The original publication is available at http://doi.ieeecomputersociety.org/10.1109/SEFM.2009.17
Appeared in Proceedings of the SEFM 2009 conference © IEEE Computer Society

Event-B Patterns and Their Tool Support

Thai Son Hoang, Andreas Fürst and Jean-Raymond Abrial
Department of Computer Science
Swiss Federal Institute of Technology Zurich (ETH-Zurich)
CH-8092, Zurich, Switzerland
Email: htson@inf.ethz.ch afuerst@student.ethz.ch jabrial@inf.ethz.ch

***Abstract*—Event-B has given developers the opportunity to construct models of complex systems which are correct by construction. However, there is no systematic approach, especially in terms of reusing, 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 future tool support within the associated Rodin Platform of Event-B. The approach has been applied successfully in some medium-size industrial case studies.**

***Keywords*-Event-B; design patterns; reuse;**

I. 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 to 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 [1]. The idea is to have some pre-defined solutions, and incorporate them into the development with some modification and/or instantiation. We want to borrow 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 investigation [2], we have already worked on several examples to understand the usefulness and applicability of the approach. We extend this work and formalise the process 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 called "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. 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 II gives a short introduction to Event-B. Section III presents a case study to illustrate the motivation for our approach. Section IV 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 V. Section VI briefly mentions the tool support. Finally, in Section VII we review related work and point out future directions.

II. The Event-B Modelling Method

Event-B [3], unlike classical B [4], 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 [5]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section II-A, and machine refinement in Section II-B.

A. 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* $S(v)$[1]. 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

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

The short form

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

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

$$x \;:=\; E(v) \tag{3}$$
$$x \;:\in\; E(v) \tag{4}$$
$$x \;:|\; Q(v, x') \quad , \tag{5}$$

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:

$$BAP\big(x \;:=\; E(v)\big) \quad \widehat{=} \quad x' \;=\; E(v) \tag{6}$$
$$BAP\big(x \;:\in\; E(v)\big) \quad \widehat{=} \quad x' \;\in\; E(v) \tag{7}$$
$$BAP\big(x \;:|\; Q(v, x')\big) \quad \widehat{=} \quad Q(v, x') \quad . \tag{8}$$

A before-after predicate describes the relationship between the state just before an assignment has occurred (represented by un-primed 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:

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

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.

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

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

$I(v)$ $G(v)$ $\vdash$ $\exists v' \cdot \boldsymbol{S}(v, v')$	**FIS**

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*:

$I(v)$ $G(v)$ $\boldsymbol{S}(v, v')$ $\vdash$ $I(v')$	**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**).

B. Machine Refinement

Machine refinement provides a mean to introduce more details about the dynamic properties of a model [5]. For more on the well-known theory of refinement, we refer to the Action System formalism that has inspired the development of Event-B [6]. 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 a *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:

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

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

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.

$I(v)$ $J(v,w)$ $H(w)$ $\vdash$ $\exists w' \cdot \boldsymbol{T}(w,w')$	**FIS_REF**

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

$I(v)$ $J(v,w)$ $H(w)$ $\vdash$ $G(v)$	**GRD**

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

$I(v)$ $J(v,w)$ $H(w)$ $\boldsymbol{T}(w,w')$ $\vdash$ $\exists v' \cdot \boldsymbol{S}(v,v') \wedge J(v',w')$	**SIM**

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.

$$\text{skip} \;\widehat{=}\; \textbf{begin}\ \text{SKIP}\ \textbf{end}$$

Moreover, it may be proved that new events do not collectively diverge, but this is not relevant here.

III. Request/Confirm Protocol

In this section, we look at the development of a simple protocol, namely *Request/Confirm* in order to understand what we mean by design patterns and how to apply them in system development. Section III-A 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 III-B we formally present a pattern, namely *single-message communication*, including its specification and refinement. Finally, we illustrate how the pattern is reused (twice) in our development of the actual Request/Confirm protocol in Section III-C.

A. Description and Formal Specification

There are two parties participating in this protocol namely the *Sender* and the *Receiver*. The protocol contains two steps as follows.

1) First, the *Sender* sends a *request* to the *Receiver*.
2) After receiving this *request*, the *Receiver* sends a *confirmation* back to the *Sender*.

Formally, we can use two Boolean variables to represent the state of the protocol: *req* to indicate that requesting has occurred, and *conf* to indicate that confirmation has occurred.

