# 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 treatment of fault-tolerance of agent interactions for the agents to be able to continue to function independently. Patterns can be used to capture fault-tolerance techniques. A set of modelling patterns is presented that specify fault-tolerance in Event-B specifications of multi-agent interactions. The purpose of these patterns is to capture common modelling structures for distributed agent interaction in a form that is re-usable on other related developments. The patterns have been applied to a case study of the contract net interaction protocol.

## 1 Introduction

A fault-tolerant system is one that can continue to function as it was designed in the presence of faults [1]. Fault-tolerance can be introduced into the design of a software system.

Multi-agent systems are systems of distributed software entities that cooperate or compete to achieve individual or shared goals [2]. Agents encapsulate their behaviour and are motivated by their internal goals. The agents can individually respond, pro-actively and reactively, to changes in their environment [3]. The agent metaphor is one approach to creating software systems that are capable of solving distributed problems.

Formal methods are the application of mathematics to model and verify software or hardware systems [4]. Event-B is a mathematical approach for developing formal models of distributed systems that can be used to analyse and reason about the system [5]. Using a formal method to model a system results in a specification of the system that is unambiguous and can be formally verified. The model can be analysed for flaws before the system based on the model is developed [6].

Patterns are intended to make software engineering easier by capturing the expertise of experienced software developers and making it available in a manner

* This research was carried out as part of the EU research projects IST 511599 RODIN (Rigorous open development environment for complex systems rodin.cs.ncl.ac.uk) and ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity www.deploy-project.eu/).

that can be re-applied in other developments [7]. The purpose of a 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.

This paper presents a set of fault-tolerance patterns that have been developed to help specify fault-tolerance in Event-B models of multi-agent systems. A case study based on a specification of the contract net interaction protocol by the Foundation for Intelligent Physical Agents (FIPA) [8] illustrates the application of the patterns.

This paper is structured as follows: Section 2 examines the aspects of multi-agent systems that require fault-tolerance. Section 3 provides an overview of Event-B. Section 4 examines how fault-tolerance patterns can be used in Event-B. Section 5 introduces the contract net case study. The following sections describe each of the patterns in turn. Related work is then examined followed by a conclusion with an outline of possible future work.

## 2 Fault-Tolerance in Agent Interaction

A fault-tolerant system is one that can continue to function as it was designed in the presence of faults [1]. A multi-agent system has to be able to cope with the faults that can occur in any distributed system.

Fault-tolerance in distributed systems requires that the system can cope with faults in communication and faults in the behaviour of the distributed components. The system must be able to continue to function if a fault leads to a failure in communication between nodes or to a node ceasing to communicate. The system must also be able to cope if a node in the system is prevented from completing a task that it has been delegated.

In this paper we understand a multi-agent system [2] as a grouping of agents that either cooperate or compete in order to fulfill individual or collective goals. Multi-agent systems require many dynamic interactions to be able to function. The agents in the system behave both rationally and autonomously. A fault-tolerant multi-agent system needs to be able to cope with this behaviour. The agent's autonomy can make their behaviour difficult to predict. Rational agents will stop pursuing a goal if they believe that the goal has already been achieved or that it cannot be achieved. An agent that is autonomous is not required to complete any tasks requested by other agents. The task may conflict with its existing goals and, therefore, not be desirable for the agent to complete. The heterogeneity and dynamic interactions of a multi-agent system may lead to agents receiving messages that they do not understand or that are out of expected order. These are not always faults in the individual agents, but they are faults in the interactions of the system. The agents must be able to handle such faults in their interactions and communicate their reactions to these faults.

Development using formal methods can help to ensure correctness by construction [9]. Using formal methods does not guarantee that the developer has not ommitted some aspects of system behaviour from the model that may lead to failure. The patterns presented in this paper add events and variables to Event-B

specifications of multi-agent systems to provide tolerance of possible faults that can occur because of the distributed and rational nature of multi-agent systems. The faults dealt with by the patterns are an excessive delay in response, a refusal in response to a request, the request to cancel a previous request, the failure to complete a committed task and the receipt of an unexpected communication.

## 3 Event-B

Event-B is a mathematical approach for developing formal models of systems [10]. An Event-B model is constructed from a collection of modelling elements. These elements include invariants and events with guards and actions. The modelling elements have attributes expressed using set theory and predicate logic. The development of an Event-B model begins with abstraction and continues with refinement of the abstraction. The abstract machine specifies the initial requirements of the system. The refinement of a model is the process of adding more detail to a model. The refinement of an Event-B abstract machine can be carried out in several steps. More detail is added to the model at each step. Refinement allows models at different abstraction levels to be related. Development is generally, but not exclusively, top-down. Refinement may highlight errors or elements missing from the model that require changes to be made to abstract models.

The focus on atomic events in Event-B creates a representation of a reactive system [11]. The guard of an event represents the necessary conditions on the state of the system for the event to be triggered. When the guard is true the actions of the event may be executed, possibly changing the state and allowing another event to be triggered.

Event-B is designed for modelling distributed systems [5]. Event-B allows new events to be added and single events to be refined into multiple concrete events. This allows a system behaviour to be modelled as a single atomic event and then refined to a set of events that separately model the behaviour. This refinement can model individual processes executing in parallel to perform the behaviour of an abstract event or different events that result in the same actions as the abstract event. Refinement ensures that refined models are consistent with the abstract machine. Creating models of reactive and distributed systems makes Event-B an appropriate formalism as a basis for modelling multi-agent systems.

To create a textual representation of the Event-B models in this paper the events will be presented using the keywords `ANY`, `WHERE`, `THEN` and `END` to structure the model. The event variables of an event will be written between `ANY` and `WHERE`. The guards of the event will be written between `WHERE` and `THEN` and the actions of the event will be written between `THEN` and `END`.

## 4 Modelling Patterns for Fault-Tolerance

Fault-tolerance is not necessarily a feature of a system that is appropriate to model in detail at the most abstract level. It is often a part of the communication infrastructure or a component of individual nodes and, therefore, will be modelled in refinement.

Each pattern includes a description, interaction diagram and Event-B extracts from the contract net case study. The description for each of the patterns includes a *name*, *fault* statement and *tolerance pattern* statement. The description can be applied to any event-based specification. The fault statement is the potential fault for which the application of the pattern will model a solution. The tolerance pattern statement describes the steps that can be taken to solve the problem in an event-based specification.

The interaction diagrams show how the fault-tolerance techniques can be included in the interactions between the different agent roles. Several of the interaction diagrams show the variations required for one-to-many interaction.

The patterns also include Event-B extracts from a case sutdy that how how the patterns can be applied to a development. These examples make the patterns specific to Event-B development. The other elements of the pattern are more generic and could be suitable for other event-based formal methods.

The fault-tolerance techniques modelled by the patterns will help an agent to continue to provide a service when a fault occurs within a particular interaction. If a tolerated fault occurs in a conversation between two agents an agent may fail to fulfill its goal, but the fault should not prevent the agent from performing its role in another conversation.

The set of fault-tolerance patterns presented in this paper model solutions for faults that can arise in multi-agent systems. This includes faults that are found in ordinary distributed systems. The Timeout pattern prevents an agent from indefinitely waiting for a communication. This allows the agent to cope with faults in either the communication medium, or other nodes or agents in the system. The failure of a node to complete a delegated task is modelled by the Failure pattern. A rational agent altering its goals is modelled by the Cancel pattern. The Refuse pattern allows the system to cope with an agent deciding not to participate in an interaction. The Not-Understood pattern models the reaction of agents to unexpected communications. With the patterns specified in an Event-B development the developer can then refine the models to include more detail on how the system or individual agents will manage these faults.

