Proposal for Generic Instantiation of Machines in the Rodin Platform

Renato Silva, Michael Butler

Dependable Systems and Software Engineering Group
Department of Electronics and Computer Science
University of Southampton

April 29, 2009

Abstract

This document is a proposal for generic instantiation of machines in the Rodin platform. Event-B [1] is the modelling language that will be used during the description of the procedure. This approach is based on the Shared Event Compositon (B-style) and Refactory plug-ins. Instantiated Machines are introduced and used for the developments of generic models instances.

1 Introduction

Generic Instantiation can be seen as a way of reusing components and solve difficulties raised by the construction of large models [1]. To create several components with similar properties, instead of creating the components from the scratch, re-usability is applied through the use of a *template or pattern* as the basic structure and afterwards each new component is generated through personalisation. This technique is well-established in areas like mathematics and even in other formal methods like B or theorem provers like Isabelle.

We propose a generic instantiation approach for Event-B by instantiating machines. The instances inherit properties (variables, sets, constants) of the generic machine (pattern) and afterwards are *personalised* by renaming/replacing those properties to more specific names to the instance. There is also a generation of proofs obligations to ensure assumptions (axioms) used in the patterns are satisfied by the instantiation.

Overview Section 2 defines how generic instantiation is interpreted by us. In section 3 instantiated machines are introduced. Section 4 gives an application of instantiation in particular together with composition. Section 5 discusses an open question that arise when instantiating theorems and invariants in a pattern. Section 6 describe the related work for generic instantiation. In the end, a summary is given and conclusions are withdrawn about the proposed implementation in Section 7. During the presentation a case study modelling a protocol communication is used.

2 Generic Instantiation

In order to explain our approach for Generic Instantiation we will use a simple case study. A protocol is modelled between two entities, a source and a target, that communicate through a channel. The content of the channel (buffer) has a maximum dimension. To send a message it is necessary to add the content of the message to the channel.

Based on the proposed requirements is possible to create a context to model the channel:

CONTEXT ChannelParameters

SETS

Message

CONSTANTS

max_size

AXIOMS

axm1 : $max\_size \in \mathbb{N}$

END

FIGURE 1: ChannelParameters context

The content of the message is of type *Message* and has a maximum dimension *max_size*.

At the machine side, a variable *channel* stores all the messages that are sent/received. The *channel* messages have type *Message* and the number of messages in the buffer is limited. *channel* is initialised empty. Messages are added to *channel* to be sent:

MACHINE Channel

SEES ChannelParameters

VARIABLES

channel

INVARIANTS

inv1 : $channel \subseteq Message$

inv2 : $card(channel) \leq max\_size$

EVENTS

Initialisation

begin

act1 : $channel := \varnothing$

end

Event *Send* $\widehat{=}$

any

m

where

grd1 : $m \in Message$

grd2 : $card(channel) < max\_size$

then

act1 : $channel := channel \cup \{m\}$

end

Event *Receive* $\widehat{=}$

any

m

where

grd1 : $m \in channel$

then

skip

end

END

Elements in *ChannelParameters* context serve as parameters (types and constant) for the *Channel* machine.

Now suppose we wish to model a bi-directional communication between two entities using two channels. Both channels are similar so an option is to *instantiate machine Channel* twice to create two instances: one channel called *Request* and the other *Response*. This model separates the entity that sends the message (Source) and the entity that receives the message (Target) as represented in Figure 2.