There is an invariant stating that if confirmation has happened then requesting must have happened. The state of the formal model is as follows.

variables: $req, conf$

invariants:
ReqCnf_0_1: $conf = \text{TRUE} \Rightarrow req = \text{TRUE}$

The dynamic system can be seen in Figure 1:

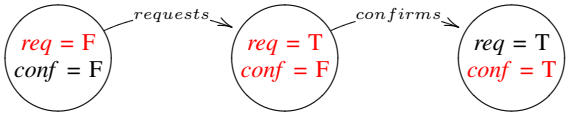

Figure 1. Request/Confirm protocol

The "requesting" phase starts when both variables are FALSE and changes the value of *req* to TRUE. The "confirmation" phase starts after the "requesting" phase and changes the value of *conf* variable from FALSE to TRUE. This is formalised as the following two events requests and confirms, representing the requesting phase and the confirmation phase, respectively.

requests
when
$req = \text{FALSE}$
then
$req := \text{TRUE}$
end

confirms
when
$req = \text{TRUE}$
$conf = \text{FALSE}$
then
$conf := \text{TRUE}$
end

The specification of the above two events are very similar, except for the additional guard $req = \text{TRUE}$ of the event confirms. Note that the guard of requests implies $conf = \text{FALSE}$ (due to the invariant **ReqCnf_0_1** and contraposition). The two events both correspond to transferring some information from one side to another which we call *single-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: one for the "requesting" phase, one for the "confirmation" phase.

B. Single-Message Communication

This section presents the development of a communication between two parties A and B for transferring some information (once) from A to B.

The specification of this protocol contains only one Boolean variable *trans*. The value of the variable is TRUE when the communication has occurred.

variables: *trans*

invariants:
SglCom_0_1: *trans* $\in$ BOOL

This single-message communication can be seen in Figure 2.

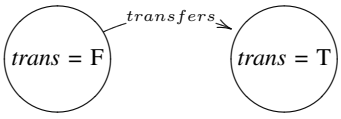

Figure 2. Single-Message Communication

There is only one event in this model to change the value of variable *trans* accordingly.

transfers
when
trans = FALSE
then
trans := TRUE
end

However this is only the abstraction of this protocol. In fact, the message needs to be sent via a 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.

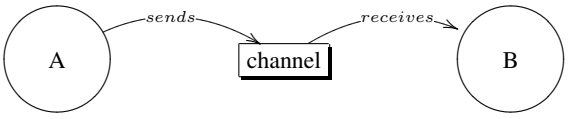

Figure 3. Communication via a channel

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

- *sent*: TRUE if A has already sent the message.
- *chan*: TRUE if there is a message in the *channel* waiting to be received.
- *recv*: TRUE if B has already received the message.

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

invariants:
SglCom_1_1: *trans* = *sent*

We also have two technical invariants about the properties of the protocol. Firstly, if B has received the message then A must have sent the message. Secondly, if there is a message in the *channel* then A must have sent the message, but B has not yet received the message.

invariants:
SglCom_1_2: *recv* = TRUE $\Rightarrow$ *sent* = TRUE
SglCom_1_3: *chan* = TRUE $\Rightarrow$
sent = TRUE $\wedge$ *recv* = FALSE

The events sends and receives are straightforward as follows.

sends
refines $transfers$
when
sent = FALSE
then
sent := TRUE
chan := TRUE
end

receives
when
chan = TRUE
then
chan := FALSE
recv := TRUE
end

The event sends is enabled if A has not yet sent the message. The action of the event specifies that A now sent the message and the message is in the *channel*. For event receives, it 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 the message. Note that event receives here is a new event (i.e. it refines skip).

C. Using the Pattern for the Protocol

In this section, we see how the pattern developed in Section III-B s used for developing the Request/Confirm protocol of Section III-A. There are four steps in doing this.

1) Firstly, we need to "match" the specification of the pattern with the problem.

2) Secondly, we need to "syntactically check" the matching to see if the pattern is applicable.
3) Thirdly, we could "rename" the variables and events in the refinement of the pattern to avoid name clash (since we can instantiate the same pattern many times). This step is optional.
4) Lastly, we "incorporate" the renamed refinement of the pattern to create a refinement of the problem.

As mentioned before, we can instantiate the single-message pattern twice for the Request/Confirm protocol: one for the "requesting" phase and one for the "confirming" phase.