### Applying a Pattern

The patterns can be applied to an existing Event-B development of a multi-agent system to introduce the fault-tolerance techniques to the model. Figure 1 shows how the patterns can be applied to the refinement chain of an existing Event-B development i.e., an abstract model and its refinement. The events and variables that model the abstraction of the pattern can be added to the abstract machine. The events and variables that model the concrete pattern can then be added to a refinement of the abstract machine. The developer can decide where in the refinement chain they want to extend a refinement model to include the concrete pattern. Several refinement steps may be required for refinements between the abstract machine and the model extended by the concrete pattern. The extend relationship requires the addition to, or modification of, the events and variables in the model for the pattern to be included. If the events and variables required for the pattern already exist in the model no additions or

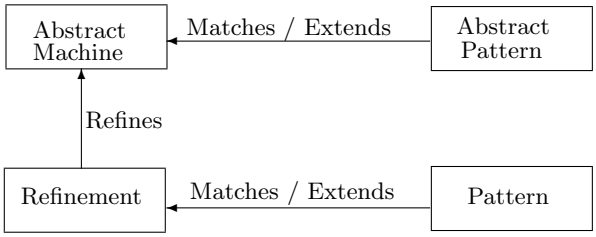

Fig. 1: Using the patterns with an existing model

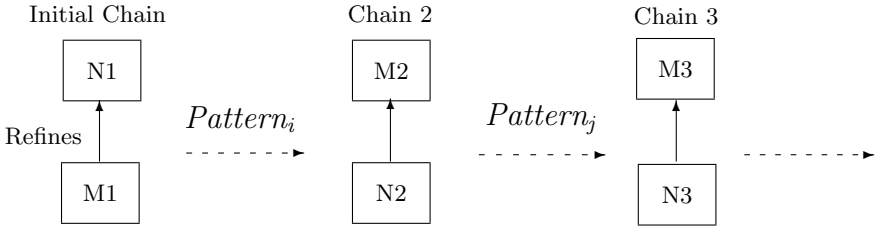

Fig. 2: Effect of applying patterns

modifications are necessary. The Event-B examples include gluing invariants that relate the abstract variables to the concrete variables. These can potentially be re-used in the extended refinement chain to help the developer specify the refines relationship.

The Event-B extracts include an abstraction of the pattern and a concrete pattern. Pattern extracts are demonstrated on the basis of the contract net case study. The abstraction of the pattern will need to be added to the Event-B abstract machine. When there are several refinement steps between the abstract machine and the refinement extended by the concrete pattern it will be necessary to make refinement steps to the intermediate models.

Not including the concept of refinement in an Event-B modelling pattern would limit the usefulness of the pattern for providing a complete solution. Including a complete refinement chain may confuse the developer should their refinement chain differ from the one provided. Not providing an abstraction may make it difficult for the developer to find an appropriate abstraction. This approach would only provide an incomplete solution and may lead to the incorrect application of the pattern.

The patterns have each been applied in separate developments to the initial chain. This ensures that there is no dependency between the patterns and, therefore, the order in which they are applied has no importance. All of the patterns have also been applied sequentially to the initial chain. This is to provide assurance that there are no conflicts between the patterns. Figure 2 illustrates how a collection of patterns can be used to extend an Event-B refinement chain. Extending the *Initial Chain* by applying $Pattern_i$ produces $Chain2$ and applying $Pattern_j$ by further extending $Chain2$ produces $Chain3$. $Pattern_j$ could be applied before $Pattern_i$ to produce the same result ($Chain3$). A possible direction for future work is to find a method to prove the orthogonality of the patterns.

## 5 Case Study

This section describes the contract net interaction protocol. A simplified version of the protocol has been modelled as an Event-B refinement chain. Each of the

patterns have been applied to the case study models. The contract net case study involves multiple participants. The developer may need to adapt the patterns for models of one-to-one interaction.

The contract net interaction protocol is a distributed negotiation process [12]. 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 required task. The initiator of the protocol advertises the existence of a task that it needs completing by broadcasting a *call for proposals*. The agents that receive the *call for proposals* can place a bid to complete the task by sending a *proposal*. Participants in the protocol are committed to the bids that they propose. When the initiator selects a bid or a group of bids the participants are informed of the decision and those selected will complete the task. The contract is completed when the participants *inform* the initiator that the task is completed. The case study has been developed using the FIPA specification of the contract net [8].

The development presented here includes an abstract model and one refinement model. The abstract model models conversations between agents and the refinement introduces the agents involved in the conversation to the model. There will be one initiator agent and one or more other agents participating in the conversation. The events of the abstract model show the behaviour of the system moving the conversation to different states to model the progression of the interaction between the agents. The refinement model shows the agents in the system and links the agents to the conversations in which they are involved and moves this relationship between the different states. Initially only successful conversations of the contract net interaction protocol are modelled. The abstract model shown in Figure 3 includes four variables that represent states that the conversation will move through. The set CONVERSATION is a set of abstract values that represent the type for conversations. The *cfp* variable represents the state after a call for proposals has been initiated by an agent. The *responded* variable represents the participating agents responding to the call for proposals. The *selected* variable represents the initiator choosing one or more proposals to accept. The *informed* variable models the state where the selected agents have informed the initiator of the successful completion of the task. The variables are not modelled as disjoint sets. Instead, the order of the conversation is enforced by specifying the variable for each state as a subset of the previous state.

The events of the abstract machine move the conversation through the different states as the conversation progresses. The *callForProposals* event adds a conversation to the *cfp* state. The *respond* event takes a conversation that is in the *cfp* state and puts it in the *responded* state. The *responded* event occurs once and represents sufficient agents sending proposals. The *select* event takes a conversation that is in the *responded* state and adds it the the *selected* state. The *inform* event takes a conversation that is in the *selected* state and adds it to the *informed* state to complete the conversation.

The refinement of the abstract model incorporates the interaction between the agents involved in the conversation. The invariants for the refinement model are shown in Figure 4. The variables of the model represent messages being sent

```
INVARIANTS
  cfp ⊆ CONVERSATION
  responded ⊆ cfp
  selected ⊆ responded
  informed ⊆ selected
EVENTS
  callForProposals
      ANY c WHERE
        c ∈ CONVERSATION
        c ∉ cfp
      THEN
        cfp := cfp ∪ {c}
      END
  select
      ANY c WHERE
        c ∈ responded
        c ∉ selected
      THEN
        selected := selected ∪ {c}
      END
  respond
      ANY c WHERE
        c ∈ cfp
        c ∉ responded
      THEN
        responded := responded ∪ {c}
      END
  inform
      ANY c WHERE
        c ∈ selected
        c ∉ informed
      THEN
        informed := informed ∪ {c}
      END
```

Fig. 3: Abstract machine of the initial chain

$$cfpS,\ proposeS,\ acceptS,\ rejectS,\ informS \in CONVERSATION \leftrightarrow AGENT,$$

$$cfpR \subseteq cfpS,\ proposeR \subseteq proposeS,\ acceptR \subseteq acceptS,$$
$$acceptS \subseteq proposeR,\ informR \subseteq informS,\ rejectR \subseteq rejectS,$$
$$informS \subseteq acceptR,\ proposeS \subseteq cfpR,$$

$$selected = dom(acceptS \cup rejectS),\ cfp = dom(cfpS)$$

Fig. 4: Invariants for the refinement of the initial chain

and received by the agents in the system. The variables that represent a message being sent are suffixed with an 'S' and those that represent a message being received are suffixed with an 'R'. The conversation is between multiple agents and so the variables are specified as relationships between a set of conversations and a set of agents. For example, $c \mapsto a \in cfpS$ means that agent $a$ has been sent a call for proposals message within conversation $c$ and $c \mapsto a \in cfpR$ means that agent $a$ has received a call for proposals message within conversation $c$. A message must be sent before it can be received and this is modelled by specifying a subset relationship between the sent variables and the received variables, e.g. $cfpR \subseteq cfpS$. Some of the variables from the abstract machine are replaced by the message variables in the refinement. The last two invariants are the gluing invariant and specify the refinement relationships between the abstract variables that represent the state of the conversation and the concrete variables that model messages being broadcast. The *responded* and *informed* variables from the abstract model represent states that are internal to the agents. Because they are not included in a conversation they are not refined by relationships between the conversation and agent and no gluing invariant is required.

