A Business Protocol: Project in Event-B

Ravi Bhoraskar

April 25, 2011

Contents

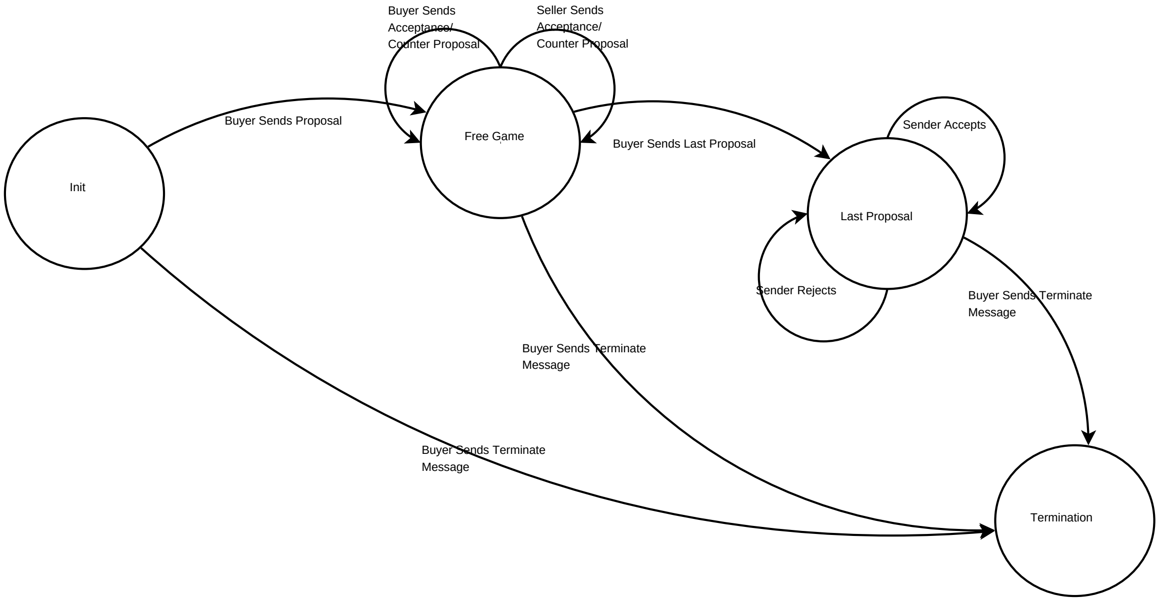
Buyer Sends Acceptance/ Counter Proposal
Seller Sends Acceptance/ Counter Proposal
Buyer Sends Proposal
Free Game
Buyer Sends Last Proposal
Sender Accepts
Init
Last Proposal
Sender Rejects
Buyer Sends Terminate Message
Buyer Sends Terminate Message
Buyer Sends Terminate Message
Termination

Figure 1: State Transition Diagram

1 Introduction

This project is based on *A Bussiness Protocol* described in [1]. Essentially, we aim at constructing the model of a business protocol. This protocol determines the negotiation taking place between a buyer and a seller of an unspecified commodity. Both the buyer and seller begin in a certain state, and through exchange of messages, they move from one state to another. Also through messages, they try and negotiate the specifics of the transaction, and the protocol may eventually end in either a success (meaning that the negotiations have been successful) or a failure (meaning that they haven't). The modeling is done using Event-B, and the model is implemented using Rodin [2]. We attempt to use *Design Patterns* in Event-B, for the ease of modeling similar patterns used multiple times. In this report, we document the details of the project, including the Specifications document, the refinement strategies, and our experiences with using Event-B and Rodin during the course of this project.

2 Problem Specification

This protocol determines the negotiation taking place between a buyer and a seller. The outcome of the protocol might be as follows:

1. The two parties agree on a final agreement by which the seller sells a certain quantity of a certain product to the buyer at a certain price. Note that the product, the quantity, and the price are all abstracted here as an INFO exchanged between the participants;

2. the two parties might end up by not succeeding in finding an agreement;

The protocol is divided into four phases: the *initial* phase, the *free game phase*, the *last proposal* phase, and the *termination* phase. (Refer Figure 1).

- In the initial phase, the buyer starts the protocol by sending a proposal to the seller, which the seller must acknowledge

- After this initial proposal has been received by the seller, and the Ack has been received by the buyer, the protocol enters the free game phase. In this second phase buyer and seller can send counter-proposal or acceptance to the other partner proposal in a fully asynchronous way. In this phase, an acceptance or a counter-proposal by either party is never definitive. We abstract the acceptance/counter proposal as just Data, which the buyer and seller may send and receive asynchronously.

- The last proposal phase is at the initiative of the buyer which makes it clear to the seller that the proposal sent to it is the last one; the seller can either accept it or reject it. It cannot send a counter-proposal. In our model, the seller does this by the Ack message itself.
- The termination phase is reached after the buyer sends a termination message, which the seller has to acknowledge. The termination phase can be broken into two states - the *success* state and the *failure* state. The *end* event is an observer, that can only be called in the termination state.

During the three first phases, the buyer can always cancel the protocol by sending a *fail* message to the seller, which needs to acknowledge it. This has the immediate effect to move the protocol to the failure phase. When in the free game state, the buyer may choose to send a *succeed* message, which causes the protocol to move to the success state.

The channels between the buyer and seller are assumed to be reliable, so that messages sent are guaranteed to be delivered to the intended recipient.

