Towards Automated Refinement: Patterns in Event B

Alexei Iliasov[1], Elena Troubitsyna[2], Linas Laibinis[2], and Alexander Romanovsky[1]

[1] Newcastle University, UK
[2] Åbo Akademi University, Finland
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk
{linas.laibinis, elena.troubitsyna}@abo.fi

Abstract. Formal modelling is indispensable for engineering highly dependable systems. However, a wider acceptance of formal methods is hindered by their insufficient usability and scalability. In this paper, we aim at assisting developers in rigorous modelling and design by increasing automation of development steps. We introduce a notion of refinement patterns – generic representations of typical correctness-preserving model transformations. Our definition of a refinement pattern contains a description of syntactic model transformations, as well as the pattern applicability conditions and proof obligations for verification of correctness preservation. This establishes a basis for building a tool supporting formal system development via pattern reuse and instantiation. We present a prototype of such a tool and some examples of refinement patterns for automated development in the Event B formalism.

1 Introduction

System development by stepwise refinement is a *formal* model-driven development approach that advocates development of systems correct by construction. Development starts from an abstract model, which is gradually transformed into implementation. Each model transformation step, called a *refinement* step, allows a designer to incorporate implementation details into the model. Correctness of each refinement step is validated by proofs.

The refinement approach supports verification and clear traceability of system properties through various abstraction levels. However, it is still poorly integrated into existing software engineering process. Among the main reasons hindering its application are complexity of carrying proofs, lack of expertise in abstract modelling, and insufficient scalability.

In this paper we propose an approach that aims at facilitating integration of formal methods into the existing development practice by leveraging automation of refinement process and increasing reuse of models and proofs. We aim at automating certain model transformation steps via instantiation and reuse of prefabricated solutions, which we call *refinement patterns*. Such patterns generalise certain typical model transformations reoccurring in a particular development method. They can be thought of as ”refinement rules in large”.

In general, a refinement pattern is a generic model transformer. Essentially it consists of three parts. The first part is the pattern applicability conditions, i.e., the syntactic and semantic conditions that should be fulfilled by the model for a refinement pattern to be applicable. The second part contains definition of syntactic manipulations over the model to be transformed. Finally, the third part consists of the proof obligations that should be discharged to verify that the performed model transformation is indeed a refinement step.

Application of refinement patterns is compositional. Hence some large model transformation steps can be represented by a certain combination of refinement patterns, and therefore can also be seen as refinement patterns per se. A possibility to compose patterns significantly improves scalability of formal modelling. Moreover, a representation of a refinement step by a number of syntactic manipulations over a model provides a basis for automation. Finally, our approach supports extensive reuse of not only models but also proofs. Indeed, by proving that an application of a generic pattern produces a valid refinement of a generic model, we at the same time verify the correctness of such a transformation for any of its instances. This allows us to significantly reduce or even avoid proving activity in a concrete development.

The theoretical work on defining refinement patterns presented in this paper established a basis for building a prototype tool for automating refinement process in Event B[10]. The tool has been developed as a plug-in for the RODIN platform [1] – an open toolset for supporting modelling and refinement in the Event B framework. We believe that by creating a large library of refinement patterns and providing automated tool support for pattern matching and instantiation, we will make formal modelling and verification more accessible for software engineers and hence facilitate integration of formal methods into software engineering practice.

The paper is organised as follows: in Section 2 we give a brief introduction into our modelling framework – Event B. In Section 3 we define a notion of a transformation rule and its special case – a refinement pattern, as well as a language for constructing transformation rules. In Section 4 we describe how to compose patterns. In Section 5 we construct a rather complex pattern for introducing well-known fault tolerance mechanism – triple modular redundancy and discuss the prototype tool for documenting and using patterns in Event B development. Finally, in Section 6 we give some concluding remarks and discuss related work.

2 Modelling and Refinement in Event B

2.1 Introduction into Event B

Event B [2] is an extension of the B Method [3] to model parallel, distributed and reactive systems. The Rodin platform [1] provides automated tool support for modelling and verification in Event B.

Event B uses the Abstract Machine Notation for constructing and verifying models. An abstract machine encapsulates a state (the variables) of the model and provides operations on its state. A simple abstract machine has the following general form:

SYSTEM AM
VARIABLES v
INVARIANT I
INITIALISATION *INIT*
EVENTS
  E_1
  $\dots$
  E_N

The machine is uniquely identified by its name *AM*. The state variables of the machine, *v*, are declared in the **VARIABLES** clause and initialised in *INIT* as defined in the **INITIALISATION** clause. The variables are strongly typed by constraining predicates of the machine invariant *I* given in the **INVARIANT** clause. The invariant is usually defined as a conjunction of the constraining predicates and the predicates defining the properties of the system that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **EVENTS** clause. An event is defined as follows:

$$E = \textbf{WHEN } g \textbf{ THEN } S \textbf{ END}$$

where the guard *g* is conjunction of predicates over the state variables *v*, and the action *S* is an assignment to the state variables.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. The action can be either a deterministic assignment to the state variables or a non-deterministic assignment from a given set or an assignment according to a given postcondition. These assignments are denoted as $:=, :\in$ and $:|$ correspondingly. If several events are enabled then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

The Event B models are formally defined using the weakest precondition semantics [7]. The weakest precondition semantics provides us with a foundation for establishing correctness of specifications and verifying refinements between them. For instance, we verify correctness of a specification by proving that its initialization and all events establish the invariant.

