Supporting Reuse of Event-B Developments through Generic Instantiation

Renato Silva and Michael Butler

School of Electronics and Computer Science
University of Southampton, UK
ras07r,mjb@ecs.soton.ac.uk

Abstract. 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 is a formal method that allows modelling and refinement of systems. Event-B supports generic developments through the context construct. Nevertheless Event-B lacks the ability to instantiate and reuse generic developments in other formal developments. We propose a way of instantiating generic models and extending the instantiation to a chain of refinements. We define sufficient proof obligations to ensure that the proofs associated to a generic development remain valid in an instantiated development thus avoiding re-proofs.

Key words: formal methods, event-B, reusability, generic instantiation

1 Introduction

Reusability has always been sought in several areas as a way to reduce time, cost and improve the productivity of developments [1]. Examples can be found in areas like software, mathematics and even formal methods. Generic Instantiation can be seen as a way of reusing components and solving difficulties raised by the construction of large and complex models [2, 3]. The goal is to reuse generic developments (single model or a chain of refinements) and create components with similar properties instead of starting from scratch. Reusability is applied through the use of a *pattern* as the basic structure and afterwards each new component is generated through parameterisation.

We propose a generic instantiation approach for Event-B by instantiating machines. The instances inherit properties from the generic development (pattern) and afterwards are *parameterised* by renaming/replacing those properties to more specific names according to the instance. Proofs obligations are generated to ensure that assumptions used in the pattern are satisfied in the instantiation. In that sense our approach avoids re-proof pattern proof obligations in the instantiation. The models are developed in the Rodin platform [4], which is a toolset for Event-B [5]. A simple case study modelling a protocol communication is described to illustrate the use of instantiation.

A brief overview of the Event-B Language is given in Section 2. Section 3 defines how generic instantiation is interpreted by us. In section 4 instantiated machines are introduced. Section 5 gives an application of instantiation in combination with shared event composition. The application of instantiation to a chain of refinements is described in Section 6. Section 7 discusses an open question that arises when instantiating theorems and invariants in a pattern.

2 Event-B Language

Event-B is a formal methodology that uses mathematical techniques based on set theory and first order logic allowing the specification of systems. An abstract Event-B specification is divided into two parts: a static part called *context* and a dynamic part called *machine*. A machine *SEES* as many contexts as desired. The context consists of sets, constants and assumptions (axioms) of the system. Sets in the context can be seen as a collection of elements or a type definition. The machine contains the state variables whose values are assigned in *events*. Events can only occur when enable by their *guards* being true and as a result *actions* are executed. Events can have *parameters* that are local variables to the event and can be used by the guards or by the actions. The *INVARIANT* defines the dynamic properties of the specification. Proof obligations are generated to verify that the invariant is maintained before and after an event is enabled. Theorems are properties of the system that have proof obligations associated and usually are discharged based on other properties of the specified system.

An abstract Event-B specification can be refined by adding more details and becoming closer to the implementation (more concrete). A context *EXTENDS* an abstract context by adding sets, constants or axioms. Nonetheless the abstract context properties are still assumed. Refinement of a machine consists in refining existing events. The relation between variables in the concrete and abstract model is given by a *gluing invariant*. Proof obligations are generated to ensure that this invariant is preserved in the concrete model. Also it is possible to add new events that refine *skip* as long as the new events do not execute forever and the abstract events are not hampered.

3 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, Source and Destination, which communicate by sending messages through a channel. The content of the channel 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 it is possible to create a context *ChannelParameters* to model the channel as seen in Fig. 1b.

The content of the message is of type *Message* and has a maximum dimension *max_size*. Figure 1a represents the machine side where a variable *channel* stores all the sent/received messages. The *channel* messages have type *Message* and

```
machine Channel sees ChannelParameters

variables channel

invariants
  @inv1 channel ⊆ Message
  @inv3 finite(channel)
  @inv2 card(channel) ≤ max_size

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
  end

  event Send
    any m
    where
      @grd1 m ∈ Message
      @grd2 card(channel) < max_size
    then
      @act1 channel ≔ channel ∪ {m}
  end

  event Receive
    any m
    where
      @grd1 m ∈ channel
    then
      @act1 channel ≔ channel\{m}
  end
end
```

(a)

```
context ChannelParameters

constants max_size

sets Message

axioms
  @axm1 max_size ∈ ℕ
end
```

(b)

Fig. 1. Machine *Channel* and respective context *ChannelParameters*

the number of messages in the channel is limited. Messages are introduced in the *channel* to be sent as seen in event *Send*. The event *Receive* models the reception of the message in the destination by extracting the messages from the *channel*. Elements in *ChannelParameters* context are the parameters (type 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*. The protocol, represented in Fig. 2 starts by a message being sent from the Source. After arriving at the Destination, the reception of the message is acknowledged in the Source. Then a response is sent from the Destination and after arriving at the Source, it is also acknowledged in the Destination.

