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 verifying correctness preservation. This work establishes a basis for building a tool that would support 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 a specification closely resembling an implementation. Each model transformation step, called a *refinement* step, allows a designer to incorporate implementation details into the model. Correctness of refinement steps 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 to be eligible for a refinement pattern application. 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 can potentially support 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 might 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[13]. 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 model-driven development process. Refinement allows us to ensure that a 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 [9] and Hoare [12], and then further developed by R.Back and J. von Wright [5] as well as C.Morgan [16].

In the refinement calculus framework, a model is represented by a composition of abstract statements. Formally, we say that the statement S is refined by the statement S', written $S \sqsubseteq S'$, if, whenever S establishes a certain postcondition, so does S' [9]. Since statement composition is monotonic with respect to the refinement relation, refinement of a model statement is also refinement of the whole model. In general, the 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.

There have been several attempts to facilitate the refinement process, by generalizing the typical refinement transformations into a set of refinement rules [5, 16]. 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 [8, 17] 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 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 by reusing the models and proofs that were constructed previously.

In this paper we propose to tackle this problem via definition and use of refinement patterns. Our definition of refinement patterns builds on the idea of refinement rules. In general, a refinement pattern is a model transformer. Unlike design patterns [10], a refinement pattern is ”dynamic” in a sense that the process of pattern application takes a model as an input and produces a new model as an output.

To formalize and automate the process of pattern application, we define a pattern as a model transformer consisting 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.

In principle, refinement patterns can be defined for any refinement-based modelling frameworks. In this paper we present our proposal for refinement patterns in the Event B formalism and also describe a prototype tool that implements them. We start by briefly introducing the Event B language and giving semantic and syntactic views on its models.

2.2 Event B

In this section we introduce our formal framework – the B Method [2]. It is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications. Recently the B method has been extended by the Event B framework [3], which enables modelling of event-based (reactive) systems. In fact, this extension has incorporated the action system formalism [6, 4] into the B Method.

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:

MACHINE 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 a 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 the predicates over the machine variables *v*, and the action *S* is an assignment to state variables. For simplicity, in this paper we do not consider Event B events with parameters or local 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 variables or a non-deterministic assignment from a given set or according to a given postcondition. The semantics of actions is defined as a before-after (BA) predicate as follows:

Action	Before-after predicate
$x := E(x,y)$	$x' = E(x,y) \ \wedge \ y' = y$
$x :\in Set$	$\exists\, t.\ (t \in Set \wedge x' = t) \ \wedge \ y' = y$
$x :\mid P(x,y,x')$	$\exists\, t.\ (P(x,t,y) \wedge x' = t) \ \wedge \ y' = y$

where x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after state.

Event B adopts interleaving semantics while treating parallelism. 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.

To check consistency of Event B machine, we should verify two types of properties: event feasibility and invariant preservation. Intuitively, event feasibility means that execution of an event from any state where both the machine invariant and the event guard hold is possible, i.e., it can produce at least one after state that satisfies the before-after predicate, i.e.,

$$I(v) \wedge G_e(v) \Rightarrow \exists\, v'.\, BA_e(v,v')$$

The invariant preservation property simply states that invariant should be maintained:

$$I(v) \wedge G_e(v) \wedge BA_e(v,v') \Rightarrow I(v')$$

The main development methodology of Event B is refinement – the process of transforming an abstract specification while preserving its correctness and gradually introducing implementation details. Let us assume that the refinement machine AM' is a result of refinement of the abstract machine *AM*:

MACHINE AM'
VARIABLES w
INVARIANT I'
INITIALISATION *INIT'*
EVENTS
  E_1
  $\ldots$
  E_M

In AM' we replace the abstract variables of *AM* (v) with the concrete ones (w). 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 establish two facts – feasibility of refined events and their correctness with respect to the abstract events. To demonstrate feasibility, we should prove the following:

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \Rightarrow \exists\, w'.\, BA'_e(w,w')$$

where $G'(w)$ is the guard of the refined event and $BA'(w,w')$ its before-after predicate.

To demonstrate that each event is a correct refinement of its abstract counterpart, we should first prove that the guard is strengthened in the refinement:

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \Rightarrow G_e(v)$$

Finally, we need to demonstrate a correspondence between the abstract and concrete postconditions:

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \wedge BA'_e(w,w') \Rightarrow \exists\, v'.\, (BA_e(v,v') \wedge I'(v',w'))$$

The refined model can also introduce new events. In this case, we have show that these new events are refinements of implicit empty (skip) events of the abstract model.

While presenting Event B above, we have slightly simplified matters by omitting the fact that Event B model consists of two separate parts. The static part, called *c*ontext, contains the declaration of new types(sets), constants and axioms. The presented, dynamic part (machine) contains the variable declarations and events. However, this simplification is of syntactic nature and is insignificant as such. Our approach to refinement pattern definition that we are presenting next can be easily extended to compensate it.

