exists x \cdot P \vdash G
\end{array}
\]  

\text{XST}_L

- Existential on the right (in the goal). Instantiate \( x \) with \( E \).
• Equality (from left to right).

\[
\begin{array}{c}
\frac{\text{H} \vdash [x := E]P}{\text{H} \vdash \exists x \cdot P} \quad \text{XST}_R
\end{array}
\]

\[
\begin{array}{c}
\frac{[E := F]\text{H} \vdash [E := F]\text{G}}{\text{H}, E = F \vdash \text{G}} \quad \text{EQ}_{LR}
\end{array}
\]

• Logic

\[
\begin{array}{c}
\frac{\text{H}' \vdash \text{G}'}{\text{H} \vdash \text{G}} \quad \text{LOGIC}
\end{array}
\]

Where \( H' \) is equivalent (pair-wise) to \( H \) and \( G' \) is equivalent to \( G \) using some logical equivalent rules. Examples of such rules are:

\[
\begin{align*}
E = E & \iff \top \\
P \land \top & \iff P
\end{align*}
\]
Appendix B

Proofs

B.1 Ingredients

obligations from the pattern specification

\[ \forall v, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \] (1)

obligations from the development

\[ I(b, c) \land H_i(b) \land N_i(b, c) \Rightarrow (\exists b', c'. R_i(b, b') \land S_i(b, c, c')) \] (2)

obligations from the pattern refinement

\[ J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \] (3)

\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow (\exists v'. T_i(v, v') \land K(v', w')) \] (4)

\[ J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w'. U_i(w, w') \] (5)
Since we do not want to carry around all the above as hypotheses in our proof, we introduce a special proof rule called AXIOM to bring in any of these above as hypotheses in the proof.

### B.2 Matching

\[
\begin{align*}
  b & \doteq v \\
  X(w) & \doteq v \\
  H_i & \doteq L_i \\
  R_{\text{init}} & \doteq T_{\text{init}} \\
  R_i & \doteq T_i
\end{align*}
\]

The SUBST rule substitutes all occurrences of the left parts of the above matching equations with the corresponding right side.
B.3  GRD_REF

\[
\begin{align*}
I(b,c) \\
K(b,w) \\
J(b) \\
c = d \\
M_i(w) \\
N_i(X(w),d) \\
\vdash H_i(b) \land N_i(b,c)
\end{align*}
\]

\[
\begin{align*}
I(v,c) \\
K(v,w) \\
J(v) \\
c = d \\
M_i(w) \\
N_i(v,d) \\
\vdash L_i(v) \land N_i(v,c)
\end{align*}
\]

\[
\begin{align*}
I(v,d) \\
K(v,w) \\
J(v) \\
M_i(w) \\
N_i(v,d) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

AXIOM (3)

\[
\begin{align*}
I(v,d) \\
K(v,w) \\
J(v) \\
M_i(w) \\
N_i(v,d) \\
\vdash L_i(v) \land K(v,w) \land M_i(w) \Rightarrow L_i(v) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
I(v,d) \\
N_i(v,d) \\
J(v) \land K(v,w) \land M_i(w) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
I(v,d) \\
N_i(v,d) \\
J(v) \land K(v,w) \land M_i(w) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
I(v,d) \\
N_i(v,d) \\
L_i(v) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
I(v,d) \\
N_i(v,d) \\
L_i(v) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
L_i(v) \land N_i(v,d) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]

\[
\begin{align*}
L_i(v) \land N_i(v,d) \\
\vdash L_i(v) \land N_i(v,d)
\end{align*}
\]
B.4 INV_REF

\[
\begin{align*}
&I(b, c) \\
&K(b, w) \\
&J(b) \\
&c = d \\
&M_i(w) \\
&N_i(X(w), d) \\
&U_i(w, w') \\
&S_i(X(w), d, d') \\
\vdash \exists b', c' \cdot R_i(b, b') \land S_i(b, c, c') \land \\
&K(b', w') \land J(b') \land c' = d'
\end{align*}
\]

\[
\begin{align*}
&I(v, c) \\
&K(v, w) \\
&J(v) \\
&c = d \\
&M_i(w) \\
&N_i(v, d) \\
&U_i(w, w') \\
&S_i(v, d, d') \\
\vdash \exists b', c' \cdot T_i(v, b') \land S_i(v, c, c') \land \\
&K(b', w') \land J(b') \land c' = d'
\end{align*}
\]

\[
\begin{align*}
&I(v, c) \\
&K(v, w) \\
&J(v) \\
&c = d \\
&M_i(w) \\
&N_i(v, d) \\
&U_i(w, w') \\
&S_i(v, d, d') \\
\vdash \exists b' \cdot T_i(v, b') \land S_i(v, c, d') \land \\
&K(b', w') \land J(b') \land d' = d'
\end{align*}
\]
\[
\begin{align*}
I(v, c) \\
K(v, w) \\
J(v) \\
c = d \\
M_i(w) \\
N_i(v, d) \\
U_i(w, w') \\
S_i(v, d, d') \\
\vdash \\
\exists b' : T_i(v, b') \land S_i(v, c, d') \land K(b', w') \land J(b') \land \top
\end{align*}
\]

