Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 00 (2010) 1–26
Science of Computer Programming

Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance

Andrew Ireland[1], Gudmund Grov[2], Maria Teresa Llano[1] and Michael Butler[3]

[1]*School of Mathematical and Computer Science, Heriot-Watt University, UK*
[2]*School of Informatics, University of Edinburgh, UK*
[3]*School of Electronics and Computer Science, Southampton University, UK*

Abstract

The activities of formal modelling and reasoning are closely related. But while the rigour of building formal models brings significant benefits, formal reasoning remains a major barrier to the wider acceptance of formalism within design. Here we propose *reasoned modelling critics* – an approach which aims to abstract away from the complexities of low-level proof obligations, and provide high-level modelling guidance to designers when proofs fail. Inspired by proof planning critics, the technique combines proof-failure analysis with modelling heuristics. Here, we present the details of our proposal, implement them in a prototype and outline future plans.

Keywords: Reasoned modelling, formal methods, formal modelling, formal verification, automated reasoning, artificial intelligence

1. Introduction

The use of mathematical techniques for system-level modelling and analysis brings significant benefits, as well as challenges. While the rigour of mathematical argument can offer early feedback on design decisions, a key challenge centres on *how* feedback derived from the analysis of complex *proof obligations* (POs) can be used to improve design decisions. Such feedback currently requires user intervention, drawing upon skilled knowledge of the subtle interplay that exists between modelling and reasoning. For example, consider a simple cruise control system with the safety requirement that the brakes (*brake*) cannot be on while the cruise control (*cc*) is enabled. This can be formalised as the following invariant:

$$cc = on \Rightarrow brake = off$$

When the driver applies the brakes, i.e. $brake := on$ it must be shown that the safety requirement (invariant) is preserved, creating a PO of the form:

$$brake = off, cc = on \;\vdash\; on = off$$

Note that this PO is false. An (unsafe) solution to this failure would be to restrict the application of the brakes via a guard, i.e. only allow the brakes to be applied when the cruise control is disabled ($cc = off$). Clearly, the desirable solution requires the introduction of an auto disable mechanism, i.e. the cruise control is disabled if the brakes are applied ($cc := off\,;\; brake := on$). While choosing between the two alternatives do not represent a significant burden to a designer, in general many solutions may arise from proof-failure analysis. In order to make good use of a designer's time, modelling heuristics are needed in order to rank the alternatives.

Here we describe an approach to formal modelling that combines proof-failure analysis with modelling heuristics. The basic idea is to use automatic analysis of failed proofs, constrained by modelling heuristics, to generate modelling suggestions. In this way we abstract away from complexities of low-level POs, and automatically provide designers with high-level decision support oriented towards modelling choices. In turn this will increase the productivity of designers as well as the accessibility of formal modelling tools. The approach has been implemented through what we call *reasoned modelling critics*.

A key inspiration for our proposal is *proof planning*, a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [1]. Central to proof planning is its proof-failure analysis and proof patching capabilities [2]. Our approach builds directly upon these features. Specifically by combining the proof-failure analysis capabilities with modelling heuristics in order to automatically generate modelling guidance.

Our longer term aim is to combine the proof plan representation with a complementary notion of *modelling patterns*, which we call *reasoned modelling methods*. This, combined with reasoned modelling critics, are the building blocks of what we call *reasoned modelling*: the study of the interplay between formal reasoning and modelling.

While our vision is generic, our starting point is Event-B, which we motivate and introduce together with proof planning in Section 2. Section 3 outlines the reasoned modelling critics mechanism and shows its application through examples. Section 4 discusses the prototype implementation of the reasoned modelling critics mechanism, and Section 5 shows the results of applying the examples discussed in Section 3 to the prototype. We discuss related and future work in Section 6 and conclude in Section 7.

This paper is an extension of [3]. Here, in addition to providing more explanation of the concepts, we provide a prototype implementation (see Section 4) of the ideas discussed in Sections 3.1 and 3.3.

2. Background

2.1. Event-B and the Rodin toolset

Event-B provides a formal framework for modelling discrete complex systems [4], and is mechanised through the Rodin toolset [5]. Event-B promotes an incremental style of formal modelling, where each step of a development is underpinned by formal reasoning. As a result, there is strong interplay between modelling and reasoning and this is partly supported by the Rodin toolset. This interplay requires skilled user interaction, *i.e.* typically a user will analyse failed proofs, and translate the analysis by hand into corrective actions at the level of modelling. This is exemplified in e.g. [4, 6]. Typical corrective actions include strengthening invariants and guards or modifying actions. Our aim is to provide high-level decision support, by automating the generation, filtering and ranking of modelling suggestions. Event-B models and POs are closely aligned [4], whilst Rodin [7] is an extensible framework. Event-B and the Rodin toolset thus represent a unique opportunity for us to investigate reasoned modelling.

We will use the term *models* for both Event-B *contexts* (the static part) and *machines* (the dynamic part). A machine includes invariants and the Rodin tool generates POs to ensure that all invariants are maintained by each event of a machine. Event-B also supports refinement of machines and Rodin generates POs to ensure the consistency of refinement. For details of the POs see [5]. We will only use the basic features of Event-B and use standard notation:

$$\textbf{EVENT}\ \langle \textit{name} \rangle \mathrel{\widehat{=}} \textbf{BEGIN}\ \langle \textit{action} \rangle\ \textbf{END}$$
$$\textbf{EVENT}\ \langle \textit{name} \rangle \mathrel{\widehat{=}} \textbf{WHEN}\ \langle \textit{guard} \rangle\ \textbf{THEN}\ \langle \textit{action} \rangle\ \textbf{END}$$

where the event is only executed when the guard holds. Moreover, the term $\langle \textit{action} \rangle$ is a list of simultaneous assignments to machine variables, while INITIALISATION is a special event without guards defining the initial state.

2.2. Proof planning and proof critics mechanism

Proof planning builds upon the tactic-based tradition of theorem proving, where primitive proof steps are packaged-up into programs known as *tactics*. Starting with a set of general purpose tactics, plan formation techniques are used to construct a customised tactic for a given conjecture. The search for a customised tactic is constrained by a set of *methods*, collectively known as a *proof plan*. Using preconditions, each method specifies the applicability of a general purpose tactic. Having explicit preconditions provides insights when proof planning fails. The proof *critics* mechanism [2] enables "interesting" failures at the level of precondition evaluation to be represented. For a given method a number of critics will typically exist. Each critic represents an alternative generic proof patch, e.g. conjecture generalisation. The analysis of a specific failure, and subsequent proof planning, provides guidance in the selection and instantiation of a generic proof patch. The method and critic schemas are presented in Figure 1.

method (name)	**critic (name)**
INPUT: *PO*	INPUT: *POs*
PRECONDITIONS:	PRECONDITIONS:
precondition 1	*precondition 1*
...	...
precondition 2	*precondition 2*
EFFECTS:	PATCH:
generation of output POs	*proof patch*
OUTPUT:	
POs	

Methods: A method takes as input a PO. The applicability of a method is determined by evaluating its associated preconditions. If the preconditions of a method succeed, then the output POs (potentially empty) are computed via the effects slot.

Critics: During proof planning, both the success and failure of a method's preconditions are recorded. In particular, when a method fails a set of partial instantiations of the preconditions are associated with the PO. The critics mechanism uses the record of partial successes to search for patchable exceptions to the method. If the pattern represented by a given critic's preconditions matches with a recorded failure pattern then the associated patch is applied. In general, a critic may require multiple failures in order to trigger a patch. In this way, critics can apply both local and global proof-failure analysis.

Figure 1: Proof method and critic schemas

To illustrate the proof patching aspect of proof planning, consider the following PO:

$$k : \tau,\ h : \tau,\ t : list(\tau),\ l : list(\tau) \vdash k \in [h|append(t, l)] \tag{1}$$

where $\in$ denotes list membership, while $[\_|\_]$ and *append* denote the constructor and concatenation function for lists. Moreover, assume the definition of $\in$ gives rise to the following rewrite rules:

$$X = Y \Rightarrow X \in [Y|W] \quad :\Rightarrow \quad true \tag{2}$$

$$X \neq Y \Rightarrow X \in [Y|W] \quad :\Rightarrow \quad X \in W \tag{3}$$

Given these rewrite rules, (1) is not directly provable. That is, a case split is required in order to allow the proof to proceed. The instances of the method and critic schemas required for this example are given in Figure 2. We now consider the **rewrite** method preconditions more closely i.e.

1. there exists an expression L at position *Pos* within the goal G,
2. there exists a rewrite rule such that L matches the left-hand side of the rule, and
3. any condition C attached to the rule is provable within the proof context.