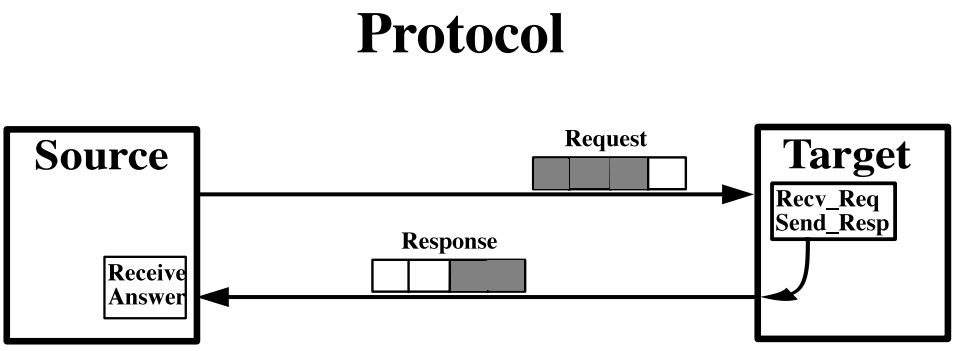


FIGURE 2: Protocol diagram

The instantiation of *Channel* is achieved by applying *machine instantiation*. An instance of the pattern *Channel* is created with more specific properties. A detailed description of the machine instantiation is described in the Section 3. Moreover, a context containing the specific instances properties is required to model the protocol. In our case study we use the context *ProtocolTypes*, where types *Request* and *Response* replace the more generic type *Message* and constants *qmax_size* and *pmax_size* replace *max_size*. The context *ProtocolTypes* can be seen in Figure 3.

CONTEXT ProtocolTypes

SETS

Request

Response

CONSTANTS

qmax_size

pmax_size

AXIOMS

axm1 : $qmax\_size \in \mathbb{N}$

axm2 : $pmax\_size \in \mathbb{N}$

END

FIGURE 3: ProtocolTypes Context

3 Generic Instantiation and Instantiated Machines

Inspired by the previous case study, having the ability to compose machines (Shared Event Composition plug-in) and renaming elements in the Rodin platform (Refactory plug-in), we propose an approach to instantiate machines. As mentioned the context plays an important role while instantiating since is where the specific properties of the instance are defined. The use of context is briefly discussed before *instantiated machines* are introduced.

3.1 Contexts

Contexts in Event-B are the static part of a model. Contexts store information that does not change during the development of specifications and contain properties of the modelled system through the use of axioms and theorems. Furthermore, having a closer look at the possible usages of contexts, there are two possible viewpoints:

Parameterisation the context is seen only by one machine and defines specific properties for that machine (sets, constants, axioms, theorems). These properties are unique for that machine and any other machine would have different properties.

Sharing a context is seen by several machines and there are some properties (sets, constants, axioms, theorems) that are shared by the machines. Therefore the context is used to share properties.

In several model developments there exist a mixture of both usages for the same context. For the general model developers this distinction is not very clear and perhaps not so important. For our approach of generic instantiation is important to distinguish since to reuse components and personalise each instance ***Parameterisation*** is used.

3.2 Example of INSTANTIATED MACHINE

An INSTANTIATED MACHINE instantiates a generic machine (pattern). It is characterised by a name, by which machine is used as generic and by which variables and events are renamed. If the generic machines sees a context, then the context elements (sets and constants) have to be replaced by instance elements. The instance elements must exist already in a context seen by the instantiated machine (in our case study, this corresponds to ProtocolTypes - see Figure 3).

Returning to the case study, the instantiated machine *QChannel* that is an instance of the machine *Channel* for requests looks like this:

INSTANTIATED MACHINE QChannel
INSTANTIATES Channel **BY** ChannelParameter
SEES ProtocolTypes /* `context containing the instance properties` */
REPLACE /* `replace parameters defined in ChannelParameters` */
    SETS *Message := Request*
    CONSTANTS *max_size := qmax_size*
RENAME /* `rename variables and events at Channel machine` */
    VARIABLES *channel := qchannel*
    EVENTS *Send := QSend*
END

FIGURE 4: Instantiated Machine: QChannel instantiates Channel

Note that context elements (sets and constants) are `replaced` because the replacement elements are already defined in *ProtocolTypes*. Machine elements (variables, parameters and events) are `renamed` instead since they do not exist before. The instantiated machine *PChannel* that is an instance of *Channel* for responses is similar.

Axioms in contexts are assumptions about sets and constants used in proofs of machines. When instantiating, we need to show that these assumptions are satisfied by the replacement sets and constants. As a result, axioms in the context are converted into theorems in the instantiated machine after the replacement is applied. A theorem has a proof obligation associated. By assuring that a proof obligation related to each axiom is generated and discharged, we are confirming the correctness of the instantiation because the assumptions in the pattern are satisfied. "Expanding" machine *QChannel* can be seen in Figure 5:

MACHINE QChannel

SEES ProtocolTypes

VARIABLES

qchannel

INVARIANTS

inv1 : $qchannel \subseteq Request$

inv2 : $card(qchannel) \leq qmax\_size$

THEOREMS

thm1 : $qmax\_size \in \mathbb{N}$

thm2 : $pmax\_size \in \mathbb{N}$

EVENTS

Initialisation

begin

act1 : $qchannel := \varnothing$

end

Event $QSend \mathrel{\widehat{=}}$

any

q

where

grd1 : $q \in Request$

grd2 : $card(qchannel) < qmax\_size$

then

act1 : $qchannel := qchannel \cup \{q\}$

end

Event $Receive \mathrel{\widehat{=}}$

any

q

where

grd1 : $q \in qchannel$

then

skip

end

END

FIGURE 5: Expanded version of instantiated machine QChannel

QChannel sees the context *ProtocolTypes* that contains the context information for the instances, the type *Message* in context *ChannelParameters* was replaced by *Request* in *ProtocolTypes*, the constant *max_size* was replaced by *qmax_size*, the variable *channel* in *Channel* was renamed to *qchannel* and event *Send* was renamed to *QSend*. The axiom that exists in *ChannelParameter* was converted to a theorem in machine *QChannel* (but easily discharged by the axioms in *ProtocolTypes*). Note that the expansion is not required in practice. We use it to show the meaning of an INSTANTIATED MACHINE.

3.3 Definition of Generic Instantiation of Machines

Owing to the example above, a general definition of generic instantiation in machines can be drawn. If we have a context C and a machine M defined as follows:

CONTEXT C
SETS $S_1 ... S_m$
CONSTANTS $C_1 ... C_n$
AXIOMS $Ax_1 ... Ax_p$

(a) Context C

MACHINE M
SEES C
VARIABLES $v_1 ... v_q$
EVENTS $ev_1 ... ev_r$

(b) Machine M

FIGURE 6: Generic view of a context and a machine

Based on context C and machine M that together can be seen as a *pattern*, we can create an Instantiatiated Machine *IM* as follows:

INSTANTIATED MACHINE IM
INSTANTIATES M **BY** C
SEES D /* `context containing the instance properties` */
REPLACE /* `replace parameters defined in context C` */
  SETS $S_1 := DS_1, \ldots, S_m := DS_m$
  CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$
RENAME /* `rename variables and events at machine M` */
  VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$
  EVENTS $ev_1 := nev_1, \ldots, ev_r := nev_r$ /* `optional` */
END

FIGURE 7: An Instantiated Machine

The context D contains the replacement parameters (sets $DS_1, \ldots, DS_m$ and constants $DC_1, \ldots, DC_n$) for the elements in context C. The variables and events are also renamed by variables $nv_1, \ldots, nv_q$ and events $nev_1, \ldots, nev_r$. From the *pattern* we are able to create several instances that can be used in a more specific *problem*. During the creation of instances validity checks are required:

1. A static validation of replaced elements is required, e.g., must replace a type with a type (or a constant set), a constant with a constant.
2. All sets and constants should be replaced, i.e., no uninstantiated parameters.
3. A static check must be done to assure that the instantiated machine specifies which generic context is being instantiated.
4. All variables should be renamed to avoid uninstantiated parameters. The same applies for variable occurrences.
5. Renaming events or replacing event parameters by an expression (optional).

4 Example of Instantiation and Composition

After the creation of the instances the system modelling carries on. In our case study, we model a protocol between entities that sends and receives messages to each other. By using the created instances and the Shared Event Composition plug-in we share events between Request and Response and model the protocol. A composed machine *Protocol* modelling this system can be seen in Figure 8:

COMPOSED MACHINE Protocol
INCLUDES
PChannel
QChannel
EVENTS
SendRequest
Combines Events QChannel.QSend
RecvReq_SendResp
Combines Events QChannel.Receive || PChannel.Send
RecvResp
Combines Events combines PChannel.Receive
END

FIGURE 8: Composed Machine Protocol

Like seen in Figure 2, while composing the instance machines *PChannel* and *QChannel* we add the events that are unique for each entity (*SendRequest* and *RecvResp*). *SendRequest* sends a message through the channel from the *Source* to *Target*. *RecvResp* models the reception of the response in the *Source* after being sent by the *Target*. Moreover the event that relates the communication between the two entities is also modelled (*RecvReq_SendResp*). While the request is received, the response to that request is also sent in parallel. (From this combined event, a possible refinement could be processing the request message before sending the response.)

The composed machine *Protocol* corresponds to the following expanded machine:

MACHINE Protocol

SEES ProtocolTypes

VARIABLES

qchannel

pchannel

INVARIANTS

inv1 : $qchannel \subseteq Request$

inv2 : $pchannel \subseteq Response$

inv3 : $card(pchannel) \leq pmax\_size$

inv4 : $card(qchannel) \leq qmax\_size$

THEOREMS

QChannel/thm1 : $qmax\_size \in \mathbb{N}$

QChannel/thm2 : $pmax\_size \in \mathbb{N}$

PChannel/thm1 : $qmax\_size \in \mathbb{N}$

PChannel/thm2 : $pmax\_size \in \mathbb{N}$

EVENTS

Initialisation

begin

act1 : $qchannel := \varnothing$

act2 : $pchannel := \varnothing$

end

Event *SendRequest* $\widehat{=}$

any

q

where

grd1 : $q \in Request$

grd2 : $card(qchannel) < qmax\_size$

then

act1 : $qchannel := qchannel \cup \{q\}$

end

Event *RecvReq_SendResp* $\widehat{=}$

any

q

p

where

grd1 : $q \in qchannel$

grd2 : $p \in Response$

grd3 : $card(pchannel) < pmax\_size$

then

act1 : $pchannel := pchannel \cup \{p\}$

end

Event *RecvResp* $\widehat{=}$

any

p

where

grd1 : $p \in pchannel$

then

skip

end

END

By creating two instance machines *Channel*, it was possible to model a bi-directional communication channel between two entities (see figure 2). Notwithstanding a simple case study, it allows us to express the applicability of generic instantiation for modelling distributed systems but not necessarily restricted to this kind of system. When modelling a finite number of similar components with some specific individual properties, instantiated machines are a suitable option.

5 Instantiating Theorems and Invariants

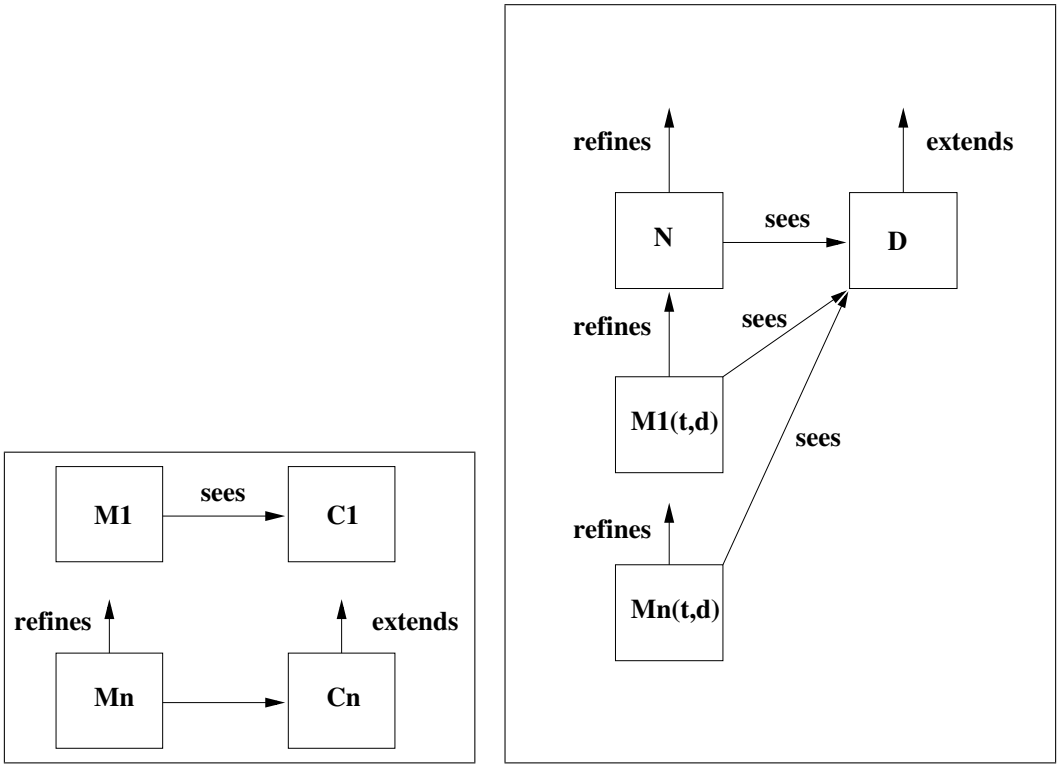
Theorems in contexts and machines are assertions about characteristics and properties of the system. Theorems have proof obligations associated that are discharged based on the model assumptions. Once the theorems are discharged, they can be used as hypothesis for discharging other proof obligations in the model since they work as a consequence of the assumptions (axioms and invariants). On the other hand, invariants in machines are assumptions about the model that need be respected in all states of the system.

An interesting question arise when a pattern is instantiated and contains theorems and invariants. Having the theorem proof obligation discharged, by creating an instance we would not want to re-prove the theorem proof. Regarding the invariants and respective proof obligations we would have a similar situation where we would not want to discharge proof obligations in the instance if they were already discharged in the pattern. Ideally we would like to add to the instance the assumptions and assertions given by the theorems and invariants without the hassle of re-proving them. Although addressed here as an open question, this situation suggest a kind of different theorem, a *proved-theorem* to be used in the instance. A *proved-theorem* would be similar to a theorem but it would not have a proof obligation associated. The invariants imported from the pattern would fall under the same category where the respective proof obligations should not be re-generated. Informally the instances are just renaming and replacing elements without changing the semantics under the original pattern (if the validity checks are followed) so theorems and invariants would work as assumptions in the instantiated machine. The assumptions in the pattern (axioms) need to be satisfied by the instances though the generation of proof obligations but the same does not apply for invariants and theorems that are assertions of the pattern.

6 Related Work

Initially our approach aimed to instantiate both machines and contexts and deal with the instantiation of each independently. But we realise that our current approach does not instantiate a context. Instead re-uses a component that already exists. Therefore the context parameterisation is done in the instantiated machine. Despite that, we believe that further study is required to figure out if context instantiation is a worthwhile approach while modelling.

[2] makes use of generic instantiation using Event-B. The *personalization* of the components is described as *parameterisation* of machines in order to reuse refinements. If a development consists in refinements of machines and respective contexts using the same list of sets s and constants c (figure 9(a)), this development is said to be generic with regard to these carrier sets and constants. A continuation of a development could simply reuse a first development with some slight changes consisting of instantiating sets s and constants c. Assuming that sets s are independent of each other is proved that is possible to replace all the contexts by a generic context D with sets t and constant d, as long as the machines are instantiated with the most recent elements (t and d), thus implicitly seeing D (figure 9(b)). The existing context D instantiates the respective context (C1 to Cn) of each refined machines (M1 to Mn). Afterwards the development could be resumed from Mn(t,d). To reuse also the proofs in M1...Mn and C1...Cn without actually redoing it, the axioms related with sets s and constant c become theorems after the instantiation.



(a) Refinement chain and respective contexts

(b) Generic Instantiation: Reuse of context `D`

FIGURE 9: Generic Instantiation according to [2]

This approach applies the instantiation at the context level which is different from our approach that applies the instantiation at the machine level. The use of generic instantiation is applied to a chain of refinements and we apply to a single machine. Since generic instantiation is a very general concept will believe that are different possible approaches for reusing components and this two options are

just in a list of more possibilities. Further study is required for instantiating context in a similar way that we do for instantiated machines and if it worths to be explored. By identifying suitable scenarios where reuse benefits the model, more generic instantiation approaches will arise.

7 Conclusions

This documents proposes an implementation of generic instantiation applied to the Rodin platform using Event-B notation. The motivation for such implementation is related with reusability of components and models that already exist. By creating an instance from a generic model, a new personalised model is created based on a generic model and on new specific properties. Generic instantiation is applied to instantiated machines and respective parameterisation contexts. An *instantiated machine* instantiates a generic machine, parameterise a context and rename/replace the necessary elements to adjust to the new instances. A practical case that models a communication protocol between two entities is presented to show the advantages of using generic instantiation and in particular how to use our approach in the Rodin platform[1]. These document outlines only a proposal for the instantiation of machine. Some methodological points arise for a possible implementation like what to do with theorems and invariants in the instances and they are left as open questions. A comparation is made between a generic instantiation applied to a refinement chain and our approach, applied to individual machines.

References

[1] C. Métayer, J.-R. Abrial, and L. Voisin, "Event-B Language," tech. rep., Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005.

[2] J.-R. Abrial and S. Hallerstede, "Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B," *Fundam. Inf.*, vol. 77, no. 1-2, pp. 1–28, 2007.

[1] The instantiation context and machine are just concepts currently and are not yet implemented in the Rodin platform.