2.3 Event-B Models as Syntactic Objects

To define refinement patterns, let us now consider an Event B model as a syntactic mathematical object. For brevity, we omit representations of some of elements of models here, though they are supported in our tool implementation [13]. The subset of Event-B models used in this paper can be described by the following data structure:

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

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

Sub-elements of a model element can be accessed by using the dot operator, e.g., $act.style$ 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:

$$
\begin{array}{ll}
\textbf{MACHINE } m0 & \\
\textbf{VARIABLES } x & \langle x \mid \\
\textbf{INVARIANT } x \in \mathbb{Z} & \quad "x \in \mathbb{Z}" \mid \\
\textbf{INITIALISATION } x := 0 & \qquad \langle \texttt{init} \mid - \mid - \mid \langle x \mid := \mid "0" \rangle \rangle, \\
\textbf{EVENTS} & \\
\quad count = \textbf{BEGIN } x := x + 1 \textbf{ END} & \qquad \langle \texttt{count} \mid - \mid - \mid \langle x \mid := \mid "x + 1" \rangle \rangle \rangle
\end{array}
$$

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 Event B.

2.4 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 [13]. 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, where 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 is represented in our notation:

$$\begin{array}{ll}
\textbf{MACHINE } m0 & \\
\textbf{VARIABLES } x & \langle\, \langle x \rangle \mid \\
\textbf{INVARIANT } x \in \mathbb{Z} & \quad \langle "x \in \mathbb{Z}" \rangle \mid \\
\textbf{INITIALISATION } x := 0 & \quad\quad \langle\, \langle \texttt{init} \mid - \mid - \mid \langle x \mid:=\mid "0" \rangle\rangle, \\
\textbf{EVENTS} & \\
\quad count = \textbf{BEGIN } x := x+1 \textbf{ END} & \quad\quad\quad \langle \texttt{count} \mid - \mid - \mid \langle x \mid:=\mid "x+1" \rangle\rangle\rangle\rangle
\end{array}$$

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 the syntax of Event B. 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 constructs a model refinement for any acceptable input model and configuration:*

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

where $\sqsubseteq$ denotes a refinement relation.

In this paper we rely on 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 the 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:

```
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 denote 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 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:

```
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.

Another 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 $\exists v \cdot Inv(v)$
```

Here $FV(x)$ is set of free variables in x, Inv stands for $(\bigwedge_{i \in s.inv \cup ii} i)$, $Guards_e$ is defined as $(\bigwedge_{g \in e.guards} g)$ and BA is the before-after predicate. 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

$p(c) = basic(c)$	*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

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.

```
MACHINE $m0$                                MACHINE $m1$
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
   $newevt(en, s)$;
   $newgrd(en, e.guard)$ ||
   $newact(en, e.action)$;
   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. 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.

```
MACHINE $m1$
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:

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

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.

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

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.

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

Here $\mathit{\Gamma}$ 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.

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

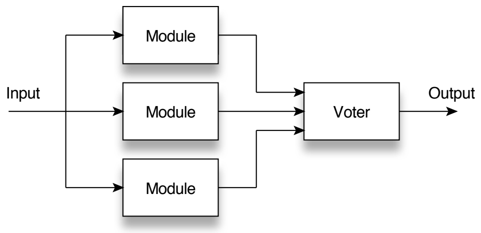

Fig. 2. TMR Arrangement

5 Triple Modular Redundancy Pattern

Triple Modular Redundancy (TMR) [15] is a fault-tolerance mechanism in which the results of executing three identical components are processed by a voting element to produce a single output that takes the majority view. This mechanism is schematically shown in Fig.2.

The purpose of the mechanism is to mask a single component failure. In this section we will demonstrate how to generalize a refinement step introducing the TMR arrangement into a model as a refinement pattern.

Before creating our new pattern, we have to decide on its applicability conditions. First, our input model should have a variable representing the output of the 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 input model should also contain an event that handles the component failure.

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 input specification.

The TMR pattern that we define uses a number of configuration parameters, as shown below. 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 the variable s.

```
**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**
```

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, i.e., 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 produced by the components; finally, the flags r_i reflect availability of new outputs in the respective output variables s_i.

The pattern consists of four major parts: the rules declaring the types and initialisation of new variables of the refined model; the definition of new events; the refinement rules for transforming a single abstract event representing the functionality 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.

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

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. For the brevity, we omit showing here the rules defining the types and initialisation for the variables s_2, s_3 and r_2, r_3 (the omitted part is indicated by ...). The shortcut notation $newini(a)$ used in the pattern description stands for declaration of the initialisation action: $newini(a) \overset{\text{df}}{=} newact(\texttt{init}, a)$. The shortcut $init(v)$ refers to an action of the initialisation event assigning to the variable v.

The refined model specifies the behavior of three components of TMR (we call them replicated components) as copies of the behaviour of the component specified in the input model. Since we assumed that a component is represented by a single event, the replicated components are created by adding three new events into the refined model in the following way.

The guard of the event modelling behaviour of a replicated component essentially coincides with the guard of an abstract component. However, it also contains an extra conjunct ensuring that the event is executed before passing control to the voter. The event actions essentially copy the corresponding actions of the abstract component (given as the pattern parameter a). The only difference that each replicated event records the result into a separate variable s_i (for the copy i) instead of the abstract variable s. In addition, a component copy also assigns to r_i to indicate the availability of result in s_i.

