**Software and Systems Modeling manuscript No.**
(will be inserted by the editor)

The original publication is available at http://dx.doi.org/10.1007/s10270-010-0183-7

Software & Systems Modeling
May 2013, Volume 12, Issue 2, pp 229-244

# Event-B Patterns and Their Tool Support [*]

**Thai Son Hoang[1], Andreas Fürst[1], Jean-Raymond Abrial[2]**

[1] Swiss Federal Institute of Technology, Zurich (ETH-Zurich)

[2] Marseille, France

Received: date / Revised version: date

**Abstract** Event-B has given developers the opportunity to construct models of complex systems that are correct by construction. However, there is no systematic approach, especially in terms of reuse, which could help with the construction of these models. We introduce the notion of *design patterns* within the framework of Event-B to shorten this gap. Our approach preserves the correctness of the models, which is critical in formal methods and also reduces the proving effort. Within our approach, an Event-B design pattern is just another model devoted to the formalisation of a typical sub-problem. As a result, we can use patterns to construct a model which can subsequently be used as a pattern to construct a larger model. We also present the interaction between developers and the tool support within the associated RODIN Platform of Event-B. The approach has been applied successfully to some medium-size industrial case studies.

**Keywords:** Event-B, formal methods, design patterns, formal modelling, model reuse.

## 1 Introduction

The purpose of our investigation here is to study the possibility of reusing models in formal modelling. Currently, formal methods are applicable to various domains for constructing models of complex systems. However, often they lack some systematic methodological approaches, in particular in reusing existing models, for helping the development process. The objective in introducing design patterns within formal methods in general, and in Event-B in particular, is to overcome this limitation.

The idea of design patterns in software engineering is to have a general and reusable solution to commonly occurring problems. In general, a design pattern is not necessarily a finished product, but rather a template on how to solve a problem which can be used in many different situations. Design patterns are further populated in object-oriented programming

[*] This is an extension of an earlier report [16]. Part of this work is supported by the DEPLOY project (`http://www.deploy-project.eu`

[14]. The idea is to have some predefined solutions, and incorporate them into the development with some modification and/or instantiation. We want to bring this idea into formal methods and in particular to Event-B. Moreover, the typical elements that we want to reuse are not only the models themselves, but also (more importantly) their correctness in terms of proofs associated with the models. In our earlier investigations [5, 11, 16], [10, Section 5.4.1], we have already worked on several examples to understand the usefulness and applicability of the approach. We summarise this work and its formalisation in this paper.

Our contribution here is the methodology for reusing existing models in Event-B. Our approach allows developers to reuse any existing models (which we call "design patterns") in a way that preserves the correctness of models, hence we can save effort on not only modelling but also on proving these models correct.

The examples that we used in this paper are models for communication protocols [23]. Note that, however, the approach is general and its applicability *is not limited* to this domain.

The structure of the paper is as follows. Section 2 gives a short introduction to Event-B. Section 3 presents a case study to illustrate the motivation for our approach. Section 4 gives an overview of the formalisation of the approach in Event-B. The list of patterns which are used in our industrial case studies is presented in Section 5. Section 6 describes our prototype tool supporting the approach. Finally, in Section 7 we review related work and point out future directions.

## 2 The Event-B Modelling Method

Event-B [2] represents a further evolution of the B-method [1], which has been simplified and is now centered around the general notion of *events*, also found in Action Systems [6] and TLA [17].

An Event-B [2] model is a collection of modelling elements that are stored in a repository. When presenting our models, we will do so in a pretty-print form, e.g. adding keywords and following a certain layout conventions to aid parsing. We proceed like this to improve legibility and help the reader to remember 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* and *axioms*, where carrier sets are similar to types [4]. 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* [1].

[1] Machine can also contain a *variant* for proving convergence properties, but it is not of our interests in this paper.

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* $S(v)$[2]. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the following form

$$\mathsf{evt} \ \widehat{=} \ \textbf{when}\ G(v)\ \textbf{then}\ S(v)\ \textbf{end} \qquad (1)$$

The short form

$$\mathsf{evt} \ \widehat{=} \ \textbf{begin}\ S(v)\ \textbf{end} \qquad (2)$$

is used if the guard always holds. A dedicated event of the form (2) is used for *initialisation*.

The action of an event is composed of several *assignments* of the form

$$x \ := \ E(v) \qquad (3)$$

$$x \ :\in \ E(v) \qquad (4)$$

$$x \ :\mid \ Q(v, x') \quad , \qquad (5)$$

where $x$ are some variables, $E(v)$ expressions, and $Q(v, x')$ a predicate. Assignment form (3) is *deterministic*, the other two forms are *non-deterministic*. Form (4) assigns $x$ to an element of a set, and form (5) assigns to $x$ a value $x'$ satisfying a predicate. The effect of each assignment can also be described by a before-after predicate $BAP$:

$$BAP\big(x \ := \ E(v)\big) \quad \widehat{=} \quad x' \ = \ E(v) \qquad (6)$$

$$BAP\big(x \ :\in \ E(v)\big) \quad \widehat{=} \quad x' \ \in \ E(v) \qquad (7)$$

$$BAP\big(x \ :\mid \ Q(v, x')\big) \quad \widehat{=} \quad Q(v, x') \quad . \qquad (8)$$

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:

$$BAP\big(S(v)\big) \quad \widehat{=} \quad A(v, x') \ \wedge \ y' = y \quad . \qquad (9)$$

Later, in proof obligations, we represent the before-after predicate $BAP\big(S(v)\big)$ of an action $S(v)$ directly by the predicate

$$\boldsymbol{S}(v, v') \quad .$$

*Proof obligations* serve to verify certain properties of a machine. Here a proof obligation is presented in the form of a sequent: "hypotheses" $\vdash$ "goal". The intuitive meaning of this sequent is that under the assumption of the *hypotheses*, the *goal* holds.

For each event of a machine, the following proof obligation which guarantees *feasibility* must be proved.

| | |
|---|---|
| $I(v)$<br>$G(v)$<br>$\vdash$<br>$\exists v' \cdot \boldsymbol{S}(v, v')$ | **FIS** |

[2] For simplicity, we do not treat events with *parameters*.

By proving feasibility, we achieve that $\boldsymbol{S}(v, v')$ 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*:

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & G(v) & \\ & \boldsymbol{S}(v,v') & \textbf{INV} \\ \vdash & & \\ & I(v') & \end{array}}$$

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 mean to introduce more details about the dynamic properties of a model [4]. For more on the well-known theory of refinement, we refer to the Action System formalism [6] that has inspired the development of Event-B. We present some important proof obligations for machine refinement.

A machine $CM$ can refine at most one other machine $AM$. We call $AM$ the *abstract* machine and $CM$ the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing 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:

$$\mathsf{ea} \mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } S(v) \textbf{ end}$$

$$\mathsf{ec} \mathrel{\widehat{=}} \textbf{when } H(w) \textbf{ then } T(w) \textbf{ end}$$

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.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{FIS\_REF} \\ \vdash & & \\ & \exists w' \cdot \boldsymbol{T}(w,w') & \end{array}}$$

2. The guard of $ec$ is *stronger* than the guard of $ea$. This is formalised by the following proof obligation.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{GRD} \\ \vdash & & \\ & G(v) & \end{array}}$$

3. The abstract event can always “simulate” the concrete event and preserve the gluing (concrete) invariant. This is formalised by the following proof obligation.

$$\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{SIM} \\ & \boldsymbol{T}(w,w') & \\ \vdash & & \\ & \exists v' \cdot \boldsymbol{S}(v,v') \land J(v',w') & \end{array}$$

For the initialisation, the corresponding proof obligations are analogue. The proofs of these above obligations ensure the correctness of the refinement model with respect to the abstract model and the gluing invariant between them.

