Event-B Patterns for Specifying Fault-Tolerance in Multi-Agent Interaction

Elisabeth Ball and Michael Butler*

Dependable Systems and Software Engineering, Electronics and Computer Science,
University of Southampton, UK
{ejb04r, mjb}@ecs.soton.ac.uk

Abstract. Interaction in a multi-agent system is susceptible to failure. A rigorous development of a multi-agent system must include the specification of fault-tolerance in agent interactions for the agents to be able to continue to function independently. Patterns are presented for the specification in Event-B of fault tolerance in multi-agent interactions.

1 Introduction

A multi-agent system is a group of autonomous agents that interact to achieve individual or shared goals [1]. The agents interact through communicative acts in the form of messages. When the communications between agents fail the communicating agents must be able to tolerate that failure for the system to continue to function. The required fault-tolerant behaviour of the agent depends on the intended affect of the communication [2].

Formal methods are the application of mathematics to model and verify software or hardware systems [3]. Event-B is a formal method for modelling and reasoning about systems based on set theory and predicate logic. The Event-B method has been devised for modelling reactive and distributed systems [4]. Formal methods have been criticised for their lack of accessibility especially for novice users [5]. Design patterns are a way of communicating expertise by capturing the solutions to similar design problems and re-using those solutions [6].

The Foundation for Intelligent Physical Agents (FIPA) specifications offer a standardised set of communicative acts [7]. In this paper we contribute a set patterns that capture how the behaviour of those communicative acts pertaining to fault-tolerance can be specified in Event-B. The patterns capture the specification of communication events that indicate the presence of faults and the events that provide the fault-tolerant behaviour in response. The patterns can be re-used to specify this behaviour as part of a specification of a multi-agent system in Event-B. The patterns can be used for any type of multi-agent interaction independent of FIPA interaction protocol specifications.

* Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems)

2 Patterns

The purpose of a design pattern is to capture structures and decisions within a design that are common to similar modelling and analysis tasks. They can be re-applied when undertaking similar tasks to in order reduce the duplication of effort [6].

The patterns described in this paper have been used to model a case study of the contract net interaction protocol [8]. The goal of the contract net is for the initiating agent to find an agent, or group of agents, that offer the most advantageous proposal to carry out a requested task. In the protocol an initiator agent broadcasts a call for proposals to the other agents in the system. The initiator selects one or more proposals from the participating agents who then carry out the required task. The contract net protocol has been chosen because it is a distributed transaction with several points of possible failure.

An extract from the abstract machine of the contract net case study is shown in Figure 1. It models an abstraction of the contract net protocol. Each interaction is modelled as a unique conversation that begins by the `callForProposals` event adding a new conversation to the `conv` variable. The successful conversation continues with the `makeProposal` event and the conversation is related to agents that make a proposal in the `proposed` variable. The `select` event moves the conversation into the next state by taking at least one and adding it to the `selected` variable. The conversation is in its final state when it is added to the `completed` variable. This will happen when the `complete` event is triggered for the successful completion of a conversation, or in the unsuccessful cases with the `failCommit`, `failedContract` and `cancel` events. The unsuccessful events model the initiator failing to select a proposal, the accepted agents failing to carry out the task and the initiator cancelling the conversation.

The abstract machine in Figure 1 abstracts away from specifying the interaction as a series of messages being passed between agents. The abstract machine will be refined to model the way in which the individual agents communicate. Before the model is refined to include the message passing the fault-tolerance patterns will be applied during a refinement step. The fault-tolerant behaviour will then be present when the model is refined to include agent communication.

Four of the patterns are described below along with examples of their specification taken from the contract net case study. Following this the other patterns are described. All of the patterns have been modelled as part of the case study.

