Rodin

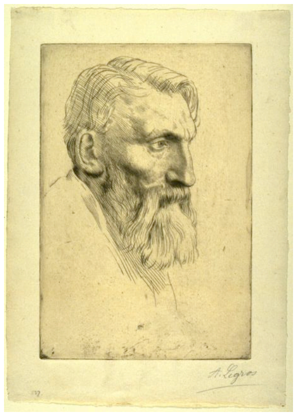
Workshop on Methods, Models and Tools for Fault Tolerance

Proceedings

July 3, 2007

Oxford

Michael Butler, Southampton University, UK
Cliff Jones, Newcastle University, UK
Alexander Romanovsky, Newcastle University, UK
Elena Troubitsyna, Aabo Akademi, Finland



FP6 IST RODIN (Rigorous Open Development Environment for Complex Systems)

http://rodin.cs.ncl.ac.uk/

The Workshop on Methods, Models and Tools for Fault Tolerance, is being held at the Integrated Formal Methods 2007 Conference on 3 July 2007 in Oxford. The aim of the workshop is to bring together researchers in design of fault tolerance systems with researchers in formal methods in order to help foster greater collaboration between these research fields. This follows on from a very the very successful Workshop on Rigorous Engineering of Fault Tolerant systems held in Newcastle in July 2005 at Formal Methods 2007. As a selection of extended papers from that workshop we published a book: M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna (Eds.). Rigorous Development of Complex Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 4157, Springer Verlag, September 2006.

This workshop is organised by the partners of FP6 IST RODIN (Rigorous Open Development Environment for Complex Systems), who are aiming to build a network of researchers from a wider community to promote integration of the dependability and formal methods fields.

Faults are unavoidable in all large systems and therefore designing for fault tolerance is essential. We believe that the use of formal methods is essential for mastering the complexity inherent in systems with faults and mechanism for tolerating those faults. Formal modelling and analysis helps designers to identify faults and to understand the effect of faults on systems behaviour. Modelling and analysis also helps designers understand the contribution of fault-tolerance mechanisms to overall system dependability. Faulty behaviour and fault tolerance provide a challenging application area for formal methods. This workshop will help researchers to further elaborate the challenges involved in applying formal methods to fault tolerance systems as well as helping to exchange ideas on addressing the challenges.

Michael Butler, Southampton University, UK
Cliff Jones, Newcastle University, UK
Alexander Romanovsky, Newcastle University, UK
Elena Troubitsyna, Aabo Akademi, Finland

July 2007

Table of Contents

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                                  INITIALISATION ...
deadline =                              failCommmit1 =
 ANY cc WHERE                            REFINES failCommit
  cc ∈ beforeTimeout                     ANY cc WHERE
 THEN                                     cc ∈ conv ∧
  beforeTimeout :=                        cc ∈ afterTimeout ∧
     beforeTimeout \ {cc} ||              cc ∉ ran(selected) ∧
  afterTimeout :=                         cc ∉ completed
     afterTimeout ∪ {cc}                 THEN
 END;                                     completed := completed ∪ {cc}
                                         END;
receiveProposal1 =                      receiveProposal2 =
 ANY aa, cc WHERE                        ANY aa, cc WHERE
  cc ∈ beforeTimeout ∧                    cc ∈ afterTimeout ∧
  cc ∉ selected ∩ completedConv ∧         cc ∉ selected ∩ completedConv ∧
  aa ↦ cc ∉ proposalR ∧                   aa ↦ cc ∉ proposalR ∧
  aa ↦ cc ∈ proposalG ∧                   aa ↦ cc ∈ proposalG ∧
 THEN                                    THEN
  proposalR := proposalR ∪ {aa ↦ cc}      rejectG := rejectG ∪ {aa ↦ cc}
 END;                                    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
taskFailure =
 ANY aa, cc WHERE
  aa ↦ cc ∈ acceptR ∧
  aa ↦ cc ∉ failureG ∧
  aa ↦ cc ∉ informG
 THEN
  failureG :=
     failureG ∪ {aa ↦ cc} ||
  participantConv :=
     participantConv \ {aa ↦ cc}
 END;

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

```
 INITIALISATION ...
failedContract =
 ANY cc WHERE
  cc ∈ conv ∧
  cc ∈ ran(selected) ∧
  cc ∉ completed ∧
  acceptG ▷ {cc} =
     failureR ▷ {cc} ∪ informR ▷ {cc} ∧
  failureR ▷ {cc} ≠ ∅
 THEN
  completed := completed ∪ {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

Mobile B Systems

Alexei Iliasov, Victor Khomenko, Maciej Koutny, Apostolos Niaouris and Alexander Romanovsky

School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract. Mobile agent systems (MAS) are complex distributed systems that are dynamically composed from communicating autonomous components. In this paper we introduce high level programming notation for the specification of MAS. This notation can faithfully capture both the behavioral and the functional model of a mobile agent. Furthermore, we provide its structured operational semantics through a set of rewriting rules together with a brief presentation of the supporting tool.
Keywords: agent systems, mobility, B Method, verification, model checking, Petri Nets

1 Introduction

Mobile agent systems (MAS) are complex distributed systems made of asynchronously communicating mobile autonomous components. Such systems have a number of advantages over traditional distributed systems, including: ease of deployment, low maintenance cost, scalability, autonomous reconfiguration and effective use of infrastructure. MAS are distinct enough to require specialised software engineering techniques. A number of methodologies, frameworks and middleware systems were proposed to support rapid development of MAS applications. However, there is as yet no single widely recognised standard and the problem of building large and dependable MAS remains open. In this paper, we propose a formal modelling based approach to developing MAS.

Our approach should be capable of addressing two critical issues. Firstly, being able to capture both the functional model (e.g., what kind of computations an agent is capable of doing) and the behavioral model of an agent (e.g., how an agent moves, how it interacts with other agents, etc.). The second issue is to develop the proper tools for the automatic verification of the produced model. While it possible to use just the Event-B notation (provided by the RODIN platform) to describe the functional model of an agent and statically verify it, it is quite challenging or even impossible to do the same with the behavioral model [**?**,**?**]. In this novel approach, we introduced a hybrid (Event-B combined with constructs inspired by two process algebras: KLAIM [**?**,**?**] π-calculus [**?**,**?**]) high level programming notation for the specification of mobile applications that can faithfully capture both the behavioral and the functional model of an agent. Moreover, we developed a plug-in for the RODIN platform based on an automatic verification engine of proven efficiency that supports the model

```
ROLE  Drinker
   BODY
      order   =   serve ∘ ();
      drink   =   skip
ROLE  Pub
   VARIABLES   int : beer = 0
   INVARIANT   beer ∈ 0...10
   BODY
      serve   =   IF
                     beer > 0
                  THEN
                     beer := beer − 1;
                     drink ∘ ()
                  END
SYSTEM
   LOCATIONS   pub1, pub2
   INVARIANT   beer@Pub ≥ drinker@pub
   AGENTS
      Student(drinker) := move(pub1).order;
      Pub(pub) := move(pub1).⟨beer@pubC < 3⟩move(home);