Note that based upon rewrite rule (3), preconditions 1 and 2 succeed, with C, L and R being instantiated to be $k \neq h$, $k \in [h|append(t, l)]$, and $k \in append(t, l)$ respectively. However, precondition 3 fails, i.e. the condition $k \neq h$ is not provable from the given hypotheses. This pattern of partial success triggers the **casesplit** critic given in Figure 2, i.e.

```
method (rewrite)
  INPUT: $\Delta \vdash G$
  PRECONDITIONS:
      1. $exp\_at(G, Pos, L)$
      2. $rewrite\_rule(C \Rightarrow L :\Rightarrow R)$
      3. $provable(\Delta \vdash C)$
  EFFECTS:
      $replace\_at(Pos, R, G, NewG)$
  OUTPUT:
      $[\Delta \vdash NewG]$
```

```
critic (casesplit)
  INPUT: $POs$
  PRECONDITIONS:
      1. $\exists failed\_po \in \{\langle rewrite, \_, PO\rangle \in POs |$
            $failed\_proof(PO)\}$.
            $failed\_po = \langle \_, Pre, \_\rangle$
      2. $\langle$ $exp\_at(\_, \_, \_)$,
            $rewrite\_rule(C \Rightarrow \_ :\Rightarrow \_)$,
            $fail \rangle \in Pre$
  PATCH:
      $insert\_casesplit(C, PO)$
```

The meta-predicates used above are defined in Figure 4. Note that Δ denotes a list of hypotheses, while G and $NewG$ denote single goal formula. Note also the use of Prolog meta-variables, in particular the use of "_" for anonymous variables. The scope of meta-variables extends across all slots of the method and critic schemas.

Figure 2: An example proof method and critic

it associates the failure (*fail*) of precondition 3 with a casesplit patch. The application of the patch gives rise to the following two POs:

$$k : \tau,\ h : \tau,\ t : list(\tau),\ l : list(\tau),\ k = h\ \vdash k \in [h|append(t, l)] \tag{4}$$

$$k : \tau,\ h : \tau,\ t : list(\tau),\ l : list(\tau),\ k \neq h\ \vdash k \in [h|append(t, l)] \tag{5}$$

Note that the **rewrite** method is applicable to both POs, i.e. rewrite rule (2) applies to (4) while rewrite rule (3) applies to (5). We should stress, that in terms of proof patching, this is a deliberately simple example – its role is purely to illustrate proof methods and the basic critics mechanism.

Proof patching has been applied successfully to the problems of inductive conjecture generalisation and lemma discovery [8], as well as loop invariant discovery [9]. Proof critics have also been developed for *abductive reasoning*, i.e. the speculation of hypotheses in order to explain a consequence, and applied to the problem of patching faulty conjectures [10]. The proof critics mechanism provides the starting point for reasoned modelling critics. Moreover, in terms of exploiting existing critics, we see the work on abduction playing an important role in our reasoned modelling critics. This is elaborated upon in Section 5.

3. From proof critics to reasoned modelling critics

As described above, our reasoned modelling critics will extend the existing proof critics mechanism. This extension will involve two key components:

1. In order to combine proof and modelling, our new critics need access to models as well as POs. In addition, critics will now have the option of providing modelling guidance, as well as proof patches.
2. Both methods and critics are associated with preconditions. We need to extend the meta-language of critics to allow us to represent preconditions which specify modelling heuristics.

To support access to models and modelling guidance, an extended critic schema will be required. Our proposed extension is described in Figure 3. In order to illustrate the new schema and the kind of meta-language extensions we propose for reasoned modelling, we develop below some critics and show their application to simple control system problems.

critic (*name*)
INPUTS:
    PO_SET *POs*
    MODEL_SET *Ms*
PRECONDITIONS:
    precondition 1
    ...
    precondition N
OUTPUTS:
    PATCH *proof patch (optional)*
    GUIDE *modelling guidance (optional)*

The above schema represents our proposed refinement to the proof critic schema:

- The INPUT slot has been extended to include models, as well as proof obligations. Note that the schema allows for a critic to access multiple models. As a consequence, critics can be developed that consider the internal consistency of individual models as well as refinements and decompositions.
- The declarative PRECONDITIONS determine the applicability of the critic. Preconditions have access to proof obligations as well as models. Note that these PRECONDITIONS can be seen as guards in the way this notion is used in B/Event-B. However, our use of the term PRECONDITIONS is more closely related to proof planning (and in general AI planning) syntax, from which our work is based on.
- The OUTPUT slot has been extended to include modelling guidance (GUIDE), as well as proof patches (PATCH). While proof patches are automatically applied, it is envisaged that modelling guidance will be communicated to the designer so as to inform their decision making.

As with the original proof critics mechanism, the output slot becomes instantiated as a side-effect of the instantiation of the input and precondition slots.

Figure 3: Reasoned modelling critic schema

3.1. Guidance for invariant proof failures via local analysis

Typically, a proof failure arising from an inconsistency in a model may be overcome in a number of ways. For example, consider an invariant which takes the form of an implication, i.e. $X \Rightarrow Y$. If a corresponding invariant proof fails, then the failure may be overcome by revising the model so that either,

1. X is false, or
2. Y is true.

At the level of modelling these effects can be achieved by changing the guards or actions associated with the invariant proof, or the invariant itself. Here we focus on *local* changes to an event, i.e. changes to guards and actions, and delay the discussion of global analysis and invariant changes until Section 3.3. In general, proof-failure analysis will generate a large number of proof patches. But not all proof patches will make sense in terms of modelling. To address this problem we wish to exploit modelling knowledge in order to rank the proof patches that are offered as guidance. To achieve this we envisage designers providing meta-data in addition to their models. The preconditions of our new critics can then exploit modelling heuristics as well as proof-failure heuristics.

To illustrate this idea, we develop two critics below. Each critic targets a different pattern of invariant proof failure, as outlined above. In addition, the critics exploit a notion of variable priority – which is based upon the observation that not all variables within a model may carry equal importance. Crucially, such "meta-data" must be supplied by the designer. For example, being able to brake is clearly more important than driving with the cruise control enabled. This

$exp\_at(X, Y, Z)$	$\widehat{=}$	*“Z is the subexpression at position Y within expression X”*
$rewrite\_rule(X \Rightarrow Y :\Rightarrow Z)$	$\widehat{=}$	*“A conditional rewrite rule: If X holds then Y is rewritten to Z”*
$insert\_casesplit(C, P)$	$\widehat{=}$	*“Progress the proof of P by a casesplit on X”*
$replace\_at(W, X, Y, Z)$	$\widehat{=}$	*“Z is obtained by replacing the subexpression at position W within Y by X”*
$dom(\{V_1 \mapsto D_1, \cdots, V_n \mapsto D_n\})$	$\widehat{=}$	$\{V_1, \cdots, V_n\}$
$disjoint\_sub(S_1, S_2)$	$\widehat{=}$	$dom(S_1) \cap dom(S_2) = \varnothing$
$failed\_proof(P)$	$\widehat{=}$	*“P is a PO which is provably false or has failed to be proven”*
$provable(P)$	$\widehat{=}$	*“P is a PO that is provable”*
$max(A)$	$\widehat{=}$	$\iota\, x.\, (x \in A \wedge \forall y \in A.\, y \leq x)$
$pri\_var(V, M)$	$\widehat{=}$	*“Priority of variable V within model M”*
$priority(S, M)$	$\widehat{=}$	$max(\{pri\_var(v, M) \mid v \in dom(S)\})$
$sub2act(\{V \mapsto D\})$	$\widehat{=}$	$V := D$
$sub2act(\{V_1 \mapsto D_1, \cdots, V_n \mapsto D_n\})$	$\widehat{=}$	$sub2act(\{V_1 \mapsto D_1\}) \parallel \cdots \parallel sub2act(\{V_n \mapsto D_n\})$
$sub2grd(\{V \mapsto D\})$	$\widehat{=}$	$V = D$
$sub2grd(\{V_1 \mapsto D_1, \cdots, V_n \mapsto D_n\})$	$\widehat{=}$	$sub2grd(\{V_1 \mapsto D_1\}) \wedge \cdots \wedge sub2grd(\{V_n \mapsto D_n\})$
$guards(E)$	$\widehat{=}$	*“Conjunction of guards of event E”*
$sat(P)$	$\widehat{=}$	*“The predicate P is satisfiable”*
$generalisable(H)$	$\widehat{=}$	$\exists r, e_1, e_2, g_1, g_2.\, \langle e_1, g_1\rangle \in H \wedge \langle e_2, g_2\rangle \in H \wedge e_1 \neq e_2 \wedge r \Rightarrow g_1 \wedge r \Rightarrow g_2 \wedge sat(r)$
$generalise(H)$	$\widehat{=}$	*“The weakest r such that sat(r) and for the most ⟨e, g⟩ ∈ H, r ⇒ p (for at least 2 events)”*
$add\_guard(G, E, M)$	$\widehat{=}$	*“Adds guard G to event E of model M”*
$add\_action(A, E, M)$	$\widehat{=}$	*“Adds action A to event E of model M”*
$add\_invariant(I, M)$	$\widehat{=}$	*“Adds invariant I to model M”*

S denotes a substitution, H denotes a set of event-guard pairs, and D is an expression. ι is the definite description operator, meaning that *max* returns the largest element of the given set

Figure 4: Meta-terms for reasoned modelling critics

notion of *priority* can be expressed as a heuristic, and used to rank or even filter modelling suggestions. Informally, where proof-failure analysis suggests that the value of a variable should be changed within the context of a given event, we adopt the following heuristics:

H1: If the priority of the candidate variable is lower than the priorities of all the variables updated by the event, then it is strongly suggestive that the change should be achieved via a new action.