3 Requirements Document

- The protocol determines the negotiation taking place between a buyer and a seller
- The two parties may agree on a final agreement or may not succeed in finding an agreement
- The protocol is divided up into five states: the initial state, the free game state, the last proposal state, the failure state and the success state.
- The protocol can end in either the failure or the success state.
- State transitions are allowed as in the FSM showed in Figure 1.
- A transition is allowed from init, free game and last proposal state to the failure state.
- A transition is allowed from last proposal state to the success state.
- Each state transition can occur exactly once in one run of the protocol
- The buyer initiates every state transition by sending a message. The seller changes state when it receives the message, and the buyer when it receives the Ack. Thus, the buyer performs a state transition only after the seller has already done so.
- In the free game state, both buyer and seller can send messages to each other in an asynchronous manner. Each message must be acknowledged, and the sender does not send the next message till the Ack is received.
- In the last proposal state, the buyer may send either a *succeed* or *fail* message, which causes appropriate state transitions.

4 Design Patterns

The same pattern is used at several places in the project:

> One party sends a message to the other, and waits for an Ack. The other party receives the message and sends an Ack. The first party stops waiting for an Ack when it receives it.

Hence the concept of design patterns was useful in this case. The *SendAndAck One Channel* design pattern fulfills exactly this requirement. This design pattern is used several times in the Machine *m03 Introducing SendAndAck*. It is used at the following places:

- Begin Events (go to free_game state)
- Fail Events (go to the failure state)
- Succeed Events (go to the success state)
- Last Proposal Events (go to the last_prop state)
- Events where the buyer sends stuff and seller acks
- Events where the seller sends stuff and buyer acks

If proper support for design patterns is available, then any change to the underlying send and receive mechanism would involve only changing the *SendAndAck One Channel* machine, and proving the invariants for it. For example, extending this model to an unreliable channel would need refining this design pattern machine for an unreliable channel (By introducing the *Demon Events*). In case of this particular project, design pattern support would lead to something similar to *Networking Protocol Layers*. The design pattern would be the Transport Layer, which guarantess lossless and in order packet delivery. The Bussiness Protocol would then be like the Application Layer, which uses the Transport Layer without really worrying about its intricacies, and how it works on the inside. Also, it would make possible independent and parallel refinements at the transport and application layers.

However, Rodin doesn't have any support for design patterns. This leads to copying of code in copious amounts. Also, any change in the Design pattern (e.g. going from a single channel to two channels for Data and Ack) requires the same change to be copied at six places, which is quite a nuisance.

5 Refinement Strategy

The refinement strategy takes place at two levels. The first one is at the Design Pattern level, where we create a more and more concrete model of sending and receiving messages (the Transport Layer) through successive refinements. The second is at the Business Protocol level (the Application Layer), which uses these design patterns, and creates a more concrete protocol through successive refinements.

Ideally, we would like these two refinement sequences to occur independently of each other, but due to the limitations of Rodin described in Section 4, we adopt the following refinement strategy.

5.1 Refinement Strategy for the Design Pattern

1. **SendAndAck** The Abstract Model. The Data channel is not present. The state transitions are handled by using 2 channels: *valid* and *ack*. Sequence Numbers of messages are maintained.
2. **SendAndAck1 Add Data** First Refinement. We add the Data channel.
3. **SendAndAck One Channel** Ideally, this should be a refinement of the above machine. But due to problems with using the Witness feature, we created a separate machine. We do away with the *valid* channel, as the Data channel takes care of the validity. We also do away with sequence numbers, as they are not required in this problem.

5.2 Refinement Strategy for the Bussiness Protocol

1. **m00 States and Transitions** The Abstract Model. All the state transitions are the events.
2. **m01 Convergence of Buyer Events** We prove that the Buyer Events converge, that is, only a finite number of state transitions can occur for the buyer.

3. **m02 Convergence of Seller Events** We prove that the Seller Events converge, that is, only a finite number of state transitions can occur for the seller.
4. **m03 Introducing SendAndAck** We introduce the communication channel from the buyer to the seller in this refinement. Design Patterns are used.
5. **m04 Introducing Reverse SendAndAck** We introduce the communication channel from the seller to the buyer in this refinement. Design Patterns are used. This communication channel is required for the asynchronous communication in the free game state.

6 Personal comments, insights and lessons learnt

- **Event-B:** The Event-B platform is an excellent one for modelling various kinds of systems. In particular, with reference to this project, it is very nice when modeling network protocols, as things happen independently and asynchronously. Events capture this requirement particularly well. Another good thing about modeling in Event-B is that it makes you prove the validity of the invariants at every step. Hence, if the requirements are sufficiently well captured by the invariants, it is very likely that no mistakes will be commited.
- **Rodin:** The Rodin platform is an eclipse based platform for modeling within the Event-B framework. It allows us to define the models, calculates the proof obligations and either solves them automatically using external theorem provers or gives the user to manually perform the proof. [3] That said, we felt that a few features were missing from Rodin, such as Design Patterns. However, Rodin is an Open Source tool, and is extensible through plugins, hence with time all such features can be expected to be present.
- **Proof Techniques:** In our first attempt, many proofs could not be automatically performed by the Rodin provers, even when we intuitively felt that the events should maintain them. Without modifying the events (e.g. by adding new guards), we realized that there are two ways to make the proofs go through.

    1. Use the interactive proof mode to manually prove the theorems.
    2. Add new redundant invariants (those which are implied by pre-existing invariants).

    The second method above is a valid one, since we are not changing the events themselves, nor are we adding additional constrainsts to the system. We are simply giving Rodin additional information about the system (which ideally it should already know), so as to help its automatic provers do their job properly. We have extensively used this method in our project. Thus, there are a very large number of invariants that need to be proved, but most of the proofs go through with the automatic provers.
- **Personal Comments:** This is the first time we are using the Event-B framework and Rodin. Thus, in hindsight, we notice a few mistakes that a seasoned Event-B programmer would not have done. Our refinement strategy was not as smooth as the ones done in [1]. Abrial does the refinements in a way that at every refinement, only a very little additional thing is added. In our project, the first two refinements were relatively small, but the third one was a large jump. Several of the requirements were added to the model only in the third refinement. In hindsight, we feel that this refinement could have been broken up into two or more refinements. With more experience in this methodology, this skill shall also be learnt.
- **Future Work:** A few more refinements would still be needed to complete the modeling of this project. One important thing to do which is not done is to prove *Deadlock Freedom.* This can either be done by proving Deadlock Freedom in the abstract machine, and Relative Deadlock Freedom in each of the refinements; or by proving Deadlock Freedom in the final refinement. This, along with the proof of the variant, is important to show that the protocol definitely finishes.

