# Refinement plans for informed formal design[⋆]

Gudmund Grov[1], Andrew Ireland[2], and Maria Teresa Llano[2]

[1] University of Edinburgh, School of Informatics, Edinburgh, UK,
[2] Heriot-Watt University, MACS, Edinburgh, UK

**Abstract.** Refinement is a powerful technique for tackling the complexities that arise when formally modelling systems. Here we focus on a posit-and-prove style of refinement, and specifically where a user requires guidance in order to overcome a failed refinement step. We take an integrated approach – combining the complementary strengths of top-down planning and bottom-up theory formation. In this paper we focus mainly on the planning perspective. Specifically, we propose a new technique called *refinement plans* which combines both modelling and reasoning perspectives. When a refinement step fails, refinement plans provide a basis for automatically generating modelling guidance by abstracting away from the details of low-level proof failures. The refinement plans described here are currently being implemented for the Event-B modelling formalism, and have been assessed on paper using case studies drawn from the literature. Longer-term, our aim is to identify refinement plans that are applicable to a range of modelling formalisms.

## 1 Introduction

We focus here on a layered style of formal modelling, where a design is developed as a series of abstract models – level by level concrete details are progressively introduced via provably correct *refinement* steps. There are two major approaches in achieving this style of formal modelling: the *rule-based* approach and the *posit-and-prove* approach; examples can be found in [25] and [21, 1], respectively.

The work reported here aims to enhance the posit-and-prove approach. Specifically, we have developed a technique called *refinement plans* which automatically generates guidance for users within posit-and-prove formal modelling. Like many approaches to design, whether informal [13] or formal [2], our technique relies upon patterns. While we focus here on relatively small patterns, we believe this will provide a foundation upon which to explore larger refinement patterns in the future.

The novelty of our refinement plans is that they combine modelling and reasoning patterns, enabling us to computationally exploit the subtle interplay that exists between modelling and reasoning – what we call *reasoned modelling*. Our refinement plans are heuristic in nature, and can be applied *flexibly* during a development. This flexibility is achieved through *partial matching* and *proof-failure*

[⋆] An earlier version of this paper appears in the informal proceedings of *AFM'10* [23].

*analysis*. While we focus here on Event-B, we believe the ideas that underpin reasoned modelling are generic with respect to posit-and-prove.

The paper is structured as follows: §2 provides background on Event-B along with our previous work on automated theory formation and reasoned modelling critics. Our refinement plans mechanism is described in §3 and an example of a refinement plan is presented in §4. The current implementation of the mechanism is outlined in §5, while §6 describes related and future work.

## 2 Background

### 2.1 Event-B refinement by example

An Event-B development is structured into *models* and *contexts*. A context describes the static part of a system, e.g. *constants* and their *axioms*, while a model describes the dynamic part. Models are themselves composed of three components: *variables*, *events* and *invariants*. Variables represent the state of the system, events are guarded actions that update the variables and invariants are constraints on the variables. By way of illustration we now consider the Event-B model shown in Figure 1. This model is a fragment of a flash-based file system developed in [9]. The fragment shown in Figure 1 deals with the function of writing the content of a file. In the abstract model, the event *writefile* is responsible for writing the content of file *f*, *wbuffer(f)*, into *fcontent* in an atomic step. In the concrete model the content is written one page at a time into a temporary storage *fcont_tmp* (event *w_step*) before being written to the actual storage *fcontent* (event *w_end_ok*). The new events *w_start* and *w_step* are said to refine *skip*, while event *w_end_ok* refines the abstract event *writefile*.

In order to prove that the refinement is indeed correct, invariants must be provided. In the example three invariants are specified in the concrete model, the two first invariants specify the type of the new variables *fcont_tmp* and *writing*, while the last invariant specifies a property of the refinement step, that is, that when the writing process starts for a given file, the content of *fcont_tmp* is a subset or is equal to the content of *wbuffer*.