In the course of refinement, often *new events* $ec$ are introduced into a model. New events must be proved to refine the implicit abstract event skip that does nothing.

$$\mathsf{skip} \;\widehat{=}\; \textbf{begin}\ \textsc{skip}\ \textbf{end}$$

Moreover, it may be proved that new events do not collectively diverge, but this is not relevant here. The new events allow us to observe the system with a finer time grain. This is an analogue of the stuttering principle in TLA [17]: a step that leaves the abstract variables unchanged.

## 3 Question/Response Protocol

In this section, we look at the development of a protocol, namely *Question/Response* in order to understand what we mean by design patterns and how to apply them in system development. Section 3.1 first gives an informal description of the protocol together with its formal specification in Event-B, then identifies *similar fragments* of the formal model that leads to the idea of using patterns. In Section 3.2 we formally present a pattern, namely *synchronous multiple message communication*, including its specification and refinement. Finally, we illustrate how the pattern is reused (twice) in our development of the actual Question/Response protocol in Section 3.3.

### *3.1 Description and Formal Specification*

There are two parties participating in this protocol namely the *Questioner* and the *Responder*. The protocol consists of an unbounded number of *rounds*. For each round, there are two steps as follows.

1. The *Questioner* sends a *question* to the *Responder*.
2. After receiving this *question*, the *Responder* sends a *response* back to the *Questioner*.

Formally, we can use two variables to represent the state of the protocol: $quest$ to denote the number of questions that have been asked, and $resp$ to indicate the number of responses that have been given. The first invariant **QuestResp_0_1** specifies that the number of responses is a natural number and the second invariant, i.e. **QuestResp_0_2** specifies that the communication is synchronous: either the number of questions is the same as the number of responses or it is greater than the number of responses by 1 – in the case where a response is expected before another question can be created.

**variables:** $quest, resp$

**invariants:**

**QuestResp_0_1:** $resp \in \mathbb{N}$

**QuestResp_0_2:** $quest = resp \ \lor\ quest = resp + 1$

Initially, there are no questions or responses hence both variables are initialised to $0$.

```
init
  begin
    $quest, resp := 0, 0$
  end
```

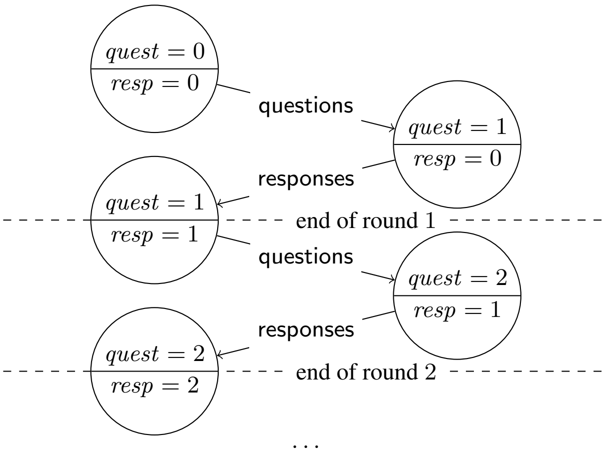


**Fig. 1** Question/Response protocol with two rounds

The dynamic system can be seen in Figure 1. For each round, the "questioning" phase starts when the number of questions and the number of responses are identical and increases the number of questions by $1$. The "responding" phase starts after the "questioning" phase of the same round (when the number of questions and responses are different) and increases the number of responses by $1$. This is formalised by the following two events, namely questions and responds, representing the two phases accordingly.

```
questions
  when
    $quest = resp$
  then
    $quest := quest + 1$
  end
```

```
responds
  when
    $quest \neq resp$
  then
    $resp = resp + 1$
  end
```

The specification of the above two events are very similar, except for their guards. The two events both correspond to transferring some information from one side to another and can be *repeated*, however, the communication is *synchronous*: a new message can be sent only when the last message has been received. We call this kind of communication *synchronous multiple message communication*. Hence if we have a development for this type of communication (to be formalised in the next section), we can instantiate it twice: once for the "questioning" phase and once for the "responding" phase.

### *3.2 Synchronous Multiple Message Communication*

This section presents the development of a communication between two parties $A$ and $B$ for transferring some information repeatedly and synchronously from $A$ to $B$.

The specification of this protocol contains only one natural number variable $trans$, to denote the number of messages that has been transferred.

**variables:** $trans$

**invariants:**

**SynchMultiCom_0_1:** $trans \in \mathbb{N}$

There is only one event in this model to increase the value of variable $trans$ denoting the fact that a message has been transferred from $A$ to $B$.

```
transfers
  begin
    $trans := trans + 1$
  end
```

This synchronous multiple message communication is illustrated in Figure 2.

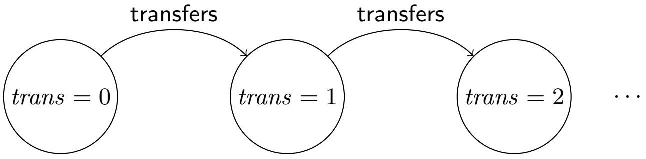


**Fig. 2** Synchronous Multiple Message Communication

However this is only the abstraction of this protocol (it might be even too abstract in the sense that it does not specify how communication happens, e.g. synchronous vs. asynchronous). In reality, the message needs to be sent via some channel between the two parties. This is illustrated in Figure 3. Here the diagram is about different parties (not states) and messages sent between them.

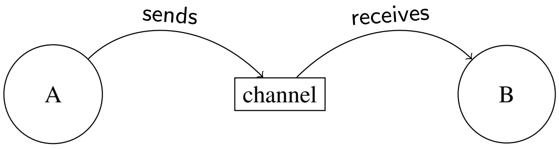


**Fig. 3** Communication via a channel

We use three variables to represent the state of the refinement.

- $snds$: the number of messages having been sent by $A$.
- $rcvs$: the number of messages having been received by $B$.
- $chan$: since there is at most one message on the channel, we use a Boolean value to denote the existence of a message on the channel.

At this point, we have a decision to make about refinement of the abstract event transfers. It could be refined by the event corresponding to “sends” or it could be refined by the event corresponding to “receives”. We presented here the refinement of event transfers when sending, *but the other alternative is also possible*. As a result of this choice, we have the following gluing invariant.

**invariants:**

**SynchMultiCom_1_1:** $trans = snds$

We also have additional technical invariants about the properties of the protocol. Firstly, if there is no message on the channel, the number of sent and received messages are the same. Secondly, if there is a message on the channel, then the number of sent messages is greater than the number of received messages by exactly 1. These two invariants correspond to the “synchronous” communication behaviour. Finally, the number of received messages must be a natural number.