H2: If the priority of the candidate variable is higher than the priorities of all the variables updated by the event, then it is strongly suggestive that the change should be achieved via a new guard.

Where priorities are the same, no ranking of the alternatives will be provided. We represent heuristics H1 and H2 as critics in Figure 5 and Figure 6 respectively. The meta-logical terms that appear within the preconditions of these critics are defined in Figure 4.

Note that meta-logical predicates, such as *priority*, require input from designers, as mentioned above. Both critics are applicable where an invariant proof has failed, and specifically where the invariant takes the form of an implication.

We focus on solutions that involve the addition of either guards or actions, where guards are restricted to equalities. Later we discuss how these basic critics could be generalised.

```
critic (priority action speculation)
  INPUTS:
      PO_SET $POs$
      MODEL_SET $\{M\}$
  PRECONDITIONS:
      1. $\exists failed\_po \in \{\langle \_, \_, \_, PO\rangle \in POs \mid failed\_proof(PO)\}.$
             $failed\_po = \langle M, E, \_/INV, (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y))\rangle$
      2. $\exists \tau \in sub.\ disjoint\_sub(\tau, \sigma) \ \wedge \ provable(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false)$
      3. $priority(\tau, M) < priority(\sigma, M)$
  OUTPUTS:
      GUIDE $add\_action(sub2act(\tau), E, M)$
```

Note that *sub* denotes the set of all substitutions, and elements of PO_SET are quadruples $\langle M, E, \_/INV, PO\rangle$, i.e. model identifier, event identifier, PO label/type, and PO. Note also that the scope of the quantification extends across the critic slots.

Figure 5: A reasoned modelling critic based on a priority heuristic to modify an action

```
critic (priority guard speculation)
  INPUTS:
      PO_SET $POs$
      MODEL_SET $\{M\}$
  PRECONDITIONS:
      1. $\exists failed\_po \in \{\langle \_, \_, \_, PO\rangle \in POs \mid failed\_proof(PO)\}.$
             $failed\_po = \langle M, E, \_/INV, (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y))\rangle$
      2. $\exists \tau \in sub.\ disjoint\_sub(\tau, \sigma) \ \wedge \ provable(\Delta \vdash (\tau \cup \sigma)Y)$
      3. $priority(\sigma, M) < priority(\tau, M)$
  OUTPUTS:
      GUIDE $add\_guard(sub2grd(\tau), E, M)$
```

Figure 6: A reasoned modelling critic based on a priority heuristic to modify a guard

The critic representing H1 (Figure 5) has three preconditions. The first precondition holds if there exists a PO of the required form that has failed to be proved, i.e. an invariant PO of the form $X \Rightarrow Y$, which is associated with the event E where substitution σ represents the actions corresponding to E. The second precondition holds if there exists a substitution (τ) which falsifies the antecedent, where τ and σ are disjoint. The third precondition expresses a modelling constraint, i.e. the variable(s) introduced by the new substitution have lower priority than the variable(s) updated by the existing actions. If the preconditions hold, then the guidance provided takes the form of a guard – constructed from τ.

The critic representing H2 (Figure 6) again has three preconditions, but illustrates a slightly different failure analysis pattern. That is, instead of making the antecedent false, the analysis tries to make the consequent true. Note that the third precondition ensures that the variable(s) introduced by the new substitution are of a higher priority than the variable(s) updated by the existing actions. Here the guidance provided takes the form of an action – constructed from τ.

3.2. The cruise control system in Event-B

```
MACHINE cruise_control
VARIABLES brake, cc
INVARIANTS inv1: cc = on ⇒ brake = off
METADATA pri_var(cc) < pri_var(brake)
EVENTS
  INITIALISATION ≙
    begin
      act1: brake := off
      act2: cc := off
    end
  EVENT enable_cc ≙
    begin
      act1: cc := on
    end
  EVENT pressbrake ≙
    begin
      act1: brake := on
    end
....
END
```

Note that since the machine name can be easily inferred, we write *pri_var(cc)* and *pri_var(brake)* in the **METADATA** field, instead of *pri_var(cc,cruise_control)* and *pri_var(brake,cruise_control)*.

Figure 7: The cruise control system in Event-B.

To illustrate the application of critics introduced above, we will first formalise in Event-B the cruise-control system that was outlined in Section 1. The key fragments of the model are presented in Figure 7 and explained below.

Recall that the status of the brakes is represented by a variable *brake*, while the status of the cruise control is represented by the variable *cc*. Both variables are of type $Val = \{on, off\}$, where $on \neq off$. The variable *brake* is *on* if the user currently presses the brake (*pressbrake*), and *off* otherwise. While the variable *cc* is *on* if the cruise control is enabled (*enable_cc*), and *off* otherwise. The system invariant is represented as follows:

$$inv1 : \ cc = on \Rightarrow brake = off$$

and initially both variables are *off*:

$$\text{INITIALISATION} \widehat{=} \textbf{BEGIN}\ brake := off \parallel cc := off\ \textbf{END}$$

We assume that the following priority relationship between the variables has been specified by the designer:

$$pri\_var(cc, cruise\text{-}control) < pri\_var(brake, cruise\text{-}control)$$

```
critic (priority guard speculation)
  INPUT
    PO_SET $pos$
    MODEL_SET { $cruise\text{-}control$ }
  PRECONDITIONS:
    1. $\langle cruise\text{-}control, enable\_cc, inv1/INV, po\rangle \in pos \wedge$
      $po = \big(cc = on \Rightarrow brake = off \vdash \{cc \mapsto on\}(cc = on \Rightarrow brake = off)\big) \wedge$
      $failed\_po\big((cc = on \Rightarrow brake = off \vdash brake = off\big)$
    2. $disjoint\_sub(\{brake \mapsto off\}, \{cc \mapsto on\}) \wedge$
      $provable\big(cc = on \Rightarrow brake = off \vdash \{brake \mapsto off, cc \mapsto on\}(brake = off)\big)$
    3. $priority(\{cc \mapsto on\}, cruise\text{-}control) < priority(\{brake \mapsto off\}, cruise\text{-}control)$
  OUTPUTS:
    GUIDE $add\_guard(brake = off, enable\_cc, cruise\text{-}control)$
```

Figure 8: An instantiation of the ‘priority guard speculation’ critic.

3.2.1. *First failure and modelling suggestion*

The model contains events that control the brakes and cruise control. The consistency proofs of invariant *inv1* over these events fails in two cases. Firstly, the event to enable the cruise control is defined as follows:

EVENT enable_cc $\widehat{=}$ **BEGIN** $cc := on$ **END**

This gives rise to the following proof obligation:

$$cc = on \Rightarrow brake = off \vdash \{cc \mapsto on\}(cc = on \Rightarrow brake = off)$$

which can be reduced to the unprovable goal:

$$cc = on \Rightarrow brake = off \vdash brake = off.$$

However, the preconditions of the ‘priority guard speculation’ critic given in Figure 6 hold, and an instantiation of this critic for this example is shown in Figure 8 (after some simplification and unfolding). This instantiation suggests the addition of a new guard of the form $brake = off$. If the suggestion is accepted by the designer, then the updated event takes the form:

EVENT enable_cc $\widehat{=}$ **WHEN** $brake = off$ **THEN** $cc := on$ **END**

3.2.2. *Second failure and modelling suggestion*

The second failure with respect to *inv1* occurs when a driver presses the brakes, as defined by the event:

EVENT pressbrake $\widehat{=}$ **BEGIN** $brake := on$ **END**

This creates the following proof obligation:

$$cc = on \Rightarrow brake = off \vdash \{brake \mapsto on\}(cc = on \Rightarrow brake = off)$$

which, when simplified, becomes false:

$$cc = on, brake = off \vdash false.$$

In this case the ‘priority action speculation’ critic given in Figure 5 triggers, and Figure 9 shows the corresponding instantiation. As a result, the following updated *pressbrake* event is suggested to the designer:

EVENT pressbrake $\widehat{=}$ **BEGIN** $brake := on \parallel cc := off$ **END**

```
critic (priority action speculation)
  INPUT
    PO_SET $pos$
    MODEL_SET { $cruise\text{-}control$ }
  PRECONDITIONS:
    1. $\langle cruise\text{-}control, pressbrake, inv1/INV, po\rangle \in pos \wedge$
       $po = \big(cc = on \Rightarrow brake = off \vdash \{brake \mapsto on\}(cc = on \Rightarrow brake = off)\big) \wedge$
       $failed\_po((cc = on, brake = off \vdash false)$
    2. $disjoint\_sub(\{cc \mapsto off\}, \{brake \mapsto on\}) \wedge$
       $provable\big(cc = on \Rightarrow brake = off \vdash \{brake \mapsto on, cc \mapsto off\}((cc = on) \Rightarrow false)\big)$
    3. $priority(\{cc \mapsto off, cruise\text{-}control) < priority(\{brake \mapsto on\}, cruise\text{-}control)$
  OUTPUTS:
    GUIDE $add\_action(cc := off, pressbrake, cruise\text{-}control)$
```

Figure 9: An instantiation of the 'priority action speculation' critic.