END
```

Fig. 1. Model of a simple multi-agent application.

checking of a given specification. By using a combination of static verification and model checking we are able to offer two different views on a model and carry out complimentary analysis of functional and dynamic properties.

The paper is structured as follows. In sections 2,3 we are introducing the new modelling language together with its structured operational semantics. In section 4, we are identifying fault tolerance properties of MAS and in the final section we are briefly presenting the developed plug-in that will be used for the automatic verification of MAS.

2 Modelling Language

The modelling language specifications, called *scenarios*, are of the following form:

$$
\begin{array}{ll}
\texttt{begin} - \texttt{scenario} & \\
\quad \ell_1 \ \dots \ \ell_k & \text{(locations)} \\
\quad rl_1 \ \dots \ rl_m & \text{(roles)} \\
\quad ag_1 = \texttt{new}(rl'_1) \ \dots \ ag_n = \texttt{new}(rl'_n) & \text{(agents)} \\
\quad E & \text{(process expression)} \\
\texttt{end} - \texttt{scenario} &
\end{array}
$$

In the above, it is assumed that $\ell_i \neq \ell_j$, $rl_i \neq rl_j$ and $ag_i \neq ag_j$, for all $i \neq j$; and that $rl'_i \in \{rl_1, \ldots, rl_m\}$, for all i.

The process expression describes a distributed system composed of agents, each agent being an instantiation of a role. Its general format is

$$ag_1 : pa\_act_{11} \,\textbf{.}\, \ldots \,\textbf{.}\, pa\_act_{1m_1} \,\textbf{.}\, \texttt{nil} \| \ldots \| ag_k : pa\_act_{k1} \,\textbf{.}\, \ldots \,\textbf{.}\, pa\_act_{km_k} \,\textbf{.}\, \texttt{nil}$$

where the ag_i's are agents and pa_act_{ij}'s are process algebra actions.

A role specification describes a set of *events* and *actions* which are procedures that update role variables and initiate further computations. A role event is invoked by the $\circ$ (trigger) statement with suitable arguments supplied by a calling agent. For example, in Figure **??**, the action `order` in the `drinker` role triggers event `serve` in role `pub` which in its turn may trigger `drinker`'s event `drink`. An action is invoked from within a process algebra expression, with constants or role variables as parameters. An action invocation may result in a chain of event invocations corresponding to communication between roles.

Executing `move`(ℓ) changes the current locality of an agent, and the function `number`(rl, ℓ) returns the number of agents associated with role rl in locality ℓ. The process expression is constructed from basic actions, which can be of one of the following forms:

– `move`(ℓ) moves the current agent (i.e., that labelling the sequential subexpression in which the action appears) to location ℓ;
– `migrate`(ℓ) moves the current agent to location ℓ provided that in its current locality there is no other agent which would want to trigger one of the events in ag;
– $act(ag, \boldsymbol{d})$ calls action act in agent ag with the actual parameters $\boldsymbol{d}$;
– $\langle bool \rangle$ is a guard, where $bool$ is a well-formed Boolean expression.

In addition to that we use prefix and (at the topmost level) parallel composition.

3 Semantics

Events and actions. We assume that $\mathbb{EN}$ and $\mathbb{AN}$ are infinite sets of, respectively, *event* and *action* names (or, simply, events and actions). With each name $x \in \mathbb{EN} \cup \mathbb{AN}$ we have associate finite sets of *parameters* and *variables*, respectively denoted by $param(x)$ and $var(x)$. The former are the parameters which are used in invocations of x, while $var(x)$ are all role variables which are involved in the execution of x (including those which are only read). Note that different actions and events belonging to a role may involve different, even disjoint, sets of variables. We assume that the sets of parameters, $param(x)$, and variables, $var(x)$, are implicitly ordered and so they can be represented as lists, and so a valuation for $param(x)$ can be represented as a list of values of appropriate type.

An execution of an event or action proceeds in two (consecutive) phases, each phase being treated as atomic. The first phase sets the parameters to correct values, while the second updates the state of local variables and possibly triggers an event. To capture the effect of these two phases, with each event or action name x we associate two mappings:

$$\begin{aligned} \mathit{store}_x &: \mathbb{EVAL}_{param(x)} \times \mathbb{EVAL}_{var(x)} \to \mathbb{EVAL}_{var(x)} \\ \mathit{trigger}_x &: \mathbb{EVAL}_{param(x)} \times \mathbb{EVAL}_{var(x)} \to \{\bot\} \cup \mathbb{IEN}\ . \end{aligned}$$

In the above, we use $\mathbb{EVAL}_{param(x)}$ and $\mathbb{EVAL}_{var(x)}$ to denote all legal *evaluations* of the parameters and variables involved in the execution of x. Moreover, $\mathbb{IEN}$ is the set of *instantiated* event names, each such name consisting of a valid event name together with a legal evaluation of its parameters (see below). The symbol $\bot$ is used in case when the second phase of x's execution does not trigger any event.

An *instantiated* event or action is a pair (x, η), where $x \in \mathbb{EN} \cup \mathbb{AN}$ and $\eta \in \mathbb{EVAL}_{param(x)}$. Each such (x, η) can be denoted as $x(\boldsymbol{d}) \overset{\mathrm{df}}{=} x(d_1, \ldots, d_k)$, where $d_i = \eta(p_i)$ for every p_i — the i-th parameter of x. The sets of all instantiated events and actions will be denoted by $\mathbb{IEN}$ and $\mathbb{IAN}$, respectively.

Both functions can be applied for any valuation *eval* of a set of variables V such that $var(x) \subset V$. In such a case, $eval' = store_x(\boldsymbol{d}, eval)$ is a valuation for V satisfying $eval'|_{var(x)} = store_x(\boldsymbol{d}, eval|_{var(x)})$ and $eval'(v) = eval(v)$ for $v \in V \setminus var(x)$. Moreover, $trigger_x(\boldsymbol{d}, eval) = trigger_x(\boldsymbol{d}, eval|_{var(x)})$.

Roles. A *role* is a triple $rl \overset{\mathrm{df}}{=} (Var, Ev, Act, init)$, where Var, Ev and Act are finite sets of, respectively, variables, events and actions, and *init* is an initial valuation of the variables in Var. It is assumed that $var(x) \subseteq Var$, for every $x \in EA_{rl} \overset{\mathrm{df}}{=} Ev \cup Act$.

Mobile B systems. A *mobile B system* (or MB-system) is a tuple $MBS \overset{\mathrm{df}}{=} (Rol, Ag, Loc, \rho)$, where Rol is a finite set of roles, Ag is a finite set of agent names (or, simply, agents), Loc is a finite set of localities, and $\rho : Ag \to Rol$ is a mapping assigning a role to every agent. It is assumed that no event or action occurs in more than one role in Rol.

Global states of an evolving MB-system are captured by the notion of a *configuration* which is a tuple $Conf \overset{\mathrm{df}}{=} (\lambda, \epsilon, \pi, \sigma, E)$, where:

– λ is a mapping returning, for every $ag \in Ag$, a locality in Loc;
– ϵ is a mapping returning, for every $ag \in Ag$, a valuation for the variables in $\rho(ag)$;
– π is a mapping returning, for every $ag \in Ag$ and $x \in EA_{\rho(ag)}$, a valuation for the parameters in $param(x)$;
– σ is a mapping returning, for every $ag \in Ag$, a mapping from the events and actions in $\rho(ag)$ to the set $\{on, off\}$,
(Note: $\sigma(ag, x) = on$ is used to indicate that x within ag is in-between the two phases of its execution);
– E is a process expression which provides a partial description of what may happen next (the other source for the continuation of behaviour comes from the events and actions which are in-between the two phases of their execution).

Evolution starts from an *initial* configuration satisfying $\lambda_0(ag) \overset{\text{df}}{=} \mathit{limbo}$, $\epsilon_0(ag) \overset{\text{df}}{=} \mathit{init}_{\rho(ag)}$, $\pi_0(ag)(x)(p) \overset{\text{df}}{=} \bot$, and $\sigma_0(ag)(x) \overset{\text{df}}{=} \mathit{off}$, for all $ag \in Ag$, $x \in EA_{\rho(ag)}$ and $p \in \mathit{param}(x)$. Note that *limbo* is a special location, not listed explicitly in the scenario, which is used to store agents before they become active, and after they become disconnected.

Rules of the operational semantics. These are given in Figure **??**, where, for all $ag \in Ag$, $\ell \in Loc$ and $ev \in Ev$, we have the following:

$$\begin{aligned} \mathit{todo}(ag) &= \{(x, \mathit{trigger}_x(\epsilon(ag), \pi(ag,x))) \mid \sigma(ag,x) = on \wedge \lambda(ag) \neq \mathit{limbo}\} \\ \mathit{waiting}(\ell) &= \{ev \mid \exists ag : \lambda(ag) = \ell \wedge \exists x, \boldsymbol{d} : (x,(ev,\boldsymbol{d})) \in \mathit{todo}(ag)\} \\ \mathit{users}(\ell, ev) &= \{ag \mid \lambda(ag) = \ell \wedge ev \in Ev_{\rho(ag)}\} \ . \end{aligned}$$

Note that $\mathit{todo}(ag)$ identifies all the second phases of executions which the agent ag tries to trigger at the current state, $\mathit{waiting}(\ell)$ lists all events which agents at the location ℓ try to trigger, and $\mathit{users}(\ell, ev)$ is the set of agents at the location ℓ which have ev as one of the events. Note that these notations depend also on an implicit *Conf*.

The various rules of the operational semantics capture essential characteristics of the mobile systems we intend to model and analyse, as follows:

- In the rules Sum and Par, we assume $E \neq E'$ to check whether the propagated execution came from the process expression rather than finishing an event or action belonging to one of the agents;
- The rule Mov moves an agent to a specific location; its main role is to simulate agent disconnection and re-connection.
- The rule Mig expresses another, conditional mobility of an agent, as it is applicable only if the agent does not have any events that other agent(s) residing at the same locality want to trigger;
- In the guarded rule Gua, the execution is possible only if the Boolean expression evaluates to true in the current configuration (cf. $\mathit{Conf} \models \mathit{bool}$);
- The rule Act governs the execution of an action belonging to an agent which is triggered from within a process expression. The action can only be triggered if it is currently dormant (in that agent);
- The rule Evt expresses the triggering of an event. It is assume that there is at least one agent which contains the event in the current location, and that no such agent is in the middle of executing this event.

4 Fault Tolerance Properties

The specific focus of our work is on identifying and checking fault tolerance properties of the mobile systems. To do this we are extending our models with a number of typical faults which can happen in the ambient campus and with the corresponding recovery actions the system conducts to tolerate them. We are defining properties of two types: properties stating the success of recovery

$$[\textsc{Par}]\quad \frac{(\lambda,\epsilon,\pi,\sigma,E) \longrightarrow (\lambda',\epsilon',\pi',\sigma',E') \quad \wedge \quad E \neq E'}{\begin{array}{c}(\lambda,\epsilon,\pi,\sigma,E \| E'') \longrightarrow (\lambda',\epsilon',\pi',\sigma',E' \| E'') \\ (\lambda,\epsilon,\pi,\sigma,E'' \| E') \longrightarrow (\lambda',\epsilon',\pi',\sigma',E'' \| E')\end{array}}$$

$$[\textsc{Mov}]\quad \frac{}{(\lambda,\epsilon,\pi,\sigma, ag : \mathtt{move}(\ell)\,\textbf{.}\,E) \longrightarrow (\lambda[ag \mapsto \ell],\epsilon,\pi,\sigma, ag : E)}$$

$$[\textsc{Mig}]\quad \frac{\mathit{waiting}(\lambda(ag)) \cap \mathit{Ev}_{\rho(ag)} \neq \varnothing}{(\lambda,\epsilon,\pi,\sigma, ag : \mathtt{migrate}(\ell)\,\textbf{.}\,E) \longrightarrow (\lambda[ag \mapsto \ell],\epsilon,\pi,\sigma, ag : E)}$$

$$[\textsc{Gua}]\quad \frac{\mathit{Conf} \models \mathit{bool} \quad \wedge \quad (\lambda,\epsilon,\pi,\sigma, z\,\textbf{.}\,E) \longrightarrow (\lambda',\epsilon',\pi',\sigma',E)}{(\lambda,\epsilon,\sigma, ag : \langle \mathit{bool} \rangle z\,\textbf{.}\,E) \longrightarrow (\lambda',\epsilon',\sigma', ag : E)}$$

$$[\textsc{Act}]\quad \frac{\sigma(ag)(\mathit{act}) = \mathit{off}}{(\lambda,\epsilon,\pi,\sigma, ag : \mathit{act}(\boldsymbol{d})\,\textbf{.}\,E) \longrightarrow (\lambda,\epsilon,\pi[(ag,\mathit{act}) \mapsto \boldsymbol{d}],\sigma[(ag,\mathit{act}) \mapsto \mathit{on}], ag : E)}$$

$$[\textsc{Evt1}]\quad \frac{(x,\bot) \in \mathit{todo}(ag)}{(\lambda,\epsilon,\pi,\sigma,E) \longrightarrow \begin{pmatrix}\lambda, \\ \epsilon[ag \mapsto \mathit{store}_x(\pi(ag,x),\epsilon(ag))], \\ \pi, \\ \sigma[(ag,x) \mapsto \mathit{off}], \\ E\end{pmatrix}}$$

$$[\textsc{Evt}]\quad \frac{\begin{array}{c}(x,(ev,\boldsymbol{d})) \in \mathit{todo}(ag) \quad \wedge \quad \mathit{users}(\lambda(ag),ev) = \{ag_1,\ldots,ag_n\} \neq \varnothing \\ \wedge \;\; \sigma(ag_1)(ev) = \ldots = \sigma(ag_n)(ev) = \mathit{off}\end{array}}{(\lambda,\epsilon,\pi,\sigma,E) \longrightarrow \begin{pmatrix}\lambda, \\ \epsilon[ag \mapsto \mathit{store}_x(\pi(ag,x),\epsilon(ag))], \\ \pi[(ag_i,ev) \mapsto \boldsymbol{d}]_{i=1}^{n}, \\ \sigma[(ag,x) \mapsto \mathit{off}][(ag_i,ev) \mapsto \mathit{on}]_{i=1}^{n}, \\ E\end{pmatrix}}$$

Fig. 2. Operational semantics rules.

and continuous provisioning of the system service and properties stating the correct application of the fault tolerance mechanisms (such as scopes, nested scopes, exception handling, disconnection detection and retry provided by the middleware).

Some of the examples of the properties of the first type are presence of all scope members in the current location to provide full recovery, presence of the agents responsible for local handling of all exceptions, successful handling of all exceptions in the system, successful dealing with lost/crashed agents, etc. Examples of the properties of the second types are correctness of scoping structure, absence of information smuggling between scopes, involving (if necessary) all agents in a scope in cooperative handling. ....

5 Supporting tool

Mobile agent systems are highly concurrent causing the state space explosion when applying model checking techniques. We therefore use an approach which copes well with such a problem, based on partial order semantics of concurrency and the corresponding Petri net unfoldings. This approach is suitable for verification of *reachability-like* (or *state*) properties, such as:

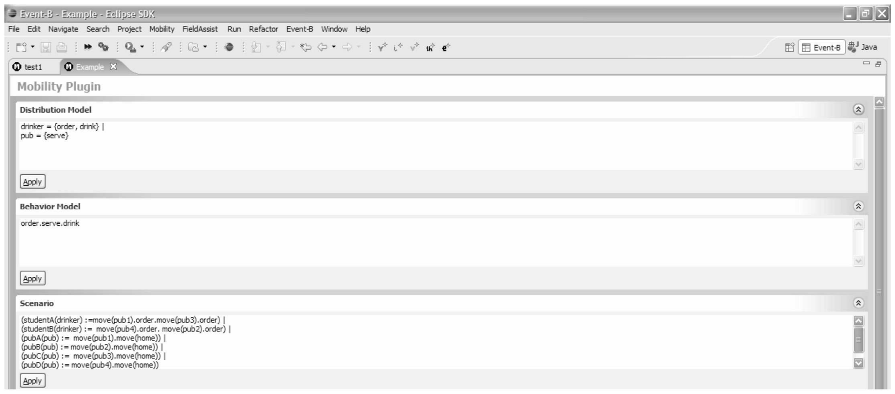
- The system never deadlocks, though it may terminate in a pre-defined set of successful termination states.
- Security properties, i.e., all sub-scope participants are participants of the containing scope.
- Proper using of the scoping mechanism, for example: a scope owner does not attempt to leave without removing the scope; agents do not leave or delete a scope when other agents expect some input from the scope; and the owner of a scope does not delete it while there are still active agents in the scope.
- Proper use of cooperative recovery: all scope exceptions must be handled when a scope completes; all scope participants eventually complete exception handling; and no exceptions are raised in a scope after an agent leaves it.
- Application-specific invariants. (Note that the negation of an invariant is a state property, i.e., the invariant holds if and only if there is no reachable state of the system where it is violated.)

As mentioned in the previous sections, the agent modelling language is a hybrid of Event-B and a process algebra with mobility characteristics tailored to our requirements. In order to apply net unfoldings, we first need to translate the hybrid programming notation (the full theoretical details of this translation will be published soon) into Petri nets.

The supporting tool was build as a extension to the RODIN platform. Using the plug-in architecture offered by the Eclipse IDE, it was possible to extend the platform to support all the necessary features for our purposes. There are several key components in this Petri net based tool:

- Rodin platform providing the Event-B specification of our model;

- Editors allowing the user to input/edit process algebra expressions corresponding to the distribution and behavior model together with the scenario part of the agent modelling language (Figure **??**);
- Translator taking as input Event-B specification and process algebra expressions and providing as output the 'Mobile B Systems' programming notation;
- Translator from the 'Mobile B Systems' notation to high-level Petri nets;
- Unfolder (PUNF [**?**,**?**]) for deriving a finite prefix of the unfolding of the translated Petri net;
- Verifier (MPSAT [**?**]) which establishes, by working with the finite prefix, whether the necessary properties of the original input hold;
- Animator where the output of tool (error traces), together with the complete model of the specification can be visualised.


Event-B - Example - Eclipse SDK
File Edit Navigate Search Project Mobility FieldAssist Run Refactor Event-B Window Help
Event-B
Java
test1
Example
Mobility Plugin
Distribution Model
Apply
Behavior Model
Apply
Scenario
Apply