**invariants:**

**SynchMultiCom_1_2:** $chan = F \;\Rightarrow\; snds = rcvs$

**SynchMultiCom_1_3:** $chan = T \;\Rightarrow\; snds = rcvs + 1$

**SynchMultiCom_1_4:** $rcvs \in \mathbb{N}$

Initially, there are no messages that have been sent, received or are in the channel.

```
init
  begin
    $snds := 0$
    $rcvs := 0$
    $chan := F$
  end
```

Events sends and receives are straightforward as follows.

```
sends
  refines transfers
  when
    $chan = F$
  then
    $chan := T$
    $snds := snds + 1$
  end
```

```
receives
  when
    $chan = T$
  then
    $chan := F$
    $rcvs := rcvs + 1$
  end
```

Event sends is enabled if there is no message in the channel. The action of the event specifies that $A$ now sent one more message and the message is in the *channel*. Event receives is enabled when there is a message in the *channel*. The action of the event removes the message from the *channel* and indicates that $B$ has received one more message. Note that event receives here is a new event (i.e. it refines skip).

### *3.3 Using the Pattern for the Protocol*

In this section, we see how the pattern developed in Section 3.2 is used for developing the Question/Response protocol of Section 3.1. There are four steps as follows.

1. We need to "match" the specification of the pattern with the problem.
2. We need to "syntactically check" the matching to see if the pattern is applicable.
3. We have to "rename" those variables and events in the pattern refinement that would lead to a name clash (since we can instantiate the same pattern many times). We can also "rename" non-conflicting variables and events if we like to.
4. Lastly, we "incorporate" the renamed refinement of the pattern to create a refinement of the problem.

As mentioned before, we can instantiate the synchronous multiple message communication pattern twice for the Question/Response protocol: once for the "questioning" phase and a second time for the "responding" phase.

*3.3.1 Pattern for "Questioning" Phase* We follow the different steps to incorporate a synchronous multiple message communication pattern for the "questioning" phase as follows.

1. As a first step we need to identify the "matching" between the specification of the pattern and the problem. The matching here is straightforward with variable $trans$ and event transfers of the pattern matched with variable $quest$ and event questions of the problem accordingly.

| **pattern** | $\leadsto$ | **problem** |
|---|---|---|
| $trans$ | $\leadsto$ | $quest$ |
| transfers | $\leadsto$ | questions |

2. The second step is to syntactically check the validity of the pattern. For example, we need to check that given the variable matching $trans \leadsto quest$, the action of event

transfers is “matched” with the action of questions. This should be done automatically by a tool. At the moment, we can assure ourselves that this step is valid. More information about this step can be seen in Section 6.2 when we discuss about tool support.

3. The third step is to rename the variables and events of the pattern refinement according to the following rules.

| **original** | $\leadsto$ | **renamed as** |
|---|---|---|
| $snds$ | $\leadsto$ | $QQuestSnds$ |
| $chan$ | $\leadsto$ | $Q2RQuestChan$ |
| $recv$ | $\leadsto$ | $RQuestRcvs$ |
| sends | $\leadsto$ | Q_sends_question |
| receives | $\leadsto$ | R_receives_question |

4. In the last step, we incorporate the renamed refinement of the pattern to create a refinement of the problem. The result is the following model.

```
variables: $resp$,
           $QQuestSnds$,
           $RQuestRcvs$,
           $Q2RQuestChan$
```

```
invariants:
  QuestResp_1_1: $quest = QQuestSnds$
  QuestResp_1_2: $Q2RQuestChan = F \Rightarrow$
                   $QQuestSnds = RQuestRcvs$
  QuestResp_1_3: $Q2RQuestChan = T \Rightarrow$
                   $QQuestSnds = RQuestRcvs + 1$
  QuestResp_1_4: $RQuestRcvs \in \mathbb{N}$
```

```
init
  begin
    $resp := 0$
    $Q2RQuestChan := F$
    $RQuestRcvs := 0$
    $QQuestSnds := 0$
  end
```

```
Q_sends_question
  refines questions
  when
    $QQuestSnds = resp$
    $Q2RQuestChan = F$
  then
    $Q2RQuestChan := T$
    $QQuestSnds = QQuestSnds + 1$
  end
```

```
R_receives_question
  when
    $Q2RQuestChan = T$
  then
    $Q2RQuestChan := F$
    $RQuestRcvs := RQuestRcvs + 1$
  end
```

```
responds
  refines  responds
  when
    $QQuestSnds \neq resp$
  then
    $resp := resp + 1$
  end
```

There are a number of important aspects of the pattern which we want to draw the readers' attention.

- The matching between event transfers and event questions is not exact.

```
transfers
  begin
    $trans := trans + 1$
  end
```

```
questions
  when
    $quest = resp$
  then
    $quest := quest + 1$
  end
```

Taking into account the matching of the variables, i.e. $trans$ becomes $quest$, only the actions of those events are matched. The guard of event questions does not correspond to any guard of event transfers.

- The additional guard of event questions, i.e. $quest = resp$ is transformed into the guard $QQuestSnds = resp$ of event Q_sends_question in the resulting refinement, because variable $quest$ is matched with variable $trans$ of the pattern and this variable is subsequently refined to $QQuestSnds$, according to the invariant **QuestResp_1_1**.

**QuestResp_1_1:** $quest = QQuestSnds$

- Similarly, the guard of event responds, i.e. $quest \neq resp$, needs to take into account the fact that variable $quest$ now becomes $QQuestSnds$.
- The rewriting of these additional guards is done automatically by the tool support.

*3.3.2 Pattern for "Responding" Phase* We now follow similar steps to use the synchronous multiple message communication pattern for the "responding" phase.

1. The matching is as follows

| **pattern** | ⇝ | **problem** |
|---|---|---|
| $trans$ | ⇝ | $resp$ |
| transfers | ⇝ | responds |

2. Similarly, we assure that the syntax checking for the given matching is successful.
3. We rename the refinement of the pattern according to the following rules.

| **original** | ⇝ | **renamed as** |
|---|---|---|
| $snds$ | ⇝ | $RRespSnds$ |
| $chan$ | ⇝ | $R2QRespChan$ |
| $rcvs$ | ⇝ | $QRespRcvs$ |
| sends | ⇝ | R_sends_response |
| receives | ⇝ | Q_receives_response |

4. We incorporate the renamed pattern refinement with the problem to obtain the following model.

```
variables:  $QQuestSnds,$
            $RQuestRcvs,$
            $Q2RQuestChan,$
            $RRespSnds,$
            $QRespRcvs,$
            $R2QRespChan$
```

```
invariants:
  QuestResp_2_1:  $resp = RRespSnds$
  QuestResp_2_2:  $R2QRespChan = F \Rightarrow$
                   $RRespSnds = QRespRcvs$
  QuestResp_2_3:  $R2QRespChan = T \Rightarrow$
                   $RRespSnds = QRespRcvs + 1$
  QuestResp_2_4:  $QRespRcvs \in \mathbb{N}$
```

```
init
  begin
    $Q2RQuestChan := F$
    $RQuestRcvs := 0$
    $QQuestSnds := 0$
    $R2QRespChan := F$
    $QRespRcvs := 0$
    $RRespSnds := 0$
  end
```

```
Q_sends_question
  refines  Q_sends_question
  when
    $QQuestSnds = RRespSnds$
    $Q2RQuestChan = F$
  then
    $Q2RQuestChan := T$
    $QQuestSnds = QQuestSnds + 1$
  end
```

```
R_receives_question
  refines  R_receives_question
  when
    $Q2RQuestChan = T$
  then
    $Q2RQuestChan := F$
    $RQuestRcvs := RQuestRcvs + 1$
  end
```

```
R_sends_response
  refines  responds
  when
    $QQuestSnds \neq RRespSnds$
    $R2QRespChan = F$
  then
    $R2QRespChan := T$
    $RRespSnds := RRespSnds + 1$
  end
```

```
Q_receives_response
  when
    $R2QRespChan = T$
  then
    $R2QRespChan := F$
    $QRespRcvs := QRespRcvs + 1$
  end
```

Again, we highlight some important aspects of our pattern application at this step.