3.2.3. Generalisations of the critics

An alternative definition of the invariant is

$$inv2 : \; brake = on \Rightarrow cc = off$$

which is the contrapositive of *inv1*. To keep our presentation as simple as possible, our critics will not trigger on POs arising from *inv2*. However, this limitation could be overcome by generalising the failure analysis, i.e. allowing the critics to consider the alternatives of falsifying an antecedent and validating a consequent. This could be achieved by changing precondition 2 as follows:

$$provable\big(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false\big) \vee provable\big(\Delta \vdash (\tau \cup \sigma)Y\big)$$

Another limitation is that currently we only consider the addition of guards that take the form of equalities. The 'priority guard speculation' critic could be extended to capture a more general notion of a guard. However, this would require a new notion of priority, which is currently not defined for arbitrary terms (only variables and substitutions). Finally, global consistency issues have been ignored. When an action is modified, the validity of other invariants may change, or a deadlock may arise. The critics should be updated with such consistency checks. Depending on how global analysis is dealt with – as discussed below – the 'priority action speculation' critic of Figure 5 may have to also include such a *consistency* predicate.

3.3. From local to global modelling suggestions

As mentioned earlier, we envisage that our critics will support global analysis, as well as the local analysis illustrated via the cruise control example. For instance, when attempting to prove the internal coherence of a machine, local analysis may suggest the need to add guards across multiple events. Taking a more global perspective, these local suggestions may reveal a more general modelling suggestion. For instance, if a formula can be found that is logically weaker than all the suggested guards, then this points to a global rather than a local problem – and the need for a global solution, i.e. an invariant.

A concrete example of this kind of global analysis is found in Abrial's 'Cars on a Bridge' example [4][1]. In this example a single-laned bridge between an island and the mainland is modelled. Since it is single-laned, cars can only travel in a given direction at a given time – and this is controlled by two traffic-lights: *ml_tl* is the traffic light on the

[1]The example is modified slightly.

mainland; and *il_tl* is the traffic light on the island. The lights can have two colours – *green* and *red* – and it is assumed that $green \neq red$. Only a finite number of cars can be on the bridge and on the island, and three variables are used to count the cars (using sensors in later refinements): a is the number of cars on the bridge travelling towards the island; b is the number of cars on the island; c denotes the number of cars on the bridge leaving the island, while d is a constant denoting the total number of cars. Our discussion will focus on the four events where cars enter the bridge. *ML_out1* and *ML_out2* captures cars entering the bridge from the mainland, and is formalised as follows:

```
EVENT ML_out1 ≘                EVENT ML_out2 ≘
  WHEN ml_tl = green              WHEN ml_tl = green
       a + b + 1 < d                   a + b + 1 = d
  THEN a := a + 1                 THEN a := a + 1
END                                    ml_tl := red
                               END
```

IL_out1 and *IL_out2* captures cars entering the the bridge from the island, and is formalised as follows:

```
EVENT IL_out1 ≘                EVENT IL_out2 ≘
  WHEN il_tl = green              WHEN il_tl = green
       b > 1                           b = 1
  THEN b := b − 1                 THEN b := b − 1
       c := c + 1                      c := c + 1
END                                    il_tl := red
                               END
```

Without giving all the details[2], the proofs of

$$il\_tl = green \Rightarrow a = 0$$

fail over events *ML_out1* and *ML_out2*, while the proofs of invariant

$$ml\_tl = green \Rightarrow c = 0$$

fail over events *IL_out1* and *IL_out2*. The corresponding POs have the following form:

$$\ldots, il\_tl = green \;\vdash\; \sigma_1(ml\_tl = green \Rightarrow \ldots) \;\; where\ ml\_tl \notin dom(\sigma_1) \tag{6}$$

$$\ldots, ml\_tl = green \;\vdash\; \sigma_2(il\_tl = green \Rightarrow \ldots) \;\; where\ il\_tl \notin dom(\sigma_2) \tag{7}$$

A local analysis of each of these failures suggests making the antecedent false, by finding a substitution τ such that:

$$disjoint\_sub(\tau, \sigma_{\{1,2\}}) \;\wedge\; provable(\Delta \vdash (\tau \cup \sigma_{\{1,2\}})X \Rightarrow false)$$

In (6), τ will be instantiated to $\{ml\_tl \mapsto red\}$ while in (7), τ will be instantiated to $\{il\_tl \mapsto red\}$. However, there is a common solution to these failures. Firstly, we assume that the substitutions are turned into guards (by *sub2grd*). Now, if an event is constrained by the guards $G_1, \cdots, G_n$, then adding an additional guard G_{n+1} to patch a failure is equivalent to adding the weaker guard $G_1 \wedge \cdots \wedge G_n \Rightarrow G_{n+1}$ (via modus ponens). Thus the suggested additional guards can be "weakened" in this way with the existing guards of the events. For this particular example, we ignore all guards except those shown in (6,7) – the other guards are irrelevant for this failure. For example, from (6) the guard $ml\_tl = red$ is derived and "weakened" to give $il\_tl = green \Rightarrow ml\_tl = red$. We then try to find a common solution for (6,7), i.e. an r such that:

$$r \Rightarrow il\_tl = green \Rightarrow ml\_tl = red$$
$$r \Rightarrow ml\_tl = green \Rightarrow il\_tl = red.$$

Moreover, we require that r is satisfiable, i.e. the trivial solution $r = false$ is not considered. The only valid such generalisation r (modulo equivalences) is:

$$il\_tl = red \vee ml\_tl = red$$

```
critic (multiple failure invariant speculation)
  INPUTS:
    PO_SET $POs$
    MODEL_SET $\{M\}$
  PRECONDITIONS:
    1. $\exists h \subseteq hs.\ h = \{\langle E, G \rangle \mid \exists po \in pos.\ \langle \_, E, \_/INV, po \rangle \in POs \ \wedge\ failed\_proof(po)$
                    $\wedge\ po = (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y))$
                    $\wedge\ \exists \tau \in sub.\ disjoint\_sub(\tau, \sigma)\ \wedge\ provable(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false)$
                    $\wedge\ G = guards(E) \Rightarrow sub2grd(\tau)\ \}$
    2. $generalisable(h) \wedge \exists i \in forms.\ i = generalise(h)$
  OUTPUTS:
    GUIDE $add\_invariant(i, M)$
```

Note that *hs* denotes the set of all event-guard pairs, while *pos* and *forms* denote the sets of all proof obligations (sequents) and formulae respectively.

Precondition holds if a local analysis suggests that proof failures can be overcome by the introduction of additional guards.

Precondition 2 holds if the suggested guards can be generalised to give an invariant.

Figure 10: A global reasoned modelling critic suggesting an invariant from multiple failures

which is suggested as an invariant. This represents a key safety requirement – at all times at least one traffic-light is red.

With regards to realising such global reasoning, we are considering two distinct alternatives via our proposed critic schema:

1. Combine both local and global analysis of the available POs within a single critic, as illustrated in Figure 10.
2. Firstly use critics to perform local analysis of the available POs, recording any modelling suggestions. Secondly, use separate critics to perform a global analysis, exploiting the results of the local analysis.

From a scientific point of view, the approaches are not very different – and should be seen more as an implementation issue. Experimentation will be required in order to determine the most practical approach.