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 Section 4. Moreover, a context containing the specific instances properties is required to model the protocol. In our case study we use the context *ProtocolTypes* in Fig. 3, where types *Request* and *Response* replace the more generic type *Message* and constants *qmax_size* and *pmax_size* replace *max_size*. This context must be provided by the modeller/developer.

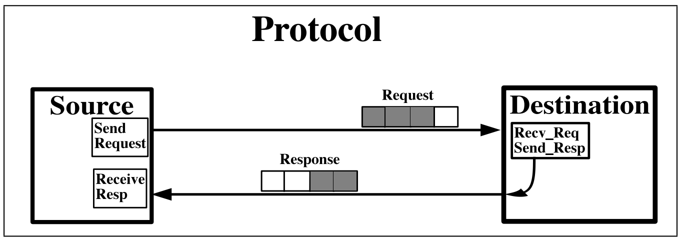

Fig. 2. Protocol diagram

```
context ProtocolTypes

constants qmax_size pmax_size

sets Request Response

axioms
  @axm1 qmax_size ∈ ℕ
  @axm2 pmax_size ∈ ℕ
end
```

Fig. 3. ProtocolTypes Context

Abrial and Hallerstede [3] and Métayer et al [2] propose the use of generic instantiation for Event-B. It is suggested that the contexts of a development (equivalent to the 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. Therefore our work proposes a mechanism to instantiate machines and extend the instantiation to a refinement chain. The reusability of a development is expressed by instantiating a development (*pattern*) according to a more specific *problem*.

4 Generic Instantiation and Instantiated Machines

Inspired by the previous case study and having the ability to compose machines (Shared Event Composition plug-in [6]) and rename elements (Refactory plug-in[7]) in the Rodin platform, we propose an approach to instantiate machines. As mentioned the context plays an important role while instantiating since this is where the specific properties of the instance are defined (parameterisation). The use of context is briefly discussed before *instantiated machines* are introduced.

4.1 Contexts

As aforementioned, contexts in Event-B are the static part of a model containing properties of the modelled system through the use of axioms and theorems. Furthermore, having a closer look at the possible usage of contexts, there are two possible viewpoints:

Parameterisation : the context is seen only by one machine (or one chain of machine refinements) 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.

Several model developments mix both usages for the same context. For the ordinary modeller this distinction is not very clear and perhaps not so important. Our approach of generic instantiation reuses components and personalises each instance implying the use of ***Parameterisation***.

4.2 Example of INSTANTIATED MACHINE

An INSTANTIATED MACHINE instantiates a generic machine (pattern). If the generic machine 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 Fig. 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 VIA ChannelParameters
SEES ProtocolTypes /* context containing the instance properties*/
REPLACE           /* replace parameters in ChannelParameters*/
    SETS Message := Request
    CONSTANTS max_size := qmax_size
RENAME          /* rename variables and events in machine Channel*/
    VARIABLES channel := qchannel
    EVENTS Send := QSend
                m := q   /*optional:rename parameter m in event Send*/
                Receive := Receive
                m := q /*optional:rename parameter m in event Receive*/
END
```

Fig. 4. Instantiated Machine: *QChannel* instantiates *Channel*

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

Axioms in contexts are assumptions about the system and are used for discharging proofs obligations. When instantiating, we need to show that assumptions in the pattern are satisfied by the replacement sets and constants. A possible solution is to convert the *pattern axioms* into *instantiated machine theorems* after the replacement is applied. A theorem has a proof obligation associated. By ensuring that a proof obligation related to each axiom is generated and discharged, we are confirming the correctness of the instantiation by satisfying the pattern assumptions (see theorem *thm1* in Fig. 5). "Expanding" machine *QChannel* can be seen in Fig. 5.

```
machine QChannel sees ProtocolTypes

variables qchannel

invariants
  @inv1 qchannel ⊆ Request
  @inv3 finite(qchannel)
  @inv2 card(qchannel) ≤ qmax_size
  theorem @thm1 qmax_size ∈ ℕ