### 2.2 Reasoned Modelling Critics

The notion of *reasoned modelling* (REMO) was first introduced in [19], where we described REMO *critics*. These critics are motivated by the way in which proof-failure analysis typically informs the activity of modelling – and is achieved by combining common patterns of proof failure with generic modelling guidance. The mechanism builds upon the notion of *proof critics* [18], a proof patching technique developed within the context of *proof planning* [4]. The key difference is that our REMO critics exploit failure at the level of modelling and proof. As a result, we reduce the burden that users experience in manually analysing low-level proof failures, presenting them instead with high-level modelling alternatives. These ideas were further developed in [20] where an implementation via the REMO tool, a prototype plug-in for the Eclipse-based Rodin toolset implemented in OCaml, is described. The work presented here aims to extend the REMO critics so as to generate modelling guidance at the level of refinement.

```
ABSTRACT MODEL:

Variables
  fcontent, w_opened_files, wbuffer, file_size,
  power_on, dateLastModified
Invariants
  w_opened_files ⊆ files
  file_size ∈ files → ℕ
  power_on ∈ BOOL
  fcontent ∈ files → CONTENT
  dateLastModified ∈ (files ∪ directories) → DATE
  wbuffer ∈ w_opened_files → CONTENT

Event writefile ≙
  any f
  where
   f ∈ w_opened_files
   power_on = TRUE
  then
   fcontent(f) := wbuffer(f)
   dateLastModified(f) := nowdate
   file_size(f) := card(wbuffer(f))
end

CONCRETE MODEL:

Variables
  fcontent, w_opened_files, wbuffer, file_size, power_on, dateLastModified, writing, fcont_tmp
Invariants
  writing ⊆ w_opened_files
  fcont_tmp ∈ writing → CONTENT
  ∀ f· f∈ writing ⇒ fcont_tmp(f) ⊆ wbuffer(f)

Event w_start ≙
  any f
  where
   f ∈ w_opened_files
   f ∉ writing
   power_on = TRUE
  then
   writing := writing ∪ { f}
   fcont_tmp(f) := ∅
end

Event w_step ≙
  any f i data
  where
   power_on = TRUE
   f ∈ writing
   i ∈ ℕ
   data ∈ DATA
   i↦ data ∈ wbuffer(f)
   i ∉ dom(fcont_tmp(f))
  then
   fcont_tmp(f) :=
   fcont_tmp(f) ∪ { i↦data}
end

Event w_end_ok ≙
refines writefile
  any f
  where
   f ∈ writing
   dom(fcont_tmp(f)) = dom(wbuffer(f))
   power_on = TRUE
  then
   fcontent(f) := fcont_tmp(f)
   dateLastModified(f) := nowdate
   file_size(f) := card(fcont_tmp(f))
   writing := writing \ { f}
   fcont_tmp := { f} ⩤ fcont_tmp
end
```

**Fig. 1.** Event-B model of a flash file system [9].

## 2.3 HRemo

HREMO [24] is an automatic approach to invariant discovery that builds upon HR [8], a machine learning system that performs descriptive induction to form a theory about a set of objects of interest which are described by a set of *core concepts*. Theories are constructed in HR via theory formation steps which attempt to construct new concepts, i.e. *non-core concepts*, through the use of a set of production rules and, if empirical relationships are found between concepts, formulate *conjectures* and evaluate the results. Thus, the theories HR produces contain concepts which relate the objects of interest, conjectures which relate the concepts; and proofs which explain the conjectures.

HREMO builds upon HR, animation and proof-failure analysis to automatically suggest candidate invariants of Event-B models. In particular, a set of heuristics are used to guide the search for invariants in HR. These heuristics exploit the strong interplay between modelling and reasoning in Event-B by using the feedback provided by failed POs to make decisions about how to configure

HR. Specifically, the approach consists of analysing the structure of failed POs to automate the:

1. Prioritisation in the development of conjectures about specific concepts.
2. Selection of appropriate production rules that increase the possibilities of producing the missing invariants.
3. Filtering of the final set of conjectures to be analysed as candidate invariants.

HRemo uses two classes of heuristics to constrain the search for invariants: those used in configuring HR, i.e. configuration heuristics, and those used in selecting conjectures from HR's output, i.e. selection heuristics. Using proof-failure analysis to prune the wealth of conjectures HR discovers, these heuristics have proven highly effective at identifying missing invariants. Further information about HRemo and examples of its application can be found in [24].

## 3 Refinement plans

Before providing details on the structure of refinement plans, we first sketch how we envisage they will be used within a development environment such as Rodin. Given a development, our approach provides a basis for classifying refinement steps against known patterns of refinement, i.e. syntactic features of abstract and concrete models.