2.1 Timeout

Name:Timeout
Problem: An agent may become deadlocked during a conversation whilst waiting for replies. Specifying a timeout will allow the agent to continue the interaction as if it were expecting no more replies.
Solution: Add an event to the specification that will change the state of the conversation from before the timeout to after the timeout. Include events for the agent have guards for receiving replies before the timeout and after the timeout.

```
MACHINE     ContractNet
SETS        CONVERSATION; AGENT
VARIABLES   conv, proposed, selected, completed, initiator
INVARIANT   conv ⊆ CONVERSATION ∧ proposed ∈ AGENT ↔ conv ∧
            selected ⊆ proposed ∧ completed ⊆ conv ∧
            initiator ∈ conv → AGENT
EVENTS              INITIALISATION = conv, selected, completed,
                                      proposed, initiator := ∅
```

```
callForProposals =
 ANY aa, cc WHERE
  cc ∈ CONVERSATION ∧
  cc ∉ conv ∧ aa ∈ AGENT ∧
  cc ∉ completed
 THEN
  conv := conv ∪ {cc} ||
  initiator(cc) := aa
 END;
```

```
makeProposal =
 ANY aa, cc WHERE
  cc ∈ conv ∧
  aa ∈ AGENT ∧
  cc ↦ aa ∉ intiator ∧
  aa ↦ cc ∉ proposed
 THEN
  proposed :=
      proposed ∪ {aa ↦ cc}
 END;
```

```
select =
 ANY cc, as WHERE
  cc ∈ conv ∧
  as ∩ selected = ∅ ∧
  as ⊆ proposed ▷ {cc} ∧
  cc ∉ completed
 THEN
  selected := selected ∪ as
 END;
```

```
complete =
 ANY cc WHERE
  cc ∈ conv ∧
  cc ∈ ran(selected) ∧
  cc ∉ completed
 THEN
  completed := completed ∪ {cc}
 END;
```

```
failCommit =
 ANY cc WHERE
  cc ∈ conv ∧ cc ∉ selected ∧
  cc ∉ completed
 THEN
  completed := completed ∪ {cc}
 END;
```

```
cancel =
 ANY cc WHERE
  cc ∈ conv ∧
  cc ∉ completed
 THEN
  completed := completed ∪ {cc}
 END
```

```
failedContract =
 ANY cc WHERE
  cc ∈ conv ∧
  cc ∈ ran(selected) ∧
  cc ∉ completed
 THEN
  completed := completed ∪ {cc}
 END;
```

Fig. 1. Abstract Model of Part of the Contract Net Interaction Protocol

```
VARIABLES conv, cfpR, proposalG, proposalR, beforeTimeout, afterTimeout,
          rejectG, completed
INVARIANT conv ⊆ CONVERSATION ∧ completed ⊆ conv ∧
          beforeTimeout, afterTimeout ⊆ conv ∧ beforeTimeout ∩ afterTimeout = ∅
          cfpR, proposalG, proposalR, rejectG ∈ AGENT ↔ CONVERSATION ∧
          proposalG = proposed ∧ proposalR ⊆ proposalG
EVENTS
deadline =
 ANY cc WHERE
  cc ∈ beforeTimeout
 THEN
  beforeTimeout :=
     beforeTimeout \ {cc} ||
  afterTimeout :=
     afterTimeout ∪ {cc}
 END;

receiveProposal1 =
 ANY aa, cc WHERE
  cc ∈ beforeTimeout ∧
  cc ∉ selected ∩ completedConv ∧
  aa ↦ cc ∉ proposalR ∧
  aa ↦ cc ∈ proposalG ∧
 THEN
  proposalR := proposalR ∪ {aa ↦ cc}
 END;
```

```
INITIALISATION ...
failCommmit1 =
 REFINES failCommit
 ANY cc WHERE
  cc ∈ conv ∧
  cc ∈ afterTimeout ∧
  cc ∉ ran(selected) ∧
  cc ∉ completed
 THEN
  completed := completed ∪ {cc}
 END;

receiveProposal2 =
 ANY aa, cc WHERE
  cc ∈ afterTimeout ∧
  cc ∉ selected ∩ completedConv ∧
  aa ↦ cc ∉ proposalR ∧
  aa ↦ cc ∈ proposalG ∧
 THEN
  rejectG := rejectG ∪ {aa ↦ cc}
 END
```

Fig. 2. Timeout Pattern in the Contract Net

The Timeout pattern prevents an agent from becoming deadlocked whilst waiting for a reply. In the contract net case study a deadline is required for when proposals may be submitted. Any proposals received after this time will be automatically rejected. Figure 2 shows part of a refinement of the abstract model that uses the Timeout pattern. The `deadline` event changes the state of the conversation from `beforeTimeout` to `afterTimeout`. These states affect the event that can be triggered when a proposal is received.

In this refinement the order of the interaction is controlled by variables that represent each type of message either being generated or received. When a proposal has been generated by an agent a relationship between the agent and conversation is added to the `proposalG` variable. When it is received the relationship is added to the `proposalR` variable.

When a proposal has been generated and not received two events that model the receiving of a proposal can be triggered. If the state of the conversation is `beforeTimeout` then the `receiveProposal1` event can be triggered and the proposal is received. If the state of the conversation is `afterTimeout` then the `receiveProposal2` event can be triggered. The action of the second event results in a reject being generated for the proposing agent and the proposal is not received.