The basic idea underlying formal stepwise development by refinement is to design the system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit unimplementable, specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artifacts.

Assume that the refinement machine AM' is a result of refinement of the abstract machine *AM*:

SYSTEM AM'
VARIABLES v'
INVARIANT I'
INITIALISATION *INIT'*
EVENTS
  E_1
  $\ldots$
  E_N

The machine AM' might contain new variables as well as replace the abstract data structures of *AM* with the concrete ones. The invariant of AM' – I' – defines not only the invariant properties of the refined model, but also the connection between the state spaces of AM and AM'. For a refinement step to be valid, every possible execution of the refined machine must correspond (via I') to some execution of the abstract machine. To demonstrate this, we should prove that $INIT'$ is a valid refinement of *INIT*, each event of AM' is a valid refinement of its counterpart in *AM* and that the refined specification does not introduce additional deadlocks, i.e.,

$$\begin{aligned}&wp(INIT',\ \neg wp(INIT, \neg I'))\ =\ true,\\ &I \wedge I' \wedge g_i'\ \Rightarrow\ g_i \wedge wp(S', \neg wp(S, \neg InvC)), \quad \text{and}\\ &I \wedge I' \wedge g_i\ \Rightarrow\ \bigvee_i^N g_i'\end{aligned}$$

2.2 Event-B Models as Syntactic Objects

To define refinement patterns, we now consider an Event B model as a syntactic mathematical object. For brevity, we omit representations of some model elements here, though they are supported in our tool implementation [10]. A subset of Event-B models used in this paper can be described by the following data structure:

$$\begin{array}{lll} \text{model} :: \mathit{var} : \text{VAR}^* & \text{event} :: \mathit{name} : \text{EVENT} & \text{action} :: \mathit{var} : \text{VAR} \\ \qquad\quad \mathit{inv} : \text{PRED}^* & \qquad\quad \mathit{param} : \text{PARAM}^* & \qquad\quad \mathit{style} : \text{STYLE} \\ \qquad\quad \mathit{evt} : \text{event}^* & \qquad\quad \mathit{guards} : \text{PRED}^* & \qquad\quad \mathit{expr} : \text{EXPR} \\ & \qquad\quad \mathit{actions} : \text{action}^* & \end{array}$$

Here VAR, PRED, EXPR, EVENT, PARAM are the carrier sets reserved correspondingly for model variables, predicates, expressions, event names and parameters. An event is represented by a tuple containing the event name, (a list of) its parameters, guards, and actions. The reserved event name `init` denotes the initialisation event. An action, in its turn, is a tuple containing a variable, an action style and an expression. An action style denotes one of the assignment types : i.e., $\text{STYLE} = \{:=, :\in, :|\}$.

Sub-elements of a model element can be accessed by using the dot operator: $act.style$ is the style of an action act. Instances of the models, events and actions are constructed using a special notation $\langle a_1 \mid \cdots \mid a_n \rangle$. The following example shows how an Event B model can be represented as a syntactic object in our notation:

SYSTEM $m0$	
VARIABLES x	$\langle\ \langle x \rangle\ \mid$
INVARIANT $x \in \mathbb{Z}$	$\langle "x \in \mathbb{Z}" \rangle\ \mid$
INITIALISATION $x := 0$	$\langle\ \langle \texttt{init} \mid - \mid - \mid \langle x \mid := \mid "0" \rangle \rangle,$
EVENTS	
$count =$ **BEGIN** $x := x + 1$ **END**	$\langle \texttt{count} \mid - \mid - \mid \langle x \mid := \mid "x+1" \rangle \rangle \rangle \rangle$

In the example, x is an element of VAR, `init` and `count` are event names from EVENT, $"x \in \mathbb{Z}"$ is a predicate, and $"0"$, $"x+1"$ are model expressions.

Now we have set a scene for a formal definition of refinement patterns that aim at automating refinement process Event B.

3 Refinement Patterns

3.1 Transformation Rules and Refinement Patterns

Usually a refinement step in Event B results in introducing several changes in all clauses of a refined model. Then we verify by proofs that these changes indeed result in a correct model refinement. Often a refinement step can be seen as a composition of ”standard” (frequently reoccurring) localized transformations distributed all over the model. However, in general it is unclear how to reuse the models and proofs constructed previously and possibly automate execution of these transformations.

In this paper we propose to tackle this problem via definition and reuse of refinement patterns. Our definition of refinement patterns builds on the idea of refinement rules [4, 13]. A refinement pattern in general is a model transformer. Unlike design patterns [8], a refinement pattern is ”dynamic” in a sense that it takes a model as an input and produces a new model as an output. To define a refinement pattern we first give a more general definition of a transformation rule

Definition 1. *Let S be a set of all well-formed Event B models. Then a transformation rule T is a function computing a new model for a given input model:*

$$T : S \times C \mathrel{+\!\!\!\rightarrow} S$$

where C contains a set of all possible configurations (i.e., additional parameters) of a transformation rule.

Note that T is defined as a partial function, i.e., it produces a new model only for some acceptable input models s and configurations c, i.e., when $(s, c) \in \mathsf{dom}(T)$.

Definition 2. *A refinement pattern is a transformation rule $P : S \times C \mathrel{+\!\!\!\rightarrow} S$ that, for any acceptable input model and configuration, constructs a model refinement:*

$$\forall\, s, c.(s, c) \in \mathsf{dom}(P) \;\Rightarrow\; s \sqsubseteq P(s, c)$$

where $\sqsubseteq$ denotes a refinement relation.