However, we are interested in situations where a refinement step is flawed, and thus the proof tools fail to discharge some of the POs. In such situations our approach attempts to automatically generate guidance, i.e. modelling alternatives that overcome the failure. This is achieved by firstly identifying which of the known patterns are closely aligned to the given failed refinement. As well as a refinement pattern, each refinement plan is associated with a set of *critics* – where a critic represents a common pattern of failure at the level of POs and models. Moreover, associated with each critic is generic modelling guidance as to how to overcome the failure, e.g. invariant speculation, event speculation, etc. When a common pattern of failure is instantiated by a particular refinement step, the associated guidance will typically only be partially instantiated. To fully instantiate the guidance for a given flawed refinement requires in general additional search and reasoning – this is where we exploit HRemo.

| | **control refinement** | | | | **data refinement** | | | |
|---|---|---|---|---|---|---|---|---|
| **Model** | **RP_1** | **RP_2** | **RP_3** | **RP_4** | **RP_5** | **RP_6** | **RP_7** | **RP_8** |
| **Cars on a bridge** [1] | ✓✓ | | ✓✓ | ✓ | | | | |
| **Mondex** [6] | ✓ | ✓ | | ✓✓✓✓ | | ✓ | ✓ | ✓✓ |
| **Flash file system** [9] | | ✓✓✓ | ✓✓ | | ✓ | | | |
| **Location access ctrl.** [1] | | ✓ | | ✓ | ✓✓✓ | | | |
| **PLC**[*] | | | | ✓ | ✓✓ | ✓ | | |
| **Network topology** [15] | | ✓ | | ✓ | | | ✓✓ | ✓✓ |

[*]Available at `http://homepages.inf.ed.ac.uk/ggrov/`

**Table 1.** Refinement pattern analysis of Event-B case studies.

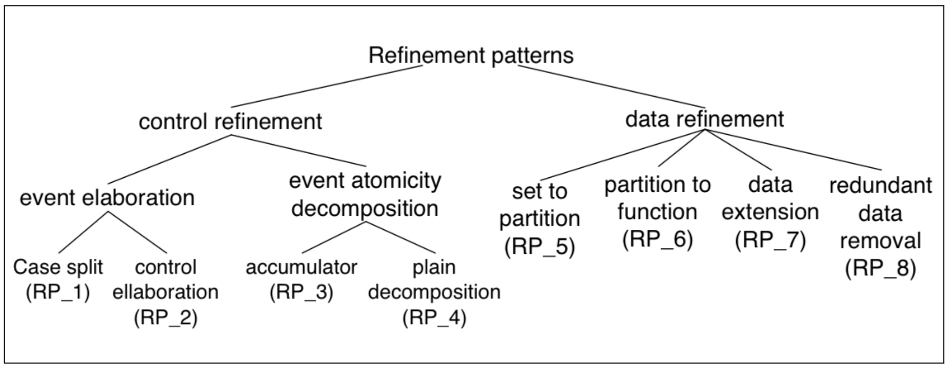


**Fig. 2.** A hierarchical classification of common refinement patterns.

Currently we have identified 8 basic refinement patterns by analysing a range of Event-B case studies from the literature. These patterns form a hierarchy as shown on Figure 2. Each leaf node denotes a distinct pattern of refinement, while the internal nodes reflect the sharing of properties between patterns. This classification provides us with a better understanding of what a user is trying to achieve in a refinement step as well as facilitates the matching process. The 8 basic patterns in Figure 2 are described briefly below:

**case split:** refers to refinement steps in which an abstract event is refined in the concrete model by two or more events.

**control elaboration:** relates to models that constrain the application of existing events based on extensions of the state and independently from the operation of new events at the concrete level.

**accumulator:** deals with models in which actions of an abstract atomic event are performed in the concrete model via iteration.

**plain decomposition:** makes reference to models in which an abstract event is refined by a sequence of new and refined events. New events are used to pre-process data used in the abstract event.

**set to partition:** refers to models in which an abstract variable is refined by partitioning it through a set of new variables in the concrete model.

**partition to function:** involves refinement steps in which an abstract partition of variables is refined into a function in the concrete model.

**data extension:** refers to models in which an abstract variable is refined into a concrete variable that extends the abstract data type in order to control membership of data in the variable.

**redundant data removal:** involves the elimination of data from the abstract level that is not being used to control the operation of any event.

The relation between this hierarchy and the case studies is given in Table 1. Currently we have explored in detail four refinement plans, i.e. *case split*, *accumulator*, *set to partition* and *partition to function*. Below in §4 we focus on the *accumulator refinement plan* and two of its associated critics.

```
ABSTRACT MODEL:

Variables: $V_1$, $V_2$

Event A $\widehat{=}$
  where
    ...
  then
    $V_1 := V_2$
end

CONCRETE MODEL:

Variables: $V_1$, $V_2$, $\overline{W}$
Invariants: $H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)$

Event $C_i$ $\widehat{=}$
  where
    $H_0$
  then
    $W_{a_i} := I_{a_i}$
end

Event $C_a$ $\widehat{=}$
  where
    $H_1$
  then
    $W_{a_i} := W_{a_i} \oplus \alpha$
end

Event $C_r$ $\widehat{=}$
  refines A
  where
    $H_1$
    $G_{=}(W_j, V_2)$
  then
    $V_1 := Wa_i$
end

Side conditions:
• $H_0 \Leftrightarrow \neg H_1$
```

