Formal Refinement Automation

Alexei Iliasov, Alexander Romanovsky

Newcastle University, Newcastle upon Tyne, UK

Elena Troubitsyna, Linas Laibinis

Åbo Akademi University, Turku, Finland

Abstract

Formal methods focus on a posteriori analysis and a modeller gets little assistance in constructing a model which makes formal modelling more expensive and laborious than it could be. In this paper we discuss a mechanism for automation of formal refinement based on reusable refinement steps, called refinement patterns. Refinement patterns offer a constructive top-down approach to formal modelling; for many patterns, correctness can be proved once-for-all. The approach is illustrated with the discussion of Event-B refinement patterns and a tool supporting pattern-based Event-B developments.

Keywords: patterns, design reuse, formal verification. refinement patterns, Event-B

1 Introduction

Although modelling is one of the more expensive stages of a system construction, models are rarely reused and adapting a legacy formal development to a new system design is nearly impossible. We propose to see a formal development not only as a way to arrive to a final model but also as a process creating important, reusable artifacts. In our approach, the model construction process is understood as a chain of design decisions. Many such decisions are specific to a problem being solved, but there are also some general ones that could be reused within the same or a different development.

The inspiration for the proposal comes from the idea of a reusable design pattern, originally introduced by an architect Christopher Alexander [1] and later adapted by Cunningham and Beck [2] to object-oriented programming. The collection of design patterns for object-oriented software development, published by Gamma et al. [3], now plays a central role in software development.

We introduce a new entity in the formal development process which sole purpose is to describe a refinement step leading from an abstract model to a refined model. We call such entity a *refinement pattern* (first introduced in [4]) and understand it as a function which, applied to an abstract model, yields a refined version of the

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

model. The name *pattern* emphasises the fact that such function is a reusable design solution applicable in a range of contexts, much like design patterns developed for object-oriented programming [3].

The paper is organised as follows: we start with the discussion of the pattern mechanism, including the pattern instantiation procedure and pattern correctness proof obligations; proceed with an overview of Event-B refinement patterns; and conclude with a detailed treatment of a simple Event-B refinement pattern.

2 Refinement Patterns

The fact that specification s is refined by s' is denoted as $s \sqsubseteq s'$ where relation $\sqsubseteq$ is reflexive, transitive and antisymmetric.

A refinement pattern is a function producing a refinement for some input model and a pattern configuration:

$rpatt : S \times C \nrightarrow S$

where

$$\forall s, c \cdot ((s, c) \in dom(rpatt) \Rightarrow s \sqsubseteq rpatt(s, c))$$

where S is a universe of models and C are pattern configurations.

This section continues with a discussion of the model transformation concept: the foundation for specifying refinement patterns. Then we introduce the pattern language, pattern instantiation rules, pattern correctness proof obligations and pattern instantiation side conditions.

2.1 Model Transformations

A model transformation is a deterministic rule producing a new model by transforming some input model. A model transformation is associated with two model classes: the input model class, describing the set of all models accepted by the transformation, and the output model class, characterising the set of possible model transformation results. The input and output models of a model transformation application must belong to, correspondingly, the input and output model classes.

A model transformation is understood as a function transforming an input model and some configuration into an output model. Unlike a refinement pattern, a model transformation does not have to a produce a model refinement. The following general form is used to define model transformations:

$\mathsf{name}(p) \equiv$ requirements $c(s, p)$ effect $s' = r(s, p)$ scope e ref q

where p is a vector of parameters, $c(s, p) = \mathbf{req}(name)$ is an applicability condition, $r(s, p)$ is a rule computing the refined version of an abstract model s, and $e = \mathbf{scope}(name)$ the transformation scope. The q flag, $q : r \times c \rightarrow \mathbb{B}, q = \mathbf{ref}(name)$, defines whether a given model transformation produces a valid model refinement. The applicability condition of a model transformation is the input model class of the transformation: $c(s, p) = \mathbf{incl}(name)(s, p)$ where $name$ is the name of the transformation. The output model class is computed as $\mathbf{outcl}(name) = c(s, p)[r(s, p)/s]$.