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
  end

  event QSend
    any q
    where
      @grd1 q ∈ Request
      @grd2 card(qchannel) < qmax_size
    then
      @act1 qchannel ≔ qchannel ∪ {q}
  end

  event Receive
    any q
    where
      @grd1 q ∈ qchannel
    then
      @act1 qchannel ≔ qchannel∖{q}
  end
end
```

Fig. 5. Expanded version of instantiated machine QChannel

The instance *QChannel* sees the context *ProtocolTypes* (provided by the modeller/developer) that contains the context information for the instances. The type *Message* in context *ChannelParameters* is replaced by *Request* in *ProtocolTypes*, the constant *max_size* is replaced by *qmax_size*, the variable *channel* in *Channel* is renamed *qchannel* and event *Send* is renamed *QSend*. The axiom that exists in *ChannelParameters* is converted into a theorem in *QChannel* (but easily discharged by the axioms in *ProtocolTypes*). We convert the axiom *axm1* from the generic context *ChannelParameters*:

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

into the theorem *thm1* in the instance *QChannel*:

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

This results from the replacement of the constant *max_size* by *qmax_size*. A proof obligation is a sequent of the shape:

$$\boxed{\begin{array}{l} Hypothesis \\ \vdash \\ Goal \end{array}}$$

For a machine theorem, the respective proof obligation is [8]:

$$\boxed{\begin{array}{l} Axioms \\ Invariants \\ \vdash \\ Theorem \end{array}}$$

For theorem *thm1*, the proof obligation to be generated is the following:

$$\boxed{\begin{array}{l} qmax\_size \in \mathbb{N} \quad \texttt{/*axiom from } \textit{ProtocolTypes}\texttt{*/} \\ pmax\_size \in \mathbb{N} \quad \texttt{/* axiom from } \textit{ProtocolTypes}\texttt{*/} \\ qchannel \subseteq Request\ \texttt{/*invariant from } \textit{QChannel}\texttt{*/} \\ \ldots \\ \vdash \\ qmax\_size \in \mathbb{N} \end{array}}$$

The first axiom of *ProtocolTypes* easily discharge this proof obligation. Note the expansion of *Qchannel* is not required in practice. We use it to show the meaning of an *instantiated machine*.

4.3 Definition of Generic Instantiation of Machines

Based on the instantiated machine *QChannel*, a general definition for generic instantiation of machines can be drawn. Considering Context *Ctx* and machine *M* in Fig. 6 together as a *pattern*, we can create a generic Instantiatiated Machine *IM* as seen in Fig. 7.

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

(a)

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

(b)

Fig. 6. Generic view of a context and a machine

```
INSTANTIATED MACHINE IM
INSTANTIATES M VIA Ctx
SEES D              /* context containing the instance properties */
REPLACE             /* replace elements defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets or Constants */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$
RENAME              /*rename elements in machine M*/
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$        /* optional */
    EVENTS $ev_1 := nev_1$        /* optional */
              $p_1 := np_1, \ldots, p_s := np_s$        /*  parameters: optional */
              :
              $ev_r := nev_r$
END
```

Fig. 7. An Instantiated Machine

The context D contains the replacement properties (sets $DS_1, \ldots, DS_m$ and constants $DC_1, \ldots, DC_n$) for the elements in context *Ctx*. The variables, events and parameters are also renamed by new variables $nv_1, \ldots, nv_q$, new events $nev_1, \ldots, nev_r$ and new parameters $np_1, \ldots, np_s$. 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., a type must be replaced with a type, or a constant set and 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 ensure that the instantiated machine specifies which generic context is being instantiated.

4.4 Avoiding reproofs

As described above, a proof obligation (P.O.) is a sequent of the form $H \vdash G$ (short for $Hypothesis \vdash Goal$). Renaming variable (or constant) v to w and type (carrier set) T to S results in instantiated P.O. as following:

$[v := w]$ $(H \vdash G)$ (variable/constant instantiation)
$[T := S]$ $(H \vdash G)$ (type instantiation)

$H \vdash G$ is valid means that the proof has been proved. We assume that if $H \vdash G$ is valid then any valid instantiation of $H \vdash G$ that avoids name clashes is also valid. Instantiation of variables and constants maintains validity since a sequent is implicitly universally quantified over its free variables. We are currently exploring a formal justification for why type instantiation maintains validity. Since instantiation maintains the validity of the sequent, the P.O. generated for the pattern can be reused in the instance and we avoid having to discharge the instantiated P.O..

5 Example of Instantiation and Composition

The creation of the instances is a intermediary step in the overall model development. In our case study, we model a protocol between entities that sends and receives messages. By using the created instances and the Shared Event Composition [9, 10] plug-in for the Rodin platform we share events between Request and Response and model the protocol. A composed machine *Protocol* modelling this system can be seen in Fig. 8.

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

Fig. 8. Composed Machine Protocol

As seen in Fig. 2, while composing the instance machines *QChannel* and *PChannel* we add the events that are unique for each entity (*SendRequest* and *RecvResp*). *SendRequest* sends a message through the channel from Source to Destination. *RecvResp* models the reception of the response in the *Source* after being sent by Destination. Moreover the event that relates the communication between the two entities is also modelled (*RecvReq_SendResp*). The request is received and acknowledged and the response to that request is sent in parallel (from this combined event, a possible refinement is processing the request message before sending the response). We opt not to refine an abstract machine in Fig. 8 (*REFINES* clause is empty: "-") although it is possible. The composed machine *Protocol* corresponds to the expanded machine in Fig. 9.

The two instances of machine *Channel* model a bi-directional communication channel between two entities. This allows us to express the applicability of generic instantiation for modelling distributed systems without being 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.