The events of the refinement are shown in Figure 5. The *sendCfp* event refines the abstract *callForProposals* event. It models the broadcast of a call for proposals message from agent $a$ to all other agents ($AGENT \setminus \{a\}$) by a set of

relationships, *as*, between a conversation and the agents in the system and adds it to the *cfpS* variable. The *receiveCfp* event models a message being received by an agent by selecting a relationship, $c \mapsto a$, that is in the *cfpS* variable and adding it to the *cfpR* variable. The *sendProposal* event can occur when there is a relationship in the *cfpR* variable and the proposal is sent when the relationship is added to the *proposeS* variable. The *receiveProposal* event adds a relationship that is in the *proposeS* variable to the *proposeR* variable. The *responded* event is a refinement of the abstract *respond* event and represents the initiator receiving the required responses. The *select* event broadcasts two different messages. One group of agents, *as*, will receive an accept message in response to their proposal and another group of agents, *ar*, will receive a reject message. The *receiveAccept* and *receiveReject* events represent those messages being received by the participants. The event and variables that model the rejection, receiveReject, rejectS and rejectR can be ommitted, as they do not affect the rest of the interaction, but in a multi-agent system it may be useful for an agent to know that it has been rejected, so it can adapt its behaviour in the future. The *sendInform* event models an agent that has received an accept message, sending an inform message following the successful completion of their task. The *receiveInform* event represents this message being received. The final *informed* event refines the abstract *inform* event and models the initiator concluding that the contract has been successfully completed following the receipt of at least one inform message.

## 6 Timeout Pattern

**Name**:Timeout
**Fault**: An agent may become blocked during a conversation whilst waiting for replies.
**Tolerance Pattern**: Specify a state for the conversation that models a deadline passing. Add an event to the specification that will change the state of the conversation from before the deadline to after the deadline. Split the event for receiving the replies into two. One event will have a guard that is true before the deadline and one will have a guard that is true after the deadline. The action of the event after the deadline will inform agents of their failure to meet the deadline.

In the case of a communication failure, or the failure of another agent or node in the system, an agent that continues to wait for a response to a communication may wait an excessively long time or may never receive the reply. This is not practical for most systems, especially a multi-agent system that may be expected to be able to adapt under such circumstances. An agent should be able to decide to either continue the conversation without waiting for a response or to resolve its goal in another way, when it becomes likely that a response will not be forthcoming. An agent may be required to make a decision on how long it should wait depending on its goals for the efficiency of its current task.

The Timeout pattern prevents an agent from becoming blocked whilst waiting for a reply. It does this by modelling a deadline after which the behaviour of the system changes. The interaction diagrams in Figure 6 show the messages that are exchanged between the roles involved in the conversation.

**sendCfp** REFINES callForProposals
ANY c, as, a WHERE
$c \in CONVERSATION$
$c \notin dom(cfpS)$
$as \in CONVERSATION \leftrightarrow AGENT$
$a \in AGENT$
$dom(as) = \{c\}$
$ran(as) = AGENT \setminus \{a\}$
THEN
$cfpS := cfpS \cup as$
END

**sendProposal**
ANY c, a WHERE
$c \mapsto a \in cfpR$
$c \mapsto a \notin proposeS$
THEN
$proposeS := proposeS \cup \{c \mapsto a\}$
END

**responded** REFINES respond
ANY c WHERE
$c \in dom(proposeS)$
$c \notin responded$
THEN
$responded := responded \cup \{c\}$
END

**receiveAccept**
ANY c, a WHERE
$c \mapsto a \in acceptS$
$c \mapsto a \notin acceptR$
THEN
$acceptR := acceptR \cup \{c \mapsto a\}$
END

**sendInform**
ANY c, a WHERE
$c \mapsto a \in acceptR$
$c \mapsto a \notin informS$
THEN
$informS := informS \cup \{c \mapsto a\}$
END

**informed** REFINES inform
ANY c WHERE
$c \in dom(informR)$
$c \notin informed$
THEN
$informed := informed \cup \{c\}$
END

**receiveCfp**
ANY c, a WHERE
$c \mapsto a \in cfpS$
$c \mapsto a \notin cfpR$
THEN
$cfpR := cfpR \cup \{c \mapsto a\}$
END

**receiveProposal**
ANY c, a WHERE
$c \mapsto a \in proposeS$
$c \mapsto a \notin proposeR$
THEN
$proposeR := proposeR \cup \{c \mapsto a\}$
END

**select** REFINES select
ANY c, as, ar WHERE
$c \in dom(proposeR)$
$c \notin dom(acceptS)$
$c \notin dom(rejectS)$
$as \subseteq \{c\} \lhd proposeR$
$ar = \{c\} \lhd proposeR \setminus as$
$c \in responded$
THEN
$acceptS := acceptS \cup as$
$rejectS := rejectS \cup ar$
END

**receiveReject**
ANY c, a WHERE
$c \mapsto a \in rejectS$
$c \mapsto a \notin rejectR$
THEN
$rejectR := rejectR \cup \{c \mapsto a\}$
END

**receiveInform**
ANY c , a WHERE
$c \mapsto a \in informS$
$c \mapsto a \notin informR$
THEN
$informR := informR \cup \{c \mapsto a\}$
END