**Fig. 3.** Accumulator plan – Modelling pattern.

## 4 The accumulator refinement plan

A technique for breaking up an atomic event has been proposed by Butler and Yadav in [6] and further developed in [5, 10, 11]. The accumulator refinement plan has been inspired by this work. The key difference with our work is that as well as the modelling patterns, we are also interested in the deductive patterns and in providing guidance when a pattern breaks in a development.

The accumulator pattern deals with models in which actions of an abstract atomic event are performed in the concrete model via iteration. This is achieved through the use of new events that iteratively accumulate the value from the abstract action. The modelling and PO patterns of the accumulator plan are shown in Figures 3 and 4, respectively. Note that we use the Vs and Ws to denote meta-variables, and specifically we use $I_{ai}$ to represent the initial value assigned to meta-variable $W_{ai}$. Note also that we use F, G and H to denote meta-predicates, where subscripts are used to restrict their instantiation, e.g. $G_=$ restricts G to be an equality. The key elements in the refinement are:
• The abstract model has an atomic event that is refined in the concrete model.
• A set of new variables $\overline{W} = \{W_1, ..., W_n\}$ are introduced.
• A subset of $\overline{W}$, $W_a$, which denotes accumulator variables. That is, for each $W_{a_i} \in W_a$ (where, $1 \leq i \leq n$) there is an accumulator event, i.e. the action pattern $W_{a_i} := W_{a_i} \oplus \alpha$ occurs, an initialisation event and a refined event.
• An initialisation event ($C_i$), accumulator event ($C_a$), and refined event ($C_r$).
• An invariant, $H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)$, that explains the refinement; i.e. that the content of the accumulator variable(s) is contained within the value assigned in the abstract model – the $\preceq$ symbol generalises the containment relationship.
• The initialisation, accumulator(s) and refined events must preserve the invariant, Figures 4(a), 4(b) and 4(c), respectively.
• The refined event must simulate the abstract action, Figure 4(d).

$$\begin{array}{l} H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2) \\ H_0 \\ \vdash \\ [W_{a_i} := I_{a_i}](H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)) \end{array}$$

(a) Init event (Inv. Preservation)

$$\begin{array}{l} H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2) \\ H_1 \\ \vdash \\ [W_{a_i} := W_{a_i} \oplus \alpha](H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)) \end{array}$$

(b) Accumulator event (Inv. Preservation)

$$\begin{array}{l} H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2) \\ H_1 \\ G_{=}(W_j, V_2) \\ \vdash \\ [V1 := W_{a_i}](H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)) \end{array}$$

(c) Refined event (Inv. Preservation)

$$\begin{array}{l} H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2) \\ H_1 \\ G_{=}(W_j, V_2) \\ \vdash \\ [V_1 := W_{a_i}](V_1 = V_2) \end{array}$$

(d) Refined event (Simulation)

$[x := e]F$ denotes the substitution of $x$ for $e$ in $F$ – and is a result of the before-after predicate [1] associated to an event.

**Fig. 4.** Accumulator plan – PO patterns.

An instance of the accumulator pattern occurs in the model presented in Figure 1, in which the action:

$$\mathit{fcontent}(f) := \mathit{wbuffer}(f)$$

within the abstract event *writefile* is achieved within the concrete model via iteration. Below we present the fragments of the events that match the modelling pattern at the concrete level:

```
Event w_start ≙
  any f
  where ...
   f ∉ writing
  then ...
   fcont_tmp(f):= ∅
   writing:=writing∪{f}
end
```

```
Event w_step ≙
  any f i data
  where ...
   f ∈ writing
  then
   fcont_tmp(f) :=
     fcont_tmp(f)∪{i↦data}
end
```

```
Event w_end_ok ≙
refines writefile
  any f
  where ...
   f ∈ writing
   dom(fcont_tmp(f))=dom(wbuffer(f))
  then ...
   fcontent(f) := fcont_tmp(f)
end
```

Note that variable *fcont_tmp* acts as the accumulator variable. Event *w_start* initialises the process by assigning the empty set to *fcont_tmp* and adding file *f* to the *writing* state, event *w_step* iteratively adds the content of each page to the accumulator variable, and event *w_end_ok* assigns the content of *fcont_tmp* to *fcontent* after all the pages have been written. Finally, the invariant:

$$\forall f \cdot f \in \mathit{writing} \Rightarrow \mathit{fcont\_tmp}(f) \subseteq \mathit{wbuffer}(f)$$

specifies that while file *f* is in the writing state, the value of *wbuffer(f)* is accumulated in *fcont_tmp(f)*.