1) Pattern for "Requesting" Phase: We follow the different steps to incorporate a single-message communication pattern for the "requesting" phase.

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 the variable *trans* and event transfers of the pattern match with variable *req* and event requests of the problem accordingly.

pattern	$\rightsquigarrow$	**problem**
trans	$\rightsquigarrow$	*req*
transfers	$\rightsquigarrow$	requests

2) The second step is to syntactically check the validity of the pattern. This should be done automatically by a tool. At the moment, we can assure ourselves that this step is valid.
3) The third step is to rename the variables and events of the pattern refinement according to the following rules.

original	$\rightsquigarrow$	**renamed as**
sent	$\rightsquigarrow$	*S_sent_req*
chan	$\rightsquigarrow$	*S2R_chan_req*
recv	$\rightsquigarrow$	*R_recv_req*
sends	$\rightsquigarrow$	S_sends_request
receives	$\rightsquigarrow$	R_receives_request

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: *S_sent_req*, *S2R_chan_req*, *R_recv_req*

invariants:
ReqCnf_1_1: $req = S\_sent\_req$
ReqCnf_1_2: $R\_recv\_req = \text{TRUE} \Rightarrow S\_sent\_req = \text{TRUE}$
ReqCnf_1_3: $S2R\_chan\_req = \text{TRUE} \Rightarrow S\_sent\_req = \text{TRUE} \wedge R\_recv\_req = \text{FALSE}$

S_sends_request
refines *requests*
when
$S\_sent\_req = \text{FALSE}$
then
$S\_sent\_req := \text{TRUE}$
$S2R\_chan\_req := \text{TRUE}$
end

R_receives_request
when
$S2R\_chan\_req = \text{TRUE}$
then
$S2R\_chan\_req := \text{FALSE}$
$R\_recv\_req := \text{TRUE}$
end

confirms
refines *confirms*
when
$S\_sent\_req = \text{TRUE}$
$conf = \text{FALSE}$
then
$conf := \text{TRUE}$
end

Note that in the resulting refinement, the event confirms needs to take into account the fact that variable *req* has been matched with variable *trans* of the pattern specification, and this variable is refined to *S_sent_req*.

2) Pattern for "Confirming" Phase: We now follow similar steps to use a single-message communication pattern for the "confirming" phase.

1) The matching is as follows

pattern	$\rightsquigarrow$	**problem**
trans	$\rightsquigarrow$	*conf*
transfers	$\rightsquigarrow$	confirms

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

original	⇝	renamed as
sent	⇝	*R_sent_conf*
chan	⇝	*R2S_chan_conf*
recv	⇝	*S_recv_conf*
sends	⇝	R_sends_confirmation
receives	⇝	S_receives_confirmation

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

variables: *S_sent_req*, *S2R_chan_req*, *R_recv_req*, *R_sent_conf*, *R2S_chan_conf*, *S_recv_conf*

invariants:

ReqCnf_2_1: $conf = R\_sent\_conf$

ReqCnf_2_1: $S\_recv\_conf = \text{TRUE} \Rightarrow R\_sent\_conf = \text{TRUE}$

ReqCnf_2_1: $R2S\_chan\_conf = \text{TRUE} \Rightarrow R\_sent\_conf = \text{TRUE} \wedge S\_recv\_conf = \text{FALSE}$

S_sends_request
status *extended*

R_receives_request
status *extended*

R_sends_confirmation
refines *confirms*
when
$S\_sent\_req = \text{TRUE}$
$R\_sent\_conf = \text{FALSE}$
then
$R\_sent\_conf := \text{TRUE}$
$R2S\_chan\_conf := \text{TRUE}$
end

S_receives_confirmation
when
$R2S\_chan\_conf = \text{TRUE}$
then
$R2S\_chan\_conf := \text{FALSE}$
$S\_recv\_conf := \text{TRUE}$
end

In the above, the *extended* events mean they extend the corresponding abstract version (with the possibility of having more guards and more actions).

There are some differences between this application of the single-message pattern for this phase compares with the application in Section III-C1.

- The matching between transfers and confirms is not exact. In fact the event confirms syntactically contains transfers (with the matching of the variables).
- The additional guard, i.e. $S\_sent\_req = \text{TRUE}$ is kept when incorporating the pattern refinement into the development.
- This guard is in fact a "cheat" in the model since event R_sends_confirmation supposes to be corresponding to the *Receiver* whereas the guard refers to variable *S_sent_req* which is a variable of the *Sender*. This problem will be handled by a standard refinement step in the next section.