```
machine Protocol sees ProtocolTypes

variables qchannel pchannel

invariants
  @inv1 qchannel ⊆ Request
  @inv2 pchannel ⊆ Response
  @inv3 card(pchannel) ≤ pmax_size
  @inv4 card(qchannel) ≤ qmax_size
  theorem @QChannel/thm1 qmax_size ∈ ℕ
  theorem @PChannel/thm2 pmax_size ∈ ℕ

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
      @act2 pchannel ≔ ∅
  end
```

(a)

```
  event SendRequest
    any q
    where
      @grd1 q ∈ Request
      @grd2 card(qchannel) < qmax_size
    then
      @act1 qchannel ≔ qchannel ∪ {q}
  end

  event RecvReq_SendResp
    any q p
    where
      @grd1 q ∈ qchannel
      @grd2 p ∈ Response
      @grd3 card(pchannel) < pmax_size
    then
      @act1 pchannel ≔ pchannel ∪ {p}
      @act2 qchannel ≔ qchannel\{q}
  end

  event RecvResp
    any p
    where
      @grd1 p ∈ pchannel
    then
      @act1 pchannel ≔ pchannel\{p}
  end
end
```

(b)

Fig. 9. *Machine Protocol*

6 Generic Instantiation applied to a chain of refinements

The above sections describe generic instantiation applied to individual machines. Although it is already an interesting way of reusing, in a large model it would be more interesting to instantiate a chain of machines, or in other words *instantiate a chain of refinements*. Suppose we have a development Dv containing several refinement levels $(Dv_1, Dv_2, \ldots, Dv_n)$. The most concrete model Dv_n matches a generic model (pattern) P_1 that is part of a chain of refinements $P_1, P_2, \ldots, P_m$ as seen in Fig. 10. By applying generic instantiation we instantiate the pattern P_1 according to Dv_n. That instantiation is a refinement of Dv_n and it is called Dv_{n+m}_abs (the suffix *abs* stands for abstract). In addition we can extend the instantiation to one of the refinement layers of the pattern and apply it to the development Dv. As an outcome we get a further refinement layer for Dv_n for free (Dv_{n+m}_abs corresponds to the instantiation of P_1 and Dv_{n+m} corresponds to the instantiation of P_m). The refinement between Dv_{n+m}_abs and Dv_{n+m} does not introduce refinement proof obligations since the proof obligations were already discharged in the pattern chain. This follows from the instantiated machines where it is avoided the re-proof of pattern proof obligations. Afterwards Dv_{n+m} can be further refined to Dv_{n+m+z}. For a better understanding of this approach, we will refine our case study and apply an instantiation over the pattern chain.

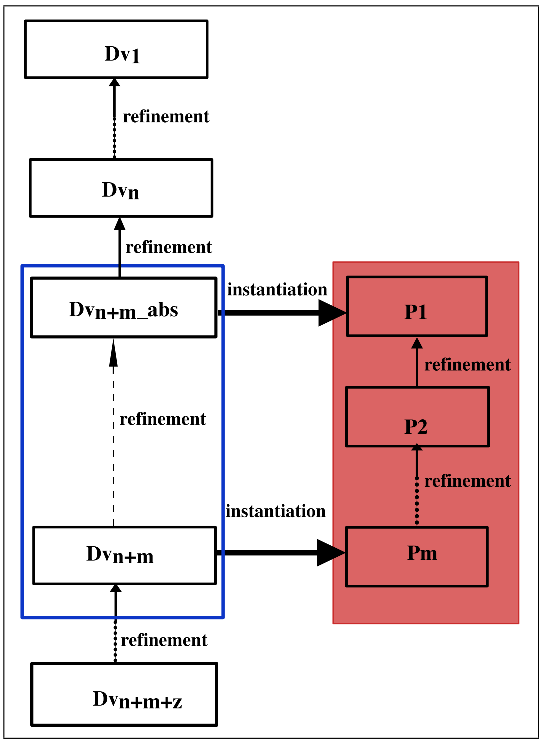

Fig. 10. Instantiation of a generic chain of refinements

6.1 Refinement of the Channel case study

We will refine the *Channel* machine. For the first refinement, the requirement is to include buffers before and after adding a message to the channel. A second refinement specifies the type *Message*. In particular, *Message* will be divided in two parts: *header* and *body*. The *header* of the *Message* contains the destination identification and the *body* represents the content of the message (data). *header* and *body* are based on the records proposal for Event-B suggested by Evans and Butler [11] and also in work developed by Rezazadeh et al [12].

The first refinement requires an introduction of two new variables *sendingBuffer* and *receivingBuffer* and a new event *addMessageBuffer* that loads the message to *sendingBuffer* before being introduced in the channel in the *Send* event. The latter event reflects the introduction of the buffers. In the event *Receive*, messages in *channel* are extracted and loaded to *receivingBuffer* as seen in Fig. 11.

The second refinement is a data refinement over the type *Message* by dividing it in *header* and *body*. The *header* contains the destination identification and the *body* contains the data of the message. Constants *header* and *body* are defined in the context *ChannelParameters_C2* as in Fig. 12.