- Similar to the previous pattern application in Section 3.3.1, the matching between event transfers and event responds are not exact: there is an additional guard in event responds.
- This guard of event responds, i.e. $QQuestSnds \neq resp$ needs to take into account the fact that variable $resp$ is matched with variable $trans$ of the pattern specification and this variable is later refined to $RRespSnds$. This guard is transformed into the guard $QQuestSnds \neq RRespSnds$ of the resulting event R_sends_response. Similarly for the guard of Q_sends_question.
- These guards are in fact "cheats" in the model. Event Q_sends_question supposes to be an event of the *Questioner*, however its guard refers to variable $RRespSnds$ of the *Responder*. The same analysis applies for event R_sends_response and variable $QQuestSnds$. This problem will be handled by a standard refinement step in the next section.

*3.3.3 Removing the "Cheating" Guards* The problem that we mentioned earlier about the "cheating" guards is better known as *local enforceability* [9]. Roughly speaking, on the abstraction level, the global interactions between partners are specified in a way that it might not be enforced during real local implementation without having more additional interactions between the different partners. In our case, it is not possible for the *Questioner* to have access to the information belonging to the *Responder*: currently event Q_sends_question has access to variable $RRespSnds$ of the *Responder*. In this section, we fix this problem by adding more information on how the two partners interact with each other.

The cheating guards, i.e.

$$QQuestSnds \neq RRespSnds$$

for event R_sends_response can be replaced by the following guard which uses only variables of the *Responder*:

$$RQuestRcvs \neq RRespSnds\ .$$

The proof for the guard strengthening obligation (**GRD**) is based on the following invariant **QuestResp_3_1** (which we need to add to the model).

```
invariants:
  QuestResp_3_1:  $RQuestRcvs \geq RRespSnds$
```

The reasoning is as follows:

- From the new guard $RQuestRcvs \neq RRespSnds$ and the new invariant $RQuestRcvs \geq RRespSnds$, we have
$$RQuestRcvs > RRespSnds\ . \qquad (10)$$
- We conclude from the existing invariants **QuestResp_1_2** and **QuestResp_1_3** that
$$QQuestSnds \geq RQuestRcvs\ . \qquad (11)$$

- From (10) and (11), we conclude that $QQuestSnds > RRespSnds$, which ensures $QQuestSnds \neq RRespSnds$, as required.

This step is a standard refinement in Event-B. Intuitively, the new invariant *links* the *questioning* and *responding* phases together and is the core of the Question/Response protocol.

Similarly, the guard $QQuestSnds = RRespSnds$ of event Q_sends_question is replaced by $QQuestSnds = QRespRcvs$. The refined events Q_sends_question and R_sends_response at their final form are as follows.

```
Q_sends_question
  refines  Q_sends_question
  when
    $QQuestSnds = QRespRcvs$
    $Q2RQuestChan = F$
  then
    $Q2RQuestChan := T$
    $QQuestSnds = QQuestSnds + 1$
  end
```

```
R_sends_response
  refines  R_sends_response
  when
    $RQuestRcvs \neq RRespSnds$
    $R2QRespChan = F$
  then
    $R2QRespChan := T$
    $RRespSnds := RRespSnds + 1$
  end
```

Note that we can consider also the guard referring to the channels, i.e. $R2QRespChan = F$ and $Q2RQuestChan = F$ as not locally enforceable, hence should be removed. However, this is not of our interest here.

Overall, this (standard) refinement step where we impose the policy for local enforceability cannot be done automatically by a tool: this corresponds to how the protocol is constructed and is usually protocol dependent.

## 4 Pattern Incorporation in Event-B

In this section, we summarise the idea of incorporating patterns into Event-B developments. The process can be seen in Figure 4.

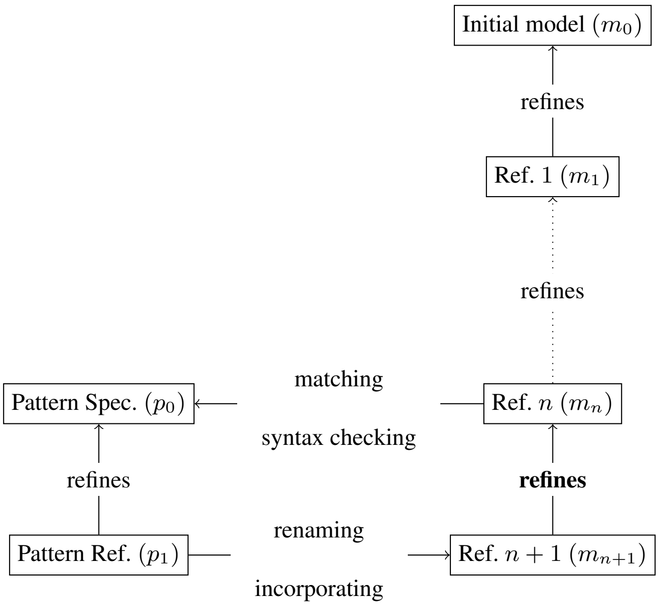


**Fig. 4** Using patterns in Event-B

First of all, in our notion, a pattern is just a development in Event-B including specification $p_0$ and a refinement $p_1$[3]. During a normal development in Event-B, at refinement $m_n$,

[3] In general, this can be extended to multiple refinement level.

developers can match part of the model with the pattern specification $p_0$. As a result of this matching, the refinement $p_1$ can be incorporated to create the refinement $m_{n+1}$ of $m_n$ (with possible "renaming" to avoid name clashes).

Moreover, we have presented here the incorporation of each synchronous multiple message communication pattern separately. However, it is possible that they could be incorporated at the same time. In other words, there can be more than one pattern that can be matched at the same time with the problem at hand. There are side conditions to guarantee that the patterns do not interfere with each other, e.g. there should be no matching to the same variable.

### *4.1 Formalisation of the Approach*

We assume that we have the following patterns containing a specification $p_0$ and its refinement $p_1$. We further assume that the pattern specification $p_0$ has some variables $v$ with invariant $J(v)$. We consider a particular event p with guard $L(v)$ and some actions $v :\mid T(v, v')$.

**variables:** $v$

**invariants:**
$J(v)$

$$\mathsf{p} \;\widehat{=}\; \textbf{when } L(v) \textbf{ then } v :\mid T(v, v') \textbf{ end}$$

In the refinement $p_1$ of $p_0$, variable $v$ is data refined by variable $w$ with gluing invariant separated into $v = X(w)$ and $K(v, w)$. Here we make the assumption that the gluing invariant can be functionally expressed as $v = X(w)$ with some other extra invariants $K(v, w)$. This assumption is valid for all our examples so far and could be relaxed later. Event p is refined by event q with concrete guard $M(w)$ and some actions $w :\mid U(w, w')$.