```
drinker = {order, drink} |
pub = {serve}
```

```
order.serve.drink
```

```
(studentA(drinker) :=move(pub1).order.move(pub3).order) |
(studentB(drinker) :=  move(pub4).order. move(pub2).order) |
(pubA(pub) :=  move(pub1).move(home)) |
(pubB(pub) := move(pub2).move(home)) |
(pubC(pub) :=  move(pub3).move(home)) |
(pubD(pub) := move(pub4).move(home))
```

Fig. 3. Mobility plug-in editors

There were two key issues to consider when building such translators. The first is a behaviour preserving translation of the combined specification into a high-level Petri net. In our work, we were following a technique used previously in translating two process algebras, KLAIM [**?**] and π-calculus [**?**] extended by the modelling of state based transformations coming from Event-B. The second issue that needed our attention is that the resulting high-level Petri net, must be accepted as input from the model-checking engine based on net unfolding. The combination of Event-B and process algebra, the translation from the newly obtained modelling language to the high level net input required by the model checker as well as the invocation of the unfolder are completely automated tasks and hidden from the user. Moreover, the verifier checks for deadlock freeness and invariant violations and it is capable to provide feedback in case of discovering an error in the specification. These error traces can be visualised with the help of the included animator, providing further assistance to the designer (Figure **??**).

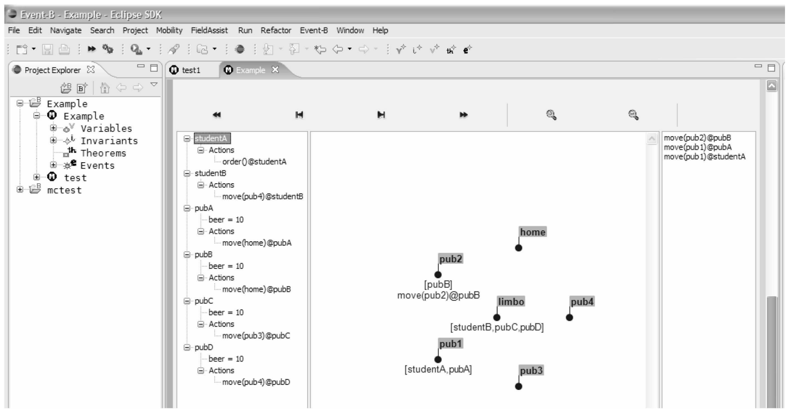


Fig. 4. Animator in action

6 Conclusions

In this paper we present a novel approach for modelling and verifying the correctness of complex mobile agent systems. None of the existing languages (e.g., Event-B) were capable of capturing the complete behaviour of mobile agents. Our achievement was the development of a single hybrid (Event-B together with a process algebra with mobility characteristics) high level programming notation that is capable of capturing both the behavioral and the functional model of agents. This language has strong theoretical foundations and its structured operational semantics are also presented here. Finally, an efficient model checker has been developed as a plug-in for the RODIN platform. The plan for this tool is to support a significant part of the Event-B notation and also behaviourally rich process algebra expressions. Internal versions of the tool are currently available and a public version is scheduled for a released within few months.

Acknowledgements This research was supported by the EC IST grant 511599 (Rodin) and the RAEng/Epsrc grant EP/C53400X/1 (Davac).

References

1. L.Bettini et al.: The KLAIM Project: Theory and Practice. Proc. of *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, Springer, LNCS 2874 (2003) 88–150.
2. L.Bettini and R.De Nicola: Mobile Distributed Programming in X-Klaim. Proc. of *Formal Methods for Mobile Computing*, M. Bernardo and A. Bogliolo. Springer, LNCS 3465 (2005) 29–68.
3. Busi, N., Gorrieri, R.: A Petri net Semantics for π-calculus. Proc. of *CONCUR'95*, LNCS 962 (1995) 145–159.

4. M.Butler and M.Leuschel: Combining CSP and B for Specification and Property Verification. Proc. of *Formal Methods 2005*, J.Fitzgerald et al.. Springer, LNCS (3582) 2005. 221-236
5. Devillers, R., Klaudel, H., Koutny, M.: Petri Net Semantics of the Finite π-Calculus Terms. *Fundamenta Informaticae* 70 (2006) 1–24.
6. R.Devillers, H.Klaudel and M.Koutny: A Petri Net Semantics of a Simple Process Algebra for Mobility. *Electronic Notes in Theoretical Computer Science* 154 (2006) 71–94.
7. A.Iliasov, V.Khomenko, M.Koutny and A.Romanovsky:, On Specification and Verification of Location-Based Fault Tolerant Mobile Systems, *RODIN Book*, (2006), 168-188
8. V.Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Computing Science, University of Newcastle upon Tyne (2003).
9. R.Milner, J.Parrow and D.Walker: A Calculus of Mobile Processes. *Information and Computation* 100 (1992) 1–77.
10. H.Treharne and S.Schneider: How to Drive a B Machine. Proc. of *ZB2000: Formal Specification and Development in Z and B*, Springer, LNCS 1878 (2000) 188–208.
11. `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/tools.html`

Formal Reasoning About Fault Tolerance and Parallelism in Communicating Systems

Linas Laibinis[1], Elena Troubitsyna[1], and Sari Leppänen[2]

[1] Åbo Akademi University, Finland
[2] Nokia Research Center, Finland
{Linas.Laibinis, Elena.Troubitsyna}@abo.fi
Sari.Leppanen@nokia.com

Abstract. Telecommunication systems should have a high degree of availability, i.e., high probability of correct provision of requested services. To achieve this, correctness of software for such systems and system fault tolerance should be ensured. In this paper we show how to formalise and extend Lyra – a top-down service-oriented method for development of communicating systems. In particular, we focus on integration of fault tolerance mechanisms into the entire Lyra development flow.

1 Introduction

Modern telecommunication systems are usually distributed software-intensive systems providing a large variety of services to their users. Development of software for such systems is inherently complex and error prone. However, software failures might lead to unavailability or incorrect provision of system services, which in turn could incur significant financial losses. Hence it is important to guarantee correctness of software for telecommunication systems.

Nokia Research Center has developed the design method Lyra [6] – a UML2-based service-oriented method specific to the domain of communicating systems and communication protocols. The design flow of Lyra is based on the concepts of decomposition and preservation of the externally observable behaviour. The system behaviour is modularised and organised into hierarchical layers according to the external communication and related interfaces. It allows the designers to derive the distributed network architecture from the functional system requirements via a number of model transformations.

From the beginning Lyra has been developed in such a way that it would be possible to bring formal methods (such as program refinement, model checking, model-based testing etc.) into more extensive industrial use. A formalisation of the Lyra development would allow us to ensure correctness of system design via automatic and formally verified construction. The achievement of such a formalisation would be considered as significant added value for industry.

In our previous work [5, 4] we proposed a set of formal specification and refinement patterns reflecting the essential models and transformations of Lyra. Our approach is based on stepwise refinement of a formal system model in the

B Method [1] – a formal refinement-based framework with automatic tool support. Moreover, to achieve system fault tolerance, we extended Lyra to integrate modelling of fault tolerance mechanisms into the entire development flow. We demonstrated how to formally specify error recovery by rollbacks as well as reason about error recovery termination.

In this paper we show how to extend our Lyra formalisation to model parallel execution of services. In particular, we demonstrate how such an extension affects the fault tolerance mechanisms incorporated into our formal models. The extension makes our formal models more complicated. However, it also gives us more flexibility in choosing possible recovery actions.

2 Previous Work

In this section we give a brief overview of on our previous results [5, 4] on formalising and verifying the Lyra development process. This work form the basis for new results presented in the next section.

2.1 Formalising Lyra

Lyra [6] is a model-driven and component-based design method for the development of communicating systems and communication protocols, developed in the Nokia Research Center. The method covers all industrial specification and design phases from pre-standardisation to final implementation.

Lyra has four main phases: *Service Specification*, *Service Decomposition*, *Service Distribution* and *Service Implementation*. The *Service Specification* phase focuses on defining services provided by the system and their users. In the *Service Decomposition* phase the abstract model produced at the previous stage is decomposed in a stepwise and top-down fashion into a set of service components and logical interfaces between them. In the *Service Distribution* phase, the logical architecture of services is distributed over a given platform architecture. Finally, in the *Service Implementation* phase, the structural elements are integrated into the target environment. Examples of Lyra UML models from the Service Specification phase of a positioning system are shown on Fig.1.

To formalise the Lyra development process, we choose the B Method as our formal framework. The B Method [1] is an approach for the industrial development of highly dependable software. Recently the B method has been extended by the Event B framework [2, 7], which enables modelling of event-based systems. Event B is particularly suitable for developing distributed, parallel and reactive systems. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [3], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping.

The B Method adopts the top-down approach to system development. The basic idea underlying stepwise development in B is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*. The refinement process starts from creating an abstract specification

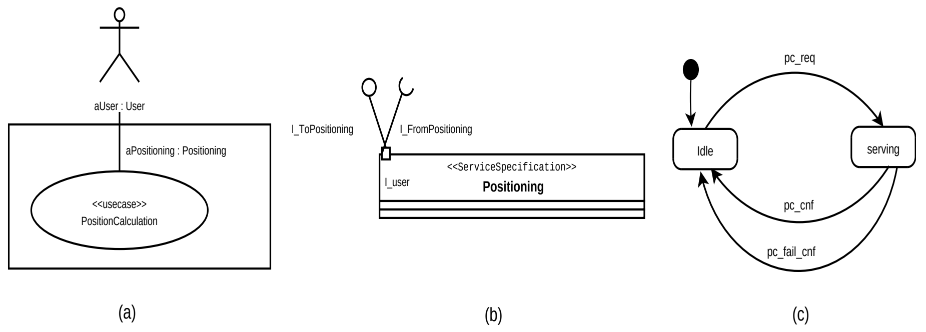


Fig. 1. (a) Domain Model. (b) Class Diagram of Positioning. (c) State Diagram.

and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artefacts.

While formalising Lyra, we single out a generic concept of a communicating service component and propose B patterns for specifying and refining it. In the refinement process a service component is decomposed into a set of service components of smaller granularity specified according to the proposed pattern. Moreover, we demonstrate that the process of distributing service components between network elements can also be captured by the notion of refinement. Below we present an excerpt from an abstract B specification pattern of a communicating service component.

The proposed approach to formalising Lyra in B allows us to verify correctness of the Lyra decomposition and distribution phases. In development of real systems we merely have to establish by proof that the corresponding components in a specific functional or network architecture are valid instantiations of these patterns. All together this constitutes a basis for automating industrial design flow of communicating systems.

```
MACHINE ACC
...
VARIABLES in_data, out_data
INVARIANT in_data ∈ DATA ∧ out_data ∈ DATA
INITIALISATION in_data, out_data := NIL, NIL
OPERATIONS
input(param, time) =
  PRE param ∈ DATA ∧ time ∈ NAT1 ∧ ¬(param = NIL) ∧ in_data = NIL
  THEN in_data := param END;
calculate =
  SELECT ¬(in_data = NIL) ∧ out_data = NIL
  THEN out_data :∈ DATA − {NIL} END;
```

```
res ← output =
  PRE ¬(out_data = NIL)
  THEN
    res := out_data ||
    in_data, out_data := NIL, NIL
  END
```

2.2 Introducing Fault Tolerance in the Lyra Development Flow

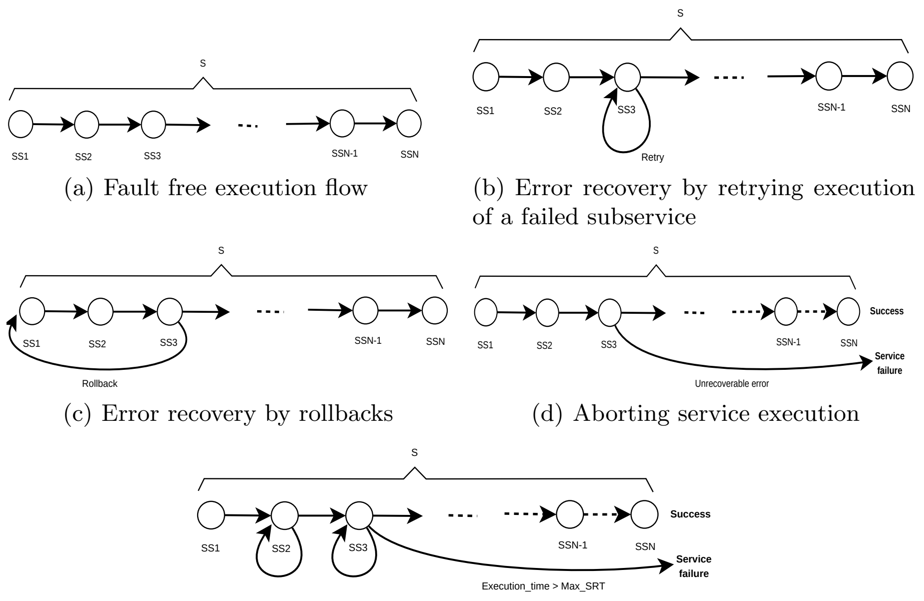
Currently the Lyra methodology addresses fault tolerance implicitly, i.e., by representing not only successful but also failed service provision in the Lyra UML models. However, it leaves aside modelling of mechanisms for detecting and recovering from errors – the fault tolerance mechanisms. We argue that, by integrating explicit representation of the means for fault tolerance into the entire development process, we establish a basis for constructing systems that are better resistant to errors, i.e., achieve better system dependability. Next we will discuss how to extend Lyra to integrate modelling of fault tolerance.

In the first development stage of Lyra we set a scene for reasoning about fault tolerance by modelling not only successful service provision but also service failure. In the next development stage – *Service Decomposition* – we elaborate on representation of the causes of service failures and the means for fault tolerance.

In the *Service Decomposition* phase we decompose the service provided by a service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request other service components to do it. According to Lyra, the flow of the service execution is managed by a special service component called *Service Director*. *Service Director* co-ordinates the execution flow by enquiring the required subservices from the external service components.

In general, execution of any stage of a service can fail. In its turn, this might lead to failure of the entire service provision. Therefore, while specifying *Service Director*, we should ensure that it does not only orchestrates the fault-free execution flow but also handles erroneous situations. Indeed, as a result of requesting a particular subservice, *Service Director* can obtain a normal response containing the requested data or a notification about an error. As a reaction to the occurred error, *Service Director* might

– retry the execution of the failed subservice,
– repeat the execution of several previous subservices (i.e., roll back in the service execution flow) and then retry the failed subservice,
– abort the execution of the entire service.



(a) Fault free execution flow (b) Error recovery by retrying execution of a failed subservice (c) Error recovery by rollbacks (d) Aborting service execution (e) Aborting the service due to timeout

Fig. 2. Service decomposition: faults in the execution flow

The reaction of *Service Director* depends on the criticality of an occurred error: the more critical is the error, the larger part of the execution flow has to be involved in the error recovery. Moreover, the most critical (i.e., unrecoverable) errors lead to aborting the entire service. In Fig.2(a) we illustrate a fault free execution of the service S composed of subservices $S_1, \ldots,\ S_N$. Different error recovery mechanisms used in the presence of errors are shown in Fig.2(b) - 2(d).

Let us observe that each service should be provided within a certain finite period of time – the *maximal service response time Max_SRT*. In our model this time is passed as a parameter of the service request. Since each attempt of subservice execution takes some time, the service execution might be aborted even if only recoverable errors have occurred but the overall service execution time has already exceeded *Max_SRT*. Therefore, by introducing *Max_SRT* in our model, we also guarantee termination of error recovery, i.e., disallow infinite retries and rollbacks, as shown in Fig.2(e).

3 Fault Tolerance in the Presence of Parallelism

Our formal model briefly described in the previous section assumes sequential execution of subservices. However, in practice, some of subservices can be executed in parallel. Such simultaneous service execution directly affects the fault tolerance mechanisms incorporated into our B models. As a result, they become more complicated. However, at the same time it provides additional, more flexible options for error recovery that can be attempted by *Service Director*.

3.1 Modelling Execution Flow

The information about all subservices and their required execution order becomes available at the Service Decomposition phase. This knowledge can be formalised as a data structure

$$Task \ : \ seq(\mathcal{P}(SERVICE))$$

Here $SERVICE$ is a set of all possible subservices. Hence, $Task$ is defined as a sequence of subsets of subservices. It basically describes the control flow for the top service in terms of required subservices. At the same time, it also indicates which subservices can be executed in parallel.

For example,

$$Task \ = \ < \{S1, S2\},\ \{S3, S4, S5\},\ \{S6\} >$$

defines the top service as a task that should start by executing the services $S1$ and $S2$ (possibly in parallel), then continuing by executing the services $S3$, $S4$, and $S5$ (simultaneously, if possible), and, finally, finishing the task by executing the service $S6$.

Essentially, the sequence $Task$ defines the data dependencies between subservices. Also, $Task$ can be considered as the most liberal (from point of view of parallel execution) model of service execution. In the Service Distribution phase the knowledge about the given network architecture becomes available. This can reduce the parallelism of service control flow by making certain services that can be executed in parallel to be executed in a particular order enforced by the provided architecture.

Therefore, $Task$ is basically the desired model of service execution that will serve as the reference point for our formal development. The actual service execution flow is modelled in by the sequence $Next$ which is of the same type as $Task$:

$$Next \ : \ seq(\mathcal{P}(SERVICE))$$

Since at the Service Decomposition phase we do not know anything about future service distribution, $Next$ is modelled as an abstract function (sequence), i.e., without giving its exact definition. However, it should be compatible with $Task$. More precisely, if $Task$ requires that certain services S_i and S_j should be executed in a particular order, this order should be preserved in the sequence $Next$. However, $Next$ can split parallel execution of given services (allowed by $Task$) by sequentially executing them in any order.

So the sequence $Next$ abstractly models the actual control flow of the top service. It is fully defined (instantiated) only in the refinement step corresponding to the Service Distribution phase. For example, the following instantiation of $Next$ would be correct with respect to $Task$ defined above:

$$Next \ = \ < \{S2\},\ \{S1\},\ \{S4\},\ \{S3, S5\},\ \{S6\} >$$

Also, we have to take into account that *Service Director* itself can become distributed, i.e., different parts of service execution could be orchestrated by distinct service directors residing on different network elements. In that case, for every service director, there is a separate $Next$ sequence modelling the corresponding part of the service execution flow. All these control flows should complement each other and also be compatible with $Task$.

3.2 Modelling Recovery Actions

As we described before, a *Service Director* is the service component responsible for orchestrating service execution. It monitors execution of the activated subservices and attempts different possible recovery actions when these services fail. Obviously, introducing parallel execution of subservices (described in the previous subsection) directly affects the behaviour of *Service Director*.

Now, at each execution step in the service execution flow, several subservices can be activated and run simultaneously. *Service Director* should monitor their execution and react asynchronously whenever any of these services sends its response. This response can indicate either success or a failure of the corresponding subservice.

The formal model for fault tolerance presented in Section 2.2 is still valid. However, taking into account parallel execution of services presents *Service Director* with new options for its recovery actions. For example, getting response from one of active subservices may mean that some or all of the remaining active subservices should be stopped (i.e., interrupted). Also, some of the old recovery action (like retrying of service execution) are now parameterised with a set of subservices. The parameter indicates which subservices should be affected by the corresponding recovery actions.

Below we present the current full list of actions that *Service Director* may take after it receives and analyses the response from any of active subservices. Consequently, *Service Director* might

– **Continue** to the next service execution step. In case of successful termination of all involved subservices (complete success).
– **Wait** for response from the remaining active subservices. In case of successful termination of one of few active subservices (partial success).
– **Abort** the entire service and send the corresponding message to the user or requesting component. In case of an unrecoverable error or the service timeout.
– **Stop** (a set of subservices) by sending the corresponding requests to interrupt their execution (partial abort). In case of a failure which requires to retry or rollback in the service execution flow.
– **Retry** (a set of subservices) by sending the corresponding requests to re-execute the corresponding subservices. In case of a recoverable failure.
– **Rollback** to a certain point of the service execution flow. In case of a recoverable failure.

Service Director makes its decision using special abstract functions needed for evaluating responses from service components. These functions should be supplied (instantiated) by the system developers at a certain point of system development.

Here is a small excerpt from the B specification of *Service Director* specifying the part where it evaluates a response and decides on the next step:

```
handle =
    ...
    resp := Eval(curr_task, curr_state);
    CASE resp OF EITHER
      CONTINUE THEN
        IF curr_task = size(Next) THEN finished := TRUE
        ELSE active_serv, curr_task := Next(curr_task + 1), curr_task + 1 END
      WAIT THEN skip
      RETRY THEN active_serv := active_serv ∪ Retry(curr_task, curr_state)
      STOP THEN active_serv := active_serv ∪ Cancel(curr_task, curr_state)
      ROLLBACK THEN curr_task := Rollback(...); active_serv := Next(curr_task)
      ABORT THEN finished := TRUE
    END
    ...
```

where the abstract functions Next, Retry, Cancel, and Rollback are defined (typed) as follows:

$\mathsf{Next} : \mathsf{seq}(\mathcal{P}(\mathsf{SERVICE}))$

$\mathsf{Eval} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \rightarrow \{\mathsf{SUCCESS}, \mathsf{RETRY}, \mathsf{CANCEL}, \mathsf{ROLLBACK}, \mathsf{ABORT}\}$

$\mathsf{Retry} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow \mathcal{P}(\mathsf{SERVICE})$

$\mathsf{Cancel} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow \mathcal{P}(\mathsf{SERVICE})$

$\mathsf{Rollback} : 2..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow 1..\mathsf{size}(\mathsf{Next}) - 1$

4 Conclusions

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalises and extends Lyra [6] – the UML2-based design methodology adopted in Nokia. The formalisation is done within the B Method [1] and its extension EventB [2] – a formal framework supporting system development by stepwise refinement. The proposed approach establishes a basis for automatic translation of UML2-based development of communicating systems into the refinement process in B. Such automation would enable smooth integration of formal methods into existing development practice.

In particular, in this paper we focused on integrating fault tolerance mechanisms into the formalised Lyra development process. A big challenge is formal modelling of parallel service execution and its effect on system fault tolerance. The ideas presented in this paper are implemented by extending our previously developed B models. The formalised Lyra development is verified by completely proving the corresponding B refinement steps using the Atelier B tool. At the moment, we are in the process of moving this development to new Event B language developed within the EU RODIN project [8].

Acknowledgements

This work is supported by IST FP6 RODIN Project.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. J.-R. Abrial. Extending B without Changing it (for Developing Distributed Systems). *Proceedings of 1st Conference on the B Method*, pp.169-191, Springer-Verlag, November 1996, Nantes, France.
3. Clearsy. *AtelierB: User and Reference Manuals.* Available at http://www.atelierb.societe.com/index_uk.html.
4. L. Laibinis, E. Troubitsyna, S. Leppänen, J.Lilius, and Q. Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems. *Rigorous Development of Complex Fault-Tolerant Systems, Lecture Notes in Computer Science*, Vol.4157, chapter 14, pp.261-287, Springer-Verlag, 2006.
5. L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, and Qaisar Malik. Formal Model-Driven Development of Communicating Systems. Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM'05), LNCS 3785, Springer, November 2005.
6. S. Leppänen, M. Turunen, and I. Oliver. Application Driven Methodology for Development of Communicating Systems. *Forum on Specification and Design Languages*, Lille, France, 2004.
7. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/.
8. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

Formal Development of Fault Tolerant Transactions for a Replicated Database using Ordered Broadcasts

Divakar Yadav[*] and Michael Butler[**]

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K
{dsy04r,mjb}@ecs.soton.ac.uk

Abstract. Data replication across several sites improves fault tolerance as available sites can take over the load of failed sites. Data is usually accessed within a transactional framework. However, updating replicated data within a transactional framework is a complex affair due to failures and conflicting transactions. Group communication primitives have been proposed to support transactions in a asynchronous distributed system. In this paper we outline how a refinement based approach with Event B can be used for the development of a reliable replicated database system that ensure atomic commitment of update transactions using group communication primitives.

1 Introduction

A replicated database system can be defined as a distributed system where copies of the database are kept across several sites. Data access in a replicated database can be done within a transactional framework. It is advantageous to replicate the data if the transaction workload is predominantly read only. However, during updates, keeping the replicas in a consistent state arises due to race conditions among conflicting update transactions. A distributed transaction may span several sites reading or updating data objects. A typical distributed transaction contains a sequence of database operations which must be processed at all of the participating sites or none of the sites to maintain the integrity of the database. The strong consistency criterion in the replicated database requires that the database remains in a consistent state despite transaction failures. The possible causes of the transaction failures include bad data input, time outs, temporary unavailability of data at a site and deadlocks.

No common global clock or shared memory exist in a distributed system. The sites communicate by the exchange of messages which are delivered to them after arbitrary time delays. In such systems up-to-date knowledge of the system is not known to any process or site. This problem can dealt by relying on group communication primitives that provide higher guarantees on the delivery of messages. Group communication has also been investigated in Isis [5], Totem [14] and Trans [12]. The protocols in these systems use varying broadcast primitives and address group maintenance, fault tolerance and consistency services. The transaction semantics in the management of replicated data is also considered in [3, 16]. In addition to providing fault tolerance, one of the important issues to be addressed in the design of replica control protocols is consistency. The *One Copy Equivalence* [4] criteria

[*] Divakar Yadav is a Commonwealth Scholar supported by the Commonwealth Scholarship Commission in the United Kingdom.
[**] Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

requires that a replicated database is in a mutually consistent state only if all copies of data objects *logically* have the same identical value.

Distributed algorithms can be deceptive, may have complex execution paths and may allow unanticipated behavior. Rigorous reasoning about such algorithms is required to ensure that an algorithm achieves what it is supposed to do [11]. Group communication services have been studied as a basic building block for many fault tolerant distributed services, however the application of formal methods providing clear specifications and proofs of correctness is rare [6]. Some of the important work on the application of formal methods to group communication services in order to verify the properties of algorithm are given in [7, 17]. The work reported in [7] uses I/O automata for the specifications and proves properties about all trace behavior of the automation. In [17] formal results are provided that defines whether or not a totally ordered protocol provides a causal order. They provide a proof of correctness by doing proofs by hand. Instead, our approach of specifying the system and verification is based on the technique of abstraction and refinement. This technique is supported by the Event B [13], an event driven approach used together with B Method [1]. This formal approach carries a step-wise development from initial abstract specifications to a detailed design of a system in the refinement steps. Through the refinement proofs we verify that design of detailed system conforms to the abstract specifications. The refinement approach of Event B has also been used for the formal development of fault tolerant communication systems [9]. We have used the Click'n'Prove [2] B tool for proof obligation generation and for discharging proof obligations.

The remainder of this paper is organized as follows: Section 2 outline the system model, Section 3 describes group communication primitives and its application to replicated database, Section 4 outline the formal development of a system of total order broadcast and Section 5 concludes the paper.

2 System Model

Our system model consist of a sets of sites and data objects. Users interact with the database by *starting transactions*. We consider the case of full replication and assume all data objects are updateable. The *Read Anywhere Write Everywhere* [4, 15] replica control mechanism is considered for updating replicas. A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction will either *commit* or *abort* the effect of all database operations. The following types of transactions are considered for this model of replicated database.

- *Read-Only Transactions* : These transaction are submitted locally to the site and *commit* after reading the requested data object locally.
- *Update Transactions* : These transactions update the requested data objects. The effect of update transactions are global, thus when committed, all replicas of data objects maintained at all sites must be updated. In case of abort, none of the sites update the data object.

Let the sequence of read/write operations issued by the transaction T_i be defined by a set of objects $objectset[T_i]$ where $objectset[T_i] \neq \varnothing$. Let the set $writeset[T_i]$ represents the set of object to be *updated* such that $writeset[T_i] \subseteq objectset[T_i]$.

A transaction T_i is a read-only transaction if $writeset[T_i] = \varnothing$. Similarly a transaction T_i is a update transaction if its $writeset[T_i] \neq \varnothing$. Two update transactions T_i and T_j are in *conflict* if the sequence of operations issued by T_i and T_j are defined on set of object $objectset[T_i]$ and $objectset[T_j]$ respectively and $objectset[T_i] \cap objectset[T_j] \neq \varnothing$.

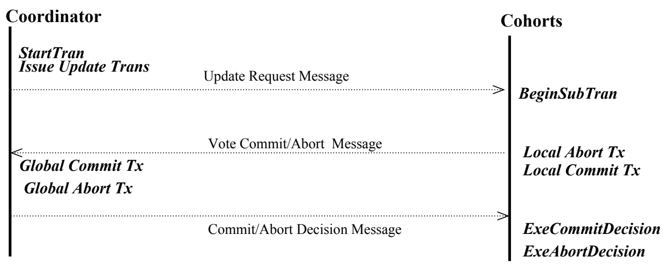


Fig. 1. Events of Update Transaction

In order to meet the strong consistency requirement where each transaction reads the correct value of a replica, *conflicting* transactions need to be executed in isolation. In our model, we ensure this property by not *issuing* a transaction at a site if there is a conflicting transaction that is *active* at that site. In our model the transactions are executed as follows.

- A read-only transaction T_i is executed locally at the initiating site of T_i(also called the coordinator site of T_i) by acquiring locks on the data object defined by $objectset[T_i]$.
- An update transaction T_i is executed by broadcasting its $objectset[T_i]$ to the participating sites. On delivery, a participating site S_j initiates a subtransaction T_{ij} by acquiring locks on $objectset[T_i]$. If the objects are currently locked by another transaction, the T_{ij} is blocked.
- The coordinator site of T_i waits for the vote commit/abort messages from all participating site. A global commit/abort message is broadcasted by coordinator site of T_i only if it receives all local commit message from all participating sites or at-least one vote-abort message from participating sites.

The commit or abort decision of global transaction T_i is taken at the coordinator site within the framework of a two phase commit protocol as shown in Fig. 1 as follows. A global transaction T_i *commits* if *all* T_{ij} *commit* at S_j. The global transaction T_i *aborts* if *some* T_{ij} *aborts* at S_j.

In [20] we have presented a formal refinement based approach using Event B to model and analyze distributed transaction. In our abstract model, an update transaction modifies the abstract one copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database. Through the refinement proofs, we verify that the design of the replicated database confirms to the one copy database abstraction despite transaction failures at a site. In this model we assume that the sites communicate by a reliable broadcast which eventually deliver messages without any ordering guarantees. Therefore, the conflicting operations of update transactions originating from different sites may arrive at different sites in the different order. This may lead to the deadlocks among the conflicting transactions which results in unnecessary abort of various transactions. The abort of these conflicting update transactions may be avoided if a reliable broadcast also provide higher ordering guarantees on the message delivery. We are currently investigating and formalizing the group communication primitive with reference to the the update transactions.

3 Ordering Properties

In our model of replicated databases [20] we assume that the sites communicate by exchange of messages using a reliable broadcast. A reliable broadcast [8] eventually deliver the messages to all participating sites and satisfies following properties.

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: All correct processes delivers a same set of message, i.e. if a process delivers a message m then all correct processes eventually delivers m.
- *Integrity* : No spurious messages are ever delivered, i.e., for any message m, every correct process delivers m at most once and only if m was previously broadcast by $sender(m)$.

A reliable broadcast imposes no restriction on the order in which messages are delivered to the processes. This may lead to the blocking of conflicting transactions and the sites may abort one or more of the conflicting transaction by timeouts. For example, consider two conflicting update transactions T_i and T_j initiated at site S_i and S_j respectively. Both of the transactions may be blocked in the following scenario :

- S_i starts transaction T_i and acquire locks on $objectset[T_i]$ at site S_i. Site S_i broadcast update messages of T_i to participating sites. Similarly, another site S_j starts a transaction T_j , acquires locks on $objectset[T_j]$ at site S_j and broadcast update messages of T_j to participating sites.
- The site S_i delivers update message of T_j from S_j and S_j delivers update message of T_i from S_i. The T_j is blocked at S_i as S_i waits for vote-commit from S_j for T_i. Similarly, T_i is blocked at S_j waiting for vote-commit from S_i for T_j

In order to recover from the above scenario where two conflicting transaction are blocked, either or both transactions may be aborted by the sites. This problem can greatly be simplified by assuming a stronger notion of reliable broadcast that provide higher order guarantees on message delivery. Various definitions of ordering properties have been discussed in [8]. A reliable broadcast can be used to deliver messages to the processes following a *FIFO Order*, *Local Order*, *Causal Order* or a *Total Order*. An informal specifications of these ordering properties are given below.

- *FIFO Order* : If a particular process broadcasts a message *M1* before it broadcasts a message *M2*, then each recipient process delivers *M1* before *M2*.
- *Local Order*: If a process delivers *M1* before broadcasting the message *M2*, then each recipient process delivers *M1* before *M2*.
- *Causal Order* : If the broadcast of a message *M1* *causally precedes* the broadcast of a message *M2*, then no correct process delivers *M2* unless it has previously delivered *M1*.
- *Total Order* : If two process *P1* and *P2* both deliver the messages *M1* and *M2* then *P1* delivers *M1* before *M2* if and only if *P2* also delivers *M1* before *M2*.

A *Causal Order Broadcast* is a reliable broadcast that satisfies the *causal order* requirement. The notion of causality is based on *causal precedence relation* ($\rightarrow$) defined by the Lamport [10]. It is also shown in the [8] that a *causal order* combines the properties of both FIFO and Local order. The FIFO Order and Local Order is shown in the Fig. 2. The dotted lines represents the delivery of message violating the respective order. Similarly, a *Total Order Broadcast*[1] is a reliable broadcast that

[1] The *Total Order Broadcast* is also known as Atomic Broadcast. Both of the terms are used interchangeably. However we prefer the former as the term *atomic* suggests the *agreement* property rather than *total order*.

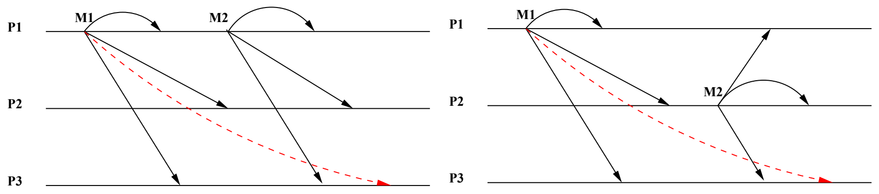


Fig. 2. [a]. FIFO order [b]. Local Order

satisfies the *total order* requirement. The *Agreement* and *Total Order* requirements of Total Order Broadcast imply that all correct processes eventually deliver the same *sequence* of messages [8]. As shown in the Fig. 3[a] the messages are delivered conforming to both causal and a total order. However, as shown in Fig. 3[b] the delivery order respects a total order but violates causal order. It can be noticed that the causality among the broadcast of message *M2* and *M3* is not preserved for delivery.

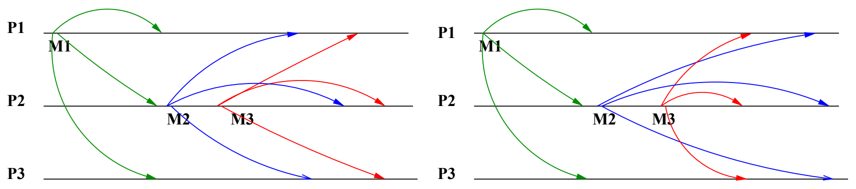


Fig. 3. [a]Total Order and a Causal Order [b]Total Order but not a Causal Order

Processing Transactions on Ordered Broadcast : In a replicated database that uses a reliable broadcast without ordering guarantees, the conflicting operations of the transactions mays arrive at different sites in different orders. This may lead to unnecessary aborts of the transaction due to blocking. The abortion of the conflicting transaction can be avoided by using a *total order broadcast* which delivers and execute the conflicting operations at all sites in the same order. Similarly, a *causal order broadcast* captures conflict as causality and transactions executing conflicting operations are executed at all sites in the same order. In [19], we used a refinement approach with Event B for formal development of broadcast system which deliver messages satisfying various ordering properties. In the abstract model we outlined how a causal order on the message can be constructed and in the refinement we show how it can correctly be implemented by vector clocks. In [18] we present a model of causal order broadcast which does not deliver messages to the sender. In a separate development in [19] we also outline the construction of abstract total order on messages and its implementation using sequencer numbers in the refinements. Lastly, we also formally develop a system where message are delivered following both a total and a causal order[2].

4 Total Order Broadcast

In this section we outline the incremental development of a system of total order broadcast. The operations of an update transaction are communicated by the co-

[2] A reliable broadcast that satisfies both causal and total order is also called *Causal Atomic Broadcast.*

ordinator site to the participating sites by broadcast of an update message. The abortion of the conflicting transaction can be avoided by using an underlying system of *total order broadcast* which delivers and execute the conflicting operations at all sites in the same order.

Our system model consists of a set of sites communicating by a reliable broadcast. The key issues with respect to a system of total order broadcast are ; how to build an order on messages? and what information is necessary for defining an abstract total order? Our approach of building a total order is based on the notion of *sequencer*, where a designated site called *sequencer* site is responsible for building a total order. A *sequencer* site may also take the role of a *sender* and *destination* in addition to the role of *sequencer*. The protocol consists of first broadcasting an *update message* m to all destinations including the sequencer. Upon receiving m, sequencer assigns it a sequence number and broadcast its sequence number to all destinations through *control messages*. Each destination site deliver m in the order of sequence numbers.

Abstract Model of Total Order Broadcast : The abstract B model of a total order broadcast is given in Fig. 4 and 5. In the abstract model a total order is built on the message when it is delivered to any site in the system for the first time. The specification consists of four variables *sender*, *order*, *deliver* and *delorder*. The *sender* is a partial function from *MESSAGE* to *SITE*. The mapping $(m \mapsto s) \in sender$ indicates that message m was sent by the site s. The variable *order* is defined as a relation among the messages. A mapping of the form $(m1 \mapsto m2\ ) \in order$ indicates that message $m1$ is *totally ordered before* $m2$. In order to represent the