Although, there are no formal constraints on the kind of rules used in a model

$$p(w) \;=\; mdr(w) \mid \begin{pmatrix} p & \\ & p \end{pmatrix} \mid \begin{pmatrix} p \\ p \end{pmatrix} \mid \textsf{forall } a \textsf{ with } d \textsf{ where } P(a,d,w) \textsf{ do } p(a,d,w)$$

$$\textsf{with } d \textsf{ where } P(d,w) \textsf{ do } p(d,w) \;\equiv\; \textsf{forall } a \textsf{ with } d \textsf{ where } P(d,w) \textsf{ do } p(d,w)$$

Fig. 1. The language of refinement patterns.

transformation, a model transformation rule $r(s,p)$ should be simple enough to define the model transformation scope e and the refinement attribute q without in-depth analysis of the properties of $r(s,p)$.

The scope of a model transformation is made of two sets: the set of elements updated by the rule and the set of elements referred by the rule. The union of scopes, characterising the scope of a combination of model transformations, is defined as

$scope((rr_1, ww_1)) \cup scope((rr_2, ww_2)) = (rr_1 \cup rr_2, ww_1 \cup ww_2)$

where rr_1 and rr_2 are the sets of referred elements, and ww_1, ww_2 are the sets of updated elements. The intersection of scopes determines whether two given rules transform overlapping model parts. It is defined as a combination of update/update and read/update conflicts:

$scope((rr_1, ww_1)) \cap scope((rr_2, ee_2)) = (ww_1 \cap ww_2) \cup (rr_1 \cap ww_2) \cup (rr_2 \cap ww_1)$

For two rules, working on non-overlapping model parts, the intersection of their scopes is an empty set.

2.2 Specifying Patterns

Pattern is a named entity. A pattern declaration starts with a construct declaring the pattern name:

pattern *name*

  p

where *name* is the name of the declared pattern and p is the pattern body. The body of a pattern is a combination of model transformations glued together using a number of control structures. The simplest form of a pattern is a single model transformation rule ($mdr(w)$ in Figure 1).

The two main operations used in a pattern declaration are the sequential and parallel compositions. In a pattern specification we use indentations to denote blocks of rules and the style of a composition operator used to connect the rules. The sequential composition requires the next rule to be written below and indented to the right (Figure 1). The sequential composition is normally used to put together semantically linked rules.

The parallel composition is denoted by an equal indentation of the composed rules (Figure 1). The parallel composition is used to connect unrelated rules, which are the rules operating on different model parts. The sequential composition binds stronger than the parallel composition and brackets are used when an instance of the parallel composition must take precedence.

The role of the parallel composition is the simplification of the pattern correctness analysis. To prove that a given instance of the parallel composition is correct it is enough to independently show the correctness of each constituent rule.

To construct a non-trivial pattern we have to provide a mechanism for matching

and selecting model parts that need to be transformed. This is achieved with the following construct: forall a with d where $P(a,d)$ do $r(a,d)$. Here a is a vector of local variables bound by the constraining predicate $P(a,d)$ (the predicate also depends on an implicit input model parameter), d is a vector of local parameters and r is a nested rule. The rule r is applied to every possible value (a,d), as defined by the predicate P. Local variables a are selected automatically during a pattern instantiation while the values for d are provided by a modeller, once for each *forall* construct. In other words, a contained rule r is applied to all possible values of a but each times uses the same vector d.

The nesting of rules controls the visibility of names. Children of a rule can see all the parameters and local variables available to their parent. This relation is transitive - a rule sees all the parameters available to its parent rule.

The pattern language we have discussed is method-neutral. It can be used to describe refinement patterns for a wide range of formal modelling methods. To connect the pattern mechanism with a specific formalism (or a family of formalisms) we have to import a library of model transformations. Symmetrically, to bring the support for the pattern mechanism into a formal method it is enough to construct a library of model transformations that can be used to describe interesting refinement cases.

2.3 Applying Patterns

A pattern is applied to a model to produce a new model. A refinement pattern takes an abstract model and produces its concrete version. Interpretation of a pattern is provided by a function of the following form:

eff : MODEL $\times$ RULE $\rightarrow$ MODEL

There is one more ingredient to this defintion: a pattern configuration providing pattern instantiation parameters. We assume that we have the complete configuration before applying a pattern and, for legibility reasons, assume pattern configuration as an implicit parameter of **eff**.

The effect of the sequential composition of two rules is computed by applying the the second rule to the result of the first rule (Figure 2, first rule). For the parallel composition the effect is computed in the same manner but the order of the rule applications should not affect the overall result (Figure 2, second rule). The *forall* construct applies the body rule to all the possible values of local variables. This is expresses with a recursive definition which, at each iteration, reduces the set of possible parameters by removing the already used values (Figure 2, third rule). Finally, the effect of a model transformation is computed by applying the model transformation to an input model (Figure 2, fourth rule).