Another thing to do is to add an unreliable channel over which the Transport Layer runs. This can be done by introducing new events, known as *Demon Events*, which tamper with the data going over a channel. For more details on how this technique is used, see the chapter on Bounded Retransmission Protocol in [1].

Another thing left to do is to add a single communication from the seller to the buyer in the Last Proposal state. Right now, we assume that the seller sends this with the Ack. However, due to this assumption, the last message sent by the seller in this state has no Ack, and hence must be sent on trust over an unreliable channel. This will be very similar to the message sent by the seller in the free game state. Also, a new state will need to be added, since exactly one message must be sent by the seller. This will also be similar to the state transitions already done. We skip this part due to lack of time.

7 The Rodin Models

This section contains the details of the models. The comments are part of the model and are included here by the LaTeXplugin. Information about the generated proof obligations is not available.

7.1 Refinements for the Design Pattern

An Event-B Specification of SendAck context
Creation Date: 11 Apr 2011 @ 08:17:08 PM

CONTEXT SendAck context

The context seen by machines which send and receive messages

SETS

`Boolean` A Boolean Set. Contains 'true' and 'false'

`DataSet` The values which the Data Channel can take

CONSTANTS

`t` True

`f` False

`fail` Message saying 'fail'

`invalid` Data channel is empty

`succeed` Message saying 'succeed'

`last_proposal` Message saying 'go to last_prop state'

`begin` Message saying 'go to free_game state'

`DataDataSet` This is the subset of DataSet that consists of all messages except control messages

AXIOMS

axm1 : $Boolean = \{t, f\}$
Definition of Boolean

axm2 : $t \neq f$
Definition of Boolean

axm3 : $fail \in DataSet$

axm10 : $succeed \in DataSet$

axm5 : $invalid \in DataSet$

axm14 : $last\_proposal \in DataSet$

axm19 : $begin \in DataSet$

axm7 : $fail \neq invalid$

axm11 : $succeed \neq invalid$

axm13 : $succeed \neq fail$

axm15 : $last\_proposal \neq fail$

axm17 : $last\_proposal \neq invalid$

axm18 : $last\_proposal \neq succeed$

axm20 : $begin \neq fail$

axm21 : $begin \neq invalid$

axm22 : $begin \neq succeed$

axm23 : $begin \neq last\_proposal$

axm9 : $DataSet \setminus \{invalid\} \neq \varnothing$

The Dataset contains some valid things

axm24 : $DataDataSet \subseteq DataSet$

Definition of DataDataSet

axm25 : $DataDataSet = DataSet \setminus \{fail, succeed, invalid, last\_proposal, begin\}$

Definition of DataDataSet

axm26 : $DataDataSet \neq \varnothing$

There exist messages other than control messages

END

An Event-B Specification of SendAndAck
Creation Date: 11 Apr 2011 @ 08:17:13 PM

MACHINE SendAndAck

Abstract model for sending a message and receiving an Ack

SEES SendAck context

VARIABLES

SenderWaitingForACK State of Sender

SendMsgNo Message Number seen by sender(not really required)

RecvMsgNo Message Number seen by receiver(not really required)

Valid Whether Data is valid or not

Ack The Ack Channel

INVARIANTS

inv7 : $Valid \in Boolean$

inv6 : $Ack \in Boolean$

inv1 : $SenderWaitingForACK \in Boolean$

inv3 : $SendMsgNo \in \mathbb{N}$

inv4 : $RecvMsgNo \in \mathbb{N}$

inv5 : $SendMsgNo = RecvMsgNo \lor SendMsgNo = RecvMsgNo + 1$

inv8 : $SenderWaitingForACK = f \Rightarrow Ack = f$

inv9 : $Valid = t \Rightarrow SenderWaitingForACK = t$

inv10 : $Ack = t \Rightarrow Valid = f$

inv12 : $Valid = t \land Ack = f \Rightarrow SendMsgNo = RecvMsgNo + 1$

inv13 : $Valid = t \land Ack = t \Rightarrow SendMsgNo = RecvMsgNo$

inv14 : $Valid = f \Rightarrow SendMsgNo = RecvMsgNo$

EVENTS

Initialisation

begin

act1 : $SenderWaitingForACK := f$

act2 : $SendMsgNo := 0$

act3 : $RecvMsgNo := 0$
act4 : $Valid := f$
act5 : $Ack := f$
end

Event $SenderSend \mathrel{\widehat{=}}$
Sender Sends Data
when
grd1 : $SenderWaitingForACK = f$
grd2 : $Valid = f$
then
act1 : $SenderWaitingForACK := t$
act2 : $Valid := t$
act3 : $SendMsgNo := SendMsgNo + 1$
end

Event $ReceiverReceive \mathrel{\widehat{=}}$
Receiver gets Data and sends ack
when
grd1 : $Valid = t$
then
act1 : $Valid := f$
act2 : $Ack := t$
act3 : $RecvMsgNo := RecvMsgNo + 1$
end

Event $SenderGetACK \mathrel{\widehat{=}}$
Receiver gets the Ack
when
grd1 : $SenderWaitingForACK = t$
grd2 : $Ack = t$
then
act1 : $Ack := f$
act2 : $SenderWaitingForACK := f$
end

END

An Event-B Specification of SendAndAck One Channel
Creation Date: 11 Apr 2011 @ 08:17:11 PM

MACHINE SendAndAck One Channel
Ideally this should refine the other SendAndAck machines. We get rid of the valid channel