Including the Timeout pattern in the model can allow the deadline to pass before any agents make a proposal. In this case the initiator will not be able to select a proposal. The Refuse pattern, described below, can also lead to the initiator being unable to select a proposal. These behaviours are a refinement of the behaviour modelled in the abstract machine by the `failCommit` event. In this refinement the `failCommit` event has been refined into two events that reflect each behaviour. The `failCommit1` event in Figure 2 models initiator failing to select a proposal after the deadline has passed. Without the specification of the fault-tolerant behaviour in the abstract model it cannot be refined to include the more detailed behaviour prescribed by the patterns.

2.2 Refuse

Name: Refuse
Problem: An agent cannot support the action requested.
Solution: Add an event for an agent to send a refuse message in response to a request and an event for an agent to receive a refuse message.

```
VARIABLES cfpR, refuseG, refuseR
INVARIANT cfpR, refuseG, refuseR ∈ AGENT ↔ CONVERSATION ∧
          refuseR ⊆ refuseG
EVENTS                                INITIALISATION ...
makeRefusal =                         receiveRefusal =
 ANY aa, cc WHERE                      ANY aa, cc WHERE
  aa ↦ cc ∈ cfpR ∧                       aa ↦ cc ∈ refuseG ∧
  aa ↦ cc ∉ refuseG                      aa ↦ cc ∉ refuseR
 THEN                                  THEN
  refuseG := refuseG ∪ {aa ↦ cc}         refuseR := refuseR ∪ {aa ↦ cc}
 END;                                  END;
failCommit2 =
REFINES failCommit
 ANY cc WHERE
  cc ∈ conv ∧ cc ∈ beforeTimeout ∧
  cc ∉ completed ∧ cc ∉ ran(selected) ∧
  dom(refuseR ▷ {cc}) = AGENT - initiator(cc)
 THEN
   completed := completed ∪ {cc}
 END
```

Fig. 3. Refuse Pattern in the Contract Net

Not all agents that receive a request will be able to fulfill it. The Refuse pattern allows an agent to respond to a request that it cannot support, that is not correctly requested or that the requesting agent is not authorised to request.

In the contract net protocol an agent that receives a call for proposals can respond with a refusal. Figure 3 shows the part of the refinement that implements the Refuse pattern. After an agent receives a call for proposals the `makeRefusal` event can be triggered. This results in a relationship between the participating agent and the conversation being added to the `refuseG` variable. After a refusal has been generated the `receiveRefusal` event can be triggered. The relationship is added to the `refuseR` variable indicating that the refusal has been received.

Similarly to the Timeout pattern the Refuse pattern refines the original model of the initiator failing to commit. If all of the agents refuse to make a proposal, no selection can be made and the `failCommit2` event can be triggered.

2.3 Cancel

Name: Cancel
Problem: The requesting agent no longer requires an action to be performed.
Solution: Add an event to the specification for an agent to send a cancel message to an agent that has agreed to perform an action on its behalf. Add events for that agent to receive a cancel message. The agent will either reply with an inform if they have cancelled the action or a failure if they have not.

Once an agent has requested an action they can then request that it is cancelled. Agents that behave rationally may require that an action is no longer performed. This may be because their beliefs about the action change [9].

Figure 4 shows the part of the refinement that implements the Cancel pattern. The Cancel pattern models the behaviour that leads to the refined `cancel` event. The cancel mechanism can be introduced as a valid refinement because the `cancel` event is modelled in the abstract machine.