**variables:** $w$

**invariants:**
$v = X(w)$
$K(v, w)$

$$\mathsf{q} \;\widehat{=}\; \textbf{when } M(w) \textbf{ then } w :\mid U(w, w') \textbf{ end}$$

We assume that we have arrived at a refinement level in a particular development which we call problem specification $m_n$. The machine has some variables $b$ which we intend to match with the above pattern. Moreover, this problem specification could have some other variables $c$ which we have to keep when incorporating the pattern into the development. We do not need to consider the invariant for this machine hence this is left out.

**variables:** $b, c$

```
e
  when
    $H(b)$
    $N(b, c)$
  then
    $b :\mid R(b, b')$
    $c :\mid S(b, c, c')$
  end
```

```
f
  when
    $G(b, c)$
  then
    $c :\mid P(b, c, c')$
  end
```

Without loss of generality, we consider two events of the problem specification: event e which is going to be matched with event p of the pattern specification, and event f which is not going to be matched. Event e is separated into the parts which are matched with event p of the pattern specification, taken into account the decision that variable $b$ is matched with variable $v$ of the pattern specification. Here we say that every variable in the pattern need to be matched with some variable in the problem. However, this condition can be relaxed to make the approach more flexible (see future work in Section 7.3). Hence the guards of the event are separated into $H(b)$ and $N(b, c)$ where $H(b)$ is matched with guard $L(v)$ of event p. Similarly, the action is separated into $b :\mid R(b, b')$ – which is a match of $v :\mid T(v, v')$ – and $c :\mid S(b, c, c')$. The validity of this matching can be syntactically checked and/or even be "discovered" by a tool. For the unmatched event f, we require that it must not change variable $b$, hence its action is of the form $c :\mid P(b, c, c')$. However, it can refer to $b$ in the guard and in the action (only as reference to the before state). The preservation of this restriction will be checked by the supporting tool (more information in Section 6.2). The matching and the extraction from the gluing invariant can be summarised as follows.

| **pattern** | $\leadsto$ | **problem** |
|---|---|---|
| $v$ | $\leadsto$ | $b$ |
| p | $\leadsto$ | e |
| $L(v)$ | $\leadsto$ | $H(b)$ |
| $v :\mid T(v, v')$ | $\leadsto$ | $b :\mid R(b, b')$ |

The refinement $m_{n+1}$ of $m_n$ is generated by combining the problem specification and the pattern refinement as follows.

```
variables:  $w, c$
```

```
invariants:
    $b = X(w)$
    $K(b, w)$
    $J(b)$
```

```
e
  when
    $M(w)$
    $N(X(w), c)$
  then
    $w :\mid U(w, w')$
    $c :\mid S(X(w), c, c')$
  end
```

```
f
  when
    $G(X(w), c)$
  then
    $c :\mid P(X(w), c, c')$
  end
```

We must guarantee that the constructed machine $m_{n+1}$ is indeed a refinement of the specification $m_n$. The detailed proofs are in [11, Section 4.5]. Intuitively, the proofs assume the correctness of the problem specification $m_n$, the pattern specification $p_0$ and the pattern refinement $p_1$ in order to prove the correctness of the problem refinement $m_{n+1}$. The obligation list includes feasibility, guard strengthening and simulation for both events e and f.

As an example, we sketch the proof for guard strengthening obligation of event e which is stated as follows.

```
    $b = X(w)$
    $K(b, w)$
    $J(b)$
    $M(w)$
    $N(X(w), c)$
$\vdash$
    $H(b) \land N(b, c)$
```

The proof of the above sequent can be split into two parts since the goal is a conjunction.

$$\boxed{\begin{array}{l} \quad b = X(w) \\ \quad K(b,w) \\ \quad J(b) \\ \quad M(w) \\ \quad N(X(w),c) \\ \vdash \\ \quad H(b) \end{array}} \quad (12) \quad \boxed{\begin{array}{l} \quad b = X(w) \\ \quad K(b,w) \\ \quad J(b) \\ \quad M(w) \\ \quad N(X(w),c) \\ \vdash \\ \quad N(b,c) \end{array}} \quad (13)$$

The second part of the proof (13) for proving $N(b,c)$ follows from the assumptions $b = X(w)$ and $N(X(w),c)$. The first part (12) of the proof relies on the fact that event q is a refinement of event p in the pattern, hence we have proof the guard strengthening obligation for q, namely.

$$\boxed{\begin{array}{l} \quad J(v) \\ \quad v = X(w) \\ \quad K(v,w) \\ \quad M(w) \\ \vdash \\ \quad L(v) \end{array}}$$

Moreover, from the matching information $v$ is matched with $b$ and guard $H(b)$ is matched with $L(v)$ (i.e. $H$ and $L$ are syntactically the same), we can derive (with renaming variable from $v$ to $b$) the following.

$$\boxed{\begin{array}{l} \quad J(b) \\ \quad b = X(w) \\ \quad K(b,w) \\ \quad M(w) \\ \vdash \\ \quad H(b) \end{array}}$$

and from there we can conclude the proof for (12).

### *4.2 What We Gain with the Pattern Approach*

So far, it seems that we have to do more work in order to apply patterns: we have to develop the pattern separately and incorporate it into the main development. But we do have the following advantages.

- We do not need to prove that $m_{n+1}$ is a refinement of $m_n$. This is because we have already done this proof when developing patterns.
- Moreover, we can reuse the pattern more than once. For example, in the development of the Question/Response protocol, we use the synchronous multiple message communication pattern twice, so we save doing proofs for one pattern.
- Since the pattern is just a normal Event-B development, the meaning of the pattern is also intuitive. Moreover, we can use any development as pattern in our approach.

The proof statistics related to the synchronous multiple message communication and Question/Response protocol is given in Table 1. As we can see, by developing the synchronous multiple message communication pattern separately, we have to prove 15 obligations. However, we do not need to prove the model “Question/Response 1” and “Question/Response 2” (which has a total of 32 obligations) since it is correct by construction. Hence in total we save $32 - 15$, that is 17 proofs. Note that the number of proof obligations for each model “Question/Response 1” and “Question/Response 2” is roughly the same as that of “Synch. Multi. Com. 1”, since in each model we apply the pattern once. The development of the two protocols is available on-line [13].

| Models | Total | Auto. (%) | Man. (%) |
|---|---|---|---|
| Synch. Multi. Com. 0 | 2 | 2 (100%) | 0 (0%) |
| Synch. Multi. Com. 1 | 13 | 12 (92%) | 1 (8%) |
| Question/Response 0 | 6 | 5 (83%) | 1 (17%) |
| Question/Response 1 | 16 | 15 (94%) | 1 (6%) |
| Question/Response 2 | 16 | 15 (94%) | 1 (6%) |
| Question/Response 3 | 5 | 4 (80%) | 1 (20%) |

**Table 1** Proof Statistics

## 5 Patterns Used in Industrial Case Studies

Our approach has been applied to formalise communication protocols from SAP. The examples are *Buyer/Seller B2B* as described in [23] and *Ordering/Supply Chain A2A Communications* as described in [10, Section 5.3.3]. Table 2 shows the proof statistics comparing the developments without patterns and with patterns for the two case studies. More importantly, our approach save on average of the two case studies 33% of the manual proofs (those that need interactive efforts to discharge).