\[
\begin{align*}
I(v, c) \\
K(v, w) \\
J(v) \\
c = d \\
M_i(w) \\
N_i(v, d) \\
U_i(w, w') \\
S_i(v, d, d') \\
\vdash \\
\exists b' : T_i(v, b') \land S_i(v, c, d') \land K(b', w') \land J(b')
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
K(v, w) \\
J(v) \\
M_i(w) \\
N_i(v, d) \\
U_i(w, w') \\
S_i(v, d, d') \\
\vdash \\
\exists b' : T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b')
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
K(v, w) \\
J(v) \\
M_i(w) \\
N_i(v, d) \\
U_i(w, w') \\
S_i(v, d, d') \\
\forall v, v' : J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow \exists v' : T_i(v, v') \land K(v', w') \\
\vdash \\
\exists b' : T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b')
\end{align*}
\]
\[ I(v, d) \]
\[ N_i(v, d) \]
\[ U_i(w, w') \]
\[ S_i(v, d, d') \]
\[ \forall u, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \]
\[ J(v) \land K(v, w) \land M_i(w) \]
\[ J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \]
\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow \exists v'. T_i(v, v') \land K(v', w') \]
\[ \exists b'. T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b') \]

\[ \text{IMP}_\perp \]

\[ I(v, d) \]
\[ N_i(v, d) \]
\[ U_i(w, w') \]
\[ S_i(v, d, d') \]
\[ \forall u, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \]
\[ J(v) \land K(v, w) \land M_i(w) \]
\[ L_i(v) \]
\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow \exists v'. T_i(v, v') \land K(v', w') \]
\[ \exists b'. T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b') \]

\[ \text{AND}_\perp \]

\[ I(v, d) \]
\[ N_i(v, d) \]
\[ S_i(v, d, d') \]
\[ \forall u, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \]
\[ J(v) \land K(v, w) \land M_i(w) \]
\[ L_i(v) \]
\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow \exists v'. T_i(v, v') \land K(v', w') \]
\[ \exists b'. T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b') \]

\[ \text{IMP}_\perp \]

\[ I(v, d) \]
\[ N_i(v, d) \]
\[ S_i(v, d, d') \]
\[ \forall u, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \]
\[ L_i(v) \]
\[ J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \Rightarrow \exists v'. T_i(v, v') \land K(v', w') \]
\[ \exists b'. T_i(v, b') \land S_i(v, d, d') \land K(b', w') \land J(b') \]

\[ \text{XST}_\perp \]
\[ v' \rightsquigarrow v' \]
\[
\begin{align*}
I(v, d) \\
N_i(v, d) \\
S_i(v, d, d') \\
\forall v, v'. J(v) \land L_i(v) \land T_i(v, v') \Rightarrow J(v') \\
L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \land U_i(w, w') \\
T_i(v, v') \land K(v', w') \\
\vdash \\
\exists v'. T_i(v, v') \land S_i(v, d, d') \land K(v', w') \land J(v')
\end{align*}
\]
APPENDIX B. PROOFS

I(v, d)
N_i(v, d)
S_i(v, d, d')
J(v) ∧ L_i(v) ∧ T_i(v, v')
J(v')
K(v, w) ∧ M_i(w) ∧ U_i(w, w')
K(v', w')

\[ ⊢ \exists b'. T_i(v, b') ∧ S_i(v, d, d') ∧ K(b', w') ∧ J(b') \]

I(v, d)
N_i(v, d)
S_i(v, d, d')
J(v) ∧ L_i(v) ∧ T_i(v, v')
J(v')
K(v, w) ∧ M_i(w) ∧ U_i(w, w')
K(v', w')

\[ ⊢ \exists b'. T_i(v, b') ∧ S_i(v, d, d') ∧ K(b', w') ∧ J(b') \]

I(v, d)
N_i(v, d)
S_i(v, d, d')
J(v) ∧ L_i(v) ∧ T_i(v, v')
J(v')
K(v, w) ∧ M_i(w) ∧ U_i(w, w')
K(v', w')

\[ ⊢ T_i(v, v') ∧ S_i(v, d, d') ∧ K(v', w') ∧ J(v') \]

S_i(v, d, d')
T_i(v, v')
J(v')
K(v', w')

\[ ⊢ T_i(v, v') ∧ S_i(v, d, d') ∧ K(v', w') ∧ J(v') \]
\[ T_i(v, v') \land S_i(v, d, d') \land K(v', w') \land J(v') \]
\[ \vdash T_i(v, v') \land S_i(v, d, d') \land K(v', w') \land J(v') \]