```
machine Channel_M1 refines Channel
sees ChannelParameters

variables channel sendingBuffer
          receivingBuffer

invariants
  @inv1 sendingBuffer ⊆ Message
  @inv2 receivingBuffer ⊆ Message

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
      @act2 sendingBuffer ≔ ∅
      @act3 receivingBuffer ≔ ∅
  end

  event addMessageBuffer
    any m
    where
      @grd1 m ∈ Message
      @grd2 m ∉ sendingBuffer
    then
      @act1 sendingBuffer≔sendingBuffer∪{m}
  end
```

(a)

```
event Send refines Send
  any m
  where
    @grd1 sendingBuffer ≠ ∅
    @grd2 m ∈ sendingBuffer
    @grd3 card(channel) < max_size
  then
    @act1 channel ≔ channel ∪ {m}
    @act2 sendingBuffer≔sendingBuffer\{m}
end

event Receive refines Receive
  any m
  where
    @grd1 m ∈ channel
    @grd2 m ∉ receivingBuffer
  then
    @act1 channel ≔ channel\{m}
    @act2 receivingBuffer≔receivingBuffer∪{m}
end
```

(b)

Fig. 11. *Channel_M1*: refinement of *Channel*

```
context ChannelParameters_C2 extends ChannelParameters

constants header body

sets DATA DESTINATION

axioms
  @axm3 header ∈ Message → DESTINATION
  @axm4 body ∈ Message → DATA
end
```

Fig. 12. Context *ChannelParameters_C2*

In Fig. 13 the machine *Channel_M2* data refines the variable *channel* and introduces a new event, *processMessage* that processes the received message after being retrieved from the receiving buffer. A variable *storeDATA* is also introduced to store the data that each destination receives.

6.2 Instantiation of a chain of refinements

We can consider the chain of refinements of *Channel* as a pattern. In that case, having all the proof obligations discharged we can reuse this pattern in a more specific development. The chain of refinements is seen as a single entity where it is possible to choose an *initial* and a *final* refinement level.

Using our case study, we intend to instantiate and refine *QChannel* with the chain of refinements of machine *Channel*, selecting *Channel* and *Channel_M2* as our initial and final refinement levels respectively. In Fig. 14 the shaded chain of refinement is seen as a single entity. After the selection of the two

```
machine Channel_M2 refines Channel_M1
sees ChannelParameters_C2

variables channel sendingBuffer
          receivingBuffer storeDATA

invariants
  @inv1 storeDATA ∈ DESTINATION ⇸ ℙ(DATA)

events
  event INITIALISATION
    then
      @act1 channel ≔ ∅
      @act2 sendingBuffer ≔ ∅
      @act3 receivingBuffer ≔ ∅
      @act4 storeDATA ≔ DESTINATION × {∅}
  end

  event addMessageBuffer
  refines addMessageBuffer
    any h b m
    where
      @grd1 header(m) = h
      @grd2 body(m) = b
      @grd3 m ∉ sendingBuffer
    then
      @act4 sendingBuffer≔sendingBuffer∪{m}
  end
```

(a)

```
event send refines Send
  any m
  where
    @grd1 sendingBuffer ≠ ∅
    @grd2 m ∈ sendingBuffer
    @grd3 card(channel) < max_size
  then
    @act1 channel ≔ channel ∪ {m}
    @act2 sendingBuffer≔sendingBuffer\{m}
end

event receive refines Receive
  any m
  where
    @grd1 m ∈ channel
    @grd2 m ∉ receivingBuffer
  then
    @act1 channel ≔ channel\{m}
    @act2 receivingBuffer≔receivingBuffer∪{m}
end