Fig. 5: Events of the refinement of the initial chain

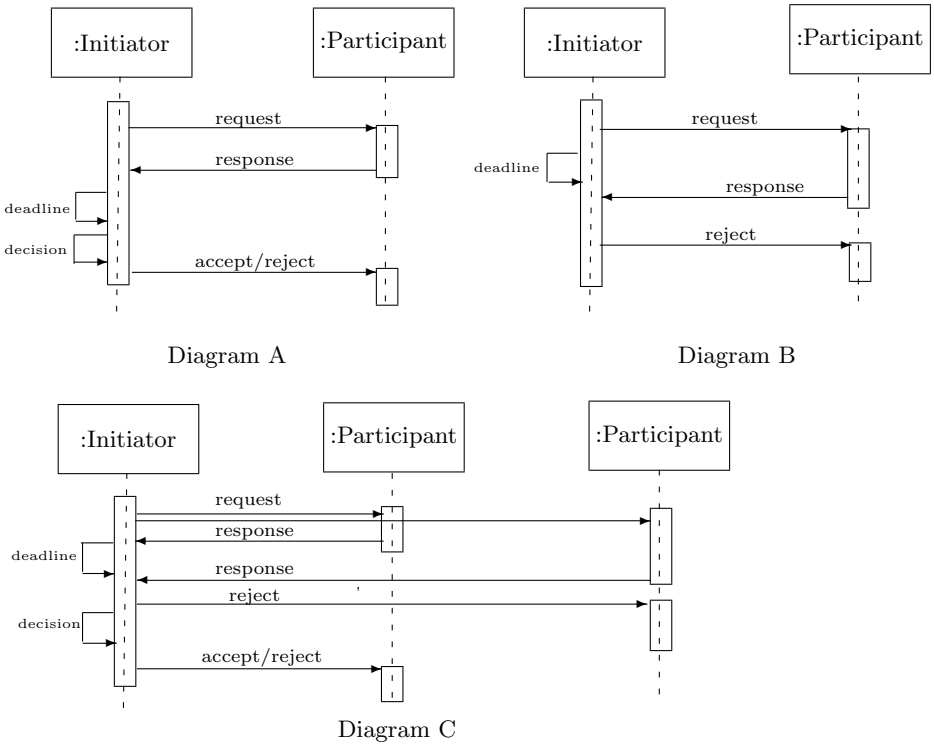


Fig. 6: Timeout: Interaction diagrams

The Timeout pattern requires that any messages received after the deadline will lead to the responding agent being informed of their failure to meet the deadline. The agent that has the role of initiating the request will be responsible for enforcing the deadline. The assumption is made in the model that the deadline will be specified by the initiator in the messages sent and that a global clock is available to all of the agents involved. Diagram A shows a successful one-to-one interaction with the response to a request being received by the initiator before the deadline. In this case the reply from the initiator will depend on the initiator's decision about the response. Diagram B shows the initiator's deadline occurring before the response is received and in this case the reply from the initiator is a rejection of the response. Diagram C shows how a one-to-many interaction can affect the Timeout pattern. Responses are received from different participating agents before and after the deadline has passed. Those received before the deadline will elicit replies that depend on a decision that is made by the initiating agent. Those received after will result in a reject notification.

Figure 7 shows the *callForProposals*, *respond* and *failure* events that are required in the abstract model for the Timeout pattern to be applied. The *callForProposals* and *respond* events are already present in the initial chain. The *failure* event has been added to model the system responding when no proposals are received before the deadline. The pattern for the timeout could be more general than that taken from the contract net case study. Any request by an agent that waits for a response could use the Timeout pattern to ensure that the requesting agent does not wait indefinitely. The abstract pattern in Figure 7 conforms to this general request-response pattern. The invariant conditions for the refinement are shown in Figure 8. To create the states for before and after

```
INVARIANTS
     failed ⊆ cfp
EVENTS
  callForProposals
       ANY c WHERE
          c ∈ CONVERSATION
          c ∉ cfp
       THEN
          cfp := cfp ∪ {c}
       END
  failure
       ANY c WHERE
          c ∈ cfp
          c ∉ failed
          c ∉ responded
       THEN
          failed := failed ∪ {c}
       END
  respond
       ANY c WHERE
          c ∈ cfp
          c ∉ responded
       THEN
          responded := responded ∪ {c}
       END
```

Fig. 7: Timeout: Abstract events

$$
\begin{aligned}
&beforeTimeout \subseteq dom(cfpS)\\
&afterTimeout \subseteq beforeTimeout\\
&proposeRD \subseteq proposeS\\
&rejectSD \subseteq proposeRD\\
&rejectRD \subseteq rejectSD\\
&failedCfp \subseteq afterTimeout\\
&failedCfp \cap dom(proposeR) = \varnothing\\
&failed = failedCfp
\end{aligned}
$$

Fig. 8: Timeout: Refinement invariants

the deadline two variables have been added to the model; *beforeTimeout* and *afterTimeout*. The pattern could have been specified with just the *afterTimeout* variable. Both variables were included to make the effect of the deadline clear in the model. The *beforeTimeout* variable is specified as a subset of the domain of the *cfpS* variable so the timeout cannot occur before the conversation has begun. Variables have been added to the model to represent the proposals that are received after the deadline, *proposeRD*, the reject messages sent in response to these proposals, *rejectSD*, and then received, *rejectRD*. The *failedCfp* variable refines the abstract *failed* variable to model the state when the deadline has passed, $failedCfp \subseteq afterTimeout$, and no proposals have been received, $failedCfp \cap dom(proposeR) = \varnothing$.

Events have been added to the initial chain and existing events have been modified to apply the Timeout pattern. The new and modified events are shown in Figure 9 where the names of the new events, and the modifications to existing events, are underlined. The *sendCfp* event has an additional action that adds the conversation to the *beforeTimeout* variable. The guard of the *receiveProposal* event has been strengthened so that it can only occur when the conversation is not in the *afterTimeout* variable. The new *deadline* event moves the conversation from the state *beforeTimeout* into the state *afterTimeout*. The new *receiveProposal2* event can only occur when the conversation is in the *afterTimeout* variable. The action of the event adds the relationship from the *proposeS* variable to the new *proposeRD* variable. The new *sendReject* event will take a relationship that is in the *proposeRD* variable and add it to the *rejectSD* variable. This

```
sendCfp  REFINES callForProposals
  ANY  c, as, a   WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfpS)
    as ∈ CONVERSATION ↔ AGENT
    a ∈ AGENT
    dom(as) = {c}
    ran(as) = AGENT \ {a}
  THEN
    cfpS := cfpS ∪ as
    beforeTimeout := beforeTimeout ∪ {c}
  END
deadline
  ANY  c    WHERE
    c ∈ beforeTimeout
    c ∉ afterTimeout
  THEN
    afterTimeout := afterTimeout ∪ {c}
  END
sendReject
  ANY  c , a    WHERE
    c ↦ a ∈ proposeRD
    c ↦ a ∉ rejectSD
  THEN
    rejectSD := rejectSD ∪ {c ↦ a}
  END

failToPropose  REFINES failure
  ANY  c  WHERE
    c ∉ dom(proposeR)
    c ∈ afterTimeout
    c ∉ failedCfp
  THEN
    failedCfp := failedCfp ∪ {c}
  END
```

```
receiveProposal
  ANY  c, a   WHERE
      c ↦ a ∈ proposeS
      c ↦ a ∉ proposeR
      c ∉ afterTimeout
  THEN
      proposeR := proposeR
          ∪ {c ↦ a}
  END
receiveProposal2
  ANY  c, a    WHERE
      c ↦ a ∈ proposeS
      c ↦ a ∉ proposeR
      c ↦ a ∉ proposeRD
      c ∈ afterTimeout
  THEN
      proposeRD := proposeRD
          ∪ {c ↦ a}
  END
receiveReject2
  ANY  c, a    WHERE
      c ↦ a ∈ rejectSD
      c ↦ a ∉ rejectRD
  THEN
      rejectRD := rejectRD
          ∪ {c ↦ a}
  END
```

Fig. 9: Timeout: Concrete events

models the initiator responding with a reject message to any proposals received after the timeout. The new *receiveReject2* event will take a relationship that is in the *rejectSD* variable and add it to *rejectRD* variable. Instead of adding this as a new event a developer could merge it with the existing *receiveReject* event from the initial chain. The new *failToPropose* event refines the abstract *failure* event that was added to the abstract model for the Timeout pattern. It can occur after the deadline has passed and no proposals have been received.