### 4.1 Accumulator refinement plan critics

We now focus on the critics aspect of refinement plans, and how partial matching, with respect to the modelling pattern, and failure analysis are used to automatically generate modelling guidance.

```
ABSTRACT MODEL:

Variables                   Event incr ≙
   x y                        then
Invariants                      x := x + y
   x ∈ ℕ                    end
   y ∈ ℕ

CONCRETE MODEL:

Variables                Event start ≙     Event step ≙              Event end_ok ≙
 y x n x_tmp flag          when              when                    refines incr
Invariants                  flag = TRUE       n < y                    when
 n ∈ ℕ                     then               flag = FALSE              flag = FALSE
 x_tmp ∈ ℕ                  n := 0           then                      then
 flag ∈ BOOL                x_tmp := x        x_tmp := x_tmp + 1        x := x_tmp
                            flag := FALSE     n := n + 1                flag := TRUE
                         end               end                       end
```

**Fig. 5.** Flawed accumulator plan instance – Addition example

We have identified a number of critics for the accumulator plan:

**postGuard_speculation critic:** considers the case when the guard of the refined event that ensures the accumulation process is complete is either flawed or missing.

**invariant_speculation critic:** handles the case when the accumulator invariant is wrong or missing.

**accumulator_speculation critic:** handles the case when an accumulator event refines an abstract event whose actions are performed in an atomic step.

**initialisation_speculation critic:** considers the case when the accumulation process does not have an initialisation phase.

**loopGuard_speculation critic:** deals with the case when the guard(s) that deal with the loop in the accumulator event is wrong or missing.

**guard_relocation critic:** deals with guards from the abstract event that need to be moved to a new event in the accumulation sequence.

Due to space constraints we only present two critics: *postGuard_speculation* and *invariant_speculation*. In order to illustrate the application of these critics we will use a simple model that adds a value to a variable. The model, taken from [9], is shown in Figure 5. The running example of the flash file system, Figure 1, is not used because it is not possible to perform the simulation of this model through the ProB animator and animation is a key component of the critics presented. We give more information about these limitations in §6.

The abstract model in Figure 5 shows an atomic event *incr* that increments the value of $x$ by the value of $y$. In the concrete model the value of $y$ is iteratively assigned in event *step* to an accumulator variable *x_tmp*, while in the event *end_ok* the value of *x_tmp* is assigned to the abstract variable $x$ after the accumulation has finished. Event *start* initialises the accumulation. Note that variable $n$ is a new variable used to control the accumulation process. Note also that the accumulator invariant as well as the post-guard are missing from the model; this gives rise to the following failed SIM PO associated to event *end_ok*:

**end_ok/SIM PO:** flag = FALSE $\vdash$ x_tmp = x + y

At this point the *postGuard_speculation* and *invariant_speculation* critics are triggered. First the critic that deals with the guard is applied because in order to reason about the invariant, the events in the model need to be correct.

**Preconditions for the postGuard_speculation critic:**

**P1.** *An accumulator pattern is identified.*
This precondition holds for the addition model since a partial match of the accumulator pattern is detected. That is, apart from the invariant and the guard, the other key elements of the pattern are identified in the model.
**P2.** *The simulation PO pattern associated to the refined event fails.*
This precondition holds since the end_ok/SIM PO fails.
**P3.** *The post-accumulator guard is missing or it is not compatible with the guard pattern, i.e. $G_=(W_j, V_2)$.*
As mentioned above, the post-accumulator guard is missing from the model in Figure 5; therefore this precondition holds.

**Guidance:**
*A guard with the shape $G_=(W_j, V_2)$ must be added to the refined event.*
As preconditions P1, P2 and P3 succeeded, the guard pattern is instantiated. The guidance is then to add a guard to event *end_ok* with the form:

$$G_=(x\_tmp, n, x, y)$$

We will revisit this guard schema below, and describe how it is instantiated. For now assume that the correct instantiation is available, i.e. $y = n$. Because the invariant is also missing, the failure persists, this triggers the invariant critic.