event processMessage
  any m dest d
  where
    @grd1 m ∈ receivingBuffer
    @grd3 header(m) = dest
    @grd4 d = body(m)
    @grd5 dest ∈ dom(storeDATA)
  then
    @act1 storeDATA(dest)≔storeDATA(dest)∪{d}
end
```

(b)

Fig. 13. *Channel_M2*: refinement of *Channel_M1*

refinement levels to be instantiated, *QChannel_M2_abs* and *QChannel_M2* are created. *QChannel_M2* is treated as a refinement of *QChannel_M2_abs* as a consequence of the instantiation. Subsequently, *QChannel_M2* can be further refined to *QChannel_Mz*.

The refinement relationship between *Channel* and *Channel_M2* is ensured by discharging all the proof obligations in the chain of refinement (all the proofs are discharged automatically in the Rodin platform). By instantiating *Channel* and *Channel_M2* implicitly we are also referring to *Channel_M1*. Some of the properties of *Channel_M2* are inherited from *Channel_M1* (for instance the buffers) but for the instantiation purpose it is not necessary to incorporate *Channel_M1* explicitly. The instantiation of a chain of refinements follows the instantiation of a single machine as seen in Fig. 15.

The initial refinement level corresponds to the most abstract machine of the pattern. The final refinement level is any of the other refinement levels in the chain. The replacement and renaming is applied to the occurrences in both instances whenever applicable. Once again it is not necessary to "expand" *QChannel_M2* but that can be seen in Fig. 16.

In an instantiation of a chain of refinements, the pattern context is seen as a *flat context* comprising all the properties seen by the refinements until the selected final refinement level is reached. Therefore context *ProtocolTypes_C2* is the parameterisation context for *QChannel_M2* and extends *ProtocolTypes*

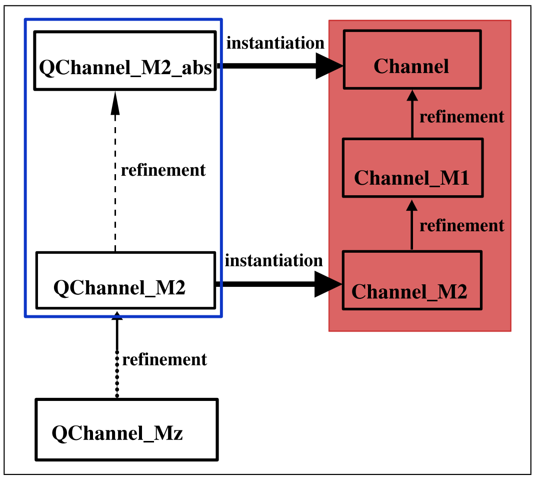

Fig. 14. Instantiation of a chain of refinements: *Channel* to *Channel_M2*

```
INSTANTIATED REFINEMENT QChannel_M2
INSTANTIATES Channel_M2 VIA ChannelParameters_C2
REFINES -
SEES ProtocolTypes_C2
REPLACE
    SETS Message := Request
    CONSTANTS max_size := qmax_size
              header := qHeader
              body := qBody
RENAME
    VARIABLES channel := qchannel
              receivingBuffer := qReceivingBuffer
              sendingBuffer := qSendingBuffer
    EVENTS Send := QSend
           m := q
           receive := Receive
           m := q
END
```

Fig. 15. Instantiation of a chain of refinements

```
machine QChannel_M2 refines QChannel_M1
sees ProtocolTypes_C2

variables qchannel qReceivingBuffer
          qSendingBuffer qStoreDATA

invariants
  @inv1 qStoreDATA ∈ DESTINATION ⇸ ℙ(DATA)
  theorem @theo1 qHeader ∈ Request → DESTINATION
  theorem @theo2 qBody ∈ Request → DATA

events
  event INITIALISATION
    then
      @act1 qchannel ≔ ∅
      @act2 qSendingBuffer ≔ ∅
      @act3 qReceivingBuffer ≔ ∅
      @act4 qStoreDATA ≔ DESTINATION × {∅}
  end

  event AddMessageBuffer
  refines qAddMessageBuffer
    any h b m
    where
      @grd1 qHeader(m) = h
      @grd2 qBody(m) = b
      @grd3 m ∉ qSendingBuffer
    then
      @act1 qSendingBuffer ≔ qSendingBuffer∪{m}
  end
```

(a)

```
event QSend refines QSend
  any q
  where
    @grd1 qSendingBuffer ≠ ∅
    @grd2 q ∈ qSendingBuffer
    @grd3 card(qchannel) < qmax_size
  then
    @act1 qchannel ≔ qchannel ∪ {q}
    @act2 qSendingBuffer≔qSendingBuffer\{q}
end

event Receive refines Receive
  any q
  where
    @grd1 q ∈ qchannel
    @grd2 q ∉ qReceivingBuffer
  then
    @act1 qchannel ≔ qchannel\{q}
    @act2 qReceivingBuffer≔qReceivingBuffer∪{q}
end

event processMessage
  any m dest d
  where
    @grd1 m ∈ qReceivingBuffer
    @grd2 qHeader(m) = dest
    @grd3 d = qBody(m)
    @grd4 qHeader(m) ∈ dom (qStoreDATA)
  then
    @act1 qStoreDATA(dest)≔qStoreDATA(dest)∪{d}
end
```

(b)