Our definition of a refinement pattern consists of three parts. The first part is the pattern applicability conditions, i.e., the syntactic and semantic conditions that should be fulfilled by the model for a refinement pattern to be applicable. The second part contains definition of syntactic manipulations on the model to be transformed. Finally, the third part consists of the proof obligations that should be discharged to verify that the performed model transformation is indeed a refinement step. It is easy to see, that a refinement pattern manipulates a model on both syntactic and semantic level.

We believe that the main benefit of refinement patterns is in possibility to construct large transformation rules that potentially automate certain domain-specific model transformations. Examples of such transformations could be integrating certain fault tolerance mechanisms or introducing communication protocols etc. We propose a special language for constructing larger transformation rules.

3.2 The Language of Transformations

Our language contains basic transformation rules as well as the constructs allowing to compose complex rules from the simpler ones. For instance, a refinement pattern is usually composed from several basic transformation rules. These rules themselves might not be refinement patterns. However, by attaching to them additional proof obligations, we can verify that their composition constitutes a refinement pattern.

The structure of the basic rules reflects the way a transformation rule or a refinement pattern is applied. First, rule applicability for a given input model and configuration parameters is checked. The applicability condition to be checked can contain both syntactic and semantic constraints on input models and configurations. Mathematically, for a transformation rule T, its applicability condition corresponds to $\mathsf{dom}(T)$. Then, the input model s for the given configuration c is syntactically transformed into the output model calculated as the function application $T(s, c)$. Finally, in case of a refinement pattern, the result $T(s, c)$ should be demonstrated to be a refinement of the input model s, i.e., $s \sqsubseteq T(s, c)$. The last expression, using the proof theory of Event B, can be simplified to the specific proof obligations on model elements to be verified.

A basic rule has the following general form:

```
rule $name(c)$
    context $Q(c, s)$
    effect $E(c, s)$
    proof obligation $PO_1(c, s)$
    ...
    proof obligation $PO_n(c, s)$
```

Here $name$ and c are correspondingly the rule name and list of its parameters. The predicate $Q(c, s)$ defines the rule application context (applicability conditions), where s is the model being transformed. The effect function $E(c, s)$ computes a new model from the current model s and the parameters c. The proof obligation part contains a list of theorems to be discharged to establish that the rule is a (part of) refinement pattern and not just a transformation rule. From now on, we write **context**(r), **effect**(r) and **proof_obligations**(r) to refer to the context, the effect computation function, and the collection of proof obligations of the rule r.

As an example, let us consider two primitive transformation rules. Below we define a transformation rule that allows us to introduce one or more new variables into the model:

```
rule $newvar(vv)$
    context $vv \cap s.var = \varnothing$
    effect $\langle s.var \cup vv \mid s.inv \mid s.evt \rangle$
    proof_obligation $\forall v \in vv \cdot (\exists a \cdot a \in s.\mathtt{init}.action \wedge v \in a.var)$
```

The rule applicability condition requires that the new variables have fresh names for the input model. The effect function simply adds the new variables to the model structure. The rule also has a single proof obligation requiring that the variable(s) is assigned in the initialisation action. Such an action would have to be added by some other basic rule for the same refinement step.

Next example is the rule for adding new model invariant(s).

```
rule $newinv(ii)$
    context $ii \subseteq \mathrm{PRED} \wedge \forall i \in ii \cdot FV(ii) \subseteq s.var$
    effect $\langle s.var \mid inv \cup ii \mid evt \rangle$
    proof obligation
        $\forall (e, v, v') \cdot e \in s.evt \wedge$
            $Inv(v) \wedge Guards_e(v) \wedge BA(v, v') \;\Rightarrow\; Inv(v')$
    proof_obligation $\exists v \cdot lnv(v)$
```

Here $FV(x)$ is set of free variables in x, Inv stands for $(\bigwedge_{i \in s.inv \cup ii} i)$, and $Guards_e$ is defined as $(\bigwedge_{g \in e.guards} g)$. Moreover, BA is the before-after relation describing the action execution in terms of the before and after values of the model variables. Both proof obligations are taken directly from the Event-B semantics (i.e., the corresponding proof obligation rules). The first obligation requires to show that the new invariant is preserved by all model events, while the second one checks feasibility of such an

$p(c) = basic(c)$	*primitive rule*
$\mid p; q$	*sequential composition*
$\mid p \| q$	*parallel composition*
$\mid$ **if** $Q(c, s)$ **then** p **end**	*conditional rule*
$\mid$ **conf** $i : Q(i, c, s)$ **do** $p(i \cup c)$ **end**	*parameterised rule*
$\mid$ **par** $i : Q(i, c, s)$ **do** $p(i \cup c)$ **end**	*generalised parallel composition*

Fig. 1. The language of transformation rules

addition by asking to prove that the new invariant is not contradictory. This example illustrates how the underlying Event B semantics is used to derive proof obligations for refinement patterns.

The table below lists the basic rules for the chosen subset of Event B. There are two classes of rules – for adding new elements and for removing existing ones. All the rules implicitly take an additional argument – the model being transformed. A double-character parameter name signifies that a rule accepts a set of elements, e.g., $newgrd(e, gg)$ adds all the guards from a given set gg to an event e.