## 7 Refuse Pattern

| |
|---|
| **Name**: Refuse<br>**Fault**: An agent cannot support the action requested.<br>**Tolerance Pattern**: 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. |

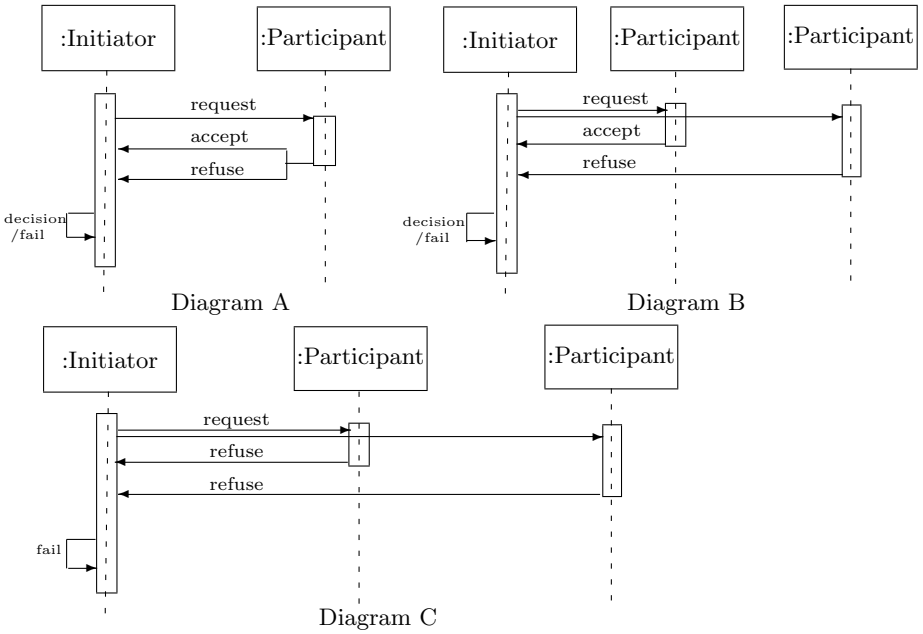

Fig. 10: Refuse: Interaction diagrams

Not all agents that receive a request will be able to fulfill it. The request may be in conflict with the agent's own goals. This could be due to the agent being overloaded, or the agent is competing against the requestor and it would not be in their interest to help. Software design does not always implement the concept of a refusal. Object-based systems use the term 'design by contract' to describe an obligation held by an object that it cannot alter at runtime [13]. The autonomy of agents means that the obligations between agents are weaker than in design by contract and a multi-agent system must be designed to cope when an agent refuses to undertake a request.

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. An agent is allowed a choice when responding to a request. The agent can either agree to fulfill the request or it can refuse.

Figure 10 shows interaction diagrams for the Refuse pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a request to a participant agent. The participant agent can respond with either an accept or refuse message. The initiator will then make a decision and the interaction may fail if the accept message is not suitable or a refuse message was sent. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of accept and refuse messages are received in response to the request. Diagram C shows the case where only refuse messages are received and the only outcome is a failure of the interaction.

The events that are required in the abstract machine for the Refuse pattern are the same as those shown in Figure 7 for the Timeout pattern. To model the Refuse pattern in the refinement of the initial chain three variables and three events have been added. In the contract net case study the refusals are

$$
\begin{array}{l}
refuseS \subseteq cfpR \\
refuseR \subseteq refuseS \\
refuseS \cap proposeS = \varnothing \\
failedCommit \subseteq dom(refuseR) \\
failedCommit \cap dom(proposeR) = \varnothing \\
failed = failedCommit
\end{array}
$$

Fig. 11: Refuse: Concrete invariants

**sendProposal**
ANY c, a WHERE
$c \mapsto a \in cfpR$
$c \mapsto a \notin proposeS$
$c \mapsto a \notin refuseS$
THEN
$proposeS := proposeS \cup \{c \mapsto a\}$
END

**sendRefusal**
ANY c, a WHERE
$c \mapsto a \in cfpR$
$c \mapsto a \notin proposeS$
$c \mapsto a \notin refuseS$
THEN
$refuseS := refuseS \cup \{c \mapsto a\}$
END

**receiveRefusal**
ANY c, a WHERE
$c \mapsto a \in refuseS$
$c \mapsto a \notin refuseR$
THEN
$refuseR := refuseR \cup \{c \mapsto a\}$
END

**failToCommit** REFINES failure
ANY c WHERE
$c \in dom(refuseR)$
$c \notin dom(proposeR)$
$c \notin failedCommit$
THEN
$failedCommit := failedCommit \cup \{c\}$
END

Fig. 12: Refuse: Concrete events

modelled so they are equivalent to the proposals. The invariants in Figure 11 specify variables that model sending and receiving refuse messages. An additional invariant specifies that the proposals and refusals for a conversation cannot be from the same agent, $refuseS \cap proposeS = \varnothing$. The *failedCommit* variable models that state of the conversation when all of the replies are refusals. This variable refines the abstract *failed* variable.

The events for the pattern are shown in Figure 12. The guard of the *sendProposal* event from the initial chain has been modified to prevent an agent that has made a refusal for the conversation from also making a proposal. The *sendRefusal* event adds a relationship that is in the *cfpR* variable to the *refuseS* variable. The *receiveRefusal* event takes a relationship that is in the *refuseS* variable and adds it to the *refuseR* variable. The *failToCommit* event models the case when all of the responses are refusals and the conversation fails. This is a refinement of the abstract *fail* event.

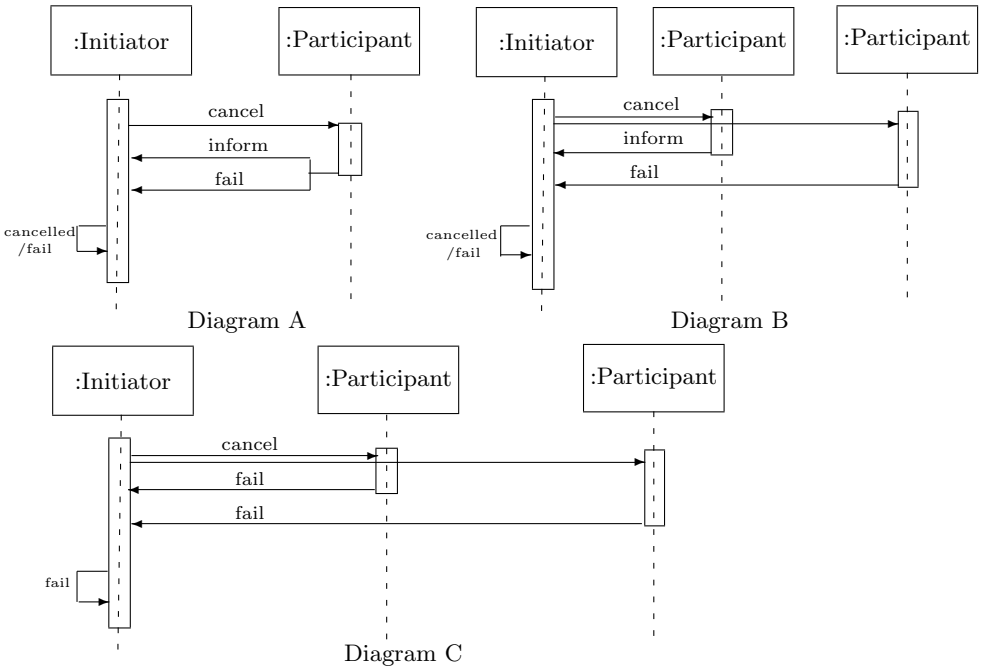


Fig. 13: Cancel: Interaction diagrams

## 8 Cancel Pattern

**Name**: Cancel
**Fault**: The requesting agent no longer requires an action to be performed.
**Tolerance Pattern**: 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 an event for that agent to receive a cancel message. Further events need to be added to allow the agent to reply with either an inform, if they have cancelled the action, or a failure, if they have not, and for those messages to be received.

Once an agent has requested an action they can then request that it is cancelled. An agent that exhibits rational behaviour may change its goals because the goal conflicts with other goals, the agent no longer desires the goal is fulfilled or the agent no longer believes that the goal can be fulfilled [2]. For the initiating agent to ensure that its beliefs about its environment are consistent it needs to know if the agents to whom it has delegated tasks have managed to undo any actions they have performed. The responses of the agents may affect the actions the initiating agent takes in response to its change of goals.