**Preconditions for the invariant_speculation critic:**

**P1.** *An accumulator pattern is identified.*
This precondition succeeds as explained for the guard critic.
**P2.** *The SIM PO pattern associated to the refined event fails.*
This precondition holds since the end_ok/SIM PO fails. The new form of the failed PO is:

flag = FALSE, y = n $\vdash$ x_tmp = x + y

**P3.** *The post-accumulator guard is not missing and it is compatible with the guard pattern, i.e. $G_=(W_j, V_2)$.*
The post-accumulator guard $y = n$ is present in the refined event and is compatible with the pattern.
**P4.** *The accumulator invariant is missing or it is not compatible with the invariant pattern, i.e. $H_1 \Rightarrow F_\preceq(\overline{W}, V_2)$.*
As mentioned above, the accumulator invariant is missing from the model; therefore, this precondition holds.

**Guidance:**
*An invariant of the shape $H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)$ must be added to the concrete model.* As with the guard critic, preconditions P1 to P4 succeeded; therefore the invariant pattern is instantiated as follows (where due to use of natural numbers $\preceq$ is instantiated to $\leq$):

$$(flag = FALSE) \Rightarrow F_{\leq}(x\_tmp, n, x, y)$$

As can be observed the guidance currently provided is in the form of partial instantiations of the schemas. At this point, there are three options to find the correct instantiation: i) through interaction with the user, ii) through the use of proof patterns, or iii) through the use of automated theory formation (ATF).

Here we use ATF, and in particular the HRemo system to search for the missing invariants and guards. However, currently HRemo cannot be used to analyse models where the events are incorrect. This prevents us from using HRemo directly to discover missing guards. On the contrary, HRemo can be used to discover missing invariants. However, with regards to the invariant schema given above, HRemo on its own fails to find the missing invariant after 1000 theory formation steps, which give rise to 7959 conjectures. This does not imply that the invariant cannot be found, rather it means that additional search is required. In the next section we show that by combining refinement plans and event error traces with HRemo these negative issues can be effectively addressed.

### 4.2 Combining modelling patterns with HRemo

The process of finding a "correct" refinement typically involves exploring many incorrect models. Refinement plans aim at providing guidance when a failed refinement is closely aligned with a known pattern. However, as shown through the guidance obtained by the critics presented in §4.1, refinement plans are limited by the patterns observed. On the other hand, as mentioned above, HRemo also exhibits some limitations. In order to overcome these limitations we combine both approaches, in particular we extend the work presented in [24] by:

– using the ProB animator [22] to generate traces that contain undesirable states which can be used by HRemo to find missing guards, and
– using the patterns of invariants and guards available in the refinement plans to automatically tailor the search in HRemo.

As mentioned in §2.3, two type of heuristics are used by HRemo, configuration heuristics (CH) and selection heuristics (SH), when a pattern of an invariant or a guard is available then the following heuristics are applied:

***Configuration heuristics:***

**CH1.** *Prioritise core and non-core concepts expected in the invariant or guard.*
**CH2.** *Follow with core and non-core concepts that occur within failed POs.*
**CH3.** *Generate conjectures that are compatible with the type of the expected invariant or, if looking for a guard, generate only equivalence conjectures.*

**CH4.** *Select only production rules which will give rise to conjectures relating to the type of the expected invariant or guard.*

*Equivalence conjectures are always generated since this optimises the theory formation process* [8].

The selection heuristics for the search of invariants based on patterns are the same than those applied in [24]. This requires selecting conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint, selecting the most general conjectures, and selecting the conjectures that discharge the failed POs and that minimise the number of additional proof failures. Note that here the selection of conjectures is focused in the core and non-core concepts that relate to the invariant pattern, as opposed to [24] which focused on core and non-core concepts from the failed POs.

In the case of missing guards the selection process differs. Through the use of the ProB animator it is possible to detect event errors which result in traces that contain undesirable states. That is, ProB can animate various refinement levels concurrently, allowing the detection of errors associated with refinement; in particular, ProB can detect violation of guard strengthening in a refined event, we exploit this animation analysis provided by ProB to tailor HRemo in the search of guards. When a trace of this type is generated we provide HRemo with the concept of *good* states, which are the steps of the trace with no guard strengthening errors associated. The selection is then focused on conjectures that express equivalences with the concept of *good*, i.e. conjectures of the form:

$$good \Leftrightarrow \phi$$

where $\phi$ represents the potential missing guard.