3) Removing the "Cheating" Guard: The cheating guard, i.e.

$$S\_sent\_req = \text{TRUE}$$

can be removed and replaced by the following guard which uses the variable of B:

$$R\_recv\_req = \text{TRUE} .$$

The reasoning for guard strengthening is based on the following invariant.

$$R\_recv\_req = \text{TRUE} \Rightarrow S\_sent\_req = \text{TRUE}$$

This step is a standard refinement in Event-B. The final model of R_sends_confirmation is as follows.

R_sends_confirmation
refines *R_sends_confirmation*
when
$R\_recv\_req = \text{TRUE}$
$R\_sent\_conf = \text{FALSE}$
then
$R\_sent\_conf := \text{TRUE}$
$R2S\_chan\_conf := \text{TRUE}$
end

IV. 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.

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

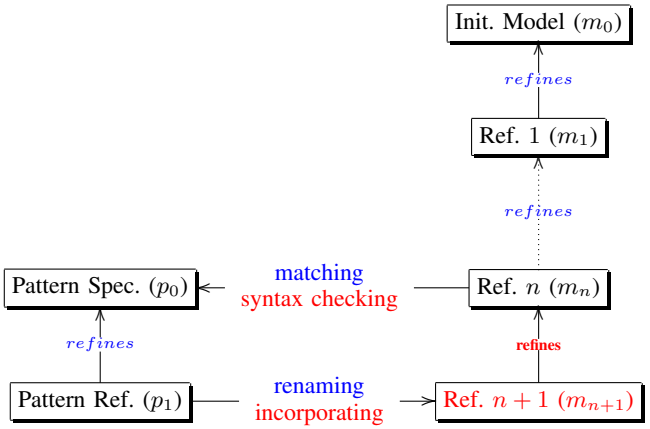

Figure 4. Using Patterns in Event-B

Moreover, we have presented here the incorporation of each single-message communication protocol 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 name clashes.

A. Formalisation of the Approach

We assume that we have the following patterns containing a specification p_0 and its refinement p_1 as follows.

p_0
variables v
invariants $J(v)$

p
when $L(v)$ **then**
$T(v, v')$
end

p_1
variables w
invariants
$v = X(w)$
$K(v, w)$

q
when $M(w)$ **then**
$U(w, w')$
end

We 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 $T(v, v')$. In the refinement p_1 of p_0, variable v is data refined by variable w with gluing invariant separated in to $v = X(w)$ and $K(v, w)$. Here we made the assumption that the gluing invariant can be functionally expressed as $v = X(w)$ with some other extra invariant $K(v, w)$. 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 $U(w, w')$.

We assume that we have arrived at a refinement level in a particular development which we called 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.

m_n
variables b, c

e
when
$H(b)$
$N(b, c)$
then
$R(b, b')$
$S(b, c, c')$
end

We consider the event e of the problem specification which is going to be matched. The event is separated into the parts which are matched with the 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 variables in the pattern need to be matched with some variables in the problems. However, this condition can be relaxed to make the approach more flexible (see future work in Section VII). Hence the guard of the events 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 $R(b, b')$ –which is a match of $T(v, v')$– and $S(b, c, c')$. The validity of this matching can be syntactically checked and/or even can be "discovered" by a tool. The matching and the extraction from the gluing invariant can be summarised as follows.

pattern		**problem**
v	$\widehat{=}$	b
p	$\widehat{=}$	e
$L(v)$	$\widehat{=}$	$H(b)$
$T(v, v')$	$\widehat{=}$	$R(b, b')$

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

m_{n+1}
variables w, c
invariants
$b = X(w)$
$K(b, w)$
$J(b)$

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

We must guarantee that the constructed machine m_{n+1} is indeed a refinement of the specification m_n. The detail proofs are in [7]. 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}.

B. 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 that 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 Request/Confirm protocol, we use the single-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 single-message communication and Request/Confirm protocol is in Table I. As we can see, by developing the single-message pattern separately, we have to prove 9 obligations. However, we do not need to prove the model "Request/Confirm 1" (which has 19 obligations) since it is correct by construction. Hence in total we save 10 obligations. Note that the number of proof obligations for "Request/Confirm 1" is roughly twice that of "Single 1", since we use the single-message communication pattern twice.