In this section, we give the description of other patterns that have been used in these protocols.

| Models/Savings | Total | Auto. (%) | Man. (%) |
|---|---|---|---|
| A2A (without pattern) | 281 | 249 (89%) | 32 (11%) |
| A2A (with pattern) | 184 | 164 (89%) | 20 (11%) |
| Savings | 97 | 85 (88%) | 12 (12%) |
| Savings percentage | 35% | 34% | 38% |
| B2B (without pattern) | 498 | 427 (86%) | 71 (14%) |
| B2B (with pattern) | 342 | 291 (85%) | 51 (15%) |
| Savings | 156 | 136 (87%) | 20 (13%) |
| Savings percentage | 31% | 32% | 28% |

**Table 2** Case studies' proof statistics (with vs. without pattern)

- Section 5.1 presents the *Single Message Communication* pattern.
- Section 5.2 presents the *Request/Confirm* pattern.
- Section 5.3 presents the *Request/Confirm/Reject* pattern.
- Section 5.4 presents the *Asynchronous Multiple Message Communication* pattern.
- Section 5.5 presents the *Asynchronous Multiple Message Communication with Repetition* pattern.

### *5.1 Single Message Communication*

The description of the pattern is as follows. There are two parties involved in the protocol, namely *Sender* and *Receiver*. There is a message sent from the *Sender* to the *Receiver*. If we denote the status of the protocol by a single variable $trans$, the (abstract) protocol can be seen in Figure 5. In the refinement, the message is transferred via a channel between the *Sender* and the *Receiver*.

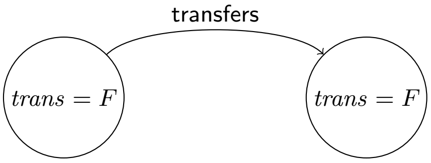


**Fig. 5** Single Message Communication

### *5.2 Request/Confirm Pattern*

The description of the protocol is as follows. There are two parties involved in the protocol, namely *Sender* and *Receiver*. The protocol contains two phases:

1. In the first phase, the *Sender* sends a request to the *Receiver*.
2. In the second phase, upon receiving the request, the *Receiver* sends a confirmation back to the *Sender*.

Using two Boolean variables $req$ and $conf$ to represent the state, the protocol can be illustrated as in Figure 6. The development of this pattern used the single message communication pattern (described in Section 5.1) twice. These two patterns are used as illustrative examples in our earlier report [16].

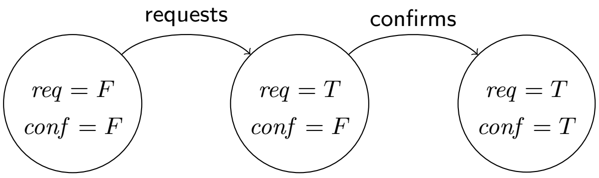


**Fig. 6** Request/Confirm protocol

### *5.3 Request/Confirm/Reject Pattern*

The description of the protocol is as follows. There are two parties involved in the protocol, namely *Sender* and *Receiver*. The protocol also contains two phases:

1. In the first phase, the *Sender* sends a request to the *Receiver*.
2. In the second phase, after receiving this request, the *Receiver* can either send a "confirmation" back to the *Sender* if he agrees; or the *Receiver* sends a "rejection" back to the *Sender* if he does not agree.

Using three Boolean variables $req$, $conf$ and $rej$ to represent the state, the protocol can be seen in Figure 7.

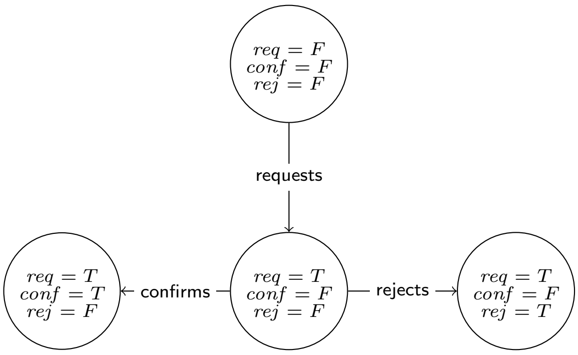


**Fig. 7** Request/Confirm/Reject protocol

The development of this pattern used the single message communication pattern (described in Section 5.1) three times.

### *5.4 Asynchronous Multiple Message Communication Pattern*

The description of the protocol is as follows. There are two parties involved in this protocol, namely *Sender* and *Receiver*.

1. The *Sender* can send many messages (multiple message) to the *Receiver*.
2. The messages are different, in other words, there is no resend.
3. To distinguish the freshness of the message, each message is stamped with a sequence number.
4. The *Receiver* can only receive new messages.
5. The *Receiver* can discard any message.

### *5.5 Asynchronous Multiple Message with Repetition Communication Pattern*

The description of the protocol is as follows. There are two parties involved in this protocol, namely *Sender* and *Receiver*.

1. The *Sender* can send many messages (multiple message) to the *Receiver*.
2. The messages can be the same, in other words, messages could be resent.
3. To distinguish the freshness of the message, each message is stamped with a sequence number.
4. The *Receiver* can receive any message which is not old.
5. The *Receiver* can discard any message.

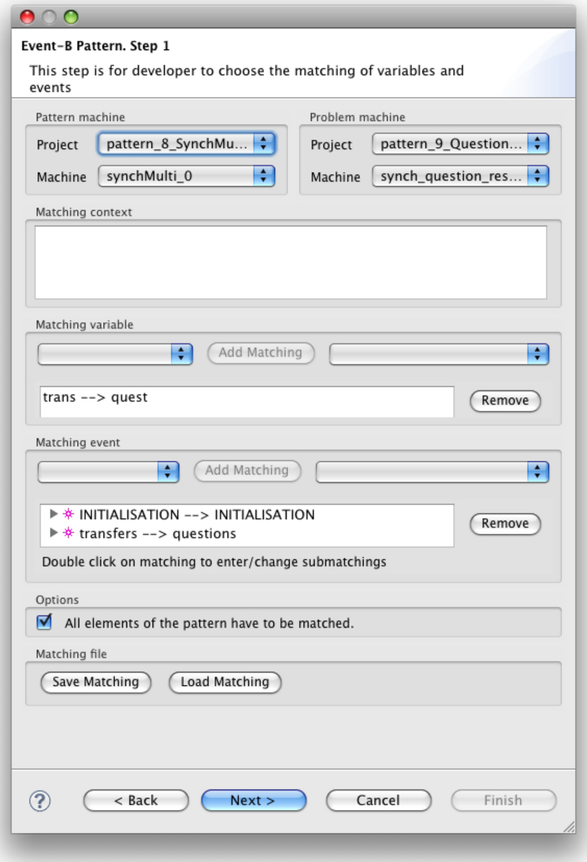
The only difference compared to the asynchronous multiple message communication (no repetition) pattern is that here messages can be resent.

## 6 Tool Support

We have implemented our prototype for supporting our approach as a plug-in for the RODIN Platform [3] which is an open source platform based on Eclipse. The plug-in provides a wizard taking users through different steps of applying patterns, namely, *matching*, *syntax checking*, *renaming* and *incorporating*.