4. Implementation

As with most theories, the notion of reasoned modelling will only be fruitful with proper tool support. Here, we describe the REMO[3] tool, which implements a prototype for the reasoned modelling critic mechanism introduced above. Since the technique proposed here has been illustrated for Event-B, the tool is necessarily integrated within the Eclipse-based Rodin toolset — implemented in Java. However, as is the case with the ProB Rodin plug-in [11], the main part of the program has been implemented in another programming language. In our case, we have chosen OCaml (*Objective Caml Language*)[4]. This has been motivated by the the following factors:

- this is a prototype implementation which we expect will change (frequently). OCaml is a declarative functional programming language which reduces the size of the source code, thus making change to the code (relatively) easy;

[2]These details can be found in [4] and `http://www.event-b.org`.
[3]The REMO acronym follows from REasoned MOdelling
[4]See `http://caml.inria.fr/ocaml/`

- a large part of the work is checking precondition, which can be handled very elegantly using inductive data types and pattern matching, which is supported by OCaml;
- compared to other languages in the ML-family, OCaml embodies an object-oriented layer, which provides a good encapsulation mechanism to represent models, and particularly critics;
- OCaml is used in the CORE project[5], which is a sister project of ours. The use of OCaml enables us to take advantage of the reasoning techniques developed within the CORE project. In particular, term synthesis and integration with the CVC3 SMT solver, discussed below.

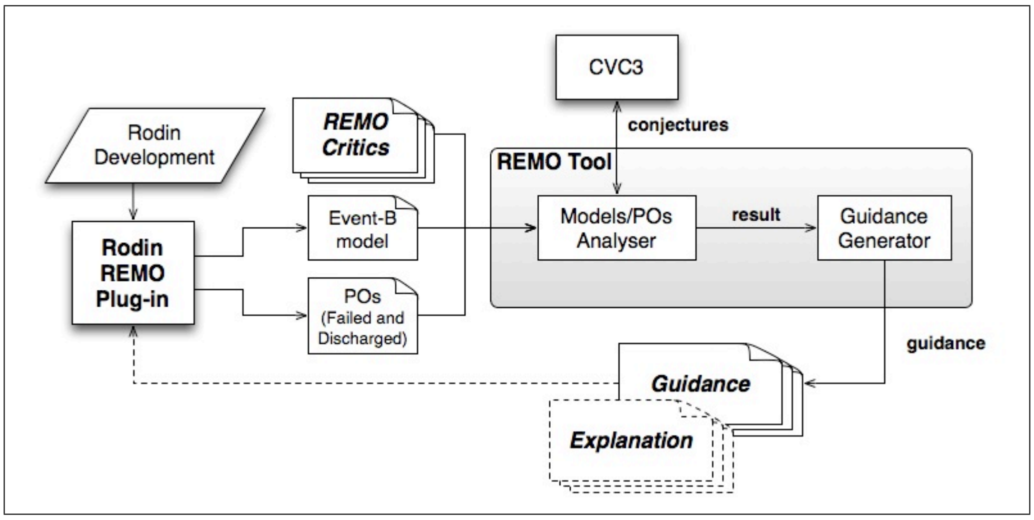

Figure 11: Tool architecture

The overall process embodied within REMO involves an analysis of the applicability of the critics with respect to Event-B models and their failed proof obligations. The results of the analysis are presented to the user in the form of guidance – it is the user that ultimately decides whether or not to accept the guidance. An overview of the architecture of the tool can be found in Figure 11. The labels with **bold** font indicate the main components of the tool chain; the stippled lines indicate work in progress; the square boxes ▭ represent processes; the document boxes ▭ represent files that are used as inputs and/or outputs of a process; while the rectangle box ▱ represents input data used in a process.

4.1. Rodin REMO Plug-in

The goal of the Rodin REMO plug-in is to provide an interface between the Rodin toolset and our REMO tool. Specifically the plug-in has two main roles:

1. translate Event-B developments (models, contexts and proof obligations) from Rodin into the REMO tool; and
2. communicate modelling guidance and explanation back from the REMO tool into Rodin.

However, at the current state only role 1 has been implemented – role 2 is in our plan of future work, and communication is currently at the level of printed messages on a terminal window. However, a description of how we envisage the guidance will be presented to the user by the Rodin plug-in is given throughout this section.

The input of the Rodin REMO plug-in is a Rodin development, from which the plug-in generates two files that serve as the input of the REMO tool. The first file contains a string representation of the Event-B models (machines and contexts) of a development, while the second file stores a string representation of all the POs (status, type, hypotheses

[5]See `http://www.macs.hw.ac.uk/core/`

and goal) associated to each model. These files are then translated into the REMO internal representation, i.e. OCaml objects.

Currently, the PO generator of Rodin applies some simplification before the POs are generated. This simplification consists of replacing the substitutions given by the events to the variables in the POs. As a consequence, the file generated by our plug-in contains the POs after the simplification has taken place. However, in the analysis of the critics the pure PO (i.e. before the substitutions are applied) is also needed in order to do the matching between the required POs (given by the preconditions) and the ones of the models. Currently, this information is hand-crafted in the file; however, we plan to update the Rodin PO generator to overcome this problem.

4.2. REMO Critics

The reasoned modelling critics represent another input to the REMO tool. Each critic is implemented as a class in OCaml that inherits from the virtual class *reasonedModellingCritic*. This super class encapsulates the basic elements of a critic as shown in Figure 12.

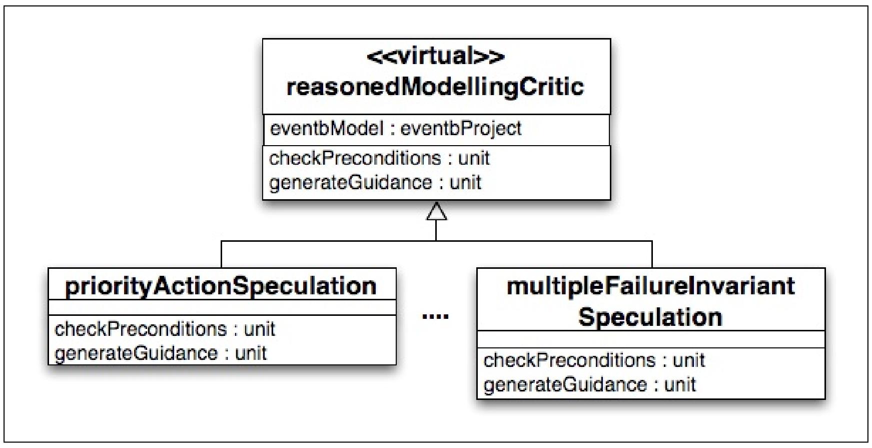
<<virtual>>
reasonedModellingCritic
eventbModel : eventbProject
checkPreconditions : unit
generateGuidance : unit
priorityActionSpeculation
checkPreconditions : unit
generateGuidance : unit
....
multipleFailureInvariant Speculation
checkPreconditions : unit
generateGuidance : unit

Figure 12: Critics implementation

In particular, each critic overrides the methods *checkPreconditions* and *generateGuidance*. The first method evaluates the applicability of the critic through the checking of the preconditions while the second method analyses the output of this check and produces the guidance that will be suggested to the user. This use of inheritance allows new critics to be added relatively easily to the REMO tool.

4.3. The REMO Tool

The REMO tool is central to our implementation. It contains two components, *Models/POs Analyser* and *Guidance Generator*, which are applied sequentially and discussed separately below.

4.3.1. Models/POs Analyser

The inputs of the Model/POs Analyser are the models, the POs and the critics. The first task of the process is to parse the model and PO files generated by the Rodin REMO plug-in. Then the analyser evaluates the Event-B models and their POs in order to find which critics are applicable (if any).

Note that currently when presented with unproven POs, REMO attempts to discover ways in which the proof failures can be overcome via the automatic generation of modelling guidance. Crucially what is on offer is guidance – whether or not the guidance is accepted is left to the user. Such guidance may address inconsistencies within a development or complexities that give rise to POs that are beyond the Rodin provers. Alternatively, where better proof automation is required, we aim in the longer-term to include a a proof planner which will enable us to improve upon the level of automation provided by the current Rodin provers (see Section 6).

The process of evaluating the applicability of a critic with respect to a model consists of the following steps (this process is further illustrated with the cruise control example in Section 5):

1. The analyser searches for matches between each of the preconditions of the critic and the models. If more than one match is found, two things can happen. If the precondition being evaluated is one that identifies different failures in the model (e.g. failed POs), each match would be considered as a separate instantiation of the critic. However, if the precondition being checked is one that identifies different solutions to the same failure (e.g. finding a substitution that is disjoint with the other substitutions of an event), all the matches will be part of the set of all possible solutions of a failure.
2. For each element of the set of possible solutions of each instance of a critic the subsequent preconditions are evaluated. When all the preconditions have been checked, the analyser verifies which instances were successful in all the preconditions.
3. The analyser collects the results of each successful instance and sends them to the guidance generator.