```
context ProtocolTypes_C2 extends ProtocolTypes

constants qHeader qBody pHeader pBody

sets DATA DESTINATION

axioms
  @axm3 qHeader ∈ Request → DESTINATION
  @axm4 qBody ∈ Request → DATA
  @axm5 pHeader ∈ Response → DESTINATION
  @axm6 pBody ∈ Response → DATA
end
```

(c)

Fig. 16. Expanded version of instantiated machine *QChannel_M2* and context *ProtocolTypes_C2*

similarly to the relation between contexts *ChannelParameters_C2* and *ChannelParameters*. As before, axioms in *ProtocolTypes_C2* must be respected in the instance, so axioms are converted in theorems in *QChannel_M2*.

6.3 Definition of Generic Instantiation of Refinements

From the case study it is possible to draw a generic definition for the instantiation of a chain of refinements. If we consider a pattern that consists of a chain of refinements *M1, M2, . . . Mt* , we can create a generic Instantiated Refinement *IR* as seen in Fig. 17.

The instantiated refinement *IR* instantiates one of the refinements of the pattern M_t via the parameterisation context Ctx_t. *IR* refines an abstract machine

```
INSTANTIATED REFINEMENT IR
INSTANTIATES $M_t$ VIA $Ctx_t$
REFINES $IR_0$        /* abstract machine */
SEES $D_w$            /* context containing the instance properties */
REPLACE               /* replace elements defined in context C */
    SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Carrier Sets or Constants */
    CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$
RENAME   /*rename variables, events and params in $M_1$ to $M_t$*/
    VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$
    EVENTS $ev_1 := nev_1$      /* optional */
              $p_1 := np_1, \ldots, p_s := np_s$     /* parameters :optional */
                ⋮
              $ev_r := nev_r$
END
```

Fig. 17. An Instantiated Refinement

IR_0 and sees the context D_w containing the instance properties. The replacement and renaming are similar to the machine instantiation but apply to both M_1 and M_t. In addition to the validity checks for instantiated machines, instantiated refinements require:

1. A static validation for the existence of a chain of refinements for M $(M_1, M_2, \ldots, M_t)$.
2. The types and constants in the contexts seen by the initial and final level of refinement should be instantiated.

The instantiation of refinements reuses the pattern proof obligations in the sense that the instantiation renames and replaces elements in the model but does not change the model itself (nor the respective properties). The correctness of the refinement instantiation relies in reusing the pattern proof obligations and ensuring the assumptions in the context parameterisation are satisfied in the instantiation.

7 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 (axioms and invariants) . Once the theorems are discharged, they can be used as hypotheses for discharging other proof obligations in the model since they work as a consequence of the assumptions. On the other hand, invariants in machines are properties of the model that need to be maintained by all events.

An interesting question arises when a pattern is instantiated and contains theorems and invariants. If a proof obligation of a theorem is 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 re-proving them. Although addressed here as an open question, this situation suggests a different kind of theorem that does not exist in Event-B, a *pre-proved theorem* to be used in the instance. A *pre-proved theorem* would be similar to a theorem but it would not have associated a proof obligation. The invariants imported from the pattern 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 through the generation of proof obligations but the same does not apply for invariants and theorems that are assertions in the pattern.

8 Conclusions

Reusability is of significant interest in the general software engineering research community. Advantages and disadvantages have been discussed in terms of how to reuse. Examples are given by Standish [1] and Cheng [13]. Reusing patterns in a style similar to design patterns is proposed in [14] using the KAOS specification language and temporal logic. The patterns are proved correct and complete and proofs can be reused. Sabatier [15] discusses the reuse of formal models as a detailed component specification or as a high level requirement and presents some real project examples. In classical B [16, 17], reuse is expressed using the keywords *INCLUDES* and *USES* where an existing machine can be used in other developments. Instantiation is a way of reusing. Instantiation is well-established in areas such as mathematics and other formal methods like classical B or theorem provers like Isabelle [18]. [19] reuses Gang of Four (GoF) design pattern adapted to formal specifications (denominated specification patterns) for classical B. Several reuse mechanisms are suggested like instantiation, composition and extension. Proof obligations are also reused when the patterns are applied. Focusing on the instantiation, this is achieved by renaming sets (machine parameters), variables and operations. Unlike our work, this approach only defines patterns as single abstract machine whereas we define the parameterisation in contexts and extend the pattern to a chain of refinements. Abrial and Hallerstede [3] and Métayer et al [2] make use of generic instantiation for Event-B. It is proposed the flattening of the context in a way that the contexts of the pattern are merged and it is suggested the reuse by instantiating the flat context. Following that approach, we decide to propose an implementation of generic instantiation.