\( \text{HYP} \)
### B.5 FIS_REF

\[
\begin{align*}
I(b, c) \\
K(b, w) \\
J(b) \\
c = d \\
M_i(w) \\
N_i(X(w), d) \\
\vdash \exists w', d' \cdot U_i(w, w') \land S_i(X(w), d, d')
\end{align*}
\]

**Axiom**: (2),(3),(5)

\[
\begin{align*}
I(b, c) \\
K(b, w) \\
J(b) \\
c = d \\
M_i(w) \\
N_i(X(w), d) \\
I(b, c) \land H_i(b) \land N_i(b, c) \Rightarrow \exists b', c' \cdot R_i(b, b') \land S_i(b, c, c') \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w' \cdot U_i(w, w') \\
\vdash \exists w', d' \cdot U_i(w, w') \land S_i(X(w), d, d')
\end{align*}
\]

**Substitution**

\[
\begin{align*}
I(v, c) \\
K(v, w) \\
J(v) \\
c = d \\
M_i(w) \\
N_i(v, d) \\
I(v, c) \land L_i(v) \land N_i(v, c) \Rightarrow \exists b', c' \cdot T_i(v, b') \land S_i(v, c, c') \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w' \cdot U_i(w, w') \\
\vdash \exists w', d' \cdot U_i(w, w') \land S_i(v, d, d')
\end{align*}
\]

**Equivalence**
\[
\begin{align*}
I(v, d) \\
K(v, w) \\
J(v) \\
c = d \\
M_i(w) \\
N_i(v, d) \\
I(v, d) \land L_i(v) \land N_i(v, d) \Rightarrow \exists b', c'. T_i(v, b') \land S_i(v, d, c') \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w'. U_i(w, w') \\
\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d') \\
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
c = d \\
N_i(v, d) \\
I(v, d) \land L_i(v) \land N_i(v, d) \Rightarrow \exists b', c'. T_i(v, b') \land S_i(v, d, c') \\
J(v) \land K(v, w) \land M_i(w) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w'. U_i(w, w') \\
\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d') \\
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
c = d \\
N_i(v, d) \\
I(v, d) \land L_i(v) \land N_i(v, d) \Rightarrow \exists b', c'. T_i(v, b') \land S_i(v, d, c') \\
J(v) \land K(v, w) \land M_i(w) \\
L_i(v) \\
J(v) \land K(v, w) \land M_i(w) \Rightarrow \exists w'. U_i(w, w') \\
\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d') \\
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
c = d \\
N_i(v, d) \\
I(v, d) \land L_i(v) \land N_i(v, d) \Rightarrow \exists b', c'. T_i(v, b') \land S_i(v, d, c') \\
J(v) \land K(v, w) \land M_i(w) \\
L_i(v) \\
\exists w'. U_i(w, w') \\
\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d') \\
\end{align*}
\]

\[
\begin{align*}
I(v, d) \\
c = d \\
N_i(v, d) \\
I(v, d) \land L_i(v) \land N_i(v, d) \Rightarrow \exists b', c'. T_i(v, b') \land S_i(v, d, c') \\
J(v) \land K(v, w) \land M_i(w) \\
L_i(v) \\
\exists w'. U_i(w, w') \\
\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d') \\
\end{align*}
\]
\[c = d\]
\[I(v, d) \land L_i(v) \land N_i(v, d)\]
\[\exists b', c'. T_i(v, b') \land S_i(v, d, c')\]
\[J(v) \land K(v, w) \land M_i(w)\]
\[\exists u'. U_i(w, w')\]
\[\vdash \exists u', d'. U_i(w, w') \land S_i(v, d, d')\]

\[c = d\]
\[I(v, d) \land L_i(v) \land N_i(v, d)\]
\[\exists b', c'. T_i(v, b') \land S_i(v, d, c')\]
\[J(v) \land K(v, w) \land M_i(w)\]
\[\exists u'. U_i(w, w')\]
\[\vdash \exists u', d'. U_i(w, w') \land S_i(v, d, d')\]

\[c = d\]
\[I(v, d) \land L_i(v) \land N_i(v, d)\]
\[T_i(v, w') \land S_i(v, d, d')\]
\[J(v) \land K(v, w) \land M_i(w)\]
\[\exists w'. U_i(w, w')\]
\[\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d')\]

\[c = d\]
\[I(v, d) \land L_i(v) \land N_i(v, d)\]
\[T_i(v, v') \land S_i(v, d, d')\]
\[J(v) \land K(v, w) \land M_i(w)\]
\[U_i(w, w')\]
\[\vdash \exists w', d'. U_i(w, w') \land S_i(v, d, d')\]

\[c = d\]
\[I(v, d) \land L_i(v) \land N_i(v, d)\]
\[T_i(v, v')\]
\[S_i(v, d, d')\]
\[J(v) \land K(v, w) \land M_i(w)\]
\[U_i(w, w')\]
\[\vdash U_i(w, w') \land S_i(v, d, d')\]
\[
S_i(v, d, d') \\
U_i(w, w') \\
\vdash
U_i(w, w') \land S_i(v, d, d')
\]