2.4 Proving Pattern Correctness

A pattern correctness can be demonstrated in a number of ways [5]. The simplest approach is to declare a pattern correct until the opposite is demonstrated and rely on the normal means of proving the refinement relation between two models. In other words, each time a pattern is applied, the result must be shown to be a valid refinement using the refinement relation conditions of the applied formalism.

$$\mathbf{eff}\left(s, \begin{matrix} a \\ \\ b \end{matrix}\right) = \mathbf{eff}(\mathbf{eff}(s,a),b)$$

$$\mathbf{eff}\left(s, \begin{matrix} a \\ b \end{matrix}\right) = \mathbf{eff}(\mathbf{eff}(s,a),b) = \mathbf{eff}(\mathbf{eff}(s,b),a)$$

$$\mathbf{eff}\left(s, \begin{matrix} \mathsf{forall}\ a\ \mathsf{with}\ p\ \mathsf{where} \\ \quad P(a,p) \\ \mathsf{do} \\ \quad r(a,p) \end{matrix}\right) = \mathbf{eff}\left(s, \begin{matrix} r(q,p) \\ \quad \mathsf{forall}\ a\ \mathsf{with}\ p\ \mathsf{where} \\ \quad\quad P(a,p) \wedge a \neq q \\ \quad \mathsf{do} \\ \quad\quad r(a,p) \end{matrix}\right)$$

$$\mathbf{eff}\,(s, mdr(p)) = mdr(p)$$

Fig. 2. Pattern instantiation rules.

Here we present an approach where a modeller tries to demonstrate that a pattern result is a valid refinement of any acceptable input model. Sometimes this is not possible. Then we rely on a combination of static pattern analysis and theorems generated for each pattern instantiation.

A pattern is correct if, for all the constituent rules, the following conditions are satisfied.

Computability

A patterns is a procedure that should be processable by a software tool. Thus, we need to make sure that a pattern is executable. A problem can arise when a constraining predicate of the *forall* construct describes an infinitely large set of parameters. Such pattern, although possibly useful, makes no sense to a tool. We express this by stating that a constraining predicate is a characteristic function of a finite set (see the first rule in Figure 3).

Conflict-freeness

The effect of the application of a parallel composition of rules should not dependent on the order of rule applications. This property holds when the rules are applied to disjoint model parts and is expressed as a requirement that a pair-wise intersection of scopes of rules, composed using the parallel composition operator, is an empty set (the second rule in Figure 3).

The *scope* function computes the scope of a rule by aggregating scopes of its constituent rules. Scope of a sequential or a parallel composition is the union of scopes of the composed rules. Scope of a model transformation is the scope property of the transformation. Finally, the scope of a *forall* construct is the scope of its body rule.

Well-formedness

A pattern rule may call its child rule only when the child rule precondition is satisfied. For each model transformation used in a pattern body it is required to demonstrate that the transformation requirement is satisfied for all the contexts in which the model transformation may be invoked. We state this by requiring that the context of a parent rule implies the requirement of a model transformation

rule	*proof obligation*
forall $a; d$ where $P(a,d)$ do $\ldots$	$finite(\{(a,d) \mid P(a,d)\})$
r_1 $\ldots$ r_k	$\forall i,j \cdot (i \in 1..k \wedge j \in 1..k \wedge i \neq j \Rightarrow$ $scope(r_i) \cap scope(r_j) = \varnothing)$
pattern *pat* q forall $a; p$ where $P(a,p)$ do $mdr(a,p)$	$\forall s,a \cdot (\mathbf{incl}(q)(s) \wedge \mathbf{eff}(s,q)$ $\wedge P(a,p) \Rightarrow \mathbf{req}(mdr)(a,p))$
q $mdr(p)$ r	$\mathbf{ref}(mdr) \vee \forall s \cdot (in(q)(s) \Rightarrow$ $s \sqsubseteq \mathbf{eff}(\mathbf{eff}(\mathbf{eff}(s,q), mdr(p)), r))$

Fig. 3. Pattern correctness proof obligations.

under analysis. The third rule in Figure 3 gives the general form of this condition when the analysed model transformed is a part of a *forall* statement. A model transformation without *forall* is seen as a special case of this condition. Note that q denotes the combination of all the parents of the analysed pattern rule.