The `cancelConversation` event can be triggered by the initiating agent at any point in the conversation. The cancel message is broadcast to every other agent in the system. In the model this is specified by a set of relationships between the agents and the conversation being added to the `cancelG` variable. When there is a relationship between an agent and the conversation in the `cancelG` variable the `receiveCancel` event can be triggered and the relationship is added to the `cancelR` variable. When the relationship is in the `cancelR` variable two events can be triggered. The first event results in the relationship being added to the `informCancelG` variable. This case models the participant successfully cancelling the task and responding with a message to inform the initiator. The second event results with the relationship being added to the `failCancelG` variable. In this case the participant responds with a message to inform the initiator that they could not cancel the task. The different responses to the cancel message are received with the `receiveInformCancel` and `receiveFailCancel` events. The `cancel` event can be triggered when a response has been received from all of the agents in the system and the conversation is completed.

```
VARIABLES conv, completed, initiator, cancelG, cancelR,
          informCancelG, failCancelG, participantConv
INVARIANT cancelG, informCancelG, failureCancelG,
          participantConv ∈ AGENT ↔ CONVERSATION ∧
          cancelR ⊆ cancelG ∧ conv ⊆ CONVERSATION ∧
          completed ⊆ conv ∧ initiator ∈ conv → AGENT
EVENTS                                    INITIALISATION ...
```

```
cancelConversation =
 ANY aa, cc, as WHERE
  cc ∈ conv ∧
  cc ∉ completed ∧
  initiator(cc) = aa ∧
  as ∈ AGENT ↔ CONVERSATION ∧
  as = (AGENT \ {aa}) * {cc}
 THEN
  completed := completed ∪ {cc} ||
  cancelG := cancelG ∪ as
 END;
```

```
receiveCancel =
 ANY aa, cc WHERE
  aa ↦ cc ∈ cancelG ∧
  aa ↦ cc ∉ cancelR ∧
  aa ↦ cc ∈ participantConv
 THEN
  cancelR := cancelR ∪ {aa ↦ cc}
 END;
```

```
sendInformCancel =
 ANY aa, cc WHERE
  aa ↦ cc ∈ cancelR ∧
  aa ↦ cc ∈ participantConv
 THEN
  informCancelG :=
    informCancelG ∪ {aa ↦ cc} ||
  participantConv :=
     participantConv \ {aa ↦ cc}
  END;
```

```
sendFailCancel =
 ANY aa, cc WHERE
  aa ↦ cc ∈ cancelR ∧
  aa ↦ cc ∈ participantConv
 THEN
  failCancelG :=
      failCancelG ∪ {aa ↦ cc} ||
  participantConv :=
      participantConv \ {aa ↦ cc }
 END;
```

```
receiveInformCancel =
 ANY aa ,cc WHERE
  aa ↦ cc ∈ informCancelG ∧
  aa ↦ cc ∉ informCancelR
 THEN
  informCancelR :=
       informCancelR ∪ {aa ↦ cc}
 END;
```

```
receiveFailCancel =
 ANY aa, cc WHERE
  aa ↦ cc ∈ failCancelG ∧
  aa ↦ cc ∉ failCancelR
 THEN
  failCancelR :=
       failCancelR ∪ {aa ↦ cc}
 END;
```

```
cancel =
 ANY cc WHERE
  cc ∈ conversation ∧
  cc ∉ completed ∧
  informCancelR ▷ {cc} ∪
    failCancelR ▷ {cc} =
      AGENT - {initiator(cc)}
 THEN
  completed := completed ∪ {cc}
 END
```

Fig. 4. Cancel Pattern in the Contract Net

2.4 Failure

Name: Failure
Problem: An agent is prevented from carrying out an agreed action.
Solution: Add an event for an agent to send a failure message after they have committed to performing an action on behalf of another agent. Add an event for an agent to receive a failure message after a commitment has been made.

An agent that makes a commitment to perform an action may be prevented from carrying it out. The agent that requested the action should be informed of this failure.

```
VARIABLES conv, selected, completed, acceptG, informR, failureG,
          failureR, informG, participantConv, proposalG
INVARIANT conv ⊆ CONVERSATION ∧ completed ⊆ conv ∧
          acceptG, informG, informR, failureG, failureR, proposalG,
          participantConv  ∈ AGENT ↔ CONVERSATION ∧
          selected ⊆ proposalG
EVENTS                               INITIALISATION ...
taskFailure =                       failedContract =
 ANY aa, cc WHERE                    ANY cc WHERE
  aa ↦ cc ∈ acceptR ∧                 cc ∈ conv ∧
  aa ↦ cc ∉ failureG ∧                cc ∈ ran(selected) ∧
  aa ↦ cc ∉ informG                   cc ∉ completed ∧
 THEN                                 acceptG ⊳ {cc} =
  failureG :=                            failureR ⊳ {cc} ∪ informR ⊳ {cc} ∧
     failureG ∪ {aa ↦ cc} ||          failureR ⊳ {cc} ≠ ∅
  participantConv :=                 THEN
     participantConv \ {aa ↦ cc}      completed := completed ∪ {cc}
 END;                                END

receiveFailure =
 ANY aa, cc WHERE
  cc ∈ conv ∧
  aa ↦ cc ∈ acceptG ∧
  aa ↦ cc ∈ failureG ∧
  aa ↦ cc ∉ failureR
 THEN
  failureR :=
     failureR ∪ {aa ↦ cc}
 END;
```

Fig. 5. Failure Pattern in the Contract Net

In the case study there are two possible outcomes to a proposal being accepted. The action can be performed successfully and the participating agent will send the initiator an inform message or the action may be unsuccessful and the participant will send a failure message. The three events that model the result of an agent being unsuccessful in completing a task are shown in Figure 5.