Models	Total	Auto. (%)	Man. (%)
Single 0	0	0 (N/A)	0 (N/A)
Single 1	9	9 (100%)	0 (0%)
Request/Confirm 0	3	3 (100%)	0 (0%)
Request/Confirm 1	19	19 (100%)	0 (0%)
Request/Confirm 2	1	1 (100%)	0 (0%)

Table I
PROOF STATISTICS FOR SINGLE-MESSAGE AND REQUEST/CONFIRM

V. 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 [8] and Ordering/Supply Chain A2A Communications. Overall, the number of proof saving by the application of patterns is around 34% for both examples. In this section, we give the description of other patterns that have been used in these protocols.

- Section V-A presents the *Request/Confirm/Reject* pattern.
- Section V-B presents the *Multiple-Message Communication* pattern.
- Section V-C presents the *Multiple-Message Communication with Repetition* pattern.
- Section V-D presents the *Synchronous Multiple-Message Communication* pattern.
- Section V-E presents the *Question/Response* pattern.

A. Request/Confirm/Reject Pattern

The description of the protocol is as follows.

1) The *Sender* sends a request to the *Receiver*.
2) After receiving this request, the *Receiver* can either send a "confirmation" back to the *Sender* if he agrees; or the *Receiver* send 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 5.

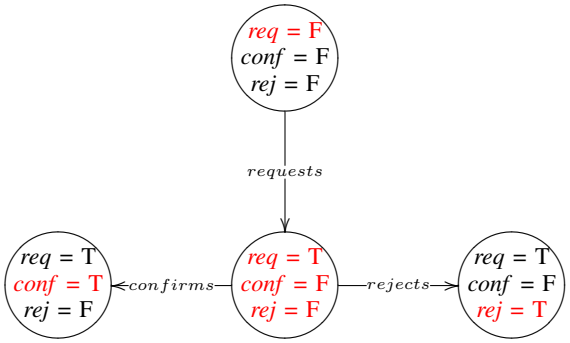

Figure 5. Request/Confirm/Reject protocol

The development of this pattern used the single-message communication pattern three times.

B. Multiple-Message Communication Pattern

The description of the protocol is as follows.

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

C. Multiple-Message with Repetition Communication Pattern

The description of the protocol is as follows.

1) There are two parties: *Sender* and *Receiver*
2) The *Sender* can send many messages (multiple message) to the *Receiver*.
3) The messages can be the same, in other words, messages could be resent.
4) To distinguish the freshness of the message, each message is stamped with a sequence number.

5) The *Receiver* can receive any message which is not old.
6) The *Receiver* can discard any message.

The only difference with the Multiple-Message Communication (no repetition) is that no messages can be resent in this protocol.

D. Synchronous Multiple-Message Communication Pattern

The description of the protocol is as follows.

1) There are two parties: *Sender* and *Receiver*
2) The *Sender* can send many messages (multiple message) to the *Receiver*, but only one at a time.
3) The *Sender* is blocked in sending a new message until the *Receiver* received the message in the channel.
4) The messages are different, in other words, there is no resend.
5) To distinguish the freshness of the message, each message is stamped with a sequence number.

The significant difference to the Multiple-Message Communication (no repetition) is that the channel is either empty or filled with at most one message. This also leads to a less complex design of the channel.

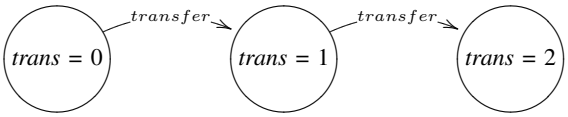

Figure 6. Synchronous Multiple-Message Communication

E. Question/Response Pattern

The description of the protocol is as follows.

1) There are two parties: *Questioner* and *Responder*
2) The protocol consists of an unbounded number of rounds.
3) In each round, the *Questioner* transfers a message (question) to the *Responder*, and in return, the *Responder* transfers a message (response) to the *Questioner*.
4) The question message and the corresponding response message have the same number.

The communication of the *Questioner* and *Responder* is synchronous. Therefore the *Questioner* cannot transfer a new question before the *Responder* has transferred a response and vice versa. The pattern was developed by using two synchronous multiple-message patterns.

VI. Proposed Tool Support

The tool support for pattern approach should closely follow the different steps for applying pattern.

- **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.