```
MACHINE          TotalOrder
SETS             SITE; MESSAGE
VARIABLES        sender, order, delorder, deliver
INVARIANT        sender ∈ MESSAGE ⇸ SITE
              ∧  order ∈ MESSAGE ↔ MESSAGE
              ∧  delorder ∈  SITE ⇸ (MESSAGE ↔ MESSAGE)
              ∧  deliver ∈ SITE ↔ MESSAGE
INITIALISATION
                  sender := ∅     ||  order :=∅   ||
                  delorder := SITE × {∅}  ||    deliver := ∅
```

Fig. 4. Abstract Model of Total Order : Initial Part

delivery order of messages at a site, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in delorder(s)$ indicate that a site s has delivered $m1$ before $m2$. The variable *deliver* represent the messages delivered to a site following a total order. A mapping of form $(s \mapsto m) \in deliver$ represents that a site s has delivered m.

The event *Broadcast* given in the Fig. 5 models the broadcast of a message. Similarly the event *Order* models the construction of a total order on *first ever* delivery of a message to any site in the system. The *TODeliver* models the delivery of the messages when a total order on the message has been constructed.

Constructing a Total Order : The event *Order* models the delivery of a message (mm) at a site (ss) when it is delivered for the *first* time. The following guards of

```
Broadcast (ss ∈ SITE , mm ∈ MESSAGE ) ≅
    WHEN      mm ∉ dom(sender)
    THEN      sender := sender ∪ {mm ↦ ss}
    END;

Order (ss ∈ SITE , mm ∈ MESSAGE ) ≅
    WHEN      mm ∈ dom(sender)
            ∧ mm ∉ ran(deliver)
            ∧ ran(deliver) ⊆ deliver[{ss}]
    THEN    deliver := deliver ∪ {ss ↦ mm}
          || order := order ∪ ( ran(deliver)  × {mm})
          || delorder(ss) := delorder(ss) ∪ ( deliver[{ss}] × {mm})
    END;