The `taskFailure` event can be triggered after an agent has had its proposal accepted. A relationship between the failing agent and the conversation is added to the `failureG` variable. The state of the participant is updated to end its participation in the conversation. When the failure has been generated the `receiveFailure` event can be triggered. The `failedContract` event can be triggered when all the agents that have been accepted have informed the initiator of either the success or failure of the task, and at least one agent has failed. Introducing the failure mechanism is a valid refinement because the failure is modelled in the abstract machine.

2.5 Further Fault-Tolerance Patterns

The remaining patterns are presented below. The Not-Understood pattern specifies the behaviour of the agents when there is a fault in communication. The final pattern prevents an agent from re-performing an action should the middleware of the system deliver multiple copies of the same message.

Name:Not-Understood
Problem: An agent receives a message that it does not expect or does not recognise.
Solution: Specify an event for receiving a message with an unknown or unexpected performative. Specify the action as replying with a not-understood message. Specify events for receiving a not-understood message for each failure recovery scenario.

Name: Sending and Receiving Agent States
Problem: An agent receives a message that has already been sent.
Solution: Specify the states of the protocol that the agents will enter when sending and receiving messages. Each sending and receiving event must be guarded on the condition that the agent is in the correct state.

Figure 2 gives an example of how the Sending and Receiving Agent States pattern can be applied. It uses the `proposalG` and `proposalR` variables to specify the state of the interaction. When an agent-conversation pair is in `proposalG`, but not in `proposalR`, the events that receive proposals can be triggered.

3 Conclusion

Event-B is a method that is suited to the specification of multi-agent systems as it has been developed for modelling reactive and distributed systems. The patterns presented above allow the developer to relate fault-tolerance behaviour to the communication events of an Event-B specification of a multi-agent system.

The fault-tolerance patterns presented in this paper can be combined with patterns for specifying different aspects of multi-agent interaction. The development of refinement patterns will improve the application of the fault-tolerant patterns. Refinement patterns would describe the link between the abstract specification of the fault-tolerant behaviour and the effect of applying the patterns during refinement. The different patterns could be formed into a pattern language [10] for multi-agent interaction.

General strategies for fault-tolerance in multi-agent systems include adapting fault-tolerance techniques, such as replication [11], redundancy [12] and checkpoints [13], to multi-agent systems. Fault-tolerance of locations that support systems of mobile agent have been specified in Event-B [14]. Patterns for the specification of fault-tolerance strategies in multi-agent systems and fault-tolerance of mobile agents are possible directions for future work.

References

1. Jennings, N.R.: On agent-based software engineering. Artificial Intelligence **117** (2000) 277–296
2. Dragoni, N., Gaspari, M., Guidi, D.: An ACL for specifying fault-tolerant protocols. In Bandini, S., Manzoni, S., eds.: AI*IA: Advances in Artificial Intelligence. Volume 3673 of Lecture Notes in Computer Science., Milan, Italy, Springer (2005) 237–248
3. Storey, N.: Safety-Critical Computer Systems. Pearson Education Limited, Bath, UK (1996)
4. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In Bert, D., ed.: Second International B Conference B'98: Recent Advances in the Development and Use of the B Method, Springer (1998) 83 – 128
5. Glass, R.: Formal methods are a surrogate for a more serious software concern. IEEE Computer **29**(4) (1996) 19
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1995)
7. FIPA: Communicative act library specification. Technical report, Available From: http://www.fipa.org/specs/fipa00037/SC00037J.pdf (2003)
8. FIPA: Contract net interaction protocol specification. Technical report, Available From: http://www.fipa.org/specs/fipa00029/SC00029H.pdf (2002)
9. Ferber, J.: Multi-Agent Systems: Introduction to Distributed Artificial Intelligence. Addison Wesley (1999)
10. Noble, J.: Towards a pattern language for object oriented design. In: Technology of Object Oriented Langauges 28, Melbourne, Australia, IEEE (1998) 2 – 13
11. Fedoruk, A., Deters, R.: Improving fault-tolerance by replicating agents. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2, ACM Press New York, NY, USA (2002) 737–744
12. Kumar, S., Cohen, P.: Towards a fault-tolerant multi-agent system architecture. In: Proceedings of the Fourth International Conference on Autonomous Agents, ACM Press New York, NY, USA (2000) 459–466
13. Wang, L., Hon, F.L., Goswami, D., Wei, Z.: A fault-tolerant multi-agent development framework. In Cao, J., Yang, L., Guo, M., Lau, F., eds.: Parallel and Distributed Processing and Applications. Volume 3358 of Lecture Notes in Computer Science., Hong Kong, China, Springer (2004) 126 – 135
14. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E., eds.: Rigorous Development of Complex Fault-Tolerant Systems. Volume 4157 of Lecture Notes in Computer Science. Springer, Berlin (2006) 241 – 260