Refinement

For a model transformation that does not construct a valid model refinement, we have to demonstrate that a combination of the transformation with other patterns rules is a model refinement rule. This can be done by finding a containing subset of pattern rules for which, together with the model transformation being examined, we are able to demonstrate that the described transformation is a subset of a refinement relation (and the refinement relation itself is imported from some formal method, such as Event-B [6]) (rule four in Figure 3).

The inability to discharge or demonstrate the falsity of these conditions may indicate that a pattern is too complex. The problem can be rectified by redesigning the pattern with an emphasise on the use of the parallel composition operator. Alternatively, a pattern may be decomposed into sub-patterns and that can be analysed independently.

2.5 Assertions

There may be situations when to prove a pattern we have to assert some non-computable property of an abstract machine. For example, a pattern implementing an array sorting algorithm may need to assert that some abstract action describes a sorted array. There are many ways to state that an array becomes sorted and the only general solution is to formulate and prove a theorem that states that the action indeed describes what it is assumed to describe. Such property is unlikely to be computable due to the size of the involved state space, which is potentially infinite. Thus, we have to rely on theorem proving and generate the required theorems each time a pattern is instantiated. This is accomplished using the assertions mechanism.

An assertion is a distinguished part of a constraining predicate. We use the

event e	new event
refines r	changes the list of refined events
label l	changes the name of an event or a variable
guard g	adds an event guard
param p	adds an event parameter
action v s exp	adds an event action or a variable initialisation
variable v	adds a new model an an action variable
style s	changes an action style
expression e	changes an action expression
invariant t	adds a new invariant or changes the typing predicate of a variable

Fig. 4. The summary of Event-B model transformations.

following syntax to declare an assertion: assert $n(p)$ $Q(p)$. Here n is the assertion name, p is a vector of parameters and $Q(p)$ is the property asserted. To use an assertion, one writes $n(u)$, where u is a vector of parameters.

During a pattern instantiation, an assertion is assumed to hold, i.e., is true. However, each time an assertion is encountered during an instantiation, a theorem is generated requiring that the assertion holds for the current values of p. If all such theorems are satisfied, then the model produced by a pattern is a refinement of the corresponding input model.

3 Event-B Refinement Patterns

Event-B [6,7] is a modern version of the B Method [8] formal modelling method. An Event-B development is a chain of model refinements starting with an abstract model and followed by a succession of refined versions. A model in Event-B is described by a collection of named variables and events. An event is understood as an instantaneous update of model variables and an observable model evolution is a number of discrete state transitions. Model invariants are used to state the desired properties which must be maintained by each event. For verification, Event-B relies mainly on theorem proving. A software tool automatically generates all the correctness conditions as a list of theorems and a theorem prover attempts to discharge them. An interactive theorem prover is used to conduct controlled manual proofs. An Event-B machine has the following general form:

system *name*

variables *var* (set of variables)

invariant *inv* (set of model invariants)

initialisation (variable initialisations)

events

...

nom = **any** *arg* **where** *grd* **then** *act* **end**

...

An Event-B event is composed of a declaration of local variables, event guard, and a list of actions updating model variables. In the model snippet above, *nom*

is the event name, arg is a vector of event parameters, grd is the event guard and act is a before-after predicate. The structure of an Event-B refinement machine is identical to the structure of an abstract machine.

To transform Event-B models with patterns, it is convenient to look at an Event-B model as a mathematical object, specifically, a syntactic tree describing a model. Such tree has the following structure:

$$
\begin{aligned}
sys &:= (var, inv, evt) \\
evt &:= (nom, ref, arg, grd, act) \\
act &:= (var, sty, exp) \\
var &:= (nom, act, typ) \\
arg &:= (nom, typ)
\end{aligned}
\qquad
\begin{aligned}
nom &:= \text{LABEL} \\
inv &:= \text{PREDICATE}^* \\
typ &:= \text{PREDICATE} \\
grd &:= \text{PREDICATE}^* \\
ref &:= \text{LABEL}^* \\
sty &:= \text{" := " } | \text{ " :}\in \text{ " } | \text{ " :| "}
\end{aligned}
$$

The root of a tree is the system tuple. The tuple contains sets of variables, invariants and events. A variable is described by a tuple of name, initialisation action and typing predicate. An action is characterised by a list of updated variables, an action style and an expression computing the new variable states. An event is described a name, a list of refined abstract events, a vector of parameters, a vector of guards (whereas an overall event guard is the conjunction of individual guards) and a collection of actions. An event parameter is a tuple of name and a typing predicate. There are two basic types: LABEL is an identifier of a model element (e.g, event name); PREDICATE is a predicate expression. In addition, an action style is one of three Event-B substitution styles [6].