$\mathbf{rule}\ newvar(vv)$	$\mathbf{rule}\ delvar(vv)$
$\mathbf{rule}\ newinv(ii)$	$\mathbf{rule}\ delinv(ii)$
$\mathbf{rule}\ newevt(ee)$	$\mathbf{rule}\ delevt(ee)$
$\mathbf{rule}\ newgrd(e, gg)$	$\mathbf{rule}\ delgrd(e, gg)$
$\mathbf{rule}\ newact(e, aa)$	$\mathbf{rule}\ delact(e, aa)$
$\mathbf{rule}\ newactexp(e, a, p)$	

To construct more complex transformations, we introduce a number of composition operators into our language. They include the sequential, $p; q$, and parallel, $p \| q$, composition constructs. In addition, there is the conditional rule construct, **if** c **then** p **end**, as well as a construct allowing to introduce additional rule parameters - **conf** $i : Q$ **do** $p(i)$ **end**. Finally, to handle rule repetitions, generalised parallel composition is introduced in the form of a loop construct: **par** $c : Q$ **do** $p(c)$ **end**. The language summary is given in Figure 1.

3.3 Constructing and Using Patterns: Examples

Below we present a couple of simple refinement patterns constructed using the proposed language.

Example 1 (New Variable). A refinement step adding a new variable can be accomplished in three steps. First, the new variable is added to the list of model variables. Second, the typing invariant is added to the model. Finally, an initialisation action is provided for the variable. The following refinement pattern adds a new variable declared to be a natural number and initalised with zero:

```
conf $v : \neg\,(v \in s.var)$ do
    $newvar(\{v\});$
    $(newinv(\{"v \in \mathbb{N}"\}, s) \,\|\, newact(\texttt{init}, \{\langle v \mid:=\mid "0"\rangle\}))$
end
```

The only pattern parameter (apart from the implicit input s) is some fresh name for the new model variable.

A pattern application example is given below. On the left-hand side there is an input model and the right-hand side there is the refined model constructed via applying the pattern ”New Variable”. Here the variable name q instantiate the parameter v.

```
SYSTEM $m0$                                    SYSTEM $m1$
VARIABLES $x$                                  VARIABLES $x, q$
INVARIANT $x \in \mathbb{Z}$                   INVARIANT $x \in \mathbb{Z} \wedge q \in \mathbb{N}$
INITIALISATION $x := 0$                        INITIALISATION $x := 0 \| q := 0$
EVENTS                                         EVENTS
  $count$ = BEGIN $x := x + 1$ END               $count$ = BEGIN $x := x + 1$ END
```

A more general (and also useful) pattern version could also accept a typing predicate and initialisation action as additional pattern parameters.

Example 2 (Action Split). In Event B, we often refine an abstract event into a choice between two or more concrete events, each of which must be a refinement of the abstract event. A simple case of such a refinement is captured by the refinement pattern below. The pattern creates a copy of an abstract event and adds a new guard and its negation to the original and new events. The guard expression is supplied as a pattern parameter.

```
conf $e, en$ : $e \in s.evt \wedge \neg\,(en \in s.evt)$ do
  $newevt(en, s)$;
  $newgrd(en, e.guard)$ $\|$
  $newact(en, e.action)$;
  conf $g$ : $g \in \mathrm{PRED} \wedge FV(g) \subseteq s.var$
    do $newgrd(e, g)$ $\|$ $newgrd(en, \neg g)$ end
end
```

The pattern configuration requires three parameters. The parameter e refers to the event to be refined from the input model s, en is some fresh event name, and g is a predicate on the model variables.

The pattern is applicable to models with at least one event. The result is a model with an additional event and a constrained guard of the original event. To exemplify pattern application, lets take the model from the previous example as an input model.

```
SYSTEM $m1$
VARIABLES $x$
INVARIANT $x \in \mathbb{Z}$
INITIALISATION $x := 0$
EVENTS
  $count$ = WHEN $x \bmod 2 = 0$ THEN $x := x + 1$ END
  $inc$   = WHEN $\neg(x \bmod 2 = 0)$ THEN $x := x + 1$ END
```

Here, the pattern parameters are instantiated as follows: e as $count$, en as inc, and x as $x \bmod 2 = 0$.

In this section we have defined refinement patterns together with the language for constructing transformations and shown small examples of pattern application. To make our approach scalable, in the next section we formally define pattern composition.

4 Pattern Composition

In the previous section we defined the notion of a basic transformation rule as a combination of the applicability conditions, transformation (effect) function, and refinement proof obligations. Moreover, In Figure 1, we also introduced various composition constructs for creating complex transformation rules. In this section we will show how to inductively define the applicability conditions, effect, and proof obligations for composed rules.

4.1 Rule Applicability Conditions

As we discussed previously, for a basic rule, the rule applicability condition is defined in its **context** clause. To define applicability conditions for more complex rules, we first introduce a function *scope*. This function returns a pair of lists, containing the model elements that the rule updates or depends on. We can compute an intersection of rule scopes: for two transformation rules it is an intersection of the elements updated by these rules and the pair-wise intersection of elements that are affected by one rule and relied upon by another.