SEES SendAck context

VARIABLES
SenderWaitingForACK
Data
Ack

INVARIANTS
inv11 : $Ack \in Boolean$
inv1 : $SenderWaitingForACK \in Boolean$
inv2 : $Data \in DataSet$
inv6 : $SenderWaitingForACK = f \Rightarrow Ack = f$
Ack may be true only when sender is waiting for it

inv13 : $Ack = t \Rightarrow SenderWaitingForACK = t$

Ack may not be sent unnecessarily

inv7 : $Data \neq invalid \Rightarrow SenderWaitingForACK = t$

Once Sender has received ack, it sets the data to invalid

inv12 : $Data = invalid \Rightarrow SenderWaitingForACK = f$

Sender cannot wait for an ack of invalid Data

EVENTS

Initialisation

begin

act1 : $Data := invalid$

act2 : $SenderWaitingForACK := f$

act5 : $Ack := f$

end

Event *SenderSend* $\widehat{=}$

when

grd1 : $SenderWaitingForACK = f$

grd2 : $Data = invalid$

then

act1 : $SenderWaitingForACK := t$

act2 : $Data :\in DataSet \setminus \{invalid\}$

end

Event *ReceiverReceive* $\widehat{=}$

when

grd1 : $Data \in DataSet \setminus \{invalid\}$

grd2 : $Ack = f$

then

act3 : $Ack := t$

end

Event *SenderGetAck* $\widehat{=}$

when

grd1 : $SenderWaitingForACK = t$

grd2 : $Data \neq invalid$

grd3 : $Ack = t$

then

act1 : $Data := invalid$

act2 : $SenderWaitingForACK := f$

act3 : $Ack := f$

end

END

7.2 Refinements for the Business Protocol

An Event-B Specification of Context_0
Creation Date: 11 Apr 2011 @ 08:16:56 PM

CONTEXT Context_0

Constext seen by the Bussiness Protocol

SETS

STATUS The state that the protocol can be in

CONSTANTS

init

free_game

last_prop

success

failure

AXIOMS

axm1 : $STATUS = \{init, free\_game, last\_prop, success, failure\}$

The various states which the protocol can be in

axm2 : $init \neq free\_game$

axm3 : $init \neq last\_prop$

axm4 : $init \neq success$

axm5 : $free\_game \neq last\_prop$

axm6 : $free\_game \neq success$

axm7 : $last\_prop \neq success$

axm8 : $failure \neq init$

axm9 : $failure \neq free\_game$

axm10 : $failure \neq last\_prop$

axm11 : $failure \neq success$

END

An Event-B Specification of m00 States and Transitions
Creation Date: 11 Apr 2011 @ 08:17:55 PM

MACHINE m00 States and Transitions

Abstract Model. State Transitions only

SEES Context_0

VARIABLES

buy_state Sender's state

sell_state Receiver's state

INVARIANTS

inv1 : $buy\_state \in STATUS$

inv2 : $sell\_state \in STATUS$

EVENTS

Initialisation

begin

act1 : $buy\_state := init$

act2 : $sell\_state := init$

end

Event *Done* $\widehat{=}$

This is the event that is called when the protocol finishes

Status anticipated

when

grd1 : $buy\_state \in \{success, failure\}$

grd2 : $sell\_state \in \{success, failure\}$

then

skip

end

Event *Buyer_Begin* $\widehat{=}$

Status anticipated

```
    when
        grd1 : $buy\_state = init$
    then
        act1 : $buy\_state := free\_game$
    end
Event  $Seller\_Begin \ \widehat{=}$
Status anticipated
    when
        grd1 : $sell\_state = init$
    then
        act1 : $sell\_state := free\_game$
    end
Event  $Buyer\_Fail \ \widehat{=}$
Status anticipated
    when
        grd1 : $buy\_state \in \{init, free\_game, last\_prop\}$
    then
        act1 : $buy\_state := failure$
    end
Event  $Seller\_Fail \ \widehat{=}$
Status anticipated
    when
        grd1 : $sell\_state \in \{init, free\_game, last\_prop\}$
    then
        act1 : $sell\_state := failure$
    end
Event  $Buyer\_Lastprop \ \widehat{=}$
Status anticipated
    when
        grd1 : $buy\_state = free\_game$
    then
        act1 : $buy\_state := last\_prop$
    end
Event  $Seller\_Lastprop \ \widehat{=}$
Status anticipated
    when
        grd1 : $sell\_state = free\_game$
    then
        act1 : $sell\_state := last\_prop$
    end
Event  $Buyer\_Succeed \ \widehat{=}$
Status anticipated
    when
        grd1 : $buy\_state = last\_prop$
    then
        act1 : $buy\_state := success$
    end
Event  $Seller\_Succeed \ \widehat{=}$
Status anticipated
    when
```

grd1 : $sell\_state = last\_prop$

then

act1 : $sell\_state := success$

end

END

An Event-B Specification of m01 Convergence of Buyer Events
Creation Date: 11 Apr 2011 @ 08:17:20 PM

MACHINE m01 Convergence of Buyer Events

REFINES m00 States and Transitions

SEES Context_0

VARIABLES

buy_state Sender's state

sell_state Receiver's state

buyer_variant This is a number that is the variant for the state that the machine is in. The value set by the event is determined by a topological sort on the FSM

INVARIANTS

inv1 : $buyer\_variant \in \mathbb{N}$

The variant is a natural number.