To specify Event-B refinement patterns we use a collection of Event-B model transformations. Due to the space limitation we are only able to present the summary of Event-B model trasformations (Figure 4). More details on Event-B model transformations can be found in [9].

4 Example

This section illustrates the patterns mechanism with an example of an Event-B refinement pattern. The presented pattern transforms an input model by refining an action of an abstract event into a simple communication mechanism. The complete pattern specification is given in Figure 5.

The pattern is applicable to models with at least one event that contains an action assigning to a single variable (in a general case, an Event-B action updates a vector of variables). This condition is expressed using the *with* construct which also defines the parameters *destevt* (some abstract event) and *copyact* (some action of *destevt*) (replacing *with* with *forall* we could make the pattern transform all the suitable pairs of events and actions in an input model). The pattern body is made of five major blocks: the construction of a variable representing a communication channel (block *ch* in Figure 5); the construction of a channel state variable (block *rd*); the defintion of a new event responsible for writing in the channel (*send*);

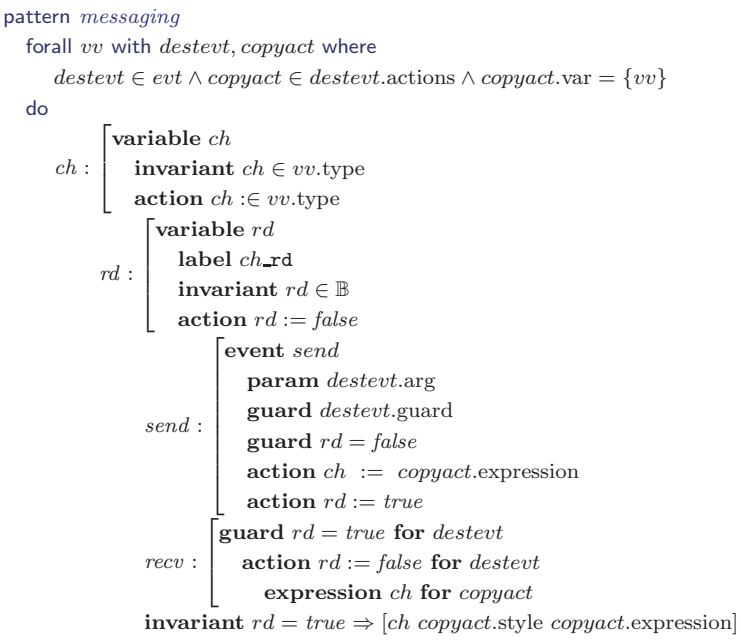

Fig. 5. The *messaging* refinement pattern specification.

the refinement rules updating the abstract event *destevt*; the invariant relating the state of the channel variable *ch* with the abstract model state (the last rule in Figure 5). Notation [*var style expr*] used in the invariant rule stands for a before-after predicate corresponding to the Event-B action *var style expr*.

4.1 Correctness

To demonstrate the pattern correctness, we routinely apply the rules given in Figure 3 to obtain the list of the pattern proof obligations. Instantiating the computability proof obligation, we get the following condition:

$finite(\{(d, c) \mid d \in evt \wedge c \in d.\text{actions} \wedge card(c.\text{var}) = 1\})$

The above trivially holds noticing that an Event-B model has a finite number of events - $finite(evt)$ - and an event has a finite number of actions: $finite(d.\text{actions})$.

The conflict-freeness condition is demonstrated by writing out scopes of the pattern rules. For example, for the pattern subset declaring new variable *ch* the scopes are computed as follows

variable *ch*	$var, \star ch$
invariant $ch \in copyact.\text{type}$	$\star ch : \text{typ}$
action $ch :\in copyact.\text{type}$	$\star ch : \text{act}$
variable *rd*	$var, \star rd$

For all the pairs of parallel rules we have to demonstrate the absence of scope conflicts. Although this results in a large number of proof obligations, all such obligations are computed automatically by a tool. Moreover, in the new version of the tool it is impossible to construct patterns with scope conflicts: the editor simply does not allow parallel composition of conflicting rules.