TODeliver (ss ∈ SITE , mm ∈ MESSAGE) ≅
    WHEN      mm ∈ dom(sender)
            ∧ mm ∈ ran ( deliver )
            ∧ ss ↦ mm ∉ deliver
            ∧ ∀m.( m ∈ MESSAGE ∧ (m↦ mm) ∈ order ⇒ (ss ↦ m) ∈ deliver)
    THEN    deliver := deliver ∪ {pp ↦ mm}
          || delorder(ss) := delorder(ss) ∪ ( deliver[{ss}] × {mm})
    END
```

Fig. 5. Abstract Model of Total Order : Events

this event ensures that the message(mm) has not been delivered elsewhere and that each message delivered at any other site has also been delivered to the site(ss):

$$mm \notin ran(deliver)$$
$$ran(deliver) \subseteq deliver[\{ss\}]$$

Later in the refinement we show that this is a function of a designated site called *sequencer*. As a consequence of the occurrence of the *Order* event, the message mm is delivered to site ss and the variable *order* is updated by mappings in $(ran(deliver) \times \{mm\})$. This indicates that all messages delivered at any site in the system are *ordered* before mm. Similarly, the delivery order at the site ss is also updated such that all messages delivered at ss precedes mm. It can be noticed that the total order for a message is built when it is delivered to a site for the *first* time.

The event $TOdeliver(ss,mm)$ models the delivery of a message mm to a site ss respecting the *total order*. As the guard $mm \in ran(deliver)$ implies that the mm has been delivered to at least one site and it also implies that the total order on the message mm has also been constructed. Later in the refinement we show that site ss represents a site other than the *sequencer*. The guards of the event ensure that message mm has already been delivered elsewhere and that all messages which precedes mm in the abstract total order has also been delivered to ss.

After constructing an abstract model of a total order we verify that this model preserves the total order properties. The agreement and total order requirements imply that all correct process eventually deliver all messages in the same order [8]. Therefore we add following invariant to our model as a primary invariant.

$$\forall(m1, m2, p).((m1 \mapsto m2) \in delorder(p) \Rightarrow \ (m1 \mapsto m2) \in order)$$

This invariant states that if two messages, irrespective of the sender, are delivered at any site then their delivery order at that site corresponds to the abstract total order. In order to discharge the proof obligations associated with this invariant we also discover new invariants. The process of discovery of invariants is explained in [19]. Similarly, in order to verify that our model also preserves the transitivity property, we added following invariants to our model and discharge the proof obligations associated with the invariant.

$$\forall(m1, m2, m3).((m1 \mapsto m2) \in order \wedge (m2 \mapsto m3) \in order \Rightarrow \ (m1 \mapsto m3) \in order)$$

Overview of the Refinement Chain : Instead of presenting the full refinement chain in the detail we will just briefly present the overview of the refinement steps. Our refinement chain consists of six levels. A brief outline of each level is given below.

L1 This consist of abstract model of total order broadcast. In this model, the abstract total order is constructed when a message is delivered to a site for the first time. At all other sites a message is delivered in the total order.

L2 This is a refinement of abstract model which introduces *sequencer*. In this refinement we demonstrate that the total order is built by the *sequencer*. In the refined specifications of the *Order* event we outline that the *first ever* delivery of a message is done at the sequencer. In order to do that the guards of *Order* event in the abstract specification [$mm \notin ran(deliver) \wedge ran(deliver) \subseteq deliver[pp]$ are replaced by [$ss{=}sequencer \wedge (sequencer \mapsto mm) \notin deliver$]. Similarly a guard $ss \neq sequencer$ is added in the specifications of *TODeliver* event. Thus on the occurrence of *TODeliver* event a message is delivered to the sites other than the sequencer.

L3 This is a very simple refinement giving a more concrete specification of the *Order* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer. Recall that a total order in the abstract specifications are constructed as below which state that all messages delivered at any process are ordered before the new message *mm*.

$$order := order \cup (ran(deliver) \times \{mm\})$$

In the refined specifications of *Order* event the *total order* is constructed as below stating that all messages delivered to the sequencer are ordered before the new message *mm*.

$$order := order \cup (deliver[\{sequencer\}] \times \{mm\})$$

L4 In this refinement we introduce the notion of *computation* messages. The computation message are those message which need to be delivered following a total order. Global sequence numbers of the computation message are generated by the sequencer. The delivery of the messages is done based on the sequence numbers. In this intermediate refinement step, the sequence number of computation messages are assigned by the sequencer. This refinement introduces the following new variables.

$$\begin{aligned}&computation \subseteq MESSAGE\\&seqno \in computation \rightarrow\!\!\!\!\!\!+\; Natural\\&counter \in Natural\end{aligned}$$

The variable *seqno* is used to assign sequence numbers to the computation messages. The *counter* is updated by one each time a *computation* message is assigned a sequence number.

Table 1. Proof Statistics

Model	Total POs	POs by Interaction	Percent. automatic
Total Order Broadcast	106	27	74
Causal Order Broadcast	80	26	67
Total Causal Order Broadcast	163	67	58

L5 In this refinement we introduce the notion of *control* messages. We also introduce the relationship of each *computation* message with the *control* messages. This refinement consists of following new state variables typed as follows,

$$control \subseteq MESSAGE$$
$$messcontrol \in control \rightarrowtail\mkern-14mu\rightarrow computation$$

The variables *control* and *computation* are used to cast a message as either a computation or a control message. The set *control* contains the control messages sent by the sequencer. The variable *messcontrol* is a partial injective function which defines relationship among a control message and its computation message. It implies that there can only be a one control message for each computation message and vice-versa. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer.

L6 A new event *Receive Control* is introduced. We illustrate that a process other than sequencer can deliver a *computation* message only if it has received *control* message for it. It can be noticed that the message delivery to the sites other than the sequencer is done using the sequence number generated by the sequencer.

5 Conclusions

In this paper we have outlined our refinement based approach for formal development of a fault tolerant models of replicated database system. We have presented the abstract B models of total order broadcast and outlined how an abstract total order can be refined by the concrete sequence numbers in the refinement steps. The abortion of the conflicting update transactions originated at different sites may be avoided if the updates are delivered to the participating sites in a total order. However, the total order broadcast does not preserve the causality among the transactions. In [19], we used a refinement approach with Event B for formal development of broadcast system which deliver messages satisfying various ordering properties. We have developed the separate models of total order broadcast, causal order broadcast and total causal order broadcast. The work was carried out on the Click'n'Prove B tool. The tool generates the proof obligations for refinement and consistency checking. These proofs helped us to understand the complexity of the problem and the correctness of the solutions. They also helped us to discover new system invariants which provide a clear insight to the system. The overall proof statistics for various developments including a total order broadcast is given in the Table 1. Our experience with these case studies strengthens our belief that abstraction and refinement are valuable technique for modelling complex distributed system.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

2. Jean-Raymond Abrial and Dominique Cansell. Click'n' Prove: Interactive proofs within set theory. In *TPHOLs*, pages 1–24, 2003.
3. Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 496–503. Springer, 1997.
4. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
5. Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
6. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
7. Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.
8. V. Hadzilacos and S.Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94 -1425, Cornell University,NY, 1994.
9. Linas Laibinis, Elena Troubitsyna, Sari Leppänen, Johan Lilius, and Qaisar A. Malik. Formal service-oriented development of fault tolerant communicating systems. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 261–287, volume 4157 of Lecture Notes in Computer Science, Springer,2006.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
11. Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. 1990.
12. P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):17–25, 1990.
13. C Metayer, J R Abrial, and L Voison. Event-B language. RODIN deliverables 3.2, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, 2005.
14. Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
15. M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition.* Prentice-Hall, 1999.
16. Andre Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communication of the ACM*, 39(4):84–87, 1996.
17. C. Toinard, Gerard Florin, and C. Carrez. A formal method to prove ordering properties of multicast systems. *ACM Operating Systems Review*, 33(4):75–89, 1999.
18. Divakar Yadav and Michael Butler. Application of Event B to global causal ordering for fault tolerant transactions. In *Proc. of Workshop on Rigirous Engineering of Fault Tolerant System,REFT05*, pages 93–103,Newcastle upon Tyne, 19 July 2005 , http://eprints.ecs.soton.ac.uk/10981/.
19. Divakar Yadav and Michael Butler. Formal specifications and verification of message ordering properties in a broadcast system using Event B. In *Technical Report,School of Electronics and Computer Science, University of Southampton, Southampton,UK*, May 2007, http://eprints.ecs.soton.ac.uk/14001/.
20. Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database systems using Event B. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 343–363, volume 4157 of Lecture Notes in Computer Science, Springer,2006.

Reasoning about System-Degradation and Fault-Recovery with Deontic Logic

Pablo F.Castro and T.S.E.Maibaum

McMaster University
Department of Computing & Software
Hamilton, Canada
castropf@mcmaster.ca
tom@maibaum.org

Abstract. In this paper we outline the main characteristics of a deontic logic which is useful for modeling of and reasoning about fault-tolerance and related concepts. Towards this goal, we describe a temporal extension of this formalism together with some of its properties. We use a simple example to show how some fault-tolerance concepts (like fault-recovery and system degradation) can be expressed using deontic constructs.

Key words: Fault-Tolerance, Formal Specification, Deontic Logics, Software Design

1 Introduction

Fault-tolerance has emerged as an important research field in recent years; the increasing complexity of software code in current applications has implied that techniques such as program verification (e.g., Hoare logic) are very expensive to be used in practice, in part because the total elimination of errors is a hard task in large programs. This implies that designers have to find other techniques to develop critical software. To produce fault-tolerant programs (software which is able to recover from errors) is an interesting option. Although the main techniques for fault-tolerance have been proposed for the implementation phase, in the past few years some techniques and formalisms have been proposed for the design phase. We intend to take some steps towards this goal; in this paper we introduce a propositional deontic logic (a detailed introduction is given in [1]) to specify and to reason about fault-tolerance at the design level. We give some examples of application to show how concepts like fault-recovery and system-violation can be formalized using our logic.

Although deontic logics (or DL for short) were created to formalize moral and ethical reasoning, they have been proposed as a suitable formalism for dealing with fault-tolerance by several authors (for example: [2], [3] and [4]). This logic has the main benefit of allowing us to distinguish between qualitative different scenarios: normative (following the rules, expected, normal) and non-normative (violating the rules, unexpected, abnormal) situations. However, it is hard to find a suitable version of DL to apply in practice. Some formalisms have been described to be used in computer science (for example:[5] or [6]), but most of them are not designed to be used in the context of fault-tolerance. In addition, the notion of time has been useful for reasoning about program properties, so we need to mix both deontic notions and temporal frameworks; as a result the logic obtained is very expressive, allowing us to express several properties, like those related to fault recovery.

The paper is organized as follows. In section 2 we present a brief description of our deontic logic. In section 3 we describe a practical example. Finally, we describe some conclusions and future works.

2 A Temporal Deontic Logic

The logic presented in this section takes some features from the *dynamic deontic logic* described by Meyer in [5], and the *modal action logic* proposed by Maibaum and Khosla in [7]. In the language, we have a set of atomic (or primitive) actions:

$$\Delta_0 = \{\alpha, \beta, \gamma, ...\}$$

and a set of atomic propositions:

$$\Phi_0 = \{\varphi, \psi, \vartheta, ...\}$$

More complex actions can be constructed from the atomic ones using the following operators: $\sqcup, \sqcap, -$, that is: nondeterministic choice, concurrent execution and action complement. In addition, we consider two special actions: $\emptyset$ and $\mathbf{U}$. The former is an impossible action (an action that cannot be executed), while the latter is universal choice (the non-deterministic choice between the enabled actions). The complement operator is particularly useful to specify wrong behavior, for example, to say that if a given system is obliged to perform an action and it executes another action (this action is in the complement), then we have an error. We must be very careful with the complement because it can bring undecidability in the logic (for example, if we combine it with the iteration operator, see [8]).

The intuition behind each construct in the logic is as follows:

- $\alpha =_{act} \beta$: *actions α and β are equal.*
- $[\alpha]\varphi$: *after all the possible executions of α, φ is true.*
- $[\alpha \sqcup \beta]\varphi$: *after the non-deterministic execution of α or β, φ is true.*
- $[\alpha \sqcap \beta]\varphi$: *after the parallel execution of α and β, φ is true.*
- $[\mathbf{U}]\varphi$: *after the non-deterministic choice of any possible action, φ is true.*
- $[\emptyset]\varphi$: *after executing an impossible action, φ becomes true.*
- $[\overline{\alpha}]\varphi$: *after executing an action other than α, φ is true.*
- $\mathsf{P}(\alpha)$: *every way of executing α is allowed.*
- $\mathsf{P_w}(\alpha)$: *some way of executing α is allowed.*

The deontic part of the logic is given by the permission predicates. Note that we consider two different versions of permission. Both are useful, in particular since we can define an obligation operator using them:

$$\mathsf{O}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow} \mathsf{P}(\alpha) \wedge \mathsf{P_w}(\overline{\alpha})$$

That is, an action is obliged if it is permitted and the remaining actions are not weakly permitted (you are not allowed to execute them in any context).

This definition of obligation allows us to avoid several deontic paradoxes (some of the most well-known paradoxes in deontic logic can be found in [9]). For example, the so-called Ross's paradox: *if we are obliged to send a letter then we are obliged to send it or burn it.* This is a paradox in the sense that we do not expect this sentence to be valid in snatural language. The formalization of this paradox is as follows: $\mathsf{O}(send) \rightarrow \mathsf{O}(send \sqcup burn)$. The reader can verify that this formula is not valid in our framework.

The semantics of our logic is defined by a labelled transition system, $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$, where:

- $\mathcal{W}$ is a (non empty) set of worlds.
- $\mathcal{E}$ is set of events (the set of events that occurs during system execution).
- $\mathcal{R}$ is a $\mathcal{E}$-labelled relation between worlds.
- $\mathcal{I}$ is an interpretation which tells us which predicates are true in which world; in addition, it maps an action to a set of events (the events that this action produces during its execution).
- the relation $\mathcal{P} \subseteq \mathcal{W} \times \mathcal{E}$ tells us which event is allowed in a given world.

The relation $\vDash$ can be defined in a standard way; we have two novel rules for the two versions of permission:

- $w, M \vDash p \overset{\mathsf{def}}{\Longleftrightarrow} w \in \mathcal{I}(p)$
- $w, M \vDash \alpha =_{act} \beta \overset{\mathsf{def}}{\Longleftrightarrow} \mathcal{I}(\alpha) = \mathcal{I}(\beta)$
- $w, M \vDash \neg\varphi \overset{\mathsf{def}}{\Longleftrightarrow}$ not $w \vDash \varphi$.
- $w, M \vDash \varphi \rightarrow \psi \overset{\mathsf{def}}{\Longleftrightarrow} w \vDash \neg\varphi$ or $w \vDash \psi$ or both.
- $w, M \vDash \langle\alpha\rangle\phi \overset{\mathsf{def}}{\Longleftrightarrow}$ there exists some $w' \in \mathcal{W}$ and $e \in \mathcal{I}(\alpha)$ such that $w \xrightarrow{e} w'$ and $w', M \vDash \phi$.
- $w, M \vDash \mathsf{P}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow}$ for all $e \in \mathcal{I}(\alpha)$, $\mathcal{P}(w, e)$ holds.
- $w, M \vDash \mathsf{P_w}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow}$ there exists some $e \in \mathcal{I}(\alpha)$ such that $\mathcal{P}(w, e)$.

They are a formalization of the intuition explained above. Note that using modalities we can define the composition of actions, that is:

$$[\alpha; \beta]\varphi \overset{\mathsf{def}}{\Longleftrightarrow} [\alpha]([\beta]\varphi)$$

However, we introduce it only as notation. (We can introduce this operator in the language but it complicates in several ways the semantics, in particular the semantics of the action complement.)

We describe in detail the semantics and axioms of the logics in [1]. We can explain intuitively the deontic operators with some diagrams. Consider the following model M:

$$\begin{tikzcd}
 & \bullet w_1 \\
w\bullet \arrow[ur, "e_1"] \arrow[r, "e_2"] \arrow[dr, dotted, "e_3"] & \bullet w_2 \\
 & \bullet w_3
\end{tikzcd}$$

The dotted arrow means that this transition is not allowed to be performed. e_1, e_2 and e_3 represent possible events during the execution of the system; we can suppose that they are generated by two actions: α and β. Suppose that α produces (during its execution) events e_1 and e_2, and action β produces event e_3. Here we have $w, M \vDash \mathsf{P}(\alpha)$, because every way of executing it is allowed, and also we have $w, M \vDash \mathsf{P}_\mathsf{w}(\alpha)$, because α is allowed to be executed in a least one way. On the other hand, we have $w, M \vDash \neg\mathsf{P}(\beta)$ and also $w, M \vDash \neg\mathsf{P}_\mathsf{w}(\beta)$. Finally, since $w, M \vDash \mathsf{P}(\alpha)$ and $w, M \vDash \neg\mathsf{P}_\mathsf{w}(\overline{\alpha})$ we obtain $w, M \vDash \mathsf{O}(\alpha)$.

Some interesing properties of the modal and deontic operators are the following:

P1. $[\alpha \sqcup \alpha']\varphi \leftrightarrow [\alpha]\varphi \wedge [\alpha']\varphi$
P2. $[\alpha]\varphi \rightarrow [\alpha \sqcap \alpha']\varphi$
P3. $\mathsf{P}(\emptyset)$
P4. $\mathsf{P}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}(\alpha) \wedge \mathsf{P}(\beta)$
P5. $\mathsf{P}(\alpha) \vee \mathsf{P}(\beta) \rightarrow \mathsf{P}(\alpha \sqcap \beta)$
P6. $\neg\mathsf{P}_\mathsf{w}(\emptyset)$
P7. $\mathsf{P}_\mathsf{w}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}_\mathsf{w}(\alpha) \vee \mathsf{P}_\mathsf{w}(\beta)$
P8. $\mathsf{P}_\mathsf{w}(\alpha \sqcap \beta) \leftrightarrow \mathsf{P}_\mathsf{w}(\alpha) \wedge \mathsf{P}_\mathsf{w}(\beta)$

P1 says that *if after executing α or β φ is true, the φ is true after executing α and after executing β*. P2 says that parallel composition preserves properties. P3 says that the impossible action is allowed. P4 and P5 are similar to P1 and P2 but for strong permission. P6, P7 and P8 are the dual properties for the weak permission. In particular, P6 says that the impossible action is not weakly permitted, i.e., there is no (allowed) way of executing it. It is in this sense that $\emptyset$ is the impossible action. This explains the seemingly paradoxical nature of P3: every way of executing the impossible action is allowed (but there is no way!).

Note that we do not have, as in dynamic logic, the iteration as a valid operation over actions. Even though it is desirable, it will bring us undecidability. Instead, we prefer to enrich our logic with temporal operators in a branching time style (precisely, similar to CTL [10]). We consider the following temporal formulae:

– $\mathsf{AN}\varphi$ (*in all possible executions φ is true at the next moment*).

- $\mathsf{AG}\varphi$ (*in all executions φ is always true*),
- $\mathsf{A}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ (*for every possible execution φ_1 is true until φ_2 becomes true*)
- $\mathsf{E}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ (*there exists some execution where φ_1 is true until φ_2 becomes true*).

As usual, using these operators we can define their dual versions. It is interesting to note that iteration and the temporal operators are related; with iteration we can define: $[\mathbf{U}^*]\varphi \overset{\mathsf{def}}{=} \mathsf{AG}\varphi$. But the temporal formulae do not make the logic undecidable because the temporal operators cannot be mixed with the modal ones.

In addition, we consider the operator $\mathsf{Done}(\alpha)$, which means *the last action executed was α*. Using it, together with the temporal operators, we can reason about the traces of our models. Some useful formulae can be expressed using the $\mathsf{Done}()$ operator; some examples are:

- $\mathsf{ANDone}(\alpha)$, *the next action to be executed will be α.*
- $\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\beta)$, *the execution of α implies the execution of β*
- $\mathsf{Done}(\alpha) \rightarrow \mathsf{O}(\beta)$, *if you performed α then you are obliged to perform β.*
- $\mathsf{A}(\mathsf{Done}(\alpha_1 \sqcup ... \sqcup \alpha_n) \,\mathcal{U}\, \mathsf{Done}(\beta))$, *on every path you perform some α_i until you perform β.*

Some of these formulae are important to express error-recovery, as we illustrate in the next section.

The $\mathsf{Done}(-)$ operator has some interesting properties:

Done1. $\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta)$
Done2. $\mathsf{Done}(\alpha \sqcap \beta) \leftrightarrow \mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta)$
Done3. $\mathsf{Done}(\alpha \sqcup \beta) \wedge \mathsf{Done}(\overline{\alpha}) \rightarrow \mathsf{Done}(\beta)$
Done4. $[\alpha]\varphi \wedge [\beta]\mathsf{Done}(\alpha) \rightarrow [\beta]\varphi$

Property $Done1$ says that if a choice between two actions was executed then one of them was executed. $Done_2$ means that if we execute the parallel composition of two actions then we have to perform both actions. $Done3$ allows us to discover which action of a choice was executed. And the last property is a kind of subsumption property: *if after executing α φ is true, and after of β α is done, then after β also φ is true.*

The semantics of the temporal operators can be defined using traces (as usual). Suppose that $\pi = s_0 \overset{e_0}{\rightarrow} s_1 \overset{e_1}{\rightarrow} ...$ is an infinite trace on a given model (note that we can extend the finite traces to infinite ones, as is usually done in temporal logics). We says that $\pi' \preceq \pi$ if π' is an initial segment of π. Then we define the formal semantics of the temporal operators as follows:

- $\pi, i, M \vDash \mathsf{Done}(\alpha) \overset{\mathsf{def}}{\iff} i > 0$ and $e_{i-1} \in \mathcal{I}(\alpha)$.
- $\pi, i, M \vDash \mathsf{AN}\varphi \overset{\mathsf{def}}{\iff} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\pi', i+1, M \vDash \varphi$.
- $\pi, i, M \vDash \mathsf{AG}\varphi \overset{\mathsf{def}}{\iff} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\forall j \geq i : \pi', j, M \vDash \varphi$.

- $\pi, i, M \models \mathsf{A}(\varphi_1 \ \mathcal{U} \ \varphi_2) \overset{\mathsf{def}}{\Longleftrightarrow} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\exists j \geq i$: $\pi', j, M \models \varphi_2$ and $\forall i \leq k < j : \pi', k, M \models \varphi_1$.
- $\pi, i, M \models \mathsf{E}(\varphi_1 \ \mathcal{U} \ \varphi_2) \overset{\mathsf{def}}{\Longleftrightarrow} \exists \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\exists j \geq i$: $\pi', j, M \models \varphi_2$ and $\forall i \leq k < j : \pi', k, M \models \varphi_1$.

Note that the relation $\models$ is now defined with respect to a sequence, an instant and a model.

3 A Practical Example

We will use a small example to illustrate why the deontic operators are useful to model fault-tolerance:

In a factory which produces some kind of object, the process of making an object is as follows: we have two mechanical hands (A and B), one press and one drill; the hand A puts an element in the press and the hand B takes the pressed element and puts it in the drill. If the hand A fails and does not put some element in the press then the hand B should put the element in the press and then it should continue doing its work. And vice-versa (if hand B fails). If both hands fail, an alarm sounds and the system is shut down.

The interesting point in the example is how a violation (when a mechanical hand fails) can be overcome using the other hand (taking advantage of the redundancy in the system); of course, using only one hand for whole the process implies a slower process of production, and therefore the entire process is more expensive. Note that here we have an important difference between prescription and description of behavior: *the hand A should put an element in the press.* We need to model this as a prescription of behavior; that is, *what the system is obliged to do in a given situation.* One of the main advantages of deontic logic is that it allows us to distinguish between the description and prescription of a system (as established in [7]). For example, if we proceed to say that the hand A puts an element in the press (in a descriptional way):

$$\neg el2press \to \mathsf{ANDone}(A.putselpress)$$

which means that if there is no element in the press then the hand A puts one in it (note that $\mathsf{ANDone}(\alpha)$ could be thought of as a *do* operator). On the other hand, the deontic version:

$$\neg el2press \to \mathsf{O}(A.putselpresser)$$

says that, if there is no element in the press, then the hand A *should* put one in the press. The difference is that, in the second case, the obligation could be violated.

Having these facts in mind, we can give a part of the specification:

A1 $\neg\mathsf{Done}(\mathbf{U}) \to \neg el2press \land \neg el2drill \land \neg v_1 \land \neg v_2$
A2 $(\neg el2press \to \mathsf{O}(A.putselpress)) \land (v_2 \land elpressed \to \mathsf{O}(A.putseldrill))$

A3 $(\neg el2drill \wedge elpressed \rightarrow \mathsf{O}(B.putseldrill)) \wedge (v_1 \rightarrow \mathsf{O}(B.putselpress))$
A4 $\neg v_1 \wedge \mathsf{O}(A.putselpress \sqcup A.putseldrill) \rightarrow [\overline{A.putselpress \sqcup A.putseldrill}]v_1$
A5 $\neg v_2 \wedge \mathsf{O}(B.putselpress \sqcup B.putseldrill) \rightarrow [\overline{B.putselpress \sqcup B.putseldrill}]v_2$
A6 $\neg v_1 \rightarrow [A.putselpress \sqcup A.putseldrill]\neg v_1$
A7 $\neg v_2 \rightarrow [B.putselpress \sqcup B.putseldrill]\neg v_2$
A8 $(v_1 \rightarrow [\overline{A.fix}]v_1) \wedge (v_1 \rightarrow [A.fix]\neg v_1)$
A9 $(v_2 \rightarrow [\overline{B.fix}]v_1) \wedge (v_2 \rightarrow [B.fix]\neg v_2)$
A10 $v_1 \wedge v_2 \rightarrow \mathsf{ANDone}(alarm)$
A11 $[alarm]\mathsf{AF}(\mathsf{Done}(A.fix \sqcap B.fix))$

Some explanation will be useful about the axioms. We have only shown the deontic axioms; some other axioms should be added (for example frame axioms). Axiom **A1** establishes the initial condition in the system: *at the beginning (when no action has occurred) there is no element to press, and no element to drill, and no violations.* **A2** says that if *there is no element in the press then the hand A should put an element there*; in addition it says that if *the hand B is not working, then A has to put pressed elements in the drill.* **A3** says that *if there is no element to drill and there exists a pressed element, then the hand B should put that element in the drill.* Axiom **A4** expresses when a violation of type v_1 is committed: *if there is no violation v_1 and hand A is obliged to put an element in the press but the hand does not do it, then v_1 becomes true.* **A5** is the specification of violation v_2: *it happens when the hand B does not fulfill its obligation.* **A6** and **A7** model when normal states are preserved. **A8** and **A9** express when we can recover from violation, that is, when some hand is repaired. Finally, **A10** and **A11** tell us when the worst situation is achieved, that is, when both hands are in violation; then an alarm is initiated and the hands are repaired.

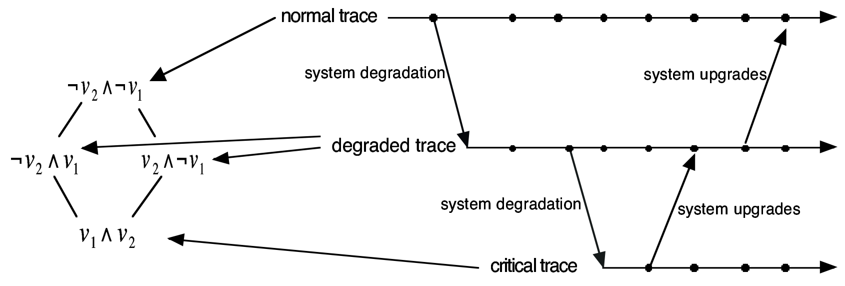


Fig. 1. ordering violations

It is interesting to analyze the different faults that we can have in the system; we can see that there exists an order relation between them. The situation is shown in figure 1. The ideal scenario is when $\neg v_1 \wedge \neg v_2$ is true, that is, when no hand is faulty. From here, the system can suffer a degradation and then it goes to violation 1 (v_1 is true) or violation 2 (v_2 is true); both situations of violation are incomparable, in the sense that none of them implies the other. Then, the system can be degraded again and both violations hold; in this case, both hands are not working correctly and there is no other option than repairing both hands. Otherwise, the entire process of production will be effected. It is important to note that, though in violation 1 (or violation 2) the system can work correctly (because of the redundancy of the hands), the process of production is slower (only one hand will do all the work).

Note that the hand A can try to put an element in the drill. Indeed, if hand B is working correctly, this scenario is obviously not desirable. We can use the *forbidden* operator to avoid this. The forbidden operator can be defined using the weak permission or the strong permission; depending on the choice, we get different results. We define it using the weak permission, as follows:

$$\mathsf{F}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow} \neg \mathsf{P_w}(\alpha)$$

That is, an action is forbidden if it is not allowed to be performed. Using this operator we can include the following formulae:

– $\neg v_1 \rightarrow \mathsf{F}(A.putseldrill)$
– $\neg v_2 \rightarrow \mathsf{F}(B.putselpress)$

Using these new formulae we can define new violations in the case that prohibitions are violated, and therefore, the corresponding recovery actions can be introduced.

Some properties can be proved from the specification. In particular, some interesting properties to prove are:

$$\mathsf{AG}(\neg v_1 \wedge \neg v_2) \wedge \neg el2drill \wedge \mathsf{AF}el2press \rightarrow \mathsf{AF}el2drill$$

if there is no violation, and eventually we have an element to press, then we will have an element to drill.

$$\mathsf{AG}(v_1 \wedge \neg v_2) \wedge \neg el2drill \wedge \mathsf{AF}el2press \rightarrow \mathsf{AF}el2drill$$

if there is a violation of type v_1 (but no violation of type v_2), then the pressed elements will be brought to the drill, that is, the system continues working, in a degraded way.

$$\mathsf{AG}(v_1 \wedge v_2) \rightarrow \mathsf{AF}(\mathsf{EG}(\neg el2drill \wedge \neg el2press))$$

if both hands are not working correctly, then there exists the possibility that the elements will not be transported to the press or to the drill.

Of course, a lot of interesting different properties can be proposed, and proven. We have described another example of application in [1]. The point to make here is the way in which system violation and fault recovery are specified; we can mix modal and deontic operators to specify these system properties. And the expressiveness that temporal operators give us allow us to prove important properties about the specification.

We note that the logic described is decidable and, therefore, techniques such as model checking could be used to validate specifications and to prove properties of corresponding programs.

4 Conclusions

We have shown, using an example, how deontic logics can be used to express some properties about fault-tolerant systems. Though the example is simple, it illustrates a non-trivial complex scenario of failure and recovery, demonstrating that these ideas themselves are non trivial. For this purpose we have developed our own version of deontic logic, which has some interesting properties (like compactness and decidability).

As we demonstrate in the example, it is possible to formalize the notion of violation and normal state, which give us the possibility of analyzing how the system is degraded and repaired through time, and therefore some interesting properties can be proved. The utilization of deontic operators allows us to differentiate between model description and prescription in a natural way. We have presented another (more complex) example in [1], and we proved several properties about it; from these examples, it seems possible to conclude that we can apply the underlying logic calculus in practice.

However, we need to do research about practical decision methods for the proposed logic. Our final goal is to provide automatic tools which allow designers to analyze models (and programs) in a practical way. Towards this goal, it is also interesting to research how we can modularize the deontic specifications, in such a way that different components have different deontic contracts (obligations and permissions) and then the system specification could be derived from the individual ones.

Another interesting branch of research seems to be *contrary to duty reasoning* (see [11]), in particular how this kind of reasoning is applied in fault-tolerance. Contrary to duty structures are sets of sentences where there exists a primary obligation and a secondary obligation which arises from the violation of the primary one. These kinds of formulae are hard to reason about, as is shown everywhere in the deontic literature; indeed, several paradoxes are contrary to duty structures. This kind of reasoning is usual in fault-tolerance; for example, if we violate some system obligation, then we are obliged to perform some corrective action. At first sight, it seems possible to extend our formalism (e.g., with several versions of obligation) to support contrary to duty reasoning. We leave this as a further work.

References

1. P.F.Castro, T.S.E.Maibaum: Towards a deontic logic for fault tolerance. Technical Report SQRL39, McMaster, Department of Computing & Software, Software Quality Research Laboratory (2007)

2. T.S.E.Maibaum: Temporal reasoning over deontic specifications. In: Deontic Logic in Computer Science, John & Wiley Sons (1993)
3. J.Magee, T.S.E.Maibaum: Towards specification, modelling and analysis of fault tolerance in self managed systems. In: Proceeding of the 2006 international workshop on self-adaptation and self-managing systems. (2006)
4. S.Kent, T.S.E.Maibaum, W.Quirk: Formally specifying temporal constraints and error recovery. In: Proceedings of IEEE International Symposium on Requirements Engineering. (1993) 208–215
5. J.J.Meyer: A different approach to deontic logic: Deontic logic viewed as variant of dynamic logic. In: Notre Dame Journal of Formal Logic. Volume 29. (1988)
6. F.Dignum, R.Kuiper: Combining dynamic deontic logic and temporal logic for the specification of deadlines. In: Proceedings of the thirtieth HICSS. (1997)
7. S.Khosla, T.S.E.Maibaum: The prescription and description of state-based systems. In B.Banieqnal, H., A.Pnueli, eds.: Temporal Logic in Computation, Springer-Verlag (1985)
8. J.Broersen: Modal Action Logics for Reasoning about Reactive Systems. PhD thesis, Vrije University (2003)
9. J.J.Meyer, R.J.Wieringa, F.P.M.Dignum: The paradoxes of deontic logic revisited: A computer science perspective. Technical Report UU-CS-1994-38, Utrecht University (1994)
10. E.A.Emerson, J.Y.Halpern: Decision procedures and expressiveness in the temporal logic of branching time. In: 14th Annual Symposiun on Theory of Computing (STOC). (1982)
11. M.Sergot, H.Prakken: Contrary-to-duty obligations. In: DEON 94 (Proc.Second International Workshop on Deontic Logic in Computer Science). (1994)

Engineering Fault-tolerance Requirements using Deviations and the FIDJI Methodology

Andrey Berlizev[1,2], Nicolas Guelfi[1]

[1] Laboratory for Advanced Software Systems, University of Luxembourg
6, rue Richard Coudenhove-Kalergi L-1359 Luxembourg-Kirchberg, Luxembourg
[2] Software Modeling and Verification group, Centre Universitaire D'Informatique
Université de Genève, Route de Drize, 7 CH-1227 Carouge, Switzerland
{Andrey.Berlizev, Nicolas.Guelfi}@uni.lu

Abstract. Contemporary software systems are usually complex and prone to errors due to their complexity. If dependability attributes are defined, some strategies must be adopted. One approach is to follow a component based approach allowing reuse of dependable components. The FIDJI methodology is a semi-formal approach that allows reuse of existing analysis, design or implementation elements in a product line perspective. FIDJI elements can be reused at any abstraction level by derivation. This paper presents how we can modify the FIDJI analysis models to adapt them to the semi-formal specification of fault-tolerance requirements. For this purpose, the main contribution of this article is to introduce the concept of deviation from requirements and to propose the specification of recovery requirements associated to deviations for the use case description of the FIDJI methodology. The proposed approach is illustrated using a running example.

Keywords: integrated approaches, fault-tolerant systems development, deviations, Semi-formal methodology, MDE.

1 Introduction

Contemporary software development is a difficult process because systems have to be built with high complexity (large size, distribution, cross platform, etc.). Because of this, systems are prone to developer mistakes and highly dependant on the hardware they use. In safety-critical applications such as medical systems, the quality requirements demanded by users are further increased and software correctness is of utmost importance. Although there are special development methods for such systems that satisfy these requirements, they are usually very costly and need special skills. Therefore, they are rarely used when developing standard systems that are not highly critical. We believe that simplification of the process for developing dependable systems will help enhance the dependability of developed software.

A widely accepted definition of *dependability* for computing systems was introduced by Jean-Claude Laprie in 1985 as, “the trustworthiness of the system by which reliance can be justifiably placed on the services the system delivers” [5].

Developer mistakes or hardware defects are known as *faults*. If they are executed and *manifest*, faults may lead to an *error*: an improper internal state of the system. If an error is not *recovered* to normal state and reaches the border of the system, it is a *failure* of the system for the environment (behavior different from the specified). *Fault-tolerance* (FT) is the ability to comply specification even in the presence of faults. Usually FT is introduced during the design phase, however, we believe that the earlier FT and dependability are introduced during development, the better the outcome [1]. For example, abnormal behavior is introduced during requirement elicitation phase [4] by extending use case based requirements elicitation with ideas from the exception handling world.

FIDJI is a software development methodology, which uses the UML 2.0 notation and textual descriptions and is based on the so called *architectural framework* concept, which is a set of models devoted to the definition of product line assets at the analysis and design level, and which is instantiated into the product line members via model transformation [2, 3]. Product Line (PL) is the development of "product families" [7], which use some functionality of one product to further develop new ones. Model Driven Engineering (MDE) is the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. The most well-known MDE is Model-Driven Architecture by OMG [6]. *Model transformation* is the process of converting one model to another model of the same system. Thus, FIDJI defines how to develop products within a PL by reusing parts of the models and then using model transformations.

There are four layers defined in FIDJI corresponding to different levels of abstraction: SPL requirements, analysis, design and implementation. In this work, we concentrate on the analysis layer. Its purpose is to specify the functionalities offered as well as the concepts manipulated by the architectural framework. FIDJI analysis model is used to describe the architectural framework, as well as the analysis of the product. It consists of the following sub-models:

- Domain Model precisely defines concepts manipulated by use case and operations.
- Use Case Model extends and refines the use cases described at the requirements elicitation time. Its purpose is to express the architectural framework's behavior in terms of sequence of operations.
- Operational Model specifies in detail each operation: informal descriptions, parameters, return values and pre/postconditions.

FIDJI does not consider fault-tolerance as a primary concept. By integrating deviation and recovery specifications coherently with the FIDJI models, this article proposes a solution for improving the FIDJI analysis models in order to cope with fault-tolerance. In order to do this, Section 2 introduces a running example and its requirements; Section 3 presents the notions of deviation and recovery; and Section 4 illustrates these notions for the running example within the FIDJI use case model.

2 Running example

As an academic running example, we will consider a software system that allows several banks to collaborate and borrow money from each other. The task of the system is to find a lender at the request of the borrower and handle the money transfer (which is conducted outside the system itself).

Our illustration is based on the Use Case Model. Due to the issue of space we are omitting Domain Model, a class diagram with all used elements, and Operational Model. Note, that FIDJI use case is not only a textual description in informal language but, whenever possible, it is formalized using OCL 2.0 expressions.

Figure 1 shows the use case for the main scenario used by the borrower. In this scenario, system finds a lender (by internal algorithms) then sends a request to the lender to send the money to the borrower and confirm this. Then, system sends the confirmation to the borrower. GetLender represents the system. This is not a fully formal specification but rather a semi-formal specification, since every step in use case has Object Constraint Language (OCL) postcondition expression. In the next section we will consider abnormal behavior, as well.

Name	Borrow Money
ID	UC1 BorrowMoney
Description	Bank requests from the system to find a lender bank agreeing to work with the requester bank send the money directly to the borrower
Primary actors	Borrower: Bank
Secondary actors	Lender: Bank
Trigger event	Borrowing is requested Post: BorrowMoney^borrowRequest(requestedAmount:Money)
Preconditions	An actor is logged into the system Pre: bankAcc.isUserSignedIn=true
Postconditions	The lender is sent Post: lender^Lend(borrower:Bank; requestedAmount:Money) and Borrower^confirmMoney(Lender: Bank)
Main success scenario: 1. Bank requests the amount Post: BorrowMoney^borrowRequest(requestedAmount:Money) 2. The system finds bank as a lender and sends request lender^Lend(borrower:Bank; requestedAmount:Money) 3. The lender candidate sends confirmation GetLender^confirm(lender:Bank; borrower:Bank) 4. The system confirms with the borrower Post: Borrower^confirmMoney(Lender: Bank)	

Fig. 1. Use case of normal behavior

3 Deviation and Recovery

A specification describes what a system should do: its normal behavior. Fault-tolerance aims at making sure that a system continues to behave as specified, even in the presence of faults. We want to change the specification in such a way that it describes abnormal behavior as well, and simplify sometimes classical FT at the design level, for which some decisions made during the analysis phase about error messages, etc., will be predefined. Fault-tolerance at the analysis level aims at providing a specification not only describing the normal system behavior, but also behavior that is still acceptable by customer behavior. In classical FT, the mechanisms used are implicit and hidden from the customer. It means that for usual specification any deviation from the specified behavior is a failure and is not accepted by customer. In FT specification we can say that some deviations are still acceptable by customer with the proper recovery or, with normal service it may be acceptable to have some degraded service.

To support fault-tolerance at the analysis level we introduce the notion of *Deviation*, which is the expression of the difference between two elements that should be equal. For any deviation defined we also want to define acceptable recovery. Thus, we extend the specification of normal behavior with deviations and recovery accordingly. We use the stereotype <<deviation>> followed by detection statement and some additional stereotypes with their statements description for the full description of fault-tolerance requirements:

- <<recovery>> - this required section describes what should be done to tolerate detected deviation.
- <<impact>> - there is an additional optional clause, which allows to define use cases, classes or other elements which will be impacted by this deviation.
- <<continue>> - this optional section defines what should happen after the recovery.

Within use case we propose to add deviations as additional blocks or lines (Fig. 2), and leave only those parts that were changed (compare with Fig. 1).

4 Running example with fault-tolerance requirements

During FT analysis we should find possible deviations to answer the question of “what can go wrong?” for every step and element of the normal specification. There needs to be a decision of what faults should be considered during the development of the system, which depends on the system type and the desired dependability (quality) level. Not every found deviation should be considered since we cannot offer a solution at the analysis level. For instance, we cannot consider communication is lost with the user, because we do not have definition of communication at the analysis level and it will appear only during the design. At the analysis level we can abstract from the details of the fault and simply consider the omission of the message from the actor. Typically this can be detected with a timeout, which is illustrated by the deviation in Fig. 2 at Step 3, on the right. On the left, we define that an error message should be sent and that the use case is finished when sentAmount is not equal to requiredAmount. In a real system it would be better to compare with the received amount, but in this case it would require new steps to be added to the use case.

For trigger event we add reaction on the situation when the borrower does not use proper type for request parameter (such a situation may happen in web services, for example, when all parameters are passed through textual XML file, and may therefore be of wrong type). For deviation we have informal error detection.

Trigger event	Borrowing is requested Post: BorrowMoney^borrowRequest(requestedAmount:Money) <<deviation>> amount is not type of Money <<recovery>> Borrower^WrongAmountType
Preconditions	An actor is logged into the system Pre: bankAcc.isUserSignedIn=true
Postconditions	The lender is sent Post: lender^Lend(borrower:Bank; requestedAmount:Money) and borrower^confirmMoney(Lender: Bank; requestedAmount:Money) and requestedAmount=sentAmount

…

3. the lender candidate sends confirmation
GetLender^confirm(lender:Bank; borrower:Bank; sentAmount:Money)

<<deviation>> not requestedAmount=sentAmount <<recovery>> Borrower^lessAmountSent(Amount:Money) <<continue>> UC finishes	<<deviation>> GetLender^confirmTimeout <<recovery>> find other lender <<continue>> from 2

…

Fig. 2. Use case with abnormal behavior

5 Perspectives & Conclusion

Future work on this topic is planned in several directions. First of all, all the models of FIDJI should be enhanced with our proposal given for use case description. Secondly, deviations and recovery deal with only a part of dependable requirements and should include other aspects of dependability like, availability, security, etc. Thirdly, testing is costly in the development process and should be simplified by reusing tests for common sets in PLs and through automatic test generation from the analysis model. We would like to study the links between fault-tolerant requirement specification and test cases.

Acknowledgments. We would like to thank Alexander Romanovsky and Jörg Kienzle for their helpful discussions. This research is partially supported by the Ministry of Culture, Higher Education and Research of Luxembourg by grant BFR04/053.

References

1. Beder D.M., Randell B., Romanovsky A., Rubira C.M.F.: On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. ISORC - 2001 Proceedings. Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. IEEE Computer Society Press, (2001) 103-112
2. Guelfi, N., Perrouin G.: Using Model Transformation and Architectural Frameworks to Support the Software Development Process: the FIDJI Approach. In 2004 Midwest Software Engineering Conference (2004) 13–22
3. Guelfi, N., Perrouin, G.; A Flexible Requirements Analysis Approach for Software Product Lines. International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'07), June 11-12th, Trondheim, Norway, LNCS 4542, Springer-Verlag (2007)
4. Shui, A., Mustafiz, S., Kienzle, J.: Exception-Aware Requirement Elicitation with Use Cases. In Romanovsky, A., Dony, C., Knudsen, J. L., and Tripathi, A., editors, *Advanced Topics in Exception Handling Techniques*. Springer Verlag, October, n. 4119 in Lecture Notes in Computer Science, (2006) 221 - 242.
5. Laprie J. C.: Dependability: A Unifying Concept For Reliable Computing and Fault Tolerance. In Anderson T (Ed) Dependability of Resilient Computers. BSP Professional Books, Oxford (1989)
6. OMG: MDA® Specifications http://www.omg.org/mda/specs.htm
7. Parnas, D.L.: On the Design and Development of Program Families. *TSE*, 2(1):1–9, (1976)

Model-based Testing Using Scenarios and Event-B Refinements

Qaisar A. Malik, Johan Lilius, and Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS), Finland
{Qaisar.Malik, Johan.Lilius, Linas.Laibinis}@abo.fi

Abstract. In this paper, we present a model-based testing approach based on user provided testing scenarios. In this approach, when software model is refined to add/modify features, the test cases are automatically refined to incorporate these changes. We use the Event-B formalism for software models, while user scenarios are represented as Communicating Sequential Process (CSP) expressions.

1 Introduction

Testing is an important activity in the software development life cycle. With advancements in the model-based approaches for software development, new ways have been explored to generate test-cases from existing models of the system. This is often referred to as *model-based testing*. A software model is usually a specification of the system which is developed from the given requirements early in the development cycle [5]. For dependable systems, software model should also include fault tolerance mechanism as part of their functionality. In this paper, we present a model-based testing approach based on user-provided testing scenarios. As our formal framework we use the Event-B method [4, 3] supporting stepwise system development by refinement. Generally, implementation code for a system-under-test (SUT) can be generated from a sufficiently detailed specification. But often, due to the remaining abstraction gap between a model and the implementation, it is not always feasible to generate implementation code. As a result, the implementation is not shown to be correct by construction but instead it is hand-coded by programmer(s). Identifying and writing testing scenarios for such an implementation is a very time consuming and error-prone process. In our approach, test scenarios are identified at an abstract specification level and are automatically refined (together with a specification) at each refinement step. These scenarios can also include tests of the incorporated fault tolerance mechanisms. In our approach, test scenarios are represented as Communicating Sequential Process (CSP) [6] expressions. In the final step, executable test cases are generated from these CSP expressions to be tested on SUT. This work is based on our earlier approach [10] for scenario-based testing from B models.

The organisation of the paper is as follows. Section 2 discusses stepwise development using the Event-B formalism. Section 3 describes our approach for model-based testing as well as addresses the topics on refinement and representation of test scenarios. In Section 4, we illustrate our approach by development of a fault-tolerant system. Section 5 contains some concluding remarks.

2 Developing Systems by Refinement using the Event-B Method

This section gives a brief introduction to the Event-B [4, 3] formalism. We also discuss the stepwise development methodology in Event-B focusing on the basic types of system refinement.We will use these basic refinement rules in our model-based testing approach described in the next section.

2.1 Modeling in Event-B

The Event-B [4, 3] is a recent extension of the classical B method [2] formalism. Event-B is particularly well-suited for modeling event-based systems. The common examples of event-based systems are reactive systems, embedded systems, network protocols, web-applications and graphical user interfaces.

In Event-B, the specifications are written in Abstract Machine Notation (AMN). An abstract machine encapsulates state (variables) of the machine and describes operations (events) on the state. A simple abstract machine has following general form