inv2 : $buy\_state = init \Rightarrow buyer\_variant = 5$

inv3 : $buy\_state = free\_game \Rightarrow buyer\_variant = 4$

inv4 : $buy\_state = last\_prop \Rightarrow buyer\_variant = 3$

inv5 : $buy\_state = success \Rightarrow buyer\_variant = 2$

inv6 : $buy\_state = failure \Rightarrow buyer\_variant = 1$

EVENTS

Initialisation

extended

begin

act1 : buy_state := init

act2 : sell_state := init

act3 : $buyer\_variant := 5$

end

Event *Done* $\widehat{=}$

This is the event that is called when the protocol finishes

extends *Done*

when

grd1 : buy_state $\in$ {success, failure}

grd2 : sell_state $\in$ {success, failure}

then

skip

end

Event *Buyer_Begin* $\widehat{=}$

Status convergent

extends *Buyer_Begin*

when

grd1 : buy_state = init

then

act1 : buy_state := free_game

act2 : $buyer\_variant := 4$

```
     end
Event   Seller_Begin ≙
Status  anticipated
extends  Seller_Begin
     when
          grd1 : sell_state = init
     then
          act1 : sell_state := free_game
     end
Event   Buyer_Fail ≙
Status  convergent
extends  Buyer_Fail
     when
          grd1 : buy_state ∈ {init, free_game, last_prop}
     then
          act1 : buy_state := failure
          act2 : buyer_variant := 1
     end
Event   Seller_Fail ≙
Status  anticipated
extends  Seller_Fail
     when
          grd1 : sell_state ∈ {init, free_game, last_prop}
     then
          act1 : sell_state := failure
     end
Event   Buyer_Lastprop ≙
Status  convergent
extends  Buyer_Lastprop
     when
          grd1 : buy_state = free_game
     then
          act1 : buy_state := last_prop
          act2 : buyer_variant := 3
     end
Event   Seller_Lastprop ≙
Status  anticipated
extends  Seller_Lastprop
     when
          grd1 : sell_state = free_game
     then
          act1 : sell_state := last_prop
     end
Event   Buyer_Succeed ≙
Status  convergent
extends  Buyer_Succeed
     when
          grd1 : buy_state = last_prop
     then
          act1 : buy_state := success
```

act2 : $buyer\_variant := 2$

end

Event *Seller_Succeed* $\widehat{=}$

Status anticipated

extends *Seller_Succeed*

when

grd1 : $\mathtt{sell\_state} = \mathtt{last\_prop}$

then

act1 : $\mathtt{sell\_state} := \mathtt{success}$

end

VARIANT

`buyer_variant`

END

An Event-B Specification of m02 Convergence of Seller Events
Creation Date: 11 Apr 2011 @ 08:17:22 PM

MACHINE m02 Convergence of Seller Events

REFINES m01 Convergence of Buyer Events

SEES Context_0

VARIABLES

`buy_state` Sender's state

`sell_state` Receiver's state

`buyer_variant` This is a number that is the variant for the state that the machine is in. The value set by the event is determined by a topological sort on the FSM

`seller_variant` This is a number that is the variant for the state that the machine is in. The value set by the event is determined by a topological sort on the FSM

INVARIANTS

inv1 : $seller\_variant \in \mathbb{N}$

The variant is a natural number.

inv2 : $sell\_state = init \Rightarrow seller\_variant = 5$

inv3 : $sell\_state = free\_game \Rightarrow seller\_variant = 4$

inv4 : $sell\_state = last\_prop \Rightarrow seller\_variant = 3$

inv5 : $sell\_state = success \Rightarrow seller\_variant = 2$

inv6 : $sell\_state = failure \Rightarrow seller\_variant = 1$

EVENTS

Initialisation

extended

begin

act1 : $\mathtt{buy\_state} := \mathtt{init}$

act2 : $\mathtt{sell\_state} := \mathtt{init}$

act3 : $\mathtt{buyer\_variant} := \mathtt{5}$

act4 : $seller\_variant := 5$

end

Event *Done* $\widehat{=}$

This is the event that is called when the protocol finishes

extends *Done*

when

grd1 : $\mathtt{buy\_state} \in \{\mathtt{success}, \mathtt{failure}\}$

```
          grd2 : sell_state ∈ {success, failure}
      then
          skip
      end
Event   Buyer_Begin ≙
extends  Buyer_Begin
      when
          grd1 : buy_state = init
      then
          act1 : buy_state := free_game
          act2 : buyer_variant := 4
      end
Event   Seller_Begin ≙
Status  convergent
extends  Seller_Begin
      when
          grd1 : sell_state = init
      then
          act1 : sell_state := free_game
          act2 : seller_variant := 4
      end
Event   Buyer_Fail ≙
extends  Buyer_Fail
      when
          grd1 : buy_state ∈ {init, free_game, last_prop}
      then
          act1 : buy_state := failure
          act2 : buyer_variant := 1
      end
Event   Seller_Fail ≙
Status  convergent
extends  Seller_Fail
      when
          grd1 : sell_state ∈ {init, free_game, last_prop}
      then
          act1 : sell_state := failure
          act2 : seller_variant := 1
      end
Event   Buyer_Lastprop ≙
extends  Buyer_Lastprop
      when
          grd1 : buy_state = free_game
      then
          act1 : buy_state := last_prop
          act2 : buyer_variant := 3
      end
Event   Seller_Lastprop ≙
Status  convergent
extends  Seller_Lastprop
      when
```

```
          grd1 : sell_state = free_game
     then
          act1 : sell_state := last_prop
          act2 : seller_variant := 3
     end
Event   Buyer_Succeed ≙
extends  Buyer_Succeed
     when
          grd1 : buy_state = last_prop
     then
          act1 : buy_state := success
          act2 : buyer_variant := 2
     end
Event   Seller_Succeed ≙
Status  convergent
extends  Seller_Succeed
     when
          grd1 : sell_state = last_prop
     then
          act1 : sell_state := success
          act2 : seller_variant := 2
     end
VARIANT
     seller_variant
END
```

An Event-B Specification of m03 Introducing SendAndAck
Creation Date: 11 Apr 2011 @ 08:17:24 PM

```
MACHINE   m03 Introducing SendAndAck
     Introducing the Channel from the Buyer to Seller
REFINES   m02 Convergence of Seller Events
SEES   Context_0, SendAck context
VARIABLES
     buy_state      Sender's state
     sell_state      Receiver's state
     buyer_variant      This is a number that is the variant for the state that the machine is in.  The
          value set by the event is determined by a topological sort on the FSM
     seller_variant
     Data      The Data Channel (visible to sender and receiver)
     Ack      The Ack Channel (visible to sender and receiver)
     BuyerWaitingForACK
INVARIANTS
     inv1 : Ack ∈ Boolean
          Invariant from the SendAndAck Design Pattern
     inv2 : Data ∈ DataSet
          Invariant from the SendAndAck Design Pattern
     inv3 : BuyerWaitingForACK ∈ Boolean
          Invariant from the SendAndAck Design Pattern
```

inv4 : $BuyerWaitingForACK = f \Rightarrow Ack = f$
Invariant from the SendAndAck Design Pattern

inv5 : $Ack = t \Rightarrow BuyerWaitingForACK = t$
Invariant from the SendAndAck Design Pattern

inv6 : $Data \neq invalid \Rightarrow BuyerWaitingForACK = t$
Invariant from the SendAndAck Design Pattern

inv7 : $Data = invalid \Rightarrow BuyerWaitingForACK = f$
Invariant from the SendAndAck Design Pattern

inv8 : $buy\_state = failure \Rightarrow sell\_state = failure$
Captures requirement. Buyer enters failure state only after seller has already done so

inv9 : $Data = fail \land Ack = t \Rightarrow sell\_state = failure$
Invariant added to enable Buyer_Fail/inv8/INV to go through

inv10 : $buy\_state = success \Rightarrow sell\_state = success$
Captures requirement. Buyer enters succeed state only after seller has already done so

inv11 : $Data = succeed \land Ack = t \Rightarrow sell\_state = success$
Invariant added to enable Buyer_Succeed/inv10/INV to go through

inv12 : $buy\_state = free\_game \land Data \neq fail \land Data \neq last\_proposal \Rightarrow sell\_state = free\_game$
Captures requirement. Buyer enters free game state when seller has already done so

inv15 : $Data = begin \land Ack = t \Rightarrow sell\_state = free\_game$
Invariant added to enable Buyer_Begin/inv12/INV to go through

inv13 : $buy\_state = last\_prop \land Data \neq fail \land Data \neq succeed \Rightarrow sell\_state = last\_prop$
Captures requirement. Buyer enters last proposal state when seller has already done so

inv14 : $Data = last\_proposal \land Ack = t \Rightarrow sell\_state = last\_prop$
Invariant added to enable Buyer_Lastprop/inv13/INV to go through

inv16 : $Data \in DataDataSet \Rightarrow (buy\_state = free\_game \land sell\_state = free\_game)$
Non-Control messages can only be sent in the free game state

EVENTS

Initialisation

extended

begin

act1 : buy_state := init
act2 : sell_state := init
act3 : buyer_variant := 5
act4 : seller_variant := 5
act5 : $Data := invalid$
act6 : $BuyerWaitingForACK := f$
act7 : $Ack := f$

end

Event *Done* $\widehat{=}$

This is the event that is called when the protocol finishes

extends *Done*

when

grd1 : buy_state $\in$ {success, failure}
grd2 : sell_state $\in$ {success, failure}

then

skip

end

Event *Buyer_Send_Begin* $\widehat{=}$

when

grd1 : $buy\_state = init$

```
          grd2 : $Data = invalid$
          grd3 : $BuyerWaitingForACK = f$
     then
          act1 : $Data := begin$
          act2 : $BuyerWaitingForACK := t$
     end
Event   $Seller\_Begin \ \widehat{=}$
extends $Seller\_Begin$
     when
          grd1 : $\mathtt{sell\_state} = \mathtt{init}$
          grd2 : $Data = begin$
          grd3 : $Ack = f$
     then
          act1 : $\mathtt{sell\_state} := \mathtt{free\_game}$
          act2 : $\mathtt{seller\_variant} := 4$
          act3 : $Ack := t$
     end
Event   $Buyer\_Begin \ \widehat{=}$
extends $Buyer\_Begin$
     when
          grd1 : $\mathtt{buy\_state} = \mathtt{init}$
          grd2 : $Data = begin$
          grd3 : $Ack = t$
          grd4 : $BuyerWaitingForACK = t$
     then
          act1 : $\mathtt{buy\_state} := \mathtt{free\_game}$
          act2 : $\mathtt{buyer\_variant} := 4$
          act3 : $Data := invalid$
          act4 : $Ack := f$
          act5 : $BuyerWaitingForACK := f$
     end
Event   $Buyer\_Send\_Fail \ \widehat{=}$
     when
          grd1 : $buy\_state \in \{init, free\_game, last\_prop\}$
          grd3 : $Data = invalid$
          grd4 : $BuyerWaitingForACK = f$
     then
          act1 : $Data := fail$
          act2 : $BuyerWaitingForACK := t$
     end
Event   $Seller\_Fail \ \widehat{=}$
extends $Seller\_Fail$
     when
          grd1 : $\mathtt{sell\_state} \in \{\mathtt{init}, \mathtt{free\_game}, \mathtt{last\_prop}\}$
          grd2 : $Data = fail$
          grd3 : $Ack = f$
     then
          act1 : $\mathtt{sell\_state} := \mathtt{failure}$
          act2 : $\mathtt{seller\_variant} := 1$
          act3 : $Ack := t$
     end
```

```
Event  Buyer_Fail ≙
extends  Buyer_Fail
     when
          grd1 : buy_state ∈ {init, free_game, last_prop}
          grd2 : Data = fail
          grd3 : Ack = t
          grd4 : BuyerWaitingForACK = t
     then
          act1 : buy_state := failure
          act2 : buyer_variant := 1
          act3 : Ack := f
          act4 : Data := invalid
          act5 : BuyerWaitingForACK := f
     end
Event  Buyer_Send_Succeed ≙
     when
          grd1 : buy_state = last_prop
          grd2 : Data = invalid
          grd3 : BuyerWaitingForACK = f
     then
          act1 : Data := succeed
          act2 : BuyerWaitingForACK := t
     end
Event  Seller_Succeed ≙
extends  Seller_Succeed
     when
          grd1 : sell_state = last_prop
          grd2 : Data = succeed
          grd3 : Ack = f
     then
          act1 : sell_state := success
          act2 : seller_variant := 2
          act3 : Ack := t
     end
Event  Buyer_Succeed ≙
extends  Buyer_Succeed
     when
          grd1 : buy_state = last_prop
          grd2 : Data = succeed
          grd3 : Ack = t
          grd4 : BuyerWaitingForACK = t
     then
          act1 : buy_state := success
          act2 : buyer_variant := 2
          act3 : Data := invalid
          act4 : Ack := f
          act5 : BuyerWaitingForACK := f
     end
Event  Buyer_Send_LastProp ≙
     when
          grd1 : buy_state = free_game
```

```
        grd2 : Data = invalid
        grd3 : BuyerWaitingForACK = f
    then
        act1 : Data := last_proposal
        act2 : BuyerWaitingForACK := t
    end
Event  Seller_Lastprop ≙
extends  Seller_Lastprop
    when
        grd1 : sell_state = free_game
        grd2 : Data = last_proposal
        grd3 : Ack = f
    then
        act1 : sell_state := last_prop
        act2 : seller_variant := 3
        act3 : Ack := t
    end
Event  Buyer_Lastprop ≙
extends  Buyer_Lastprop
    when
        grd1 : buy_state = free_game
        grd2 : Data = last_proposal
        grd3 : Ack = t
        grd4 : BuyerWaitingForACK = t
    then
        act1 : buy_state := last_prop
        act2 : buyer_variant := 3
        act3 : Data := invalid
        act4 : Ack := f
        act5 : BuyerWaitingForACK := f
    end
Event  Buyer_Send_Stuff ≙
    Buyer sends something and waits for ack, during free game state
    when
        grd1 : Data = invalid
        grd2 : buy_state = free_game
    then
        act1 : Data :∈ DataDataSet
        act2 : BuyerWaitingForACK := t
    end
Event  Seller_Get_Stuff ≙
    Seller sends the Ack for the buyer's message in the free game state
    when
        grd3 : sell_state = free_game
        grd1 : Data ∈ DataDataSet
        grd2 : Ack = f
    then
        act1 : Ack := t
    end
Event  Buyer_Get_Ack_for_Stuff ≙
    when
```

grd1 : $BuyerWaitingForACK = t$
grd2 : $buy\_state = free\_game$
grd3 : $Data \in DataDataSet$
grd4 : $Ack = t$

then

act1 : $BuyerWaitingForACK := f$
act2 : $Data := invalid$
act3 : $Ack := f$

end

END

An Event-B Specification of m04 Introducing Reverse SendAndAck
Creation Date: 11 Apr 2011 @ 08:17:27 PM

MACHINE m04 Introducing Reverse SendAndAck
Introducing the channel from the seller to the buyer

REFINES m03 Introducing SendAndAck

SEES Context_0, SendAck context

VARIABLES

buy_state Sender's state

sell_state Receiver's state

buyer_variant This is a number that is the variant for the state that the machine is in. The value set by the event is determined by a topological sort on the FSM

seller_variant

Data The Data Channel (visible to sender and receiver)

Ack The Ack Channel (visible to sender and receiver)

BuyerWaitingForACK

ReverseData The Reverse Data Channel (visible to sender and receiver)

ReverseAck The Reverse Ack Channel (visible to sender and receiver)

SellerWaitingForACK

INVARIANTS

inv1 : $Ack \in Boolean$

inv2 : $SellerWaitingForACK \in Boolean$

inv3 : $ReverseData \in DataDataSet \cup \{invalid\}$
ReverseData can't be control packets

inv4 : $SellerWaitingForACK = f \Rightarrow ReverseAck = f$

inv5 : $ReverseAck = t \Rightarrow SellerWaitingForACK = t$

inv6 : $ReverseData \neq invalid \Rightarrow SellerWaitingForACK = t$

inv7 : $ReverseData = invalid \Rightarrow SellerWaitingForACK = f$

inv8 : $ReverseData \neq invalid \Rightarrow sell\_state = free\_game$

EVENTS

Initialisation

extended

begin

act1 : buy_state := init
act2 : sell_state := init
act3 : buyer_variant := 5
act4 : seller_variant := 5
act5 : Data := invalid

```
          act6 : BuyerWaitingForACK := f
          act7 : Ack := f
          act8 : ReverseData := invalid
          act9 : SellerWaitingForACK := f
          act10 : ReverseAck := f
     end
Event   Done ≙
     This is the event that is called when the protocol finishes
extends  Done
     when
          grd1 : buy_state ∈ {success, failure}
          grd2 : sell_state ∈ {success, failure}
     then
          skip
     end
Event   Buyer_Send_Begin ≙
extends  Buyer_Send_Begin
     when
          grd1 : buy_state = init
          grd2 : Data = invalid
          grd3 : BuyerWaitingForACK = f
     then
          act1 : Data := begin
          act2 : BuyerWaitingForACK := t
     end
Event   Seller_Begin ≙
extends  Seller_Begin
     when
          grd1 : sell_state = init
          grd2 : Data = begin
          grd3 : Ack = f
     then
          act1 : sell_state := free_game
          act2 : seller_variant := 4
          act3 : Ack := t
     end
Event   Buyer_Begin ≙
extends  Buyer_Begin
     when
          grd1 : buy_state = init
          grd2 : Data = begin
          grd3 : Ack = t
          grd4 : BuyerWaitingForACK = t
     then
          act1 : buy_state := free_game
          act2 : buyer_variant := 4
          act3 : Data := invalid
          act4 : Ack := f
          act5 : BuyerWaitingForACK := f
     end
Event   Buyer_Send_Fail ≙
```

```
extends Buyer_Send_Fail
    when
        grd1 : buy_state ∈ {init, free_game, last_prop}
        grd3 : Data = invalid
        grd4 : BuyerWaitingForACK = f
    then
        act1 : Data := fail
        act2 : BuyerWaitingForACK := t
    end
Event  Seller_Fail ≙
extends Seller_Fail
    when
        grd1 : sell_state ∈ {init, free_game, last_prop}
        grd2 : Data = fail
        grd3 : Ack = f
        grd4 : SellerWaitingForACK = f
            Change the state only when the previous ack has been received
    then
        act1 : sell_state := failure
        act2 : seller_variant := 1
        act3 : Ack := t
    end
Event  Buyer_Fail ≙
extends Buyer_Fail
    when
        grd1 : buy_state ∈ {init, free_game, last_prop}
        grd2 : Data = fail
        grd3 : Ack = t
        grd4 : BuyerWaitingForACK = t
    then
        act1 : buy_state := failure
        act2 : buyer_variant := 1
        act3 : Ack := f
        act4 : Data := invalid
        act5 : BuyerWaitingForACK := f
    end
Event  Buyer_Send_Succeed ≙
extends Buyer_Send_Succeed
    when
        grd1 : buy_state = last_prop
        grd2 : Data = invalid
        grd3 : BuyerWaitingForACK = f
    then
        act1 : Data := succeed
        act2 : BuyerWaitingForACK := t
    end
Event  Seller_Succeed ≙
extends Seller_Succeed
    when
        grd1 : sell_state = last_prop
        grd2 : Data = succeed
```

```
          grd3 : Ack = f
     then
          act1 : sell_state := success
          act2 : seller_variant := 2
          act3 : Ack := t
     end
Event   Buyer_Succeed ≙
extends Buyer_Succeed
     when
          grd1 : buy_state = last_prop
          grd2 : Data = succeed
          grd3 : Ack = t
          grd4 : BuyerWaitingForACK = t
     then
          act1 : buy_state := success
          act2 : buyer_variant := 2
          act3 : Data := invalid
          act4 : Ack := f
          act5 : BuyerWaitingForACK := f
     end
Event   Buyer_Send_LastProp ≙
extends Buyer_Send_LastProp
     when
          grd1 : buy_state = free_game
          grd2 : Data = invalid
          grd3 : BuyerWaitingForACK = f
     then
          act1 : Data := last_proposal
          act2 : BuyerWaitingForACK := t
     end
Event   Seller_Lastprop ≙
extends Seller_Lastprop
     when
          grd1 : sell_state = free_game
          grd2 : Data = last_proposal
          grd3 : Ack = f
          grd4 : SellerWaitingForACK = f
              Change state only when previous ack has been received
     then
          act1 : sell_state := last_prop
          act2 : seller_variant := 3
          act3 : Ack := t
     end
Event   Buyer_Lastprop ≙
extends Buyer_Lastprop
     when
          grd1 : buy_state = free_game
          grd2 : Data = last_proposal
          grd3 : Ack = t
          grd4 : BuyerWaitingForACK = t
     then
```

```
        act1 : buy_state := last_prop
        act2 : buyer_variant := 3
        act3 : Data := invalid
        act4 : Ack := f
        act5 : BuyerWaitingForACK := f
    end
Event   Buyer_Send_Stuff ≙
    Buyer sends something and waits for ack, during free game state
extends Buyer_Send_Stuff
    when
        grd1 : Data = invalid
        grd2 : buy_state = free_game
    then
        act1 : Data :∈ DataDataSet
        act2 : BuyerWaitingForACK := t
    end
Event   Seller_Get_Stuff ≙
    Seller sends the Ack for the buyer's message in the free game state
extends Seller_Get_Stuff
    when
        grd3 : sell_state = free_game
        grd1 : Data ∈ DataDataSet
        grd2 : Ack = f
    then
        act1 : Ack := t
    end
Event   Buyer_Get_Ack_for_Stuff ≙
extends Buyer_Get_Ack_for_Stuff
    when
        grd1 : BuyerWaitingForACK = t
        grd2 : buy_state = free_game
        grd3 : Data ∈ DataDataSet
        grd4 : Ack = t
    then
        act1 : BuyerWaitingForACK := f
        act2 : Data := invalid
        act3 : Ack := f
    end
Event   Seller_Send_Stuff ≙
    when
        grd1 : SellerWaitingForACK = f
        grd2 : ReverseData = invalid
        grd3 : sell_state = free_game
    then
        act1 : SellerWaitingForACK := t
        act2 : ReverseData :∈ DataDataSet
    end
Event   Buyer_Get_Stuff ≙
    when
        grd1 : ReverseData ∈ DataDataSet
        grd2 : ReverseAck = f
```

```
            grd3 : $buy\_state = free\_game$
        then
            act1 : $ReverseAck := t$
        end
Event   $Seller\_Get\_Ack\_for\_Stuff$ $\widehat{=}$
        when
            grd1 : $SellerWaitingForACK = t$
            grd2 : $ReverseData \neq invalid$
            grd3 : $ReverseAck = t$
            grd4 : $sell\_state = free\_game$
        then
            act1 : $ReverseData := invalid$
            act2 : $ReverseAck := f$
            act3 : $SellerWaitingForACK := f$
        end
END
```

References

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

[2] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT 12*, 6 (2010), 447–466.

[3] Breitner, J. Proving vizings theorem with rodin. Tech. rep., Indian Institute of Technology, Bombay, March 2011.