### *6.1 Matching*

The tool assists developers in inputting the matching between the problem and the specification. This includes a dialog for the developers to choose the matching between variables and events. Moreover, in some cases, we need to also match the context information, i.e. carrier sets and constants which can also be chosen through the wizard page (in fact, this "matching context" is better known as generic instantiation in Event-B [4]). Information about this matching can be persistently saved for reuse later. A screen-shot of the wizard page for this step is in Figure 8.



Event-B Pattern. Step 1

This step is for developer to choose the matching of variables and events

Pattern machine
Project: pattern_8_SynchMu...
Machine: synchMulti_0

Problem machine
Project: pattern_9_Question...
Machine: synch_question_res...

Matching context

Matching variable
trans --> quest

Matching event
INITIALISATION --> INITIALISATION
transfers --> questions

Double click on matching to enter/change submatchings

Options
☑ All elements of the pattern have to be matched.

Matching file

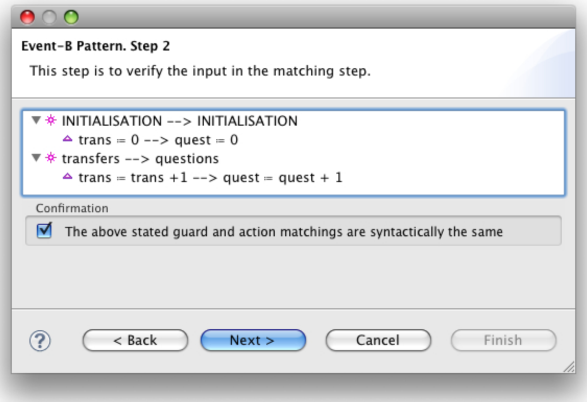
**Fig. 8** First step. Matching

### *6.2 Syntax Checking*

In this step, the tool needs to check the consistency of the matching provided by the user in the previous steps. The consistency checking at this step includes:

- For events matched with some events in the pattern, we need to check the signature of these events against the corresponding pattern events.
- For remaining (unmatched) events, we need to check that they do not modify the matched variables (as mentioned earlier in Section 4.1).

A screen-shot of the relevant wizard page is in Figure 9.



Event-B Pattern. Step 2

This step is to verify the input in the matching step.

```
▼ INITIALISATION --> INITIALISATION
    trans = 0 --> quest = 0
▼ transfers --> questions
    trans = trans +1 --> quest = quest + 1
```

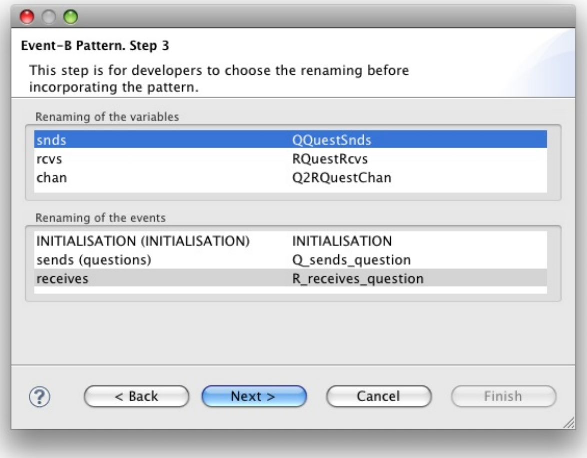
Confirmation

☑ The above stated guard and action matchings are syntactically the same

**Fig. 9** Second step. Syntax Checking

### *6.3 Renaming*

The tool assists developers in inputting renaming patterns. This includes a dialog for the developers to give renaming pattern of variables and events. Consistency (e.g. name clash) for this renaming is verified at this step. A screen-shot of the renaming wizard page is in Figure 10.



Event-B Pattern. Step 3

This step is for developers to choose the renaming before incorporating the pattern.

Renaming of the variables

| | |
|---|---|
| snds | QQuestSnds |
| rcvs | RQuestRcvs |
| chan | Q2RQuestChan |

Renaming of the events

| | |
|---|---|
| INITIALISATION (INITIALISATION) | INITIALISATION |
| sends (questions) | Q_sends_question |
| receives | R_receives_question |

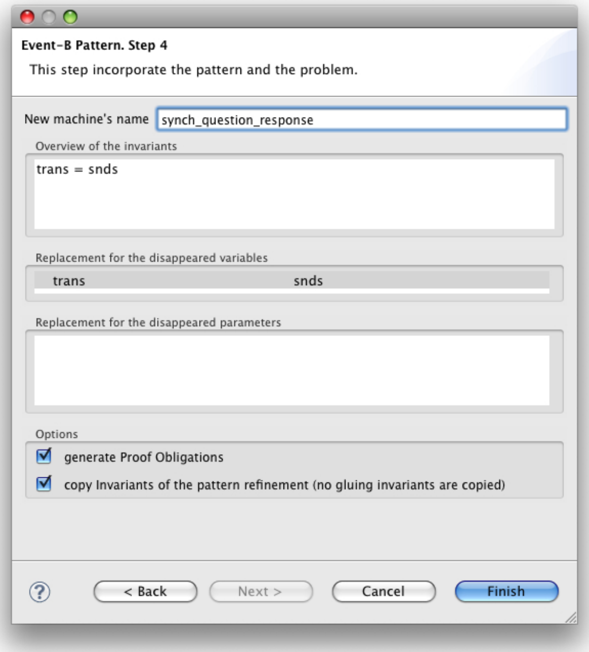
**Fig. 10** Third step. Renaming

### *6.4 Incorporating*

Finally, the tool generates the refinement of the problem according to the input in the previous steps. In order to incorporate the refinement of the pattern into the development, the tool needs to extract information from the gluing invariant on how the abstract variables $v$ in the pattern are refined. Usually, the information is of the form $v = X(w)$. At the moment this information is also entered manually by the user in the wizard. A screen-shot of the wizard page for the incorporating step is in Figure 11.

## 7 Conclusion

We have presented an approach for reusing formal models as patterns in Event-B. During a development, patterns can be discovered by either identifying the part of the model matched by existing patterns, or by recognising similar elements of the model which could be developed separately as a new pattern themselves.



Event-B Pattern. Step 4

This step incorporate the pattern and the problem.

New machine's name: synch_question_response

Overview of the invariants

trans = snds

Replacement for the disappeared variables

| | |
|---|---|
| trans | snds |

Replacement for the disappeared parameters

Options

☑ generate Proof Obligations

☑ copy Invariants of the pattern refinement (no gluing invariants are copied)

**Fig. 11** Fifth step. Incorporating

Even though we presented in Section 4.1 a formalisation of our approach when there is only a single refinement step in the pattern development, the approach is also valid when there are multiple refinement steps. This is the same as applying patterns step by step for each level of refinement. Since refinement is monotonic, the final resulting model will be a refinement of the original model. Practically, only the last refinement model of the pattern's refinement-chain is incorporated in the development. This is already supported by our tool presented in Section 6. This feature allows us to reuse our formal models more flexibly, for example, using the Question/Response protocol in the development of the A2A Communications [10, Section 5.3.3].

### *7.1 Scalability*

We have applied our approach to two medium-size case studies from SAP, namely the Buyer/Seller B2B [23] and Ordering/Supply Chain A2A Communications [10, Section 5.3.3]. However, our approach is general and is not restricted to this specific domain. The efforts on modelling and proving are replaced by specifying how patterns are identified and incorporated into the development. Our experiments show that this process is scalable. In particular, the patterns can be nested, i.e., a pattern can be used to develop another pattern, which then can be reused in a larger development.