The REMO tool uses the CVC3 SMT solver [12] to check if a match is found for a specific precondition. This choice of SMT solver follows from an existing OCaml interface developed within the CORE project. Currently, we only support first-order predicates with equality, as well as arithmetic. Thus, the full underlying (set-theoretic) Event-B mathematical notation is not supported. In the future, we plan to take advantage of both work with SMT solvers for Event-B[6], as well as the use of a proof planner (see above). Furthermore, we may possibly use generic purpose theorem provers.

4.3.2. Guidance Generator

The guidance generator 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 plug-in and presented to the user.

Two processes take place here. The first process consists of interpreting the guidance given as an input in order to produce written explanations. These are linked to the critic preconditions of each suggestion. The purpose behind this, is to help the user understanding the nature of the guidance, to (hopefully) make a more informed decision about which modification to apply. We have classified the guidance produced by the generator into three categories:

1. **Global guidance:** This refers to a global solution that addresses multiple local failures (e.g. the addition of an invariant to the model instead of guards in multiple events). This type of guidance is given alongside an explanation of how the local suggestions are used to propose the general one.
2. **Local guidance:** This is concerned with a solution that addresses a local failure (e.g. a guard is missing from an event). An explanation of how the failure is overcome with the solution is provided.
3. **Conditional guidance:** This type of guidance is related to solutions that are subject to the fulfilment of certain conditions (e.g. priorities of variables that were not defined in the model). A list of conditions together with explanations of why these conditions are needed is given with the suggested solution.

The second process, which is currently under development, deals with the linking of the guidance and explanation to the Rodin plug-in.

4.3.3. Guidance and Explanation

The result of the guidance generator is a set of files capturing both the guidance and explanations for each critic instance. These files will then be sent to the Rodin plug-in for presentation to the user.

From the user perspective, the guidance will consist of the fragment of Event-B model that is relevant to the suggested modification. For instance, in the case of the priority action speculation critic, the guidance will contain the current event plus the new action as shown in Figure 13. The instantiation of this schema for our running cruise control example is shown in Figure 16.

The explanations will be represented as templates, i.e. a natural language explanation of the critic preconditions, with slots to denote instance specific details. To illustrate, the proposed template associated to the priority action speculation critic is shown in Figure 14. An instantiation of this template is presented in Figure 16.

Regarding conditional guidance, consider again precondition 3 of the priority action speculation critic (see Figure 5). Whether or not this precondition holds depends upon if the user has provided the relevant meta-data, i.e. priority information with respect to their model. If a critic fails to apply because of the lack of such meta-data, the user can still be presented with the relevant guidance, but conditional with respect to the missing meta-data.

[6]See `http://www.cprover.org/SMT-LIB-LSM/`

Guidance
(priority action speculation critic)

EVENT *event_name*
  WHERE *guards*
  THEN *old_actions*
    $\boxed{\textbf{\textit{new\_action}}}$
END

Figure 13: Guidance provided by the priority action speculation critic analysis.

Explanation template (priority action speculation critic)

1. There exists an unproven invariant PO ______.
2. There exists a variable update ______ that does not interfere with the existing actions ______ of event ______ and which makes the PO provable.
3. Update preserves the variables priority relationship ______.

Therefore:
Failure can be overcome by adding action ______ to event ______.

Figure 14: Explanation template associated to the priority action speculation critic.

5. Results - Examples Revisited

Here, we apply the 'cruise control' and 'cars on a bridge' examples to the REMO tool, and demonstrate step by step the analysis process. These examples are discussed separately below.

5.1. The cruise-control example – the (local) priority critics

The initial step is the evaluation of the preconditions. The analysis of the critic's preconditions involves two parts, a syntactic check followed by a semantic check. The syntactic checking is a filtering process that consists of searching for elements in the models and the POs that have specific structures. After this filtering is done, the semantic checking consists of the analysis of specific properties that these elements must satisfy in order to be an instantiation of the precondition. Both of these checks rely heavily on pattern matching, thus illustrating the advantages of functional languages such as OCaml.

To demonstrate, we analyse the priority action speculation critic in the 'cruise control' model. Precondition 1 (PC_1) of this critic states:

$$\begin{aligned}\mathbf{PC}_1 : \exists \mathit{failed\_po} \in \{\langle \_, \_, \_, PO\rangle \in POs \mid \mathit{failed\_proof}(PO)\}.\\ \mathit{failed\_po} = \langle M, E, \_/INV, (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y))\rangle\end{aligned}$$

We need to search for all failed POs, whose type is *INV* (i.e. invariant preservation POs) with the form $X \Rightarrow Y$ (since invariant proofs have an inductive structure, this form will be found in both the goal and the hypothesis). The POs are represented as objects in the tool – where they are stored with their attributes, status and type. Using this meta-information we can directly check if the type is *INV*. The next part checks that the form of the goal/hypothesis is (syntactically) $X \Rightarrow Y$. This is achieved by pattern matching. Two (failed) POs succeed this check:

$$\textbf{po}_1\textbf{:}\ \ldots.\wedge cc = on \Rightarrow brake = off \vdash \{cc \mapsto on\}\ cc = on \Rightarrow brake = off$$
$$\textbf{po}_2\textbf{:}\ \ldots.\wedge cc = on \Rightarrow brake = off \vdash \{brake \mapsto on\}\ cc = on \Rightarrow brake = off$$

Each of these POs represents a different failure, as a result, each of them will be treated as an instance of the critic. Let $instance_1$ be the instance of the critic associated to po_1, and $instance_2$ be the instance associated to po_2. In the following precondition we look for a substitution to falsify the antecedent, and the precondition is formalised as follows:

$$\textbf{PC}_2\textbf{:}\ \exists \tau \in sub.\ disjoint\_sub(\tau, \sigma)\ \wedge\ provable(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false)$$

The checking of this precondition is broken into three steps:

- $\textbf{PC}_{2.1}$**:** Find the possible substitutions that can be applied to the antecedent of the goal.
- $\textbf{PC}_{2.2}$**:** From $\text{PC}_{2.1}$ find the substitutions that are disjoint with the current substitutions of the event.
- $\textbf{PC}_{2.3}$**:** From $\text{PC}_{2.2}$ find the substitutions that are provable.

The antecedent of the goals of po_1 and po_2 is $cc = on$; thus, in the analysis of $\text{PC}_{2.1}$ we need to find all possible values that the variable cc can take. In order to do this, we need to identify the type of the variable, which is given by the invariant $cc \in state$, and then we look for the members of the set $state$. This is analysed by looking for a "type axiom" of the form '$state = \{\_, \_, ..., \_\}$', or in the case this is not given, by axioms of the form '$\_ \in state$'. From this, in the cruise control model, *on* and *off* are defined as the members of the set *state*. Therefore, two possible substitutions are found for each instance of the critic: $cc \mapsto on$ and $cc \mapsto off$. These are considered as two potential solutions for each failed PO.

The next step, $\text{PC}_{2.2}$, is the verification of the disjointedness of the substitutions found in $\text{PC}_{2.1}$. We find that for $instance_1$, the suggested substitutions are not disjoint with $cc \mapsto on$, which is the substitution of event *enable_cc*. As a result, both solutions are considered as failures of this instance of the critic. On the contrary, in $instance_2$, the suggested substitutions are both disjoint with $brake \mapsto off$, which is the substitution of event *pressbrake*.

Finally, in $\text{PC}_{2.3}$ we need to verify if the statement $\Delta \vdash (\tau \cup \sigma)X \Rightarrow false$ is provable for the substitutions found in $\text{PC}_{2.2}$. The instantiation of this statement generates the following conjectures:

$$\textbf{conj}_1\textbf{:}\ cc = on \Rightarrow brake = off \vdash \{cc \mapsto on, brake \mapsto on\}\ cc = on \Rightarrow false$$
$$\textbf{conj}_2\textbf{:}\ cc = on \Rightarrow brake = off \vdash \{cc \mapsto off, brake \mapsto on\}\ cc = on \Rightarrow false$$

Both conjectures are passed to CVC3 in order to be validated. The result is that conj_2 is valid but conj_1 is not. This rules out $cc \mapsto on$ as a possible solution of the critic. Thus, up to this point, we have $instance_2$ as the only instance of the critic, and $cc \mapsto off$ as the only potential solution for that instance.

Finally, the last precondition verifies the priorities. The precondition is:

$$\textbf{PC}_3\textbf{:}\ priority(\sigma, M) < priority(\tau, M)$$

In this case, the priority of the new substitution should be less than the priority of the current substitutions (actions) of the event. The priority of a substitution is the priority of the variable being modified; therefore, in order to perform the analysis of the precondition, we must compare the priorities of the variables. Event *pressbrake* has only one substitution $brake \mapsto off$; then, we need to check that

$$pri\_var(cc,cruise\text{-}control) < pri\_var(brake,cruise\text{-}control).$$

This is evaluated by the REMO tool through pattern matching of the meta-data defined in the model. The precondition succeeds, creating as guidance the addition of the action $cc := off$ in event *pressbrake*.