Regarding the *postGuard_speculation* and *invariant_speculation* critics, presented in §4.1, the guidance is achieved by using the partially instantiated guard and invariant schemas to tailor HRemo in the search. To illustrate, lets revisit the instantiated guard schema obtained by the *postGuard_speculation* critic:

$$G_{=}(xtmp, n, x, y)$$

based on this, we instantiate the configuration heuristics as follows:

**CH1:** Prioritised concepts from the guard schema: $x\_tmp$, $n$, $x$ and $y$ .
**CH2:** Concepts from the failed POs: *flag*, *x+y*, *x_tmp=x+y* and *flag=FALSE*.
**CH3:** Searching for a guard; thus, only equivalence conjectures are generated.
**CH4:** As the top-level symbol in the guard is = and the involved variables are natural numbers, the *numrelation* and *arithmetic* PRs are selected.

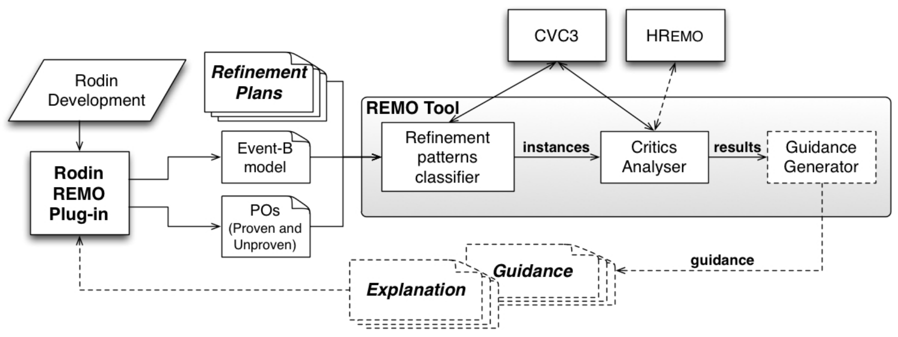
After 65 seconds and 1000 theory formation steps HRemo returns 1 conjecture:

$$good \Leftrightarrow y = n$$

which means that the missing guard is y = n. A similar approach is followed in the search for the missing invariant. After 45 seconds and 1000 theory formation steps HRemo returns 1 conjecture:

$$flag = FALSE \Rightarrow x\_tmp = x + n$$

which represents the missing invariant.



The Rodin Remo plug-in provides the interface between the Rodin toolset and the Remo tool. The plug-in generates two files, i) and Event-B model, and ii) the associated POs. The refinement plans are also given as an input to the tool. The Refinement Patterns Classifier uses the refinement plans to classify the patterns of refinement used in a development. The role of the Critics Analyser is to find ways of overcoming failures. This is achieved via the critics mechanism which analyses the instances given by the classifier together with the POs. The classifier and the analyser interact with CVC3 when a precondition requires proof. The analyser also interacts with HRemo in order to search for missing/wrong invariants or guards. The results of this analysis are passed to the Guidance Generator which takes the raw results from the analyser and produces a formatted, ranked (ordered) list of alternative guidance suggestions, which will then be sent to the Rodin Remo plug-in and presented to the user. Note that the stippled lines indicate work in progress.

**Fig. 6.** The Remo tool architecture.

## 5 Implementation & results

We have implemented and tested the *set to partition* and *partition to function* refinement plans. Moreover we have conducted the experiments described above with the *accumulator* plan. This implementation effort was partially integrated into the Remo toolset, which we mentioned in §2.2. An architectural view of the implementation is given in Figure 6. The prototype is partial in that the integration of the guidance from Remo back into Rodin is still under development. Note that in terms of results, our implementation is still at the experimental stage, and we are now looking to undertake more extensive testing (see §6).

## 6 Related and future work

The motivation behind the work described here is to correct a refinement which almost matches an existing pattern. Similar tools and techniques we are familiar with – such as the BART tool for classical B [26]; the ZRC refinement calculus for Z [7]; and more relevant, Event-B based tools and techniques as described in [16, 17, 2, 12] – instead focus on automating the refinement from a given step to a more concrete step. None of the tools can handle the failure-analysis we have described here.

Our implementation of the refinement plans highlighted in this paper is ongoing. We plan to automate the link with HRemo and the external theorem prover(s) as well as to automate the communication of the results from Remo back to the user[3]. We also plan to further test and develop our existing plans, drawing upon industrial case studies arising from the DEPLOY project[4]. We are also interested in exploring the potential for using machine learning techniques to automate the discovery of new plans.

Animation traces from ProB are used in the analysis phase of our work-flow. Such information about how the events relate to each other should naturally be part of the pre-conditions of plans and critics, and we will extend them with such information. Bendisposto and Leuschel [3], have developed a tool which turn ProB traces into a more abstract *flow graphs* which shows the order events may be executed[5]. We plan to add support for such "event flow" information in the preconditions, either as described in [3], or ideally extended with support for infinite systems ([3] only supports finite models), which undoubtedly will require theorem proving support.

Finally, animation is key to our approach, where the quality of the invariants produced by HRemo strongly depends on the quality of the animation traces. We believe that increasing the randomness in the production of the traces is an area where the ProB animator requires improvement. Specifically, this limitation arose during our analysis of the Mondex [6] case study.

## 7 Conclusions

We have described refinement plans, a technique which provides automatic modelling guidance for users of posit-and-prove style formal refinement. Building upon common patterns of refinement, the technique uses an automated analysis of refinement failure at the level of models and POs in order to focus the search for modelling guidance. To provide flexibility in terms of the guidance that can be generated, we have experimented with the HRemo theory formation tool. Through these experiments we have shown that combining refinement plans with HRemo improves the search for invariants and has suggested how missing guards can be discovered automatically.

**Acknowledgements** Thanks to Alison Pease, Simon Colton, Julian Gutierrez, Alan Bundy and the Mathematical Reasoning Group at Edinburgh University. This work was supported by EPSRC grants EP/F037058, EP/H024204, EP/E005713, EP/E035329, EP/J001058. Maria Teresa Llano was also supported by a BAE systems studentship. Finally, we thank the anonymous ABZ reviewers for their constructive feedback.

[3] One possible route is via Lopatkin's *transformation patterns* plug-in. For details see `http://wiki.event-b.org/index.php/Transformation\_patterns`.

[4] See `http://www.deploy-project.eu/`

[5] Hallerstede [14] suggests an approach achieving a similar goal, but here the user has to add more structure to the model.

## References

1. J.-R. Abrial. *Modelling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. J.-R. Abrial and T. S. Hoang. Using Design Patterns in Formal Methods: an Event-B Approach. In *ICTAC*, LNCS **5160**, 2008.
3. J. Bendisposto and M. Leuschel. Automatic flow analysis for event-B. In *FASE*, LNCS **6603**, 2011.
4. A. Bundy. A science of reasoning. In *Computational Logic: Essays in Honor of Alan Robinson.* MIT Press, 1991.
5. M. Butler. Decomposition structures for Event-B. In *iFM*, LNCS **5423**, 2009.
6. M. Butler and D. Yadav. An incremental development of the mondex system in Event-B. *Formal Aspects of Computing*, 20(1), 2008.
7. A. Cavalcanti and J. Woodcock. ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3), 1998.
8. S. Colton. *Automated Theory Formation in Pure Mathematics.* Springer, 2002.
9. K. Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B.* PhD thesis, University of Southampton, 2010.
10. A. S. Fathabadi and M. Butler. Applying Event-B atomicity decomposition to a multi media protocol. In *FMCO*, LNCS **6286**, 2010.
11. A. S. Fathabadi, A. Rezazadeh, and M. Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In *NFM*, LNCS **6617**, 2011.
12. A. Fürst. Design Patterns in Event-B and Their Tool Support. Master's thesis, ETH Zürich, 2009.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
14. S. Hallerstede. Structured Event-B models and proofs. In *ABZ*, LNCS **5977**, 2010.
15. T. S. Hoang, D. Basin, H. Kuruma, and J.-R. Abrial. Development of a network topology discovery algorithm. DEPLOY project Repository. Available at http://deploy-eprints.ecs.soton.ac.uk/82/.
16. A. Iliasov. Refinement Patterns for Rapid Development of Dependable Systems. In *EFTS*. ACM Press, 2007.
17. A. Iliasov. *Design Components.* PhD thesis, University of Newcastle, 2008.
18. A. Ireland. The use of planning critics in mechanizing inductive proofs. In *LPAR*, LNCS **624**, 1992.
19. A. Ireland, G. Grov, and M. Butler. Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. In *ABZ*, LNCS **5977**, 2010.
20. A. Ireland, G. Grov, M. Llano, and M. Butler. Reasoned modelling critics: turning failed proofs into modelling guidance. In *Science of Computer Programming.* Elsevier, 2011. (In Press).
21. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.
22. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, LNCS **2805**, 2003.
23. M. Llano, G. Grov, and A. Ireland. Automatic guidance for refinement based formal methods. In *AFM workshop*, 2010.
24. M. T. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. In *Refine workshop*, EPTCS **55**, 2011.
25. C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.
26. A. Requet. BART: A tool for automatic refinement. In *ABZ*, LNCS **5238**, 2008.