So far, our patterns are quite specific since they arose from some domain specific problems that we are trying to solve. More general patterns can be "parameterised" by some carrier sets and constants, which can be "instantiated" upon application to a problem (see our discussion on future work in Section 7.3). This makes the patterns more reusable in distinct problems within different contexts.

Finally, tool support is important for making our approach scalable. Our aim is to have as less interaction from the user as possible by providing different assistances for users when using the tool. Our initial experiments with the implemented tool support is encouraging.

### *7.2 Related Work*

Design patterns are well-known concepts in object-oriented programming, in particular in the work of the Gang-of-Four (GoF) [14]. In their work, each pattern is usually represented by some informal description and some diagram in UML. There is no formal semantics associated with patterns, hence the meanings of these patterns are imprecise. There is some work on formalising these classic software design patterns in different formal notations, e.g., using predicate logic [7], using TLA+ [22], using DisCo [18]. In these papers, the first step is to give some formal meaning to the pattern before the verification of its correctness can take place. This also needs to be done for any newly defined pattern. To overcome this problem, one needs to give some formal semantics to the diagrams used to define patterns. LePUS3 [15] is designed precisely for this purpose. However, verification in LePUS3 emphasises on the consistency between a specification (diagram) and a program. In our opinion, this is quite different from using patterns consistently to design the future system.

Our approach is related to decomposition [8,4] where developers can separate a model into sub-models and can subsequently refine these sub-models independently. The similarity with our approach is when some of the sub-models already exist as some off-the-shelve components (patterns). In this case the advantage of reusing is similar, however decomposition is not intended for reusing.

Another related work to ours is the “automatic refinement tool” [19]. Our patterns are just formal models which encode some design decisions about refining some abstract models. However, the automatic refinement tool still requires proofs in order to make sure that the proposed refinement is correct. This approach does not necessarily preserve correctness.

Comparing with classical B [1], reusing of components is facilitated by the *INCLUDES* clause in the specification level and *IMPORTS* clause at the implementation level to compose different components. In order to reuse the same components several times, classical B supports a renaming mechanism by prefixing the name of the included/imported components with some certain identifier. In our approach, we allow the user to specify the renaming of the pattern, but it could also be done systematically with a prefixing mechanism. The main difference between our approach and the including/importing mechanism is that the including/importing mechanism does not support incorporation refinement, i.e. only reuse of the specification of the pattern is possible.

In Z [21], schemas can be reused conveniently by combining together using operators of the *schema calculus*. Moreover, *instances* of schema can be created by *schema referencing* mechanisms which include both *generic constructions* and *renaming*. Similar to classical B, this technique allows reusing of a single specification component only.

### *7.3 Future Work*

As for future work, we intend to implement the missing features from the current prototype plug-in for the RODIN Platform, e.g. syntax checking and support for extracting information from the pattern refinement. The current documentation for tool support is at the Event-B wiki documentation system [12]. At the same time, we are going to investigate more examples in other domains that could benefit from our approach.

Furthermore, we also need to “instantiate” the context of the pattern development. In our examples so far, the contexts of the pattern and the problem are the same. However, we would like to use the patterns in a more general context. For example, the model of the communication for transferring a certain (abstract) message should be instantiated for any kind of (concrete) message, e.g., if the message is just a Boolean value, or if the message contains some numbers or some complicated data structure. This requires the context of the pattern to be instantiated accordingly. Generic instantiation [4] is a more general concept and could be used in association with other applications, for example with shared-event composition as shown in [20].

As mentioned before, it is not necessarily the case that all the variables of the pattern need to be matched with some variables in the problem. It could be the case that only a part of the variables needs to be matched or even none of them, which corresponds to the case where we do superposition refinement [4]. This makes the approach more flexible.

Moreover, we have specifically chosen to have the “syntax checking” rather than raising proof obligations when applying patterns. In the future, if this turns out to be too restrictive, we can choose to generate the corresponding proof obligations, again for more flexibility. Note that if a pattern matching can be syntactically checked successfully, the proof obligations generated should be trivial to be discharged.

## Acknowledgement

We would like to thank Matthias Schmalz for his comments on early drafts of the paper. We also like to thank anonymous reviewers for their constructive comments to improve the quality of the paper.

## References

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. RODIN: An open toolset for modelling and reasoning in Event-B. *Internation Journal on Software Tools for Technology Transfer (STTT)*, April 2010.
4. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
5. Jean-Raymond Abrial and Thai Son Hoang. Using design patterns in formal methods: An Event-B approach. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.
6. Ralph-Johan Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag.
7. Ian Bayley. Formalising design patterns in predicate logic. In *SEFM*, pages 25–36. IEEE Computer Society, 2007.

8. Michael Butler. *Decompostion Structures for Event-B*, volume 5423 of *Lecture Notes in Computer Science*, chapter Integrated Formal Methods, pages 20–38. Springer, 2009. http://www.springerlink.com/content/3202127567642301/.

9. Gero Decker and Mathias Weske. Local enforceability in interaction petri nets. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2007.

10. DEPLOY Project. Deliverable JD1 – Report on Knowledge Transfer. http://www.deploy-project.eu/pdf/fv-d5-jd1-reportonknowledgetransfer.zip, February 2009.

11. Andreas Fürst. Design patterns in Event-B and their tool support. Master's thesis, Deparment of Computer Science, ETH Zurich, March 2009. http://e-collection.ethbib.ethz.ch/view/eth:41612.

12. Andreas Fürst. Documentation on tool support for Event-B design patterns. http://wiki.event-b.org/index.php/Pattern, April 2010.

13. Andreas Fürst and Thai Son Hoang. Rodin platform archive of question/response protocol. http://deploy-eprints.ecs.soton.ac.uk/230/, June 2010.

14. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995. ISBN-10: 0201633612 ISBN-13: 978-0201633610.

15. Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. Lepus3: An object-oriented design description language. In Gem Stapleton, John Howse, and John Lee, editors, *Diagrams*, volume 5223 of *Lecture Notes in Computer Science*, pages 364–367. Springer, 2008.

16. Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. In Dang Van Hung and Padmanabhan Krishnan, editors, *SEFM*, pages 210–219. IEEE Computer Society, 2009.

17. Leslie Lamport. The temporal logic of actions. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.

18. Tommi Mikkonen. Formalizing design patterns. In *ICSE*, pages 115–124, 1998.

19. Antoine Requet. BART: A tool for automatic refinement. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 345. Springer, 2008.

20. Renato Silva and Michael Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.

21. Michael Spivey. *The Z Notation: A reference manual*. Prentice Hall International, 2nd edition, 1992.

22. Toufik Taibi, Ángel Herranz-Nieva, and Juan José Moreno-Navarro. Stepwise refinement validation of design patterns formalized in TLA+ using the TLC model checker. *Journal of Object Technology*, 8(2):137–161, 2009.

23. S. Wieczorek, A. Roth, A. Stefanescu, and A. Charfi. Precise steps for choreography modeling for SOA validation and verification. In *Proceedings of the Fourth IEEE International Symposium on Service-Oriented System Engineering*, December 2008. http://deploy-eprints.ecs.soton.ac.uk/41/.