For a complex rule (constructed using the proposed language of transformation rules), the rule applicability is derived inductively according to the following definition:

$$\begin{array}{ll}
\mathbf{app}(basic)(c,s) & = \mathbf{context}(basic)(c,s) \\
\mathbf{app}(p;q)(c,s) & = \mathbf{app}(p)(c,s) \wedge \mathbf{app}(q)(c,\mathbf{eff}(p)(c,s)) \\
\mathbf{app}(p\|q)(c,s) & = \mathbf{app}(p)(c,s) \wedge \mathbf{app}(q)(c,s) \wedge \\
 & \qquad inter(\mathbf{scope}(p),\mathbf{scope}(q)) = \oslash \\
\mathbf{app}(\mathbf{if}\ G(c,s)\ \mathbf{then}\ p\ \mathbf{end})(c,s) & = G(c,s) \Rightarrow \mathbf{app}(p)(c,s) \\
\mathbf{app}(\mathbf{conf}\ i : Q(i,c,s)\ \mathbf{do}\ p(i)\ \mathbf{end})(c,s) & = \forall\, i \cdot Q(i,c,s) \Rightarrow \mathbf{app}(p(i))(c,s) \\
\mathbf{app}(\mathbf{par}\ i : Q(i,c,s)\ \mathbf{do}\ p(i)\ \mathbf{end})(c,s) & = \forall\, i \cdot Q(i,c,s) \Rightarrow \mathbf{app}(p(i))(c,s) \wedge \\
 & \quad \forall (i,j) \cdot Q(i,c,s) \wedge Q(j,c,s) \wedge i \neq j \Rightarrow \\
 & \qquad inter(\mathbf{scope}(p(i)),\mathbf{scope}(p(j))) = \oslash
\end{array}$$

The conditions for the sequential composition, conditional and parameterised rules are quite standard. Two rules can be applied in parallel if they are working on disjoint scopes. For instance, a rule transforming an event (e.g., adding a new guard) cannot be composed with another rule transforming the same event. A similar requirement is formulated for the loop rule, since it is realised as generalised parallel composition.

4.2 Effect of Pattern Application

Once the rule applicability conditions are met, an output model can be syntactically constructed in a compositional way. For a basic rule, the effect function is directly applied to transform an input model. For more complex rules, a new model is constructed according to an inductive definition of the function **eff** given below.

$$\begin{array}{ll}
\textbf{eff}(basic)(c,s) & = \textbf{effect}(basic)(c,s) \\
\textbf{eff}(p;q)(c,s) & = \textbf{eff}(q)(c,\textbf{eff}(p)(c,s)) \\
\textbf{eff}(p\|q)(c,s) & = \textbf{eff}(q)(c,\textbf{eff}(p)(c,s)),\ \text{or} \\
 & = \textbf{eff}(p)(c,\textbf{eff}(q)(c,s)) \\
\textbf{eff}(\textbf{if}\ G(c,s)\ \textbf{then}\ p\ \textbf{end})(c,s) & = \textbf{eff}(p)(c,s),\ \text{if}\ G(c,s) \\
 & = s,\ \text{otherwise} \\
\textbf{eff}(\textbf{conf}\ i : Q(i,c,s)\ \textbf{do}\ p(i)\ \textbf{end})(c,s) & = \textbf{eff}(p(i))(c,s),\ \text{if}\ Q(i,c,s) \\
 & = s,\ \text{otherwise} \\
\textbf{eff}(\textbf{par}\ i : Q(i,c,s)\ \textbf{do}\ p(i)\ \textbf{end})(c,s) & = (\| i \in Q(i,s,c) \cdot \textbf{eff}(p(i))(c,s)), \\
 & \qquad \text{if}\ \exists(i,c,s) \cdot Q(i,c,s) \\
 & = s,\ \text{otherwise}
\end{array}$$

Not supprisingly, the result of sequential composition of two rules is computed by applying the second rule to the result produced by the first one. For the parallel composition, the result is computed similarly. However, here the order of the rule application should not affect the final result. The model resulting from an application of the loop construct is computed as a generalised parallel composition of an indexed family of transformation rules. Finally, the last three cases depend on some additional application conditions (i.e., $G(c,s)$ or $Q(i,c,s)$). If these conditions are not satisfied then the rule application leaves the input model unchanged.

The rule application procedure based on the presented definition can be easily automated. Probably the only non-trivial detail here is to provide the input values for the rule parameters. In our prototype tool implementing the ideas described in this paper, the user is requested to give the parameter values during the rule instantiation, while appropriate contextual hints and descriptions are provided by the tool.

4.3 Pattern Proof Obligations

The modest complex part of our approach is to define proof obligations needed to demonstrate that a transformation rule is actually a refinement pattern. To achieve this, in general we have to discharge all the proof obligations of individual basic rules constituting the pattern. These proof obligations cannot be discharged without considering the context produced by the neighboring rules. The following inductive definition shows how the list of proof obligations is built for a particular refinement pattern. The context information for each proof obligation is accumulated while traversing the structure of a pattern. It forms a set of additional hypotheses that later can be used in automated proofs.

$$\begin{array}{ll}
\textbf{po}(\Gamma, basic)(c,s) & = \{\Gamma \models \textbf{proof\_obligations}(basic)\} \\
\textbf{po}(\Gamma, p;q)(c,s) & = \textbf{po}(\Gamma \cup \{s' = \textbf{eff}(p;q)(c,s)\}, p(c,s')) \cup \\
 & \quad \textbf{po}(\Gamma \cup \{s' = \textbf{eff}(p;q)(c,s)\}, q(c,s')) \\
\textbf{po}(\Gamma, p\|q)(c,s) & = \textbf{po}(\Gamma, p) \cup \textbf{po}(\Gamma, q) \\
\textbf{po}(\Gamma, \textbf{if}\ G(c,s)\ \textbf{then}\ p\ \textbf{end})(c,s) & = \textbf{po}(\Gamma \cup \{G(c,s)\}, p) \\
\textbf{po}(\Gamma, \textbf{conf}\ i : Q(i,c,s)\ \textbf{do}\ p(i)\ \textbf{end})(c,s) & = \bigcup i \in Q(i,c,s) \cdot \textbf{po}(\Gamma \cup \{Q(i,c,s)\}, p(i)) \\
\textbf{po}(\Gamma, \textbf{par}\ i : Q(i,c,s)\ \textbf{do}\ p(i)\ \textbf{end})(c,s) & = \bigcup i \in Q(i,c,s) \cdot \textbf{po}(\Gamma \cup \{Q(i,c,s)\}, p(i))
\end{array}$$

Here Γ is a set of accumulated hypothesis containing pattern parameters c and the initial model s as free variables. For each basic rule, we formulate a theorem whose right-hand side is a list of the rule proof obligations and the left-hand side is a set of hypotheses containing the knowledge about the context in which the rule is applied.

4.4 Assertions

The described procedure of building a list of proof obligations for a refinement pattern aims at including all available information as a proof obligation hypothesis. This can be very complex for larger patterns, since the large number of accumulated hypotheses makes a proof obligation intractable. To circumvent this problem, in the tool implementation we allow a modeller to manually add fitting hypotheses, called assertions, that can be inferred from the context they appear in. On the one side, typically an assertion is simple enough to be discharged automatically by a theorem prover. On the other hand, it can be used to assist in demonstrating the proof obligations of the rule immediately following the assertion.

An assertion is written as **assert**$(A(c,s))$ and is delimited from the neighboring rules by semicolons. An assertion has no effect on rule instantiation and application. The following additional cases of the po definition are used to generate additional proof obligations for assertions as well as insert an asserted knowledge into the set of collected hypotheses of a refinement pattern.

$$\begin{array}{l}\mathbf{po}(\mathit{\Gamma}, p; \mathbf{assert}(A(c,s)))(c,s) = \mathit{\Gamma} \cup \{s' = \mathbf{eff}(p)(c,s)\} \models A(c,s') \\ \mathbf{po}(\mathit{\Gamma}, \mathbf{assert}(A(c,s)); p)(c,s) = \mathbf{po}(\mathit{\Gamma} \cup \{A(c,s)\}, p)(c,s)\end{array}$$

Pattern composition enables construction of large refinement patterns. We believe that a promising area of pattern application is in product-line developments. Indeed, a product-line development significantly relies on reuse of certain design solutions. Refinement patterns can be collected and composed to formally define these solutions. By discharging proof obligations for a general pattern representation, we enhance the reuse not only on the modelling but also on verification level.

To demonstrate scalability of our approach, next we demonstrate how to construct a rather complex refinement pattern allowing to introduce a well-known fault tolerance mechanism into a model.

5 Towards Refinement Automation: Case Study and Tool Support

5.1 Case Study: Triple Modular Redundancy Pattern

Triple Modular Redundancy (TMR) is a fault-tolerance mechanism in which the results of three similar components are processed by a voting element to produce a single output [12]. The purpose of the mechanism is to mask a single component failure. In this section we will demonstrate how a refinement step that introduces the TMR arrangement into a model can be generalized as a refinement pattern.

Our initial specification should have a variable representing the output of a component for which TMR will be introduced. Moreover, it should have an event that models the behaviour of a component by non-deterministically updating this variable. Non-determinism is used here to model unpredictable (possibly faulty) results produced by the component. We do not make any assumptions about the variable type. Furthermore, the event can contain some additional actions on other variables. Finally, our initial model should also contain a special event handling a failure (abort) of the component.

In the refined model, we replace the single abstract component with three similar components. The outputs of the new components are modelled by fresh variables. The variable types and initialisation of these variables are simply copied from their abstract counterpart in the initial specification.

The TMR pattern we define uses a number of configuration parameters. The parameter s identifies a variable modelling the output of a component; u is an event updating the variable s (in addition to possible update of other variables); zz is an event handling a failure of the component modelled by u; finally, a is an action from u updating variable s.

Also, as a result of pattern application, the new variables ph, s_i and r_i are introduced into the refined model. The variable ph keeps track of the current phase in the TMR implementation – reading from the new components, voting on them, or delivering the final result; the variables s_i, $i = 1..3$, are used to record the outputs from the three new components introduced by the pattern; finally, the flags r_i reflect availability of new outputs in the respective output variables s_i.

```
**conf** $s, u, zz, a$ :
  $s \in var \wedge u \in evt \wedge zz \in evt \wedge u \neq zz \wedge$
  $a \in u.actions \wedge a.style \neq (:=) \wedge \{s\} = a.var$
**do**
  **conf** $ph, s_1, s_2, s_3, r_1, r_2, r_3$ :
    $\{s_1, s_2, s_3, r_1, r_2, r_3, ph\} \subseteq (\mathrm{VAR} - var) \wedge$
    $part(\{\{s_1\}, \{s_2\}, \{s_3\}, \{r_1\}, \{r_2\}, \{r_3\}, \{ph\}\})$
  **do**
    $variables;\ events;\ voter;\ abort;\ invariant$
  **end**
**end**
```

The pattern is made of four major parts: the rules declaring the types and initialisation of new models variables; the definition of new events; the refinement rules for transforming a single abstract event representing the functioning of a sole component into the voter event; and, finally, the addition of an invariant characterising the behaviour of a TMR block. The condition using the operator $part$ simply states that its arguments are disjoint sets.

```
$variables \overset{\mathrm{df}}{=}$
  $(newinv(\text{"}ph \in BOOL\text{"}); newini(\langle ph \mid:=\mid \text{"}FALSE\text{"}\rangle)) \parallel$
  $(newinv(\text{"}s_1 \in s.type\text{"}); newini(\langle s_1 \mid init(s).style \mid init(s).expr\rangle))\parallel$
  $(newinv(\text{"}r_1 \in BOOL\text{"}); newini(\langle r_1 \mid:=\mid \text{"}FALSE\text{"}\rangle))$
  ...
```

Each new variable definition should come with a typing invariant and an initialisation action. These are normally grouped together so that the related proof obligation rules would work with a smaller context. In the above, . . . stand for the omitted rules defining the types and initialisation for the variables s_2, s_3 and r_2, r_3. The shortcut notation $newini(a)$ used in the pattern description stands for declaration of the initialisation action: $newini(a) \overset{\mathrm{df}}{=} newact(\mathtt{init}, a)$. The shortcut $init(v)$ refers to an action of the initialisation event assigning to the variable v.

The refined model constructed by the pattern would contain three copies of a component modelled in the abstract model. As we have made an assumption that a component is represented by a single event, the component copies are modelled by adding three new events into the refined model. A component copy has the guard of an abstract component, conjuncted with an an additional condition ensuring that it is executed before passing control to a voter, and an action that is the copy of the selected action of an abstract component (the pattern parameter a) except for saving the result into s_i (for the copy i) instead. In addition, a component copy also assigns to r_i to indicate the availability of result in s_i.

```
$\mathit{events} \overset{\text{df}}{=}$
**conf** $u_1, u_2, u_3$ :
    $\{u_1, u_2, u_3\} \subset \text{EVENT} \setminus s.evt \wedge part(\{\{u_1\}, \{u_2\}, \{u_3\}\})$
**do**
    $\mathit{copy_1} \parallel \mathit{copy_2} \parallel \mathit{copy_3}$
**end**
```

The above creates three component copies, each constructed according to the following rule.

```
$\mathit{copy_1} \overset{\text{df}}{=}$
  $newevt(\langle u_1 \mid - \mid \{\text{"}r_1 = \mathit{FALSE}\text{"}\} \cup u.guards \mid$
      $\langle s_1 \mid a.style \mid a.expression \rangle, \langle r_1 \mid:=\mid \text{"}\mathit{TRUE}\text{"}\rangle, \langle ph \mid:=\mid \text{"}\mathit{FALSE}\text{"}\rangle\rangle$
...
```

The rule $\langle s_1 \mid a.style \mid a.expression \rangle$ above constructs an action from the abstract action a in such a way that it would have the same effect but update a new variable s_1. Since $a.style$ is one of non-deterministic substitution styles (see the top-level rule above), a further refinement steps could diversify component specification.

The voter event is simply a refined version of the event modelling the abstract component. Whereas the abstracted version was computing results itself, its refined counterpart votes on the results of component copies. The voter is enabled once all the components have produced a result (which is ensured by the first guard in the rule below). The final result is computed according to a simple majority voting protocol. The event parameter rr is set to the voting outcome in the second guard.

```
$\mathit{voter} \overset{\text{df}}{=}$
  $newpar(u, \text{"}rr\text{"});$
  $newgrd(u, \text{"}r_1 = \mathit{TRUE} \wedge r_2 = \mathit{TRUE} \wedge r_3 = \mathit{TRUE}\text{"});$
  $newgrd(u, \text{"}(s_1 = s_2 \vee s_1 = s_3 \wedge rr = s_1) \vee (s_2 = s_1 \vee s_2 = s_3 \wedge rr = s_2\text{"});$
  $(delact(u, a); newact(u, \langle s \mid:=\mid \text{"}rr\text{"}\rangle));$
  $(newact(u, \langle r_1 \mid:=\mid \text{"}\mathit{FALSE}\text{"}\rangle) \parallel$
    $newact(u, \langle r_2 \mid:=\mid \text{"}\mathit{FALSE}\text{"}\rangle) \parallel$
    $newact(u, \langle r_3 \mid:=\mid \text{"}\mathit{FALSE}\text{"}\rangle));$
  $newact(u, \langle ph \mid:=\mid \text{"}\mathit{TRUE}\text{"}\rangle)$
```

The abstract action a of the component is removed, replaced by a deterministic assignment (to the same variable s) of the result of the winning component. The flags r_i and ph are reset in the preparation for a next iteration.

In case all the component copies disagree, no final result may be computed. This corresponds to an *abort* event of the abstract specification. The refined model simply constraints the guard of the event so it only gets enabled in the situations when the voting has failed.

```
$\mathit{abort} \overset{\text{df}}{=}$
  $newgrd(zz, \text{"}r_1 = \mathit{TRUE} \wedge r_2 = \mathit{TRUE} \wedge r_3 = \mathit{TRUE}\text{"});$
  $newgrd(zz, \text{"}s_1 \neq s_2 \wedge s_2 \neq s_3 \wedge s_1 \neq s_3\text{"});$
```

A new invariant is added to the refined model to characterise the state of a refined system after the voting is completed. It summarises the cases when the majority voting on component results succeeds.

```
$\mathit{invariants} \overset{\text{df}}{=}$
  $newinv(\text{"}ph = \mathit{TRUE} \wedge (s_1 = s_2 \vee s_2 = s_3)) \Rightarrow s = s_1\text{"});$
  $newinv(\text{"}ph = \mathit{TRUE} \wedge s_2 = s_3) \Rightarrow s = s_2\text{"})$
```

Application of the pattern to a simple abstract model (two events, two variables) saves a user from analysing 14 proof obligations, three of which would have to be done manually in an interactive theorem prover. For input larger model or a more elaborate pattern, the benefits are even greater.

5.2 Tool for Refinement Automation

A proof of concept implementation of the pattern tool for Event B has been implemented as a plug-in to the RODIN Platform [1]. The plug-in seamlessly integrates with the RODIN Platform interface so that a user does not have to switch between different tools and environments while applying patterns in an Event B development. The input and output models of the tool are fully semantically and syntactically compatible with their representation in the RODIN Platform. It allows a developer to interleave automated and manual refinement steps and proofs.

The core of the tool is the pattern instantiation engine. The engine uses an Event B input model and a pattern, from the pattern library, to produce a model refinement.

The process of a pattern instantiation is controlled by the pattern instantiation wizard. The wizard is an interactive tool which inputs pattern configuration from a user. It validates user input and provides hints on selecting configuration values. Pattern configuration is constructed in a succession of steps: the values entered at a previous step influence the restrictions imposed on the values of a current step configuration.

The result of a successful pattern instantiation is a new model and, possibly, a set of instantiation proof obligations - additional conditions that must be verified every time when a pattern is applied. The output model is added to a current development as a refinement of the input model.

The tool is equipped with a pattern editor and the pattern library. Patterns in the library are organised in a catalogue tree, according to the categories stated in pattern specifications. A user can browse through the library catalogue using a graphical dialogue. This dialogue is used to select a pattern for instantiation or editing.

The current version of the tool is freely available from our web site [10]. Several patterns developed with this tool were applied during formal modelling of the Ambient Campus case study of the RODIN Project [11].

6 Conclusions

In this paper we proposed an approach for automation of refinement process in Event B. We introduced the notion of refinement patterns – model transformers that generically represent typical refinement steps. Refinement patterns allow us to replace a process of devising a refined model and discharging proof obligations by a process of pattern instantiation. While instantiating refinement patterns, we reuse not only models but also proofs. All together, this establishes a basis for automation of formal development. In this paper we also described a prototype tool allowing us to automate refinement steps in Event B.

Our work was inspired by several works on automation of refinement process. The Refinement Calculator tool [6] has been developed to support program development using the Refinement Calculus theory by R.Back and J. von Wright. [4] The theory was formalised in the HOL theorem prover, while specific refinement rules were proved as HOL theorems. The HOL Window Inference library[9] has been used to facilitate transformational reasoning. The library allows us to focus on a particular part of a model and transform it, while guaranteeing that the transformation, if applicable, will produce a valid refinement of the entire model.

A similar framework consisting of refinement rules (called tactics) and the tool support for their application has been developed by Oliveira, Cavalcanti, and Woodcock [14]. The framework (called ArcAngel) provides support for the C.Morgan's version of the Refinement Calculus. The obvious disadvantage of both these frameworks is that the refinement rules that can be applied usually describe small, localised transformations. An attempt to perform several transformations on independent parts of the model at once, would require deriving and discharging additional proof obligations about the context surrounding transformed parts, that are rather hard to generalise. However, while implementing our tool, we found the idea of using the transformational approach for model refinement very useful.

Probably the closest to our tool is the proprietary domain-specific automatic refiner tool created by Siemens/Matra [5]. The tool automatically produces an implementable model in B0 language (a variant of implementable B) by applying the predefined rewrite rules. A large library of such rules has been created specifically to handle the specifications of train systems. The use of this proprietary tool resulted in significant growth of developer productivity. Our work aims at creating a similar tool yet publicly available and domain-independent.

Obviously the idea to use refinement patterns to facilitate the refinement process was inspired by the famous collection of software design patterns [8]. However in our approach the patterns are not only descriptions of the best engineering practice but rather ”active” model transformers that allow a designer to refine the model by reusing and instantiating the generic prefabricated solutions.

As a future work we are planning to further explore the theoretical aspects of proof reuse in the proposed approach as well as extend the existing collection of patterns. Obviously, this work will go hand-in-hand with the tool development. We believe that by building a sufficiently large library of patterns and providing designers with an automatic tool supporting refinement process, we will facilitate better acceptance of formal methods in practice.

Acknowledgements

This work is supported by IST FP7 DEPLOY project.

References

1. RODIN Event-B Platform. `http://rodin-b-sharp.sourceforge.net/`, 2007.
2. J. R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.
3. J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
4. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
5. L. Burdy and J.-M. Meynadier. Automatic Refinement. *Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, FM'99*, 1999.
6. M. Butler, J. Grundy, T. Løangbacka, R. Rukšenas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. *Proc. of Formal Methods Pacific*, 1997.

7. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2, 1995.
9. J. Grundy. Transformational Hierarchical Reasoning. *The Computer Journal*, 39(4):291–302, 1996.
10. A. Iliasov. Finer Plugin. `http://finer.iliasov.org`, 2008.
11. Alexei Iliasov, Alexander Romanovsky, Budi Arief, Linas Laibinis, and Elena Troubitsyna. On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 141–145, Washington, DC, USA, 2007. IEEE Computer Society.
12. R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal*, pages 200–209, April 1962.
13. Carroll Morgan. *Programming From Specifications*. Prentice Hall International (UK) Ltd., 1994.
14. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Arcangel: a tactic language for refinement. *Formal Asp. Comput.*, 15(1):28–47, 2003.