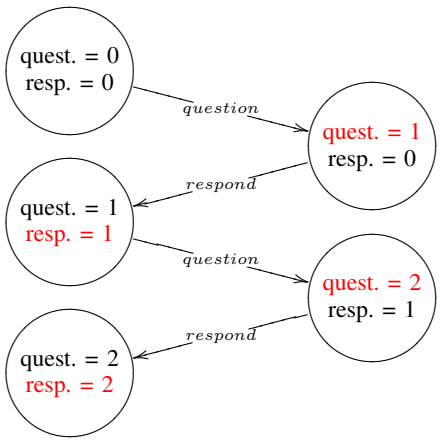

Figure 7. Question/Response protocol with two rounds

- **Syntax Checking.** Consistency (e.g. types of variables, signature of events, etc.) for the matching should be verified at this step.
- **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.
- **Incorporation.** Finally, the tool generate the refinement of the problem according to the input in the previous steps.

VII. Related and Future Work

We applied our approach to two medium-size case studies from SAP, namely the Buyer/Seller B2B [8] and Ordering/Supply Chain A2A Communications. However, our approach is general and is not be restricted to this specific domain.

Our approach is related with decomposition [5], [9] 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 "automatic refinement tool" [10]. Our patterns which are simply a model which encodes some design decision about refining a certain model. 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.

As for future work, we are currently implementing the support for this approach as plug-in for the Rodin Platform [11]–[13] which is an open source platform based on Eclipse. The current documentation for tool support is at the Event-B wiki documentation system http://wiki.event-b.org/index.php/Pattern. At the same time, we are going to

investigate more examples in other domains which could benefit from our approach.

Moreover, we also need to "instantiate" the context of the pattern development. In our example so far the contexts of the pattern and the problem are the same. However, we would like to use the pattern in a more general context. For example, the model of the communication for transferring a certain (abstract) messages should be instantiated for and kind of (concrete) messages, e.g. if the message is just a Boolean, or if the message contains some numbers or some complicated data structure. This requires that the context of the pattern to be instantiated accordingly.

As mentioned before, it is not necessarily the case that all the variables of the patterns need to be matched with some variables in the patterns. 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 super-position refinement [5]. 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.

ACKNOWLEDGMENT

This work has been supported by DEPLOY —an European Commission Information and Communication Technologies FP7 project [14]. We would like to thank Matthias Schmalz for his useful and constructive comments.

We would like to thank anonymous reviewers for constructive comments to improve the quality of the paper.

REFERENCES

[1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Mar. 1995, iSBN-10: 0201633612 ISBN-13: 978-0201633610.

[2] J.-R. Abrial and T. S. Hoang, "Using design patterns in formal methods: An Event-B approach," in *ICTAC*, ser. Lecture Notes in Computer Science, J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, Eds., vol. 5160. Springer, 2008, pp. 1–2.

[3] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009, to appear.

[4] ——, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to Event-B," *Fundam. Inform.*, vol. 77, no. 1-2, pp. 1–28, 2007.

[6] R.-J. Back, "Refinement Calculus II: Parallel and Reactive Programs," in *Stepwise Refinement of Distributed Systems*, ser. Lecture Notes in Computer Science, J. W. deBakker, W. P. deRoever, and G. Rozenberg, Eds., vol. 430. Mook, The Netherlands: Springer-Verlag, May 1989, pp. 67–93.

[7] A. Fürst, "Design patterns in Event-B and their tool support," Master's thesis, Deparment of Computer Science, ETH Zurich, Mar. 2009, http://e-collection.ethbib.ethz.ch/view/eth:41612.

[8] 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/.

[9] M. Butler, *Decompostion Structures for Event-B*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5423, ch. Integrated Formal Methods, pp. 20–38, http://www.springerlink.com/content/3202127567642301/.

[10] A. Requet, "BART: A tool for automatic refinement," in *ABZ*, ser. Lecture Notes in Computer Science, E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., vol. 5238. Springer, 2008, p. 345.

[11] J.-R. Abrial, "A system development process with Event-B and the Rodin Platform," in *ICFEM*, ser. Lecture Notes in Computer Science, M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, Eds., vol. 4789. Springer, 2007, pp. 1–3.

[12] J.-R. Abrial, M. J. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for event-b," in *ICFEM*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, 2006, pp. 588–605.

[13] Event-B.org, "The Rodin Platform," http://www.event-b.org/.

[14] DEPLOY Project, "Industrial deployment of system engineering methods providing high dependability and productivity," http://www.deploy-project.eu.