Generic Instantiation Proposal

Renato Silva and Michael Butler

October 12, 2012

1 Motivation

It is believed that reusability in formal development should reduce the time and cost of formal modelling within a production environment. Along with the ability to reuse formal models, it is desirable to avoid unnecessary re-proof when reusing models. Event-B supports generic developments through the context construct. However Event-B lacks the ability to instantiate and reuse generic developments in other formal developments. We propose a methodology to instantiate generic models and extend the instantiation to a chain of refinements.

Abrial and Hallerstede [2] and Métayer et al [3] propose the use of generic instantiation for Event-B. It is suggested that the contexts of a development (*pattern*) can be merged and reused through instantiation in other developments. That proposal lacks a mechanism to apply the instantiation from the *pattern* to the instances.

We propose a Generic Instantiation tool for Event-B by instantiating machines. The instances inherit properties from the generic development (*pattern*) and are parameterised by renaming/replacing those properties to specific instance element names. Proof obligations are generated to ensure that assumptions used in the *pattern* are satisfied in the instantiation. In that sense our approach avoids re-proof of *pattern* proof obligations in the instantiation. The reusability of a development is expressed by instantiating a development (*pattern*) according to a more specific *problem*.

2 Parameterisation of contexts

The instantiation is achieved with the *parameterisation of contexts*: contexts are only seen by one machine (or one chain of machine refinements) and define specific properties for that machine (sets, constants, axioms, theorems). These properties are unique for that machine and any other machine would have different properties. This contrasts with the other usage of contexts: a *sharing context* is seen by several machines and there are some properties (sets, constants, axioms, theorems) shared by the machines. Therefore in this case, the context is used to share properties.

3 Definition of Generic Instantiation of Machines

Here we define the generic instantiation of machines. Consider context C_{G0} and machine G_0 (*G stands for Generic*) in Fig. 1 together as a *pattern*.

CONTEXT C_{G0}
SETS sg_0
CONSTANTS cg_0
AXIOMS ag_0

(a)

MACHINE G_0
SEES C_{G0}
VARIABLES vg_0
EVENTS eg_0

(b)

Figure 1: Pattern - Context C_{G0} and machine G_0

Machine G_0 could be part of a *pattern chain* made up by a chain of refinements as seen in Fig. 2.

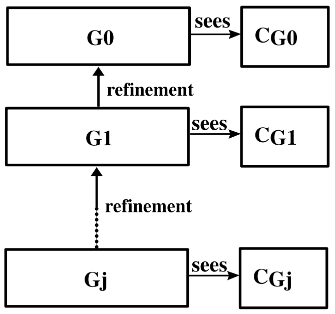
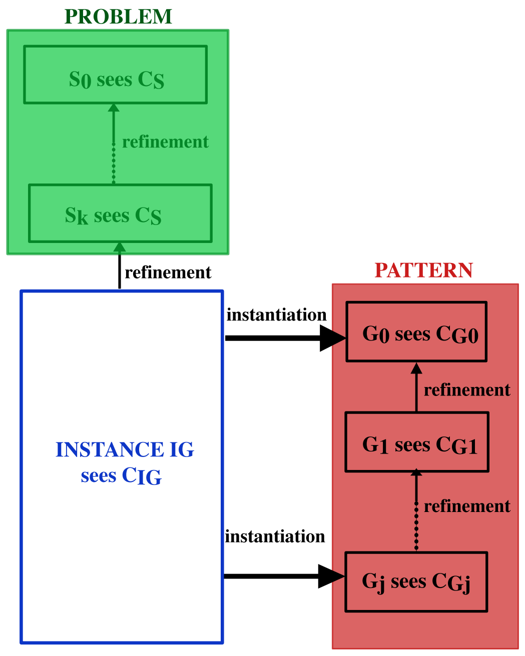


Figure 2: *Pattern*: chain of refinements

The goal is to reuse the *pattern* as an *instance* in an existing development (*problem*) consisting of a chain of refinement of machines S_0 to S_k (*S stands for Specific problem*) as seen in Fig. 3.

This is achieved by creating an *instance* of the generic pattern named *IG* as seen in Fig. 4. The *instance* sees context C_{IG} (that could extend the specific problem context C_S) containing the replacement properties (sets sig and expressions E_i that may



(a)

CONTEXT C_{IG}
EXTENDS C_S
SETS sig
CONSTANTS cig
AXIOMS aig

(b)

Figure 3: Instantiation of *pattern* $G_0 \ldots G_j$ via parameterisation context C_{IG} creating instance IG to fit specific *problem* $S_0 \ldots S_k$.

contain constants cig) for the elements in context C_{Gi}. The variables, events and parameters can also be renamed to fit new elements or existing elements in the specific *problem*: variables vs, new events es and new parameters ps.

If the *instance* is to be used as a continuation of an existing *problem*, then *problem* and *instance* are linked via refinement: the *instance* corresponding to the pattern G_0 must be a valid refinement of the last refinement of the *problem*. The relation between the states of the two machines is given by the gluing invariant J_S. The renaming of variables is required to ensure that the gluing invariant is strong enough to prove the refinement link (in particular between abstract variables vs and concrete variables vig). In Fig. 4, machine G_0 is abstractly instantiated via context C_{G0}. Consequently to ensure the aforementioned link, refinement proof obligations (PO) between S_k and IG must be discharged as described in Sect. 3.2. For each instantiated event, if an abstract parameter from the set ps disappears, then a witness needs to be explicitly added: $w1(ss, cs, sig, cig, ps, pig, vig)$; similarly, if an abstract variable from the set vs has a non-deterministic assignment and disappears, then a witness also needs to be provided: $w2(ss, cs, sig, cig, vs', pig, vig')$.

```
INSTANCE IG
REFINES $S_k$                /* problem machine; optional */
SEES $C_{IG}$     /* context containing the instance properties:$ss, cs, sig, cig$ */
INSTANTIATES ABSTRACT $G_0$ VIA $C_{G0}$ /* mandatory */
   REPLACE             /* replace parameters in context $C_{G0}$ */
     SETS $sg_i := sig_k$ .../* Carrier Sets:  $ss, sig$  */
     CONSTANTS $cg_i := E_k$ .../* Constants replaced by expressions */
   INVARIANTS $J_S$              /* Gluing invariant:$ss, cs, sig, cig, vs, vig$ */
   RENAME                        /*rename elements in machine $G_0$ */
     VARIABLES $vg_i := vig_k$ ...       /* $vs \subseteq vig$; optional */
     EVENTS $eig_k$ INSTANTIATES $eg_i$          /* optional */
                 $pg_l := pig_m$ ...           /*  parameters:  $ps \subseteq pig$;  optional */
                 WITH $w_i$ ...          /*  witnesses:  $w1(ss, cs, sig, cig, ps, pig, vig)$  or
$w2(ss, cs, sig, cig, vs', pig, vig')$; optional */
                  :

INSTANTIATES CONCRETE $G_j$ VIA $C_{Gj}$ /* optional */
   REPLACE             /* replace parameters in contexts $C_{G1}$ to $C_{Gj}$ */
     SETS $sg_j := sig_k$ /* Carrier Sets */
     CONSTANTS $cg_j := E_k$ /* Constants */
   RENAME              /*rename elements in machine $G_j$ */
     VARIABLES $vg_j := vig_k$        /* optional */
     EVENTS $eig_j$ INSTANTIATES $eg_k$        /* optional */
                 $pg_l := pig_m$        /*  parameters:  optional */
END
```

Figure 4: A generic instance IG

If the refinement PO are proved, then the rest of the *pattern* chain can be instantiated and taken for free (after the required replacement of sets and constants and renaming of variables, events and parameters). This can be specified by selecting which concrete instance is to be instantiated and which context is to be used (in Fig. 4, the machine G_j is instantiated via context C_{Gj}). Several instances can be created from the same *pattern* to fit specific a *problem*.

3.1 Static Checks

To ensure a valid instantiation of machines, several static checks must be taken into account:

1. A static validation of replaced elements is required, e.g., a type must be replaced by a type and a constant by another constant or an expression.

2. All sets and constants should be replaced, i.e., no uninstantiated parameters.

3. Renaming the constants, variables and events must be injective (not introducing name clashes) in order to reuse all the existing proof obligations.

4. Replacing sets does not have to be injective. Different sets in the *pattern* can be replaced by the same *instance* set.

5. Only given sets (defined by the user) can be replaced. Built-in types such as integer numbers $\mathbb{Z}$ and boolean BOOL cannot be replaced.

6. At most one machine (S_k in Fig. 4) can be refined per instance.

7. Contexts may be seen by the instance (context C_{IG} in Fig. 4). The sets and constants used in the replacement section are extracted from these contexts or are built-in types.

8. An abstract *pattern* machine must be defined to be instantiated. At least one context is to be used during this instantiation (context C_{G0} in Fig. 4).

   (a) The sets and constants to be replaced are extracted from these contexts. They should be replaced by the elements available the seen contexts (i.e. context C_{IG}).

   (b) The variables, events and parameters can be renamed as long as they do not introduce name clashes. Moreover if the instance refines a machine, then variable, event and parameter should be renamed accordingly to ensure a valid refinement.

9. A concrete *pattern* machine can be defined to be instantiated. At least one context is to be used during this instantiation (context C_{Gj} in Fig. 4).

   (a) The sets and constants to be replaced are extracted from these contexts. They should be replaced by the elements available the seen contexts (i.e. context C_{IG}).

   (b) The variables, events and parameters can be renamed as long as they do not introduce name clashes.

   (c) The concrete *pattern* machine must be a refinement of the abstract *pattern* machine (i.e. $S_0 \sqsubseteq S_k$).

3.2 Proof Obligations

In this section, we address the proof obligations necessary to validate the generic instantiation of machines.

3.2.1 Pattern Assumptions and Instance Theorems

Axioms in contexts are assumptions about a system and are used to help discharge proofs obligations. When instantiating, we need to show that assumptions in the *pattern* are satisfied by the replacement sets and constants. The verification of this assumption is achieved through the generation of proof obligations where the *pattern* axioms are converted into instantiated theorems after the replacement is applied.

3.2.2 Instantiation and Proof Obligations

A specific machine S_k is characterised by:

$$\boxed{\begin{array}{l}\textbf{MACHINE } S_k\\ \textbf{SEES } C_{Sk}\\ \textbf{VARIABLES } vs\\ \textbf{INVARIANTS } Is(ss, cs, vs)\\ \textbf{EVENTS } es\end{array}}$$

An event es_i of S_k is defined as:

$$\begin{array}{l} es_i \mathrel{\widehat{=}} \textbf{ANY } ps \textbf{ WHERE } Gs(ss, cs, ps, vs)\\ \qquad \textbf{THEN } Ss(ss, cs, ps, vs, vs') \textbf{ END}.\end{array}$$

A generic machine G_0 is characterised by:

$$\boxed{\begin{array}{l}\textbf{MACHINE } G_0\\ \textbf{SEES } C_{G0}\\ \textbf{VARIABLES } vg_0\\ \textbf{INVARIANTS } Ig_0(sg_0, cg_0, vg_0)\\ \textbf{EVENTS } eg\end{array}}$$

An event eg_i of G_0 is defined as:

$$\begin{array}{l} eg_i \mathrel{\widehat{=}} \textbf{ANY } pg \textbf{ WHERE } Hg(sg_0, cg_0, pg, vg_0)\\ \qquad \textbf{THEN } Tg(sg_0, cg_0, pg, vg_0, vg_0') \textbf{ END}.\end{array}$$

Similarly, an instance IG_0 is characterised by:

$$\boxed{\begin{array}{l}\textbf{MACHINE } IG_0\\ \textbf{SEES } C_{IG}\\ \textbf{VARIABLES } vig\\ \textbf{INVARIANTS } Iig(ss, cs, sig, cig, vig)\\ \textbf{EVENTS } eig\end{array}}$$

Moreover, we can add that $vs \subseteq vig$ and $ps \subseteq pig$. An event eig_i of IG_0 is defined as:

$$\begin{array}{l} eig_i \mathrel{\widehat{=}} \textbf{ANY } pig \textbf{ WHERE } Hig(ss, cs, sig, cig, pig, vig)\\ \quad \textbf{WITH } ps : Wig_1(ps, ss, cs, sig, cig, vig, pig)\\ \qquad\qquad vs' : Wig_2(vs', ss, cs, sig, cig, vig, pig, vig')\\ \quad \textbf{THEN } Tig(ss, cs, sig, cig, pig, vig, vig') \textbf{ END}. \end{array}$$

For the generic instantiation, there are two different situations where proof obligations are required:

- Ensure that an ***instance*** is a valid instantiation of a ***pattern***. This requires that ***pattern*** axioms are proved in the instance (and that the renaming of elements does not introduce name clashes). This was addressed in the previous section. For each generic axiom ag_i, the proof obligation that needs to be discharged is:

$$\boxed{ag_i/AXM\text{:} \quad \begin{array}{l} AS \wedge AIG\\ \vdash \sigma ag_i \end{array}} \tag{1}$$

  AS stands for the axioms of the specific machine given by $as(sg, cg)$; AIG stands for the axioms of the instance given by $aig(sig, cig)$; σ represents all substitutions of sets, constants and variables given by
  $[sg_i := sig_k, \ldots, cg_i := E_k, \ldots, vg_i := vig_k, \ldots]$; σag_i is the result of the substitution applied to the generic axiom ag_i.

- When an instance refines an existing model, the abstract instance machine must be a valid refinement of the existing model. This means that $S_k \sqsubseteq IG_0$ (where IG_0 is an valid instance of pattern G_0). This is addressed below in Sect. 3.2.2.

Refinement PO (REF): For each event in IG_0, the refinement PO ensure that abstract actions of events in S_k are simulated by the concrete ones, that each abstract guard is at least as weak as the concrete one and that when an abstract variable is data refined by a concrete one and disappears, gluing invariants exist linking the abstract and concrete variables.

For event eig_i, the refinement PO between S_k and IG_0 is given by:

$$\boxed{\textit{eig}_i\textit{/REF}: \begin{array}{l} AS \wedge \sigma AG \\ IS \\ JS \\ \sigma HG \\ \sigma TG \\ \vdash \exists vs' \cdot GS \wedge SS \wedge JS \end{array}} \tag{2}$$

AS stands for the axioms of the specific machine S_k given by $as(ss, cs)$; σAG is the substitution applied to the generic axioms AG ; IS stands for the invariant of S_k given by $Is(ss, cs, vs)$; JS is the gluing invariant given by $Js(ss, cs, sig, cig, vig)$; HG are the guards of the event eg_i and TG are the assignments for the same event: therefore σHG represents the substitution applied over the guards HG resulting in guard of event eig_i Hig (similar for σTG); GS and SS are the guards and actions for the event es_i in S_k.

The use of witnesses allows the separation of the previous proof rule in three parts: proof rules Gluing Invariant Preservation (3), Guard Strengthening (4) and Simulation (5). In practice, when discharging POs, it is simpler to deal with one part of the refinement PO at a time instead of dealing with all at once. We do not address the technical parts about the *partition* of the refinement POs but more details can be found in [1]. When non-deterministic witnesses are used, a proof obligation is generated to ensure that the witness is feasible. If the parameters and variables between refined events do not match, we may need to provide a mechanism to add this information.

Gluing Invariant Preservation (INV): In a refinement, concrete invariants must be preserved for each concrete event. The hypotheses include axioms, abstract invariants and theorems plus concrete invariants and theorems, concrete guards, witnesses predicates for variables and concrete before-after predicates. The goal is each concrete invariant from the set of invariants in the refinement. For event eig_i and each of the invariants j_S in JS, the respective proof obligation rule is given by (3).

$$\boxed{\textit{eig}_i\textit{/inv/INV}: \begin{array}{l} AS \wedge \sigma AG \\ IS \\ JS \\ \sigma HG \\ WIG_2 \\ \sigma TG \\ \vdash j_S' \end{array}} \tag{3}$$

WIG_2 stands for the witness predicates corresponding to each disappearing abstract variable vs' with non-deterministic assignments and given by $Wig_2(vs', ss, cs, sig, cig, vig, pig, vig')$; $j_S{}'$ is each invariant from the set J_S where the variables are replaced by their before-after state: $j_S{}'(ss, cs, sig, cig, vig')$.

Guard Strengthening (GRD): It ensures that each abstract guard is at least as weak as the concrete one in the refining event. As a consequence, when a concrete event is enabled, the corresponding abstract one is also enabled. The hypotheses include axioms, abstract invariants and theorems, concrete invariants and theorems, concrete guards and witness predicates for parameters. The goal is each individual abstract guard from the set of abstract guards. For event eig_i and each of the abstract guards $g_S(ps, ss, cs, vs)$, this proof obligation is given by (4).

$$\boxed{\textit{eig}_i\textit{/grd/GRD}: \begin{array}{l} AS \wedge \sigma AG \\ IS \\ JS \\ \sigma HG \\ WIG_1 \\ \vdash g_S \end{array}} \tag{4}$$

WIG_1 stands for each disappearing abstract parameter in an abstract event and it is given by $Wig_1(ps, ss, cs, sig, cig, vig, pig)$.

Simulation (SIM): It ensures that each action in a concrete event simulates the corresponding abstract action. When a concrete action is executed, the corresponding abstract one should not be contradicted. The hypotheses include axioms, abstract invariants and theorems, concrete invariants and theorems, concrete guards, witness predicates for refined parameters, witness predicate for refined abstract variables and the concrete before-after predicate for each concrete event. The goal is each individual abstract before-after predicate from the set of abstract assignments. For event eig_i and one of the respective actions *act*, this proof obligation is given by (5).

$$\boxed{\textit{eig}_i\textit{/act/SIM}: \begin{array}{l} AS \wedge \sigma AG \\ IS \\ JS \\ \sigma HG \\ WIG_1 \\ WIG_2 \\ \sigma TG \\ \vdash SS \end{array}} \tag{5}$$

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, April 2010.

[2] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28, 2007.

[3] Christophe Métayer, Jean-Raymond Abrial, and Laurent Voisin. Event-B Language. Technical report, Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005.

[4] Matthias Schmalz. Term Rewriting in Logics of Partial Functions. *Proceedings of ICFEM 2011*, 2011.