The Cancel pattern allows the agent that initiated the conversation to cancel the conversation at any point. The Cancel pattern will cancel a single request in a one-to-one conversation and will broadcast the cancellation in a one-to-many conversation to cancel all of the requests. Figure 13 shows interaction diagrams for the Cancel pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a cancel message to a participant agent. The participant agent can respond with either an inform message if they have successfully cancelled or a fail message if they have not. The initiator will then act according to its knowledge

```
INVARIANTS
 cancelled ⊆ cfp
EVENTS
  cancel
    ANY c WHERE
      c ∈ cfp
      c ∉ cancelled
    THEN
      cancelled := cancelled ∪ {c}
    END
```

Fig. 14: Cancel: Abstract events

$$
\begin{aligned}
&cancelS \subseteq cfpS\\
&cancelR \subseteq cancelS\\
&informCancelS \subseteq cancelR\\
&informCancelR \subseteq informCancelS\\
&failCancelS \subseteq cancelR\\
&failCancelR \subseteq failCancelS\\
&informCancelled \subseteq dom(informCancelR)\\
&failCancelled \subseteq dom(failCancelR)\\
&informCancelS \cap failCancelS = \varnothing\\
&informCancelled \cap failCancelled = \varnothing\\
&cancelled = informCancelled \cup failCancelled
\end{aligned}
$$

Fig. 15: Cancel: Refinement invariants

about the state of the system. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the cancel message. Diagram C shows the case where only fail messages are received and the cancelling of the action fails.

The Cancel pattern requires a new variable and event to be added to the abstract machine of the initial chain. The abstract pattern in Figure 14 shows the *cancel* event moving the conversation into the *cancelled* state.

The Cancel Pattern is modelled in the refinement as a collection of events that can occur at any point in the conversation. Events model a cancel message being sent from the initiating agent and received by the other agents involved. Events are also required to model the participating agents responding to the cancel request to inform the initiating agent whether they have managed to cancel their actions.

Figure 15 shows the invariant conditions from the Event-B extract of the Cancel pattern. The variables represent the states of the system as messages are sent and received. The *cancelS* variable is a subset of the *cfpS* variable so a conversation cannot be cancelled before it has begun. All of the other variables are specified as subsets according to the order of the messages that they represent being sent and received. *InformCancelS* and *failCancelS* are specified so the same agent cannot send an inform and fail message in the same conversation. The *informCancelled* and *failCancelled* variables are specified so the conversation cannot be in both states. An invariant condition specifies the intersection of the two variables as empty. The final invariant condition is the gluing invariant that relates the abstract *cancel* variable to a conjunction of the *informCancelled* and *failCancelled* variables.

Figure 16 shows the events that have been added to the initial refinement model to specify the Cancel pattern. The *sendCancel* event can be triggered by the initiating agent at any point in the conversation. The cancel message is broadcast to every agent involved in the conversation, $as = \{c\} \lhd cfpS$. The *receiveCancel* event allows the participants to receive the cancel message. The *sendInformCancel* and *sendFailCancel* events model the participants sending a message to the initiator about the success or failure of the cancellation. The *receiveInformCancel* and *receiveFailCancel* events model the initiator receiving the message. The last two events, *informCancelled* and *failCancelled*, refine the abstract *cancel* event and model the initiator evaluatings the success of the cancellation. The guards for the two events specify that at least one inform or

**sendCancel**
ANY c , as WHERE
$c \in dom(cfpS)$
$as = \{c\} \lhd cfpS$
$c \notin dom(cancelS)$
THEN
$cancelS := cancelS \cup as$
END

**sendInformCancel**
ANY c, a WHERE
$c \mapsto a \in cancelR$
$c \mapsto a \notin informCancelS$
$c \mapsto a \notin failCancelS$
THEN
$informCancelS := informCancelS \cup \{c \mapsto a\}$
END

**receiveInformCancel**
ANY c, a WHERE
$c \mapsto a \in informCancelS$
$c \mapsto a \notin informCancelR$
THEN
$informCancelR := informCancelR \cup \{c \mapsto a\}$
END

**informCancelled** REFINES cancel
ANY c WHERE
$c \in dom(informCancelR)$
$c \notin informCancelled$
$c \notin failCancelled$
THEN
$informCancelled := informCancelled \cup \{c\}$
END

**receiveCancel**
ANY c, a WHERE
$c \mapsto a \in cancelS$
$c \mapsto a \notin cancelR$
THEN
$cancelR := cancelR \cup \{c \mapsto a\}$
END

**sendFailCancel**
ANY c, a WHERE
$c \mapsto a \in cancelR$
$c \mapsto a \notin failCancelS$
$c \mapsto a \notin informCancelS$
THEN
$failCancelS := failCancelS \cup \{c \mapsto a\}$
END

**receiveFailCancel**
ANY c, a WHERE
$c \mapsto a \in failCancelS$
$c \mapsto a \notin failCancelR$
THEN
$failCancelR := failCancelR \cup \{c \mapsto a\}$
END

**failCancelled** REFINES cancel
ANY c WHERE
$c \in dom(failCancelR)$
$c \notin failCancelled$
$c \notin informCancelled$
THEN
$failCancelled := failCancelled \cup \{c\}$
END

Fig. 16: Cancel: Concrete events

fail cancel message has been received. The developer may want to strengthen these guards. For example, the guard of the *informCancelled* event could be strengthened to specify that all of the agents have replied with an inform message, $\{c\} \lhd informCancelR = AGENT \setminus \{a\}$, or that no fail messages have been received, $c \notin dom(failCancelR)$.

## 9 Failure Pattern

| |
|---|
| **Name**: Failure<br>**Fault**: An agent is prevented from carrying out an agreed action.<br>**Tolerance Pattern**: Add an event for an agent to send a failure message after they have committed to perform an action on behalf of another agent. Add an event for an agent to receive a failure message and an event for the system to respond to the failure. |

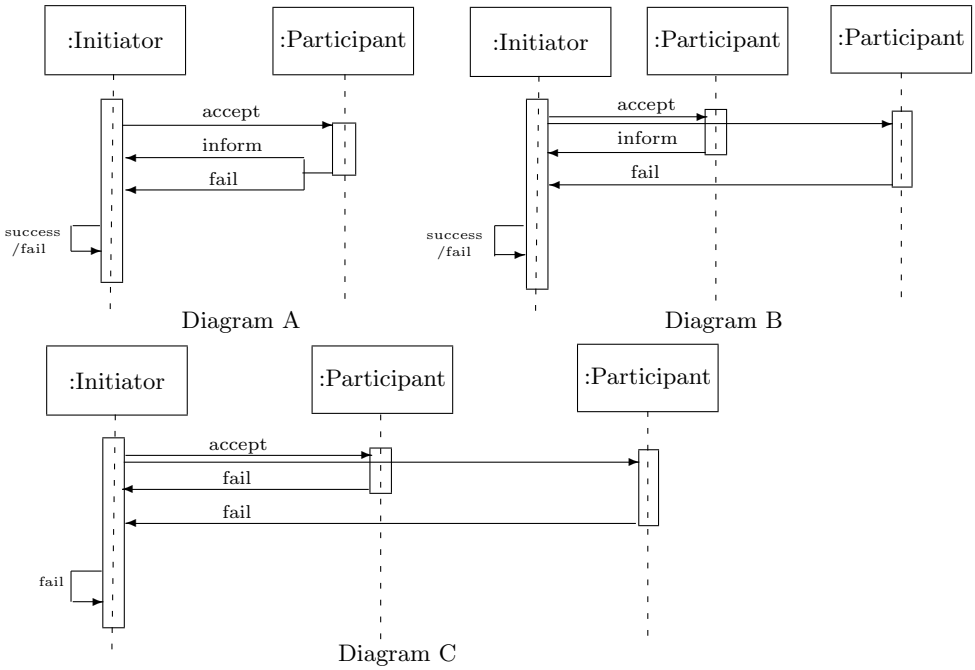


Fig. 17: Failure: Interaction diagrams

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 so that its beliefs do not become inconsistent.