```
MACHINE AM
SETS TYPES
VARIABLES v
INVARIANT I
INITIALISATION INIT
EVENTS
 E_1 = ...
  ...
 E_N = ...
END
```

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLE** clause defines state variables, which are then initialized in the **INITIALISATION** clause. The variables are strongly typed by constraining predicates of the machine invariant I given in the **INVARIANT** clause. The invariant defines essential system properties that should be preserved during system execution. The operations of event based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E = \textbf{WHEN}\ \ g\ \ \textbf{THEN}\ \ S\ \ \textbf{END}$$

$$E = \textbf{ANY}\ \ i\ \ \textbf{WHERE}\ \ C(i)\ \ \textbf{THEN}\ \ S\ \ \textbf{END}$$

where g is a predicate over the state variables v, and the body S is an Event-B statement specifying how the variables v are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where i is the parameter and *C(i)* contains condition(s) over i. The occurrence of the events represents the observable behavior of the system. The event guard (g or *C(i)*) defines the condition under which event is enabled.

2.2 Refinement of Event-Based Systems

The basic idea underlying the formal stepwise development is to design system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit implementable, specification and finishes with generating executable code. In general, refinement process can be seen as a way to reduce non-determinism of the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

We are interested how refinement affects the external behavior of a system under construction. Such external behavior can be represented as a trace of observable events, which then can be used to produce test cases. From this point of view, we can distinguish two different types of refinement called *atomicity* refinement and *superposition* refinement.

In ***Atomicity*** refinement, one event operation is replaced by several operations, describing the system reactions in different circumstances the event occurs. Intuitively, it corresponds to a branching in the control flow of the system. Let us consider an abstract machine AM_A and a refinement machine AM_AR given below. It can be observed that an abstract event E is split (replaced) by the refined events *E1* and *E2*. Any execution of *E1* and *E2* will correspond to some execution of abstract event *E1*. It is also shown graphically in Fig.1(a).