The motivation for such implementation is concerned with reusability of components and existing developments. By creating an instance from a generic model, a new parameterised model is created based on the pattern with new specific properties.

Event-B supports generic developments but lacks capacity to instantiate and reuse those generic developments. As a solution, generic instantiation is applied to patterns and as an outcome instantiated machines are created and parameterised. An *instantiated machine* instantiates a generic machine, is parameterised by a context and the pattern elements are renamed/replaced according to the instance. In a similar style, an *instantiated refinement* instantiates a chain of refinements reusing the pattern proof obligations assuming that the instantiated proof obligations are as valid as the pattern ones. As future work we intend to prove this assumption. By quantifying the variables/constants and types we want to ensure that pattern proof obligations remain valid when instantiating. Event-B is not a high-order formalism: although it is possible to quantify variables and constants, it is not possible to quantify types. So we need to use a higher-order formalism to ensure that the instantiation of types maintains the validity of associated proof obligations. A practical case that models a communication protocol between two entities illustrates the advantages of using generic instantiation and in particular how to use our approach in the Rodin platform. Although a simple case study, we believe that it can be applied to more complex cases.

Further study is required to determine if context instantiation similar to instantiated machines is a worthwhile approach while modelling. Some methodological points will arise in a possible implementation of instantiated machines and refinements in the Rodin platform. As an example, Section 7 addresses the situation of instantiating theorems and invariants and is left as an open question. A future step for the instantiation of a chain of refinements is to study the possibility of selecting any of the refinement levels as the initial refinement level giving more freedom to the modeller. In a long term perspective, any refinement chain could be considered a pattern or a library of patterns should be provided when modelling: whenever a formal development fits in a pattern, instantiation could be applied taking advantage of the reusability of the model and respective proof obligations.

Acknowledgments We would like to thank Jean-Raymond Abrial and Thai Son Hoang for valuable discussions about this paper. Also Hannah Warren for helping in the review of initial versions of the paper. We also thank the anonymous reviewers for their inputs and suggestions.

References

1. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. **10**(5) (1984) 494–497

2. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language. Technical report, Deliverable 3.2, EU Project IST-511599 - RODIN (May 2005)
3. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inf. **77**(1-2) (2007) 1–28
4. Rodin: RODIN project Homepage. `http://rodin.cs.ncl.ac.uk` (September 2008)
5. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: ICFEM. (2006) 588–605
6. Silva, R., Butler, M.: Parallel Composition Using Event-B. `http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B` (July 2009)
7. Silva, R.: Renaming Framework. `http://wiki.event-b.org/index.php/Refactoring_Framework` (July 2009)
8. Abrial, J.R.: Summary of Event-B Proof Obligations. `http://www.docstoc.com/docs/7055755/Summary-of-Event-BProof-Obligations` (March 2008)
9. Butler, M.: An Approach to the Design of Distributed Systems with B AMN. In: Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212. (1997) 221–241
10. Butler, M.: Synchronisation-based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate report on methodology. (2006)
11. Evans, N., Butler, M.: A Proposal for Records in Event-B. In Nipkow, T., Misra, J., Sekerinski, E., eds.: Formal Methods 2006. Volume LNCS 4085., Springer (2006) 221–235
12. Rezazadeh, A., Evans, N., Butler, M.: Redevelopment of an Industrial Case Study Using Event-B and Rodin. In: BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry. (December 2007)
13. Cheng, J.: A Reusability-Based Software Development Environment. SIGSOFT Softw. Eng. Notes **19**(2) (1994) 57–62
14. Darimont, R., van Lamsweerde, A.: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In: SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, New York, NY, USA, ACM (1996) 179–190
15. Sabatier, D.: Reusing Formal Models. In: IFIP Congress Topical Sessions. (2004) 613–620
16. Schneider, S.: The B method: an introduction. Palgrave (2001)
17. Abrial, J.R.: The B-Book: Assigning programs to meanings. Cambridge University Press (1996)
18. Paulson, L.C.: Isabelle: a Generic Theorem Prover. Volume 828 of Lecture Notes in Computer Science. Springer – Berlin (1994)
19. Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In Springer-Verlag SEP, ed.: ZB 2003: Formal Specification and Development in Z and B Lecture Notes in Computer Science. Volume 2651 of Lecture Notes in Computer Science., Turku, Finland (June 2003) 40–57