Figure 15 sums up the steps taken by the tool for the analysis of the priority action speculation critic. In the figure, we present the evaluation of the two instances of the critic for the cruise control system. For each instance, the branches represent the different alternatives to solve the failure while each node represents the result (or matches) of the analysis of each precondition with respect to the specific alternative. Each branch terminates either when a failure of a precondition is found or when all the preconditions have succeeded in that branch.

Finally, Figure 16 shows the guidance that would be provided by the tool. The analysis of the priority guard speculation critic is performed in a similar fashion.

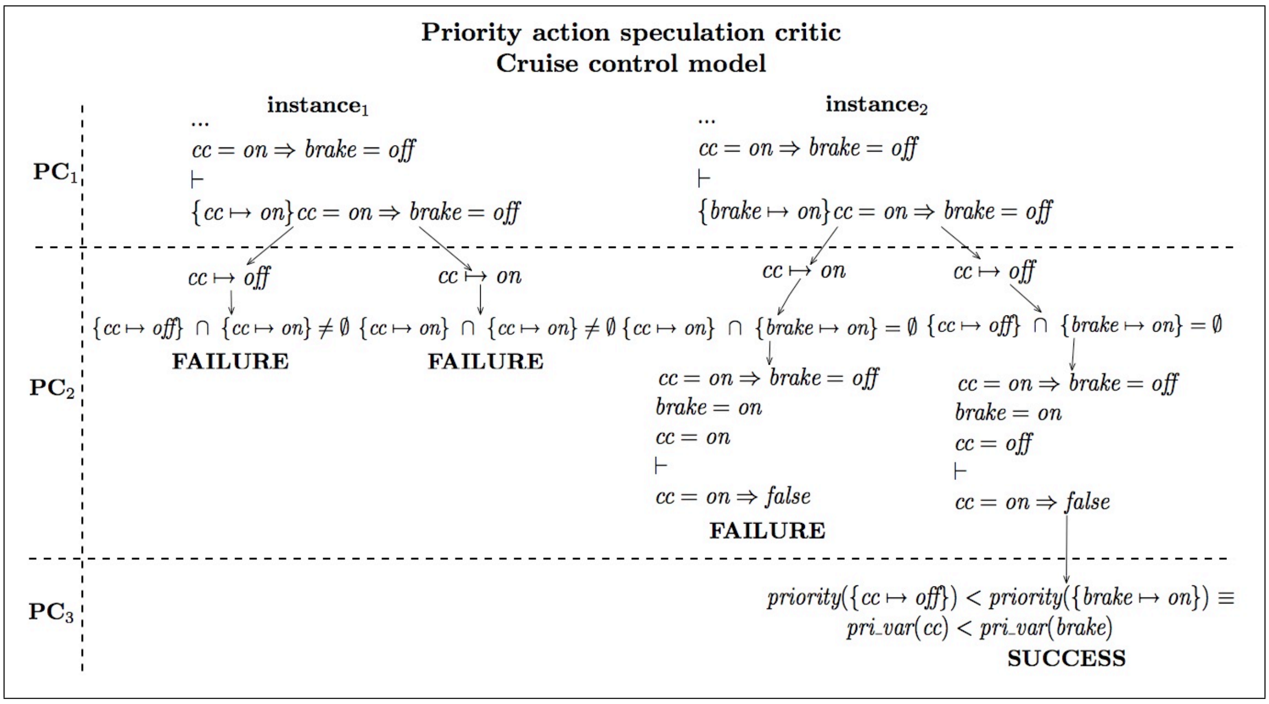
Priority action speculation critic
Cruise control model
instance$_1$
instance$_2$
PC$_1$
...
$cc = on \Rightarrow brake = off$
$\vdash$
$\{cc \mapsto on\} cc = on \Rightarrow brake = off$
$\{brake \mapsto on\} cc = on \Rightarrow brake = off$
PC$_2$
$cc \mapsto off$
$cc \mapsto on$
$\{cc \mapsto off\} \cap \{cc \mapsto on\} \neq \emptyset$
$\{cc \mapsto on\} \cap \{cc \mapsto on\} \neq \emptyset$
$\{cc \mapsto on\} \cap \{brake \mapsto on\} = \emptyset$
$\{cc \mapsto off\} \cap \{brake \mapsto on\} = \emptyset$
FAILURE
FAILURE
$cc = on \Rightarrow brake = off$
$brake = on$
$cc = on$
$\vdash$
$cc = on \Rightarrow false$
FAILURE
$cc = on \Rightarrow brake = off$
$brake = on$
$cc = off$
$\vdash$
$cc = on \Rightarrow false$
PC$_3$
$priority(\{cc \mapsto off\}) < priority(\{brake \mapsto on\}) \equiv$
$pri\_var(cc) < pri\_var(brake)$
SUCCESS

Figure 15: Sequence of steps performed by REMO in the analysis of the priority action speculation critic for the cruise control example.

Guidance

EVENT pressbrake $\widehat{=}$
  begin
    act1: brake := on
    $\boxed{\textit{act2: } \text{cc := off}}$
  end

Explanation for action speculation

1. There exists an unproven invariant PO ***pressbrake/inv1/INV***.
2. There exists a variable update ***cc ↦ off*** that does not interfere with the existing actions ***brake := on*** of event ***pressbrake*** and which makes the PO provable.
3. Update preserves the variables priority relationship ***pri_var(cc) < pri_var(brake)***.

Therefore:
Failure can be overcome by adding action ***cc := off*** to event ***pressbrake***.

Figure 16: Proposed guidance and explanation provided by REMO for the cruise control system.

5.2. *The cars on a bridge example – the (global) multiple failure critic*

The multiple failure invariant speculation critic is implemented by combining both local and global analysis within a single critic. This methodology has been chosen since it facilitated the collection of the results of the local analysis, as well as avoiding duplication of information since all the processes are achieved within the same critic, instead of being distributed over many critics.

In the 'cars on a bridge' example, the analysis of the first precondition involves a very similar process to the process described for the priority action speculation critic. Thus, we will focus on how the global analysis was implemented – i.e. the second precondition of Figure 10. This precondition is used to generalise the solutions found in the first precondition into an invariant which ideally captures all the failures – however, we only require it to capture more than one failure over more than one event. The precondition is:

$$\mathbf{PC}_2\textbf{:}\ \mathit{generalisable}(h) \wedge \exists i \in \mathit{forms}.\ i = \mathit{generalise}(h)$$

with the following definitions (as defined in Figure 4):

$$\begin{array}{lcl} \mathit{sat}(P) & \widehat{=} & \text{“}\mathit{The\ predicate\ P\ is\ satisfiable}\text{”} \\ \mathit{generalisable}(H) & \widehat{=} & \exists r, e_1, e_2, g_1, g_2.\ \langle e_1, g_1\rangle \in H\ \wedge \langle e_2, g_2\rangle \in H \\ & & \qquad\qquad \wedge\ e_1 \neq e_2 \wedge r \Rightarrow g_1 \wedge r \Rightarrow g_2 \wedge \mathit{sat}(r) \\ \mathit{generalise}(H) & \widehat{=} & \text{“}\mathit{The\ weakest\ r\ such\ that\ sat(r)\ and\ for\ the} \\ & & \quad \mathit{most}\ \langle e, g\rangle \in H,\ r \Rightarrow p\ \mathit{(for\ at\ least\ 2\ events)}\text{”} \end{array}$$

The definition of *generalise* is (deliberately) under-defined, and finding good implementation of it is a hard problem. Below, we have implemented a rather specialised version of it, but in the future we plan to rely on *abduction* [10] (see Section 2.2), possibly implement using *term-synthesis* [13], which using forward chaining, generates formulas which are then checked. An efficient implementation of term-synthesis would require finding good heuristics to avoid generating too many uninteresting formulas.

However, before that we will expand the ideas behind precondition 1 and 2 of this critic. In the first step we first find a guard which will cause the PO to be proven. We then weaken the solution as much as we can with the existing guards of the event (it can be even further weakened by existing invariants – but this would have complicated the presentation too much). For example, a solution G for an event with guards G_1 and G_2 becomes $G_1 \wedge G_2 \Rightarrow G$. In the second precondition, we then try to find the weakest strengthening r which is as least as strong as two of the weakened solutions (of two different events).