```
$events \overset{\text{df}}{=}$
**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**
    $copy_1 \parallel copy_2 \parallel copy_3$
**end**
```

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

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

The above rule $\langle s_1 \mid a.style \mid a.expression \rangle$ constructs an action from the abstract action a in such a way that it would have the same effect but update the new variable s_1. Here $a.style$ is one of non-deterministic assignment styles.

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.

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

As a result, the abstract action a of the component is 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 preparation for the 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.

$$
\begin{array}{l}
abort \overset{\text{df}}{=} \\
\quad newgrd(zz, "r_1 = TRUE \wedge r_2 = TRUE \wedge r_3 = TRUE"); \\
\quad newgrd(zz, "s_1 \neq s_2 \wedge s_2 \neq s_3 \wedge s_1 \neq s_3");
\end{array}
$$

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

$$
\begin{array}{l}
invariants \overset{\text{df}}{=} \\
\quad newinv("ph = TRUE \wedge (s_1 = s_2 \vee s_2 = s_3)) \Rightarrow s = s_1"); \\
\quad newinv("ph = TRUE \wedge s_2 = s_3) \Rightarrow s = s_2")
\end{array}
$$

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

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.

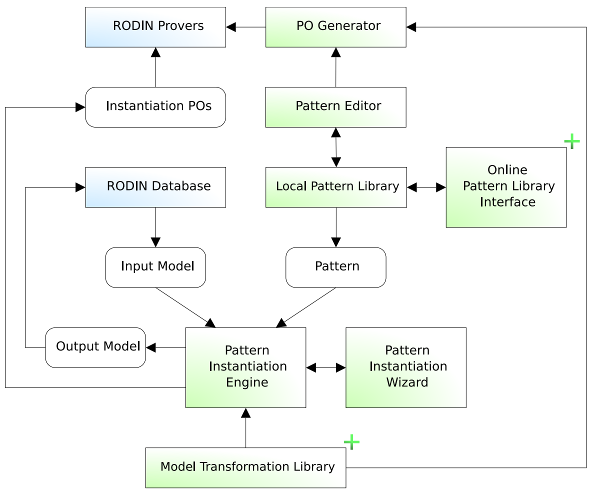

Fig. 3. The Event-B refinement patterns tool architecture.

The overall tool architecture is presented in Figure 3. 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)[13] 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 [13].Several patterns developed with this tool were applied during formal modelling of the Ambient Campus case study of the RODIN Project [14].

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 [8] has been developed to support program development using the Refinement Calculus theory by R.Back and J. von Wright. [5] The theory was formalised in the HOL theorem prover, while specific refinement rules were proved as HOL theorems. The HOL Window Inference library[11] 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 [17]. 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 [7]. 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. The idea of reuse via instantiation of generic Event B models has also been explored by Silva and Butler [18]. However, they focus on the instantiation of the static part of the model – the context – while our approach mainly manipulates its dynamic part. Nevertheless, these two approaches are complementary and can be integrated.

Obviously the idea to use refinement patterns to facilitate the refinement process was inspired by the famous collection of software design patterns [10]. 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. J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
3. J.-R. Abrial. Extending B without Changing it. *Proceedings of 1st Conference on the B Method*, pp.169-191, Springer-Verlag, November 1996, Nantes, France.
4. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), pp.1-23, 1996.
5. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998.
6. R.-J. Back and K. Sere. Stepwise Refinement of Action Systems. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 115–138, London, UK, 1989. Springer-Verlag.
7. L. Burdy and J.-M. Meynadier. Automatic Refinement. *Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, FM'99*, 1999.
8. 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.
9. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley. ISBN 0-201-63361-2, 1995.
11. J. Grundy. Transformational Hierarchical Reasoning. *The Computer Journal*, 39(4):291–302, 1996.
12. C. A. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.
13. A. Iliasov. Finer Plugin. `http://finer.iliasov.org`, 2008.
14. A. Iliasov, A. Romanovsky, B. Arief, L. Laibinis, and E. 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.
15. R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal*, pages 200–209, April 1962.
16. C. Morgan. *Programming From Specifications.* Prentice Hall International (UK) Ltd., 1994.

17. M. Oliveira, A. Cavalcanti, and J. Woodcock. Arcangel: a tactic language for refinement. *Formal Asp. Comput.*, 15(1):28–47, 2003.
18. R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In K. Breirman and A.Cavalcanti, editors, *11th International Conference on Formal Engineering Methods (ICFEM) 2009*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.