Well-formedness of a pattern is analysed by generating corresponding proof obligations for all the pattern rules. For our example, there are 17 such proof obligation. The declaration of new variable *ch* results in the following proof obligations

invariant preservation: $ch' \in vv.\text{type} \Rightarrow ch' \in vv.\text{type}$

feasibility: $\mathcal{I}(v) \Rightarrow \exists ch' \cdot ch' \in vv.\text{type}$

The names on the left-hand side indicate the Event-B refinement conditions [6] from which the proof obligations are derived. Note that vv.type is some unknown predicate. While the first proof obligation is trivially true, to discharge the second one we have to rely on the properties of the abstract model being transformed by the pattern. This properties are summarised in predicate $\mathcal{I}(v)$, introduced later in this section. For brevity, we present only few interesting proof obligations.

The **guard** $rd = true$ **for** *destevt* rule leads to a proof obligation requiring that the overall guard of concrete event *destevt* is stronger than the guard of its abstraction. This condition trivially holds since the new guard is stronger due to the conjunction with an additional condition:

guard strengthening: $destevt.\text{guard}(v) \wedge rd = true \Rightarrow destevt.\text{guard}(v)$

The update of the expression of the *copyact* (rule **expression** *ch* **for** *copyact*) leads to two proof obligations. The first requires us to show that the new action is feasible: always delivers a result under the given assumptions. The second states that new action must be a refinement of the corresponding action of abstract event *destevt*:

feasibility: $$\mathcal{I}(v) \wedge J(v,u) \wedge destevt.\text{guard}(v) \wedge rd = true \Rightarrow \exists u' \cdot [copyact.\text{var} := ch](u,u')$$

refinement: $$\mathcal{I}(v) \wedge J(v,u) \wedge destevt.\text{guard}(v) \wedge rd = true \wedge [copyact.\text{var} := ch](u,u') \Rightarrow \exists v' \cdot ([copyact](v,v') \wedge J(v',u'))$$

where v is a vector of abstract model variables. This vector contains at least one variable vv, $\{vv\} = copyact.\text{var}$. Vector u denotes concrete model variables, which, among the variables inherited from an abstract model ($v \subset u$), contains variables *ch* and *rd*.

Predicate $\mathcal{I}(v)$, used in several proof obligations above, is a collection of known facts about the abstract model. For the messaging pattern, we know that a suitable abstract model contains at least one event which must have at least one action assigning to a variable. We also assume that an abstract model transformed by the pattern is well-formed (in sense of Event-B well-formedness [6]). Thus we may state that the abstract action *copyact* has a solution when its guard is enabled (the feasibility condition). This condition helps us to discharge the *feasibility* proof obligation above. The complete $\mathcal{I}(v)$ definition is

$$\mathcal{I}(v) = I(v) \wedge \exists e \cdot (e \in evt \wedge \exists a \cdot (a \in e.\text{act} \wedge card(a.\text{var}) = 1)) \wedge (I(v) \wedge destevt.\text{guard}(v) \Rightarrow \exists v' \cdot ([copyact](v,v') \wedge J(v',u')))$$

where $I(v)$ is an arbitrary predicate corresponding to the abstract model invari-

ant.

Predicate $J(v, u)$ is a concrete invariant. In addition to constraints on concrete variables, a concrete invariant in Event-B also has a gluing invariant linking the states of concrete and abstract model. In our case, such gluing invarint is the invariant produced by the rule **invariant** $rd = \ldots$. This invariant links the abstract action expression with the value of concrete variable ch. The complete definition of $J(v, u)$ is as follows:

$$J(v,u) = ch :\in copyact.\text{type}(v) \wedge rd \in \mathbb{B} \wedge$$
$$rd = true \Rightarrow [ch\ copyact.\text{style}\ copyact.\text{expression}](u)$$

The *feasibility* and *refinement* conditions are discharged in a number of simple steps by writing out $\mathcal{I}(v)$ and $J(v, u)$ and then demonstrating that $[copyact.\text{var} := ch](u, u')$ is equivalent to $[copyact](v, v')$ provided $rd = true$.

Several model transformations require further analysis to ensure that the pattern construct a model refinement. These rules are **invariant** $rd = \ldots$, **variable** *destevt*.arg and **event** *send*. The former two cases are rather trivial and we only consider the condition generated for the **event** *send* rule. This condition requires us to prove that new event *send* terminates in a finite number of steps and passes the control to some other event. For this we have to find a variant which is a finite natural number decremented by the new event:

$$\textit{non-divergence}: \quad \mathcal{I}(v) \wedge J(v,u) \wedge destevt.\text{guard}(v) \wedge rd = false \wedge$$
$$[\{ch\}\ copyact.\text{style}\ copyact.\text{expression}](ch, ch') \wedge$$
$$rd' = true \Rightarrow \exists V \cdot (V(u) \in \mathbb{N} \wedge V(u') < V(u))$$

One possible witness for V is $V(rd), V : \mathbb{B} \rightarrow \{0, 1\} = \{true \mapsto 1\} \cup \{false \mapsto 0\}$. Event *send* decreases V: in the before-state state of *send*, V must be 1 and in the after-state it is always 0.

Figure 6 demonstrates an example of the pattern instantiation. The input model contains a single event with two actions. Any of these actions can be selected as a value for the *copyact* parameter and it is assumed that the parameter selection is handled externally (e.g., by a modeller). In this example the pattern is instantiated with $copyact = (n := (m + 1)\ mod\ 6)$. In Figure 6, model $m0$ is the input model and model $m1mes$ is the the model produced by the messaging refinement pattern (Figure 5).

The *messaging* pattern is applicable to a wide range of models. One example is a combination with the recovery block [10], parity bit [11] and hamming code [11] patterns to produce a fault-tolerant communication protocol. The protocol starts with a communication loop with a single parity check bit. Upon detection of an error, the protocol switches to a communication loop using the hamming code. The described development can be constructed entirely from the currently available refinement patterns.

system $m0$

variables n, m

invariant $n \in 0 \mathinner{..} 5 \wedge m \in \{1, 2, 3\}$

initialisation $n := 0 \| m := 1$

events

$work$ = **begin** $n := (m + 1)\ mod\ 6 \| m :\in \{1, 2\}$ **end**

system $m1mes$

refines $m0$

variables n, m, ch, ch_rd

invariant

$n \in 0 \mathinner{..} 5 \wedge m \in \{1, 2, 3\} \wedge ch \in 0 \mathinner{..} 5 \wedge ch\_rd \in \mathbb{B}$

$ch\_rd = true \Rightarrow ch = (m + 1)\ mod\ 6$

initialisation $n := 0 \| m := 1 \| ch :\in 0 \mathinner{..} 5 \| ch\_rd := false$

events

$send$ = **when** $ch\_rd = false$ **then** $ch := (m + 1)\ mod\ 6 \| ch\_rd := true$ **end**

$work$ = **when** $ch\_rd = true$ **then** $n := ch \| m :\in \{1, 2\} \| ch\_rd := false$ **end**

Fig. 6. Instantiation example for the *messaging* refinement pattern.

5 Event-B Refinement Patterns Tool

A proof of concept implementation of the pattern mechanism support for the Event-B method has been realised as a plug-in to the RODIN Platform [12]. The plug-in seamlessly integrates with the RODIN Platform interface so that a user does not have to switch between different tools and environments when to apply 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 an interactive prover.

The overall tool architecture is presented in Figure 7. 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. 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

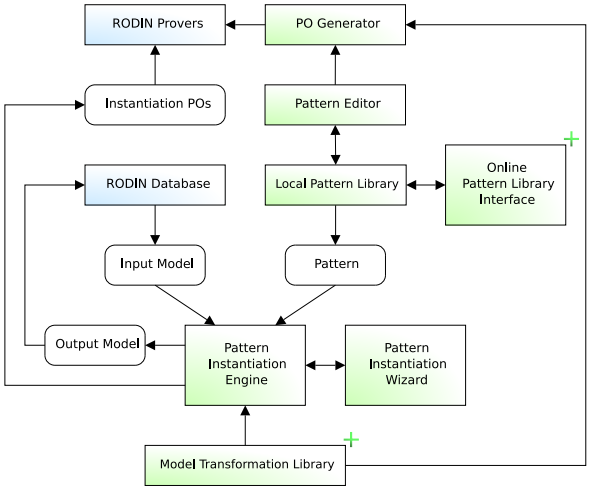

Fig. 7. The Event-B refinement patterns tool architecture.

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 demonstrated each time 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, [11]) 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 Figure 3. The result is a new context file populated with theorems corresponding to the pattern proof obligations. The normal 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 pattern instantiation engine and the proof obligation generator.

The current version of the tool is freely available from our web site [11]. To install, download the jar file and place it in the *plugins* folder of the RODIN Platform installation. Several large-scale patterns were developed with the tool and are available for download from the plug-in web page [11]. On the web page an interested reader will find examples of Event-B specifications constructed from the patterns. Several patterns developed with this tool were applied during formal modelling of the Ambient Campus case study of the RODIN Project [13].

6 Conclusions

There is a large number of related works in the area of specification and refinement automation and we briefly overview some of them. The pattern reuse mechanism described in [14] proposes specifying design patterns as B machines. A similar approach, although with the focus on object orientation, is described in [15]. Abrial's Mechanical Press Controller [16] case study is a notable example of application patterns to simplify development and facilitate proof reuse. The patterns are constructed independently from the main development using the normal tools and techniques. Pattern instantiation process is manual and thus proof reuse is based on informal assumptions. We believe, however, that the B language is generally inadequate for specifying reusable patterns.

Siemens/MATRA, working on specifications of a fully automated train system, have developed a tool for automatically refining B specifications [17]. An interesting aspect of the tool is that it is connected to a development method narrowly oriented on modelling train system. The tool was successfully applied in large specification projects (apparently train systems) but is proprietary and is used privately by the company.

There is a large body of research in the area of formalising the GoF [3] design patterns, The notable examples are the RAISE Specification Language [18] and the LePUS formal framework [19].

To conclude, we believe that the application of refinement patterns in formal modelling offers a number of advantages:

- patterns can be used in more than one context within the same development and can be applied in completely unrelated developments;
- since a pattern deals with a specific aspect of a system functionality, in many cases, refinement patterns can be constructed independently from the main development;
- proofs done for a refinement pattern are automatically reused each time a pattern is instantiated, in a large development this results in a considerable proofs economy.

In addition, since a refinement pattern is self-contained, the pattern mechanism facilitates the exchange of ideas between designers. A pattern can be transferred, or, even, purchased and sold. A development can be, at least partially, constructed from ready-made third-party refinement or abstraction patterns, which, ultimately, means reduced modelling costs and a wider adoption of formal modelling.

References

[1] Christopher, A.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, USA, 1216. ISBN 0195019199 (1977)

[2] Beck, K., Cunningham, W.: Using Pattern Languages for Object-Oriented Programs. Technical Report No. CR-87-43 (1987)

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley. ISBN 0-201-63361-2 (1995)

[4] Iliasov, A.: Refinement Patterns for Rapid Development of Dependable Systems. Proc. Engineering Fault Tolerant Systems Worshop (at ESEC/FSE, Croatia), ACM (September 4, 2007)

[5] Iliasov, A.: Refinement Patterns. Technical report, School of Computing Science, Newcastle University (2008)

[6] C. Metayer, J.-R. Abrial, L.V., ed.: Rodin Deliverable D7: Event B language. Project IST-511599, School of Computing Science, University of Newcastle (2005)

[7] Abrial, J.R.: Extending B without changing it (for developing distributed systems). In Habrias, H., ed.: 1st Conference on the B method, IRIN Institut de recherche en informatique de Nantes (1996) 169–190

[8] Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)

[9] Iliasov, A.: Event-B Model Transformations. http://finer.iliasov.org/transdef.pdf (2008)

[10] Iliasov, A., Romanovsky, A.: Refinement Patterns for Fault Tolerant Systems. In: EDCC 7: the Seventh European Dependable Computing Conference (EDCC-7). (2008)

[11] Iliasov, A.: Finer Plugin. http://finer.iliasov.org (2008)

[12] RODIN: Event-B Platform. http://rodin-b-sharp.sourceforge.net/ (2007)

[13] Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: 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, Washington, DC, USA, IEEE Computer Society (2007) 141–145

[14] Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In: ZB 2003: Formal Specification and Development in Z and B. Springer (2003) 626–626

[15] Chan, E., Robinson, K., Welch, B.: Patterns for B: Bridging Formal and Informal Development. In Julliand, J., Kouchnarenko, O., eds.: B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 125–139

[16] Abrial, J.R.: Case study of a complete reactive system in Event-B: A Mechanical Press Controller. http://www.zb2005.org/PRESS-slides.tar.gz (2005)

[17] Burdy, L., Meynadier, J.M.: Automatic Refinement. Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, FM'99 (1999)

[18] Flores, A., Moore, R., Reynoso, L.: A Formal Model of Object-Oriented Design and GoF Design Patterns. In: FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, London, UK, Springer-Verlag (2001) 223–241

[19] Eden, A.H., Hirshfeld, Y., Yehudai, A.: Towards a Mathematical Foundation for Design Patterns. Technical report, Department of Computer Science, Tel Aviv University (1999)