Figure 17 shows interaction diagrams for the Failure pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a message that requests an action to a participant agent. The participant agent can respond with either an inform message, if they have successfully carried out the action, or a fail message, if they have not. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the accept message. The initiator will then be able to evaluate whether the task was carried out successfully. Diagram C shows the case where only fail messages are received. The Failure pattern is similar to the Refuse pattern where the responding agent has a choice of two replies that affect the outcome of the interaction differently. It occurs at a different point in the conversation. The Refuse pattern is used before a commitment is made and the Failure pattern is required after a commitment has been made.

Figure 18 shows the events from the abstract machine that are related to the Failure pattern. The failure pattern specifies the failure of the conversation after the selection of the proposals has been made. Either the *inform* event or the *failure* event can complete the conversation.

Figure 19 shows the invariants added for the Failure pattern. The Event-B models the agents involved in the contract net interaction protocol sending failure messages instead of inform messages after they have had their proposal accepted. The conversation cannot succeed and fail and this is modelled by

```
inform
    ANY  c  WHERE
      c ∈ selected
      c ∉ informed
      c ∉ failed
    THEN
      informed := informed ∪ {c}
    END
failure
    ANY  c  WHERE
      c ∈ selected
      c ∉ failed
      c ∉ failed
      c ∉ informed
    THEN
      failed := failed ∪ {c}
    END
```

Fig. 18: Failure: Abstract Events

$$failS \subseteq acceptR$$
$$failR \subseteq failS$$
$$failed1 \subseteq dom(failR)$$
$$informed \cap failed1 = \varnothing$$
$$failed = failed1$$

Fig. 19: Failure: Refinement invariants

```
sendFail
    ANY  c, a  WHERE
      c ↦ a ∈ acceptR
      c ↦ a ∉ failS
      c ↦ a ∉ informS
    THEN
      failS := failS ∪ {c ↦ a}
    END
failed REFINES failure
    ANY  c  WHERE
      c ∈ dom(failR)
      c ∉ failed1
      c ∉ informed
    THEN
      failed1 := failed1 ∪ {c}
    END
```

```
receiveFail
   ANY  c, a  WHERE
      c ↦ a ∈ failS
      c ↦ a ∉ failR
   THEN
      failR := failR ∪ {c ↦ a}
   END
```

Fig. 20: Failure: Concrete events

an invariant condition that specifies the intersection of the informed and failed variables as empty.

Figure 20 shows the three events that are added to the initial concrete model. The *sendFail* event models a participant having received an accept message that instructs it to carry out a task, $c \mapsto a \in acceptR$, sending a failure message in response. The *receiveFail* event models the initiator receiving the failure message. The *failed* event refines the abstract *failure* event and can occur after a failure message has been received.

## 10 Not-Understood Pattern

**Name**:Not-Understood
**Fault**: An agent receives a message that it does not expect or does not recognise.
**Tolerance Pattern**: Specify an event for receiving a message with an unknown or unexpected performative. Specify the action as replying with a not-understood message. Specify an event for receiving a not-understood message.

The autonomy of the agents means that there is no guarantee of their behaviour and the non-hierarchical nature of multi-agent systems often means that there is no single point of control. For agents in a multi-agent system to maintain a correct understanding of their environment they need to communicate with the other agents in the system to be aware of the actions of the other agents. This can create a large number of messages being passed between agents for them to be able to negotiate, query and inform. The possible heterogeneity of the agents means that they may have a different understanding of interaction protocols. The possibility of receiving arbitrary messages increases with each of these factors and the system needs to be able to cope with such faults. In a multi-agent system that has been developed in a top-down manner the faults that may lead to an arbitrary message being sent should not occur. However, it may be that some of the system components have been developed separately or that the formal development of the system is limited to modelling the interactions. In these cases the inclusion of the Not-Understood pattern will provide assurance that the system can still tolerate the faults outlined above.

The concept of the not-understood message is described in [8]. The not-understood message communicates that the sending agent has received a message that it does not understand. A not-understood message can be sent or received at any point in the conversation.

It is suggested in [8] that the action taken in response to a not-understood message should be different when the conversation involves broadcast messages and sub-protocols than that taken as part of a one-to-one conversation. It may be inappropriate to cancel the conversation when there are multiple agents performing sub-protocols. Each response to a not-understood message should be evaluated depending on the status of the conversation and is not specified by the Not-Understood pattern.

The Not-Understood pattern involves agents receiving an arbitrary message, responding with a not-understood message and agents receiving a not-understood message. The action taken by the agent to cope with the potential fault is not modelled and is left for the developer to treat.

Figure 21 shows an interaction diagram for the Not-Understood pattern. The interaction diagram shows an interaction between any two agent roles. One agent sends another agent a message that the receiving agent does not understand. The response from the receiving agent will be to reply with a not-understood message. The action taken by the agent that receives the not-understood message depends on their role in the conversation and the stage of the conversation.

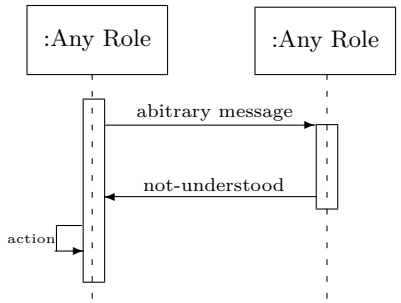


Fig. 21: Not-Understood: Interaction diagram

**INVARIANTS**
*recUnknown* $\subseteq$ *CONVERSATION*
*recNotUnderstood* $\subseteq$ *cfp*
**EVENTS**
**arbitraryComm**
**ANY** c **WHERE**
$c \in cfp$
**THEN**
$recUnknown := recUnknown \cup \{c\}$
**END**

**receiveNotUnderstood**
**ANY** c **WHERE**
$c \in cfp$
**THEN**
$recNotUnderstood :=$
$recNotUnderstood \cup \{c\}$
**END**

Fig. 22: Not-Understood: Abstract events

To model the Not-Understood pattern two events have been added to the initial abstract machine for the contract net case study. The events and variables are shown in Figure 22. The *arbitraryComm* event models the unrecognised message being received. The *receiveNotUnderstood* event abstractly model an agent receiving a not-understood message.

Figure 23 shows the extract from the Event-B refinement model that models the Not-Understood pattern in the Contract Net case study. The *unknownR* variable represents an arbitrary message being received and the *notUnderstoodS* variable represents a not-understood message being sent in response to the receipt of an arbitrary message. The *notUnderstoodR* variable represents a not-understood message being received.

The *receiveArbitraryComm* event models the receipt of a message that is not understood by the receiving agent. The *sendNotUnderstood* event models a not-understood message being sent in response to the receipt of this message. The *receiveNotUnderstood* event models an agent receiving a not-understood message. Further refinements of the pattern will model the agent's reactions to receiving the not-understood message. An initiator agent may decide to cancel the conversation or they may decide that the conversation has failed. The decisions by the agents will depend on the stage of the conversation when the not-understood message is received. This is left for the developer to decide and model.

```
INVARIANTS
        unknownR ⊆ cfpS
        notUnderstoodS ⊆ unknownR
        recUnknown = dom(notUnderstoodS)
        notUnderstoodR ⊆ notUnderstoodS
        recNotUnderstood = dom(notUnderstoodR)

EVENTS
receiveArbitraryComm
    ANY c, a WHERE
      c ↦ a ∈ cfpS
    THEN
      unknownR := unknownR ∪ {c ↦ a}
    END
sendNotUnderstood REFINES arbitraryComm
    ANY c, a WHERE
      c ↦ a ∈ unknownR
      c ↦ a ∉ notUnderstoodS
    THEN
      notUnderstoodS := notUnderstoodS ∪ {c ↦ a}
    END
receiveNotUnderstood REFINES receiveNotUnderstood
    ANY c, a WHERE
      c ↦ a ∈ notUnderstoodS
      c ↦ a ∉ notUnderstoodR
    THEN
      notUnderstoodR := notUnderstoodR ∪ {c ↦ a}
    END
```

Fig. 23: Not-Understood: Concrete invariants and events

## 11 Related Work

This section describes work that is related to the ideas presented in this paper. This work outlines approaches for constructing patterns in the B-Method and Event-B. Other work of interest are design patterns for multi-agent systems, particularly patterns that can be integrated with goal models that are used in multi-agent system design. Other fault-tolerance techniques for multi-agent systems have been investigated and are summarised in this section.