```
                                 REFINEMENT $AM\_AR$
MACHINE $AM\_A$                  REFINES $AM\_A$
 ...                               ...
EVENTS                           EVENTS
 $E$ = WHEN $g$                  $E_1$ ref $E$ = WHEN $g \wedge g_1$ THEN $S_1$ END
        THEN $S$ END             $E_2$ ref $E$ = WHEN $g \wedge g_2$ THEN $S_2$ END
END                              END
```

In ***Superposition*** refinement, new implementation details are introduced into the system in the the form of new events that were invisible in the previous specification. These new events can not affect the variables of the abstract specification and only define computations on newly introduced variables. For our purposes, it is convenient to further distinguish two basic kinds of superposition refinement, where

– a non-looping event is introduced,
– a looping but terminating event is introduced.

Let us consider an abstract machine AM_S and a refinement machine AM_SR as shown below

```
                                 REFINEMENT $AM\_SR$
MACHINE $AM\_S$                  REFINES $AM\_S$
 ...                               ...
EVENTS                           EVENTS
 $E$ = WHEN $g$                   $E$ = WHEN $g$ THEN $S$ END
        THEN $S$ END              $E_1$ = WHEN $g_1$ THEN $S_1$ END
END                              END
```

It can be observed that the refined specification contains both the old and the new events, *E* and *E1* respectively. To ensure termination of the new event(s), the **VARIANT** clause is added in a refinement machine. This **VARIANT** clause contains an expression over a well-founded type (e.g., natural numbers). The new events should decrease the value of the variant, thus guaranteeing that the new events will eventually return the control as the variant expression can not be decreased indefinitely. These two types of refinements are also shown graphically in Fig.1(b) and (c).

Let us note that the presented set of refined types is by no means complete. However, it is sufficient for our approach based on user defined scenarios.

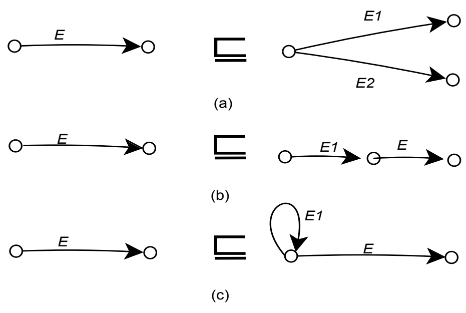


Fig. 1. Basic refinement transformations

3 Our Approach for Model-Based Testing

3.1 Scenario-based approach for testing

In the literature, we can find several definitions of the term *scenario*. Scenarios are generally used to represent system requirements, analysis, user and component interaction,test cases etc [9].

We use the term *scenario* to represent a *test scenario* for our system under test (SUT). A test scenario is one of possible valid execution paths that the system can follow. In other words, it is one of expected functionalities of the system. For example, in a hotel reservation system, booking a room is one functionality, while canceling a pre-booked room is another one. In this article, we use both terms functionality and scenario interchangeably.

Each scenario usually includes more than one system-level procedure/event, which are executed in some particular sequence. In a non-trivial system, identifying such a sequence may not be an easy task. Our testing approach is based on stepwise system development, where an abstract model is first constructed and then further refined to include more details (e.g., functionalities) of the system. On the abstract level, an initial scenario is provided by the user. Afterwards, for each refinement step, scenarios are refined automatically. In Fig.2, an abstract model M_i is refined by M_{i+1} (denoted by $M_i \sqsubseteq M_{i+1}$). Scenario S_i is an abstract scenario, formally satisfiable ($\models$) by specification model M_i, provided by the user. In the next refinement step, scenario S_{i+1} is constructed automatically from M_i, M_{i+1} and S_i in such a way that S_{i+1} formally satisfies model M_{i+1}.

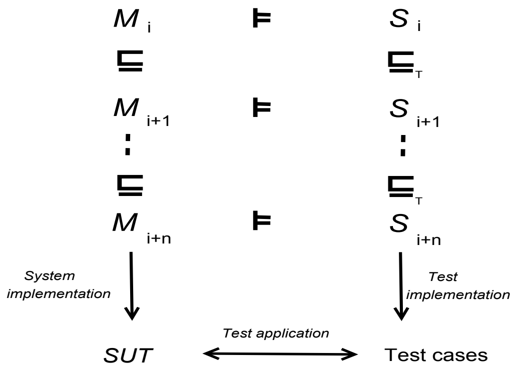


Fig. 2. Overview of our Model-based testing approach

Each scenario can be represented as a Communicating Sequential Process (CSP) [6] expression. Since we develop our system in a controlled way, i.e. using basic refinement transformations described in Section 2.2, we can associate these Event-B refinements with syntactic transformations of the corresponding CSP expressions. Therefore, knowing the way model M_i was refined by M_{i+1}, we can automatically refine scenario S_i into S_{i+1}. To check whether a scenario S_i is a valid scenario of its model M_i, i.e., model M_i satisfies ($\models$) scenario S_i, we use Pro-B model checker [8]. Pro-B supports execution (animation) of Event-B specifications, guided by CSP expressions. The satisfiability check is performed at each refinement level as shown in the Fig.2. The refinement of scenario S_i is the CSP trace-refinement denoted by $\sqsubseteq_T$.

After the final refinement, the system is implemented from the model M_{i+n}. This implementation is called *system under test (SUT)*. The scenario S_{i+n}, expressed as a CSP expression, is unfolded into the executable test cases that are then applied to SUT. In the next sections we will demonstrate how scenarios are represented and refined as CSP expressions.

3.2 Scenario Refinement and Representation

As we have described before, the scenarios are represented as CSP expressions. We refine our models in a controlled way targeting at individual events. We assume that the events are only executed when their guards are enabled. For simplicity, we omit the guard information from CSP expressions. Here we will discuss how individual refinement steps effect the scenarios. Let us assume we are given an abstract specification M_0 with three events, namely, `A`, `B` and `C`, and a scenario S_0 representing the execution order of these events: first the event `A`, then the event `B`, and finally the event `C`. As a regular expression, we can write this sequence as:

$$\texttt{A.B.C}$$

and its corresponding CSP expression is given by

$$\texttt{S}_0 = \texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{C} \rightarrow \texttt{SKIP}$$

In the next refinement step, the model M_0 is refined by M_1. This refinement step may involve any of three types of the supported refinements discussed in Section 2.2. We will discuss them one by one.

Atomicity Refinement. Let us suppose an event `B` is refