COMPUTING SCIENCE

Patterns for Refinement Automation

A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky.

TECHNICAL REPORT SERIES

No. CS-TR-1125 October, 2008

TECHNICAL REPORT SERIES

No. CS-TR-1125 October, 2008

Patterns for Refinement Automation

Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, and Alexander Romanovsky

Abstract

Formal modelling is indispensable for engineering highly dependable systems. However, a wider acceptance of formal methods is hindered by their in- sufficient 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.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

ILIASOV, A., TROUBITSYNA, E., LAIBINIS, L., ROMANOVSKY, A..

Patterns for Refinement Automation
[By] A. Iliasov, E. Troubitsyna, L. Labinis, A. Romanovsky..

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2008.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1125)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1125

Abstract

Formal modelling is indispensable for engineering highly dependable systems. However, a wider acceptance of formal methods is hindered by their in- sufficient 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.

About the author

Alexei Ilasov is a Research Associate within the School of Computing Science, Newcastle University.

Alexander Romanovsky is a Research Professor. He has been involved in a number of ESPRIT, FP and EPSRC/UK projects on system dependability within which a wide range of general fault tolerance mechanisms and architectures have been developed (DSoS, PDCS, DeVa, CaberNet, MAFTIA, ReSIST, DISCS, RODIN). He has been a co-investigator of two EPSRC/UK projects (DOTS and TrAmS). Now he is coordinating a major FP7 Integrated Project DEPLOY aiming to make major advances in engineering methods for dependable systems through the deployment of formal engineering methods in 5 sectors of European industry. His main interests are in fault tolerance, rigorous design of resilient systems, software architectures, exception handing, mobile agents and service oriented architectures.

Suggested keywords

REFINEMENT,
FORMA SYSTEM DEVELOPMENT,
REUSE,
TOOLS,
EVENT-B,
RODIN

Patterns for Refinement Automation

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

Over the recent years model-driven development has became a leading paradigm in software engineering. 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 mathematical proofs.

The refinement approach significantly reduces the required testing efforts and, at the same time, supports a 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, reducing execution of a refinement step to 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.

2 Towards Refinement Automation

2.1 Formal Development by Refinement

System development by refinement is a formal counterpart of model-driven development process. Refinement allows us to ensure that the refined, i.e., more elaborated, model retains all the essential properties of its abstract counterpart. Since refinement is transitive, the model-driven refinement-based development process enables development of systems correct by construction.

The precise definition of refinement depends on the chosen modelling framework and hence might have different semantics and the degree of rigor. The foundations of formal reasoning about correctness and stepwise development by refinement were established by Dijkstra [6] and Hoare [9], and then further developed by R.Back[2], C.Morgan [13], and J. von Wright [3].

In the refinement calculus framework, a model is represented by a composition of abstract statements. Formally, we say that statement *S* is refined by statement S', written $S \sqsubseteq S'$, if, whenever *S* establishes a certain postcondition, so does S' [6]. In general, refinement process can be seen as a way to reduce non-determinism of the abstract model, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

To facilitate the refinement process, the typical refinement transformations have been generalized into a set of refinement rules [3, 13]. These rules can be seen as generic templates (or patterns) that define the general form of the statement to be transformed, the resultant statement, and the proof obligations that should be discharged to verify refinement for that particular transformation. However, a refinement rule usually describes a small localized transformation of a certain model part. Obviously, the tools developed to automate application of such refinement rules [5, 14] lack scalability.

On the other hand, such frameworks as Z, VDM, Event B support the formal development by entire model transformation. For instance, the RODIN platform – a tool support for refinement in Event B allows us to perform refinement by introducing many changes at once and verify by proofs that these changes 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. It remains unclear, though, if we can employ transformational approach to automate execution of these transformations, i.e., reuse the models and proofs constructed previously.

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. A refinement pattern in general is a model transformer. Unlike design patterns [7], a refinement pattern is ”dynamic” in a sense that it takes a model as an input and produces a new model as an output. 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.

Although the notion of a refinement pattern is independent of the chosen modelling framework, for clarity we explain our idea of refinement patterns and describe a prototype tool that implements them for the Event B formalism. Next we briefly introduce Event B and give semantic and syntactic views on its models.

2.2 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 the state. A simple abstract machine has the following general form:

SYSTEM AM
VARIABLES v
INVARIANT I
INITIALISATION *INIT*
EVENTS
E_1
$\ldots$
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 **EVENTS** clause. An event is defined as follows:

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

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

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
...
E_N

In AM' we replace the abstract data structures of *AM* with the concrete ones. The invariant of AM' – I' – defines now not only the invariant properties of the refined model, but also the connection between the newly introduced variables and the abstract variables that they replace. 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.,

$$\begin{array}{l} wp(INIT', \neg wp(INIT, \neg I')) \;=\; true, \\ I \wedge I' \wedge g'_i \;\Rightarrow\; g_i \wedge wp(S', \neg wp(S, \neg InvC)), \quad \text{and} \\ I \wedge I' \wedge g_i \;\Rightarrow\; \bigvee_i^N g'_i \end{array}$$

2.3 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:

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

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, where an action style denotes one of the assignment types : i.e., $\mathrm{STYLE} = \{:=, :\in, :|\}$.

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 is represented in our notation:

SYSTEM $m0$
VARIABLES x
INVARIANT $x \in \mathbb{Z}$
INITIALISATION $x := 0$
EVENTS
$count$ = **BEGIN** $x := x + 1$ **END**

$$\langle \langle x \rangle \mid \langle \text{"}x \in \mathbb{Z}\text{"} \rangle \mid \langle \langle \texttt{init} \mid - \mid - \mid \langle x \mid :=\mid \text{"}0\text{"} \rangle \rangle, \langle \texttt{count} \mid - \mid - \mid \langle x \mid :=\mid \text{"}x + 1\text{"} \rangle \rangle \rangle \rangle$$

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 in general and Event B in particular.

3 Refinement Patterns

3.1 Definitions

Definition 1. *Let S be a set of all well-formed models defined according to a syntax of a chosen modelling language. Then a transformation rule T is a function computing a new model for a given input model:*

$$T : S \times C \nrightarrow S$$

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 $P : S \times C \nrightarrow S$ 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.

In this paper we use the Event-B modelling method to give concrete meanings to the concepts of models S, configurations C, and a refinement relation $\sqsubseteq$. This allows us to reuse much of the Event-B proof theory when demonstrating that a transformation rule is indeed a refinement pattern.

3.2 The Language of Transformations

We propose a special language to construct transformation rules. The proposed language contains basic transformation rules as well as the constructs allowing to compose complex rules from 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 becomes 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 given configuration c is syntactically transformed into the output model calculated as 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 specific proof obligations on model elements to be verified.

A basic rule has the following general form:

$$
\begin{array}{l}
\textbf{rule}\ name(c) \\
\quad \textbf{context}\ Q(c, s) \\
\quad \textbf{effect}\ E(c, s) \\
\quad \textbf{proof obligation}\ PO_1(c, s) \\
\quad \ldots \\
\quad \textbf{proof obligation}\ PO_n(c, s)
\end{array}
$$

Here $name$ and c are correspondingly the rule name and list of its parameters. 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 a current model s and 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, effect computation function, and collection of proof obligations of a rule r.

As an example, let us consider two primitive rules for the Event-B method. The first transformation adds one or more new variables:

$$
\begin{array}{l}
\textbf{rule}\ newvar(vv) \\
\quad \textbf{context}\ vv \cap s.var = \varnothing \\
\quad \textbf{effect}\ \langle s.var \cup vv \mid s.inv \mid s.evt \rangle \\
\quad \textbf{proof\_obligation}\ \forall v \in vv \cdot (\exists a \cdot a \in s.\mathtt{init}.action \wedge v \in a.var)
\end{array}
$$

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.

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

$$
\begin{array}{lll}
p(c) = basic(c) & & \textit{primitive rule} \\
\quad \mid p; q & & \textit{sequential composition} \\
\quad \mid p \| q & & \textit{parallel composition} \\
\quad \mid \textbf{if } Q(c,s) \textbf{ then } p \textbf{ end} & & \textit{conditional rule} \\
\quad \mid \textbf{conf } i : Q(i,c,s) \textbf{ do } p(i \cup c) \textbf{ end} & & \textit{parameterised rule} \\
\quad \mid \textbf{par } i : Q(i,c,s) \textbf{ do } p(i \cup c) \textbf{ end} & & \textit{generalised parallel composition}
\end{array}
$$

Fig. 1. The language of transformation rules

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

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 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 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.

rule $newvar(vv)$	**rule** $delvar(vv)$
rule $newinv(ii)$	**rule** $delinv(ii)$
rule $newevt(ee)$	**rule** $delevt(ee)$
rule $newgrd(e, gg)$	**rule** $delgrd(e, gg)$
rule $newact(e, aa)$	**rule** $delact(e, aa)$
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 Examples

In this section we present a couple of simple examples of 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**
  $newvar(\{v\})$;
  $(newinv(\{"v \in \mathbb{N}"\}, s) \parallel newact(\texttt{init}, \{\langle v \mid:=\mid "0"\rangle\}))$
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. The left-hand side model is an input model and the righ-hand side is the refined version constructed by the pattern. The example assumes that variable name q for chosen for 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 accept a typing predicate and initialisation action as additional pattern parameters.

Example 2 (Action Split). In Event B, an abstract event may be refined 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 refinement is implemented 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**
  $newevt(en, s)$;
  $newgrd(en, e.guard) \parallel$
  $newact(en, e.action)$;
  conf $g \ : g \in \mathrm{PRED} \wedge FV(g) \subseteq s.var$
    do $newgrd(e, g) \parallel newgrd(en, \neg g)$ **end**
end

The pattern configuration requires three parameters. 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. As an input model we use the model from the previous example.

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 $x \bmod 2 = 0$.

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 we can inductively define the applicability conditions, effect, and proof obligations for composed rules.

4.1 Rule Applicability Conditions

For a basic rule, the rule applicability condition is defined in its **context** clause. For more complex rules constructed using the proposed language of transformation rules, rule applicability is derived inductively according to the following definition:

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

The consistency requirements 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.

The rule scopes are calculated by using the predefined function $\mathbf{scope}$, which returns a pair of lists, containing the model elements that the rule updates or depends on. Intersection of rule scopes is computed as an intersection of the elements updated by the transformations and the pair-wise intersection of elements updated by one rule and depended on by another:

$$inter((r_1, w_1), (r_2, w_2)) = (w_1 \cap w_2) \cup (r_1 \cap w_2) \cup (r_2 \cap w_1)$$

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.

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

As expected, the result of sequential composition of two rules is computed by applying the second rule to the result of the first rule. For parallel composition, the result is computed in the same manner but the order of the rules should not affect the overall result. The resulting model of the loop construct is computed as generalised parallel composition of an indexed family of transformation rules. 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 true, rule application leaves the input model unchanged.

The rule application procedure based on the presented definition can be easily automated. The only interesting detail is in providing input values for the rule parameters. In our tool implementation for the Event-B method, briefly covered later, the user is requested to provide the parameter values during rule instantiation, while appropriate contextual hints and descriptions are provided by the tool.

4.3 Pattern Proof Obligations

To demonstrate that a rule is a refinement pattern, we have to discharge all the prooof obligations of individual basic rules occuring in the rule body. These proof obligations cannot be discharged without considering the context produced by the neighbour 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, as a set of additional hypotheses that can be then 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 tries to include every possible fact as a proof obligation hypothesis. This can be a problem for larger patterns

as the size of a list of accumulated hypotheses makes a proof obligation intractable. To rectify the problem, we allow a modeller to manually add fitting hypotheses, called assertions, that can be inferred from the context they appear in. An assertion would be typically simple enough to be discharged automatically by a theorem prover. At the same time, 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.

$$\textbf{po}(\Gamma, p; \textbf{assert}(A(c,s)))(c,s) = \Gamma \cup \{s' = \textbf{eff}(p)(c,s)\} \models A(c,s')$$
$$\textbf{po}(\Gamma, \textbf{assert}(A(c,s)); p)(c,s) = \textbf{po}(\Gamma \cup \{A(c,s)\}, p)(c,s)$$

5 Triple Modular Redundancy Pattern

Triple Modular Redundancy (TMR) is a fault-tolerance mechanism in which three components produce in parallel results that are processed by a voting element [12]. The mechanism masks a single component failure. In this paper we demonstrate that a refinement step that introduces TMR arrangement into the model can be generalized as a refinement pattern as shown below.

Our initial specification should have a variable representing a component for which TMR is introduced. Moreover, it should have an event that non-deterministically updates this variable. Non-determinism is used to model faulty and hence unpredictable results produced by the component. We do not make any assumptions about the variable type. Furthermore, the event can contain some other actions in addition to updating the variable modelling component.

In the refined model we replace the single abstract component with three components. The new components are modelled by fresh variables. The variable types and initialisation are simply copied from the variable modelling abstract component.

The pattern uses a number of configuration parameters. The parameters s selects a variable modelling the component; u is an event updating the variable; a is an action from u updating variable s (u is allowed to contain actions assigning to other variables). ph, fl, s_i and r_i are the new variables defined by the pattern. The variable ph keeps track of the current phase in the TMR implementation; fl is a flag indicating a failure to get a majority vote; variables s_i, $i = 1..3$, record the output from the three new components introduced by the pattern; flag r_i indicates the availability of a result from component s_i.

conf $s, u, a, ph, fl, s_1, s_2, s_3, r_1, r_2, r_3$:
$s \in s.var \wedge u \in s.evt \wedge a \in u.actions \wedge a.style \neq (:=) \wedge \{s\} = a.var \wedge$
$\{s_1, s_2, s_3, r_1, r_2, r_3, ph, fl\} \subseteq (\mathrm{VAR} - var) \wedge$
$part(\{\{s_1\}, \{s_2\}, \{s_3\}, \{r_1\}, \{r_2\}, \{r_3\}, \{ph\}, \{fl\}\})$
do
$vardefs;\ evtdefs;\ evtrefine;\ invariants$
end

$vardefs \stackrel{\text{df}}{=}$
$block_1 \parallel block_2 \parallel block_3 \parallel$
$(newinv(\text{"}ph \in BOOL\text{"}); newini(\langle ph \mid := \mid \text{"}FALSE\text{"}\rangle)) \parallel$
$(newinv(\text{"}fl \in BOOL\text{"}); newini(\langle fl \mid := \mid \text{"}FALSE\text{"}\rangle))$
$block_1 \stackrel{\text{df}}{=}$
$(newinv(\text{"}s_1 \in s.type\text{"}); newini(\langle s_1 \mid init(s).style \mid init(s).expr\rangle)) \parallel$
$(newinv(\text{"}r_1 \in BOOL\text{"}); newini(\langle r_1 \mid := \mid \text{"}FALSE\text{"}\rangle))$
...

$eventdefs \stackrel{\text{df}}{=}$
conf u_1, u_2, u_3 :
$\{u_1, u_2, u_3\} \subset \mathrm{EVENT} \setminus s.evt \wedge part(\{\{u_1\}, \{u_2\}, \{u_3\}\})$
do $\parallel$
$copy_1 \parallel copy_2 \parallel copy_3$
end
$newevt(\langle alt \mid - \mid \text{"}s_2 = s_3\text{"} \mid \langle s \mid := \mid \text{"}s_2\text{"}\rangle\rangle) \parallel$
$newevt(\langle fail \mid - \mid \text{"}s_1 \neq s_2 \wedge s_2 \neq s_3 \wedge s_1 \neq s_3\text{"} \mid \langle fl \mid := \mid \text{"}TRUE\text{"}\rangle\rangle)$

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

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

$invariants \stackrel{\text{df}}{=}$
$newinv(\text{"}ph = TRUE \wedge (s_1 = s_2 \vee s_2 = s_3)) \Rightarrow s = s_1\text{"});$
$newinv(\text{"}ph = TRUE \wedge s_2 = s_3) \Rightarrow s = s_2\text{"});$
$newinv(\text{"}ph = TRUE \wedge s_1 \neq s_2 \wedge s_2 \neq s_3 \wedge s_1 \neq s_3) \Rightarrow fl = TRUE\text{"})$

The shortcut notation $newini(a)$ used in the pattern source stands for declaration of the initialisation action: $newini(a) \stackrel{\text{df}}{=} newact(\texttt{init}, a)$. The shortcut $init(v)$ refers to an action of the initialisation event assigning to a variable v. The predicate $part$, used in $eventdefs$, requires that its argument is a set of disjoint subsets.

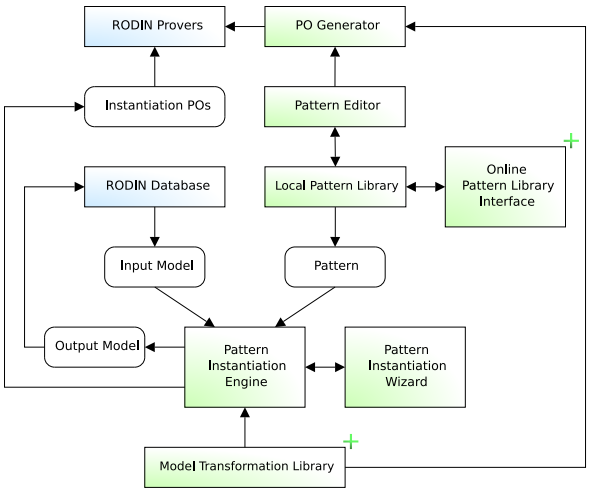

Fig. 2. The Event-B refinement patterns tool architecture.

6 Tool for Refinement Automation

A proof of concept implementation of the pattern tool for the Event B method has been realised 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 plug-in relies on two major RODIN Platform components: the Platform database, which stores models, proof obligations and proofs constituting a development; and the prover which is a collection of automated theorem provers supplemented by the interactive prover.

The overall tool architecture is presented in Figure 2. The core of the tool is the *pattern instantiation engine*. The engine uses an input model, imported from the Platform database, and a pattern, from the pattern library, to produce a model refinement. The engine implements only the core pattern language: the sequential and parallel composition, and *forall* construct. The method-specific model transformations (in this case, Event-B model transformations) are imported from the *model transformation library*.

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 and is saved in the Platform database. The instantiation proof obligations are saved in an Event B *context* file. The RODIN platform builder automatically validates and passes them to the Platform prover.

The tool is equipped with a *pattern editor*. The current version (0.1.7)[10] uses the XML notation and an XML editor to construct patterns. The next release is expected to employ a more user-friendly visual editor. The available refinement patterns are stored in the *local 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.

When constructing a pattern, a user may wish to generate the set of pattern correctness proof obligations. Proof obligations are constructed by the proof obligation generator component. The component combines a pattern declaration and the definitions of the used model transformations to generate a complete list of proof obligations, based on the rules given in Section 4.3. The result is a new context file populated with theorems corresponding to the pattern proof obligations. The standard Platform facilities are used to analyse and discharge the theorems.

We believe it is important to facilitate pattern exchange and thus the tool includes a component for interfacing with an on-line pattern library. The on-line pattern library and the model transformation library are the two main extension points of the tool. The pattern specification language can be extended by adding custom model transformations to the library of model transformation; addition of a model transformation should not affect the pattern instantiation engine and the proof obligation generator.

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].

7 Conclusions

In this paper we proposed a theoretical basis for automation of refinement process. 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. In this paper we also demonstrated how to define refinement patterns for the Event B formalism and 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 [5] has been developed to support program development using the Refinement Calculus theory by R.Back and J. von Wright. [3] The theory was formalised in the HOL theorem prover, while specific refinement rules were proved as HOL theorems. The HOL Window Inference library[8] has been used to to facilitate transformational reasoning. The library allows us to focus on and transform a particular part of a model, 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 automatic refiner tool created by Siemens/Matra [4]. 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 [7]. However in our approach the patterns are not just 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 the proposed language of refinement patterns 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 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. R. Back. On correct refinement of programs. *Journal of Computer and Systems Sciences*, 23(1):49–68, 1981.
3. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
4. L. Burdy and J.-M. Meynadier. Automatic Refinement. *Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, FM'99*, 1999.
5. 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.
6. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2, 1995.
8. J. Grundy. Transformational Hierarchical Reasoning. *The Computer Journal*, 39(4):291–302, 1996.
9. C. A. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.
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.