The B-Method is used in [14] to specify patterns, such as those identified in [7], as abstract machines. The pattern machines are instantiated by including another B model in the machine using the B-Method's inclusion mechanism. Pattern models can be composed to create a new pattern by using the inclusion mechanism to construct a new machine from the separate patterns. These patterns are specified at a single level of abstraction and are based on object-oriented development methods.

A set of patterns that solve design problems that are common when using the B-Method has been produced in [15]. The patterns they present include a pattern to associate multiple B machines, a pattern to produce unique objects

and patterns for creating sub and super-types of B machines. The patterns are implemented as either extracts of B machines or a description of how different mechanisms from the B-Method can be used to solve a described problem alongside an example of the patterns use. As with those described above, these patterns attempt to introduce some object-oriented concepts into B machines, are at a single level of abstraction and mainly address structural relationships between machines.

A refinement pattern for modelling time constraints in Event-B is presented in [16]. A pattern is produced by constructing a generic Event-B model that specifies the time constraints as a superpostion refinement. This model can be re-used to produce new refinements of the model to which the pattern is being applied. The authors suggest that it would be possible to prove the pattern model and the proof obligations generated by the pattern would not need to be discharged for the development model.

There are several methods for the use of patterns in the development of multi-agent systems. They are described and used with informal models. Coordination patterns, including a pattern of the contract net protocol, are presented in [17], patterns for mobile agent design are presented in [18], and [19] present patterns for implementing agents in object-oriented architectures. A strategy for constructing and using design patterns for agent systems that uses goals can be found in [20]. The patterns can be combined using a pattern language to construct a multi-agent system design.

The *extend* relationship used in Figure 1 of this paper is similar to those found in [21], but has not been formally defined.

The patterns presented in this paper provide fault-tolerance for the agents so they can continue to provide a service. Further strategies for managing faults in agent conversations include adapting general fault-tolerance techniques, such as replication [22], redundancy [23] and checkpoints [24], to multi-agent systems. Creating patterns for the specification of these fault-tolerance strategies in multi-agent systems is a possible direction for future work.

## 12 Summary

Event-B has been developed for modelling reactive and distributed systems and our experience shows that it is suited to the specification of multi-agent systems. The patterns presented above allow the developer to incorporate fault-tolerant behaviour in an Event-B development of a multi-agent system.

The patterns are presented as three elements: a description, interaction diagrams and Event-B examples. The Event-B examples make the patterns specific to Event-B development. The other elements of the pattern are more generic and could be suitable for other event-based formal methods. The inclusion of an abstraction of the pattern creates a pattern that can be fully integrated into the refinement chain of a development. The Event-B extracts included from the Contract Net case study show how the patterns can be applied to the model of

a complex multi-agent system. They also provide a re-usable specification of the pattern at a single level of refinement.

Providing an abstract and concrete pattern example will offer the developer guidance on how the pattern can be integrated into an Event-B development that uses refinement. The related work described above use patterns either as a superposition refinement to an Event-B model or as a component to a model. Integrating a pattern into the refinement chain of a development offers the advantages of making the pattern a fundamental part of the development. It is present in the abstraction of the model and can be analysed at all levels of abstraction.

## References


1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable Secure Computing **1**(1) (2004) 11–33
2. Ferber, J.: Multi-Agent Systems: Introduction to Distributed Artificial Intelligence. Addison Wesley (1999)
3. Jennings, N.R.: On agent-based software engineering. Artificial Intelligence **117** (2000) 277–296
4. Storey, N.: Safety-Critical Computer Systems. Pearson Education Limited, Bath, UK (1996)
5. 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. Volume 1393 of Lecture Notes in Computer Science., Montpellier, France, Springer (1998) 83 – 128
6. Jackson, D.: Dependable software by design. Scientific American - American Edition **294**(6) (2006) 68
7. 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)
8. FIPA: FIPA contract net interaction protocol specification. Available From: http://www.fipa.org/specs/fipa00029/SC00029H.pdf. Technical report, FIPA (2002)
9. Leavens, G., Abrial, J., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S.: Roadmap for enhanced languages and methods to aid verification. In: Proceedings of the 5th international conference on Generative programming and component engineering, Portland, Oregan, USA, ACM Press New York, NY, USA (2006) 221–236
10. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, XXI **77**(1 - 2) (2006) 1 – 28
11. Jones, C.B.: RODIN deliverable D9. preliminary report on methodology. Available From: http://rodin.cs.ncl.ac.uk/deliverables/rodinD9.pdf. Technical report, University of Newcastle-upon-Tyne, UK (2005)
12. Smith, R.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers **29**(12) (1980) 1104–1113

13. Meyer, B.: Object-oriented software construction. Prentice-Hall, Inc. Upper Saddle River, NJ, USA (1997)
14. Blazy, S., Gervais, F., Laleau, R.: Reuse of specification patterns with the B Method. In Bert, D., Bowen, J.P., King, S., Waldén, M., eds.: ZB 2003: Formal Specification and Development in Z and B. Volume 2651 of Lecture Notes in Computer Science., Turku, Finland, Springer-Verlag (2003) 40 – 57
15. Chan, E., Robinson, K., Welch, B.: Patterns for B: Bridging formal and informal development. In Julliand, J., Kouchnarenko, O., eds.: B 2007: Formal Specification and Development in B. Volume 4355 of Lecture Notes in Computer Science., Besancon, France, Springer-Verlag (2007) 125 – 139
16. Cansell, D., Méry, D.: Time constraint patterns for Event B development. In Julliand, J., Kouchnarenko, O., eds.: B 2007: Formal Specification and Development in B. Volume 4355 of Lecture Notes in Computer Science., Besancon, France, Springer-Verlag (2007) 140 – 154
17. Deugo, D., Weiss, M., Kendall, E.: Reusable patterns for agent coordination. In: Coordination of Internet Agents: Models, Technologies and Applications. Springer (2001) 347 – 368
18. Aridor, Y., Lange, D.: Agent design patterns: elements of agent application design. In: Proceedings of the second international conference on autonomous agents, ACM Press New York, NY, USA (1998) 108–115
19. Schelfthout, K., Coninx, T., Helleboogh, A., Holvoet, T., Steegmans, E., Weyns, D.: Agent implementation patterns. In Debenham, J., Henderson-Sellers, B., Jennings, N., Odell, J., eds.: Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies. (2002) 119–130
20. Weiss, M.: Pattern-driven design of agent systems: Approach and case study. In Eder, J., Missikoff, M., eds.: Conference on Advanced Information Systems Engineering (CAiSE). Volume 2681 of Lecture Notes in Computer Science., Klagenfurt/Velden, Austria, Springer (2003) 711 723
21. Back, R.: Incremental software construction with refinement diagrams. In Broy, M., Gruenbauer, J., Harel, D., Hoare, T., eds.: Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, Springer (2005) 3 – 46
22. 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
23. 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
24. 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