A straightforward such strengthening is to remove one or more of the guards, e.g. creating $G_2 \Rightarrow G$, and this approach is followed here, where the use of common (free) variables in the goal is used to guide the elimination of guards. From this, we divide the analysis for this precondition into four steps:

1. The set of free variables of the goals of the events (FV) is found.
2. All conjuncts (each representing a guard of the event) that use free variables not in FV are omitted.
3. We then have four conjectures – however two and two conjectures are (syntactically) equivalent, given the following two (different) conjectures (resulting from POs of different events):

$$\mathit{il\_tl} = \mathit{green} \Rightarrow \mathit{ml\_tl} = \mathit{red} \qquad \text{and} \qquad \mathit{ml\_tl} = \mathit{green} \Rightarrow \mathit{il\_tl} = \mathit{red}$$

4. A check for equivalence between the resulting conjectures is then performed. If two conjectures (from different events) are equivalent then one of them is picked. The (equivalent) solutions are then ranked according to which capture most different events – followed by the number of conjectures it captures (in a lexicographical order). This highest ranked solution will then form the weakest such generalisation, and is returned. These steps can therefore be seen as an implementation of *generalise*. In this case, they are all equivalent, and can be rewritten to

$$\mathit{ml\_tl} = \mathit{red} \vee \mathit{il\_tl} = \mathit{red}.$$

This is then presented as a global alternative to address the multiple failures.

6. Related and future work

Our notion of *reasoned modelling* represents a new paradigm for exploring the interplay between reasoning and modelling. As evident from the above discussion, our general ideas on reasoned modelling are strongly influenced by the proof planning paradigm [1], while the particular work discussed here follows from the notion of *proof critics* [2]. Currently, Event-B users have to manually analyse failed proof attempts and patch their models, e.g. [4, 6]. This form of user interaction represents a significant barrier to the accessibility of the Event-B toolset.

We have discussed how we can turn failed proofs into modelling guidance. Another source of failure is counter-examples generated by model checkers. We have previously discussed turning counter-examples from model checkers into modelling guidance [14]. This experiment was (manually) conducted by using the ProB model checker [11] in Rodin. The advantage of using proof failures is that the nature of the failure is clearer than from a counter-example. However, we may be able to extend the work in [14] into critic descriptions, however, this is a matter of future work. Moreover, we believe model checkers, ProB in our case, can have at least two roles to play within our preconditions:

- a critic precondition can demand a counter-example. This can be used to separate the cases between say a false invariant, and a correct invariant where the proof fails due to limitations of the theorem prover;
- additional preconditions, such as adding a guard will not introduce a deadlock, can be added to reduce the number of suggestions to the user. Here, ProB can be integrated with our REMO tool, and used to check such preconditions.

Previously, the application of *design patterns* [15] has been suggested for Event-B [16]. The ability to reuse design patterns, and their associated proofs, has obvious benefits over the conventional notion of design patterns. Our work with failure-analysis is more closely aligned with *anti-patterns* [17], i.e. common patterns of bad design, coupled with solutions. Bad design, however, does not necessarily mean incorrect design – i.e. the POs may still be provable. For this reason the failure-driven nature of the critics presented here differ from the conventional notion of an anti-pattern. Another related area is ontology repair plans [18] – proof failure is explored using proof planning techniques to repair ontologies – i.e. proof planning is used to describe evolving models.

In terms of future work, our short-term aim is to complete and fully integrate the REMO tool within Rodin. Moreover, we aim to investigate modelling heuristics in addition to variable priorities. While we have relied upon existing external provers, in the longer term we aim to incorporate a proof planner. This will enable us to also develop critics that analyse successful proofs and provide guidance along the lines of conventional anti-patterns. The most likely candidate is IsaPlanner [19], which is based on the Isabelle theorem prover.

7. Conclusion

We have motivated the need to abstract away from the complexities of proof obligations and proof-failure analysis. To address this need, we have proposed a technique of proof management that attempts to turn proof-failure analysis into modelling guidance via *reasoned modelling critics*. In doing so we aim to increase the productivity of designers as well as the accessibility of formal modelling. Our technique is motivated by proof planning and, in particular, proof critics. The work differs from previous work in that it provides a uniform framework in which proof-failure heuristics can be combined with modelling heuristics. It is this combined approach that will enable us to abstract away from the complexity of proof obligations. While our initial critics have focused on variable priorities, our framework will be extensible, allowing designers to record meta-data relating to other kinds of modelling decisions. Our REMO prototype represents a proof of concept, and the first step towards a reasoned modelling plug-in for the Rodin toolset.

Acknowledgements

This research is supported by EPSRC grants EP/F037058 and EP/E005713. Gudmund Grov is supported by EPSRC grant EP/H024204/1 “AI4FM: the use of AI to automate proof search in Formal Methods” (`www.ai4fm.org`).. Butler’s involvement is part of the EU research project ICT 214158 DEPLOY (`www.deploy-project.eu`), and Maria Teresa Llano is partially supported by a BAE Studenship. Our thanks go to Alan Bundy, who motivated the use of variable priorities. Thanks also go to Jean-Raymond Abrial, Cliff Jones and Ewen Maclean for their feedback and encouragement with this work. Finally, we thank the anonymous ABZ 2010 and SCP reviewers for their constructive feedback.

References

[1] A. Bundy, The Use of Explicit Plans to Guide Inductive Proofs, in: R. Lusk, R. Overbeek (Eds.), CADE’09, Springer-Verlag, 1988, pp. 111–120.

[2] A. Ireland, The Use of Planning Critics in Mechanizing Inductive Proofs, in: A. Voronkov (Ed.), LPAR’92, no. 624 in LNCS, Springer-Verlag, 1992, pp. 178–189.

[3] A. Ireland, G. Grov, M. Butler, Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance, in: M. Frappier, U. Glasser, S. Khurshid, R. Laleau, S. Reeves (Eds.), Proceedings of ABZ’10, no. 5977 in Lecture Notes in Computer Science, Springer, 2010, pp. 189–202.

[4] J.-R. Abrial, Modelling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[5] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: An open toolset for modelling and reasoning in event-b, International Journal on Software Tools for Technology Transfer (STTT).
URL `http://dx.doi.org/10.1007/s10009-010-0145-y`

[6] M. Butler, D. Yadav, An Incremental Development of the Mondex System in Event-B, Formal Aspects of Computing 20 (1) (2008) 61–77.

[7] J.-R. Abrial, M. J. Butler, S. Hallerstede, L. Voisin, An Open Extensible Tool Environment for Event-B, in: ICFEM'06, Vol. 4260 of LNCS, Springer, 2006, pp. 588–605.
URL `http://dx.doi.org/10.1007/11901433_32`

[8] A. Bundy, D. Basin, D. Hutter, A. Ireland, Rippling: Meta-level Guidance for Mathematical Reasoning, Cambridge University Press, 2005.

[9] A. Ireland, J. Stark, Proof Planning for Strategy Development, Annals of Mathematics and Artificial Intelligence 29 (1-4) (2001) 65–97.

[10] R. Monroy, A. Bundy, A. Ireland, Proof Plans for the Correction of False Conjectures, in: F. Pfenning (Ed.), LPAR'94, no. 822 in LNAI, Springer-Verlag, 1994, pp. 54–68.

[11] M. Leuschel, M. Butler, ProB: an Automated Analysis Toolset for the B Method, Journal Software Tools for Technology Transfer 10 (2) (2008) 185–203.

[12] C. Barrett, C. Tinelli, CVC3, in: W. Damm, H. Hermanns (Eds.), Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07), Vol. 4590 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 298–302, berlin, Germany.

[13] E. Maclean, A. Ireland, R. Atkey, L. Dixon, Refinement and term synthesis in loop invariant generation, in: A. Ireland, L. Kovacs (Eds.), 2nd International Workshop on Invariant Generation (WING'09), 2009, pp. 72–86.

[14] A. Pease, A. Ireland, S. Colton, R. Ramezani, A. Smaill, M. T. Llano, G. Grov, M. Guhe, Thinking Machines and the Philosophy of Computer Science: Concepts and Principles, IGI Global, 2010, Ch. 10: Applying Lakatos-Style Reasoning to AI Problems.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[16] J.-R. Abrial, T. S. Hoang, Using Design Patterns in Formal Methods: An Event-B Approach, in: ICTAC'08 1-3, 2008. Proceedings, Vol. 5160 of LNCS, Springer, 2008, pp. 1–2.
URL `http://dx.doi.org/10.1007/978-3-540-85762-4_1`

[17] W. Brown, R. Malveau, H. W. S. McCormick, T. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, 1998.

[18] A. Bundy, M. Chan, Towards ontology evolution in physics, in: W. Hodges (Ed.), Proceedings of Wollic 2008, LNCS, Springer-Verlag, 2008.

[19] L. Dixon, J. Fleuriot, IsaPlanner: A Prototype Proof Planner in Isabelle, in: Proceedings of CADE'03, Vol. 2741 of LNCS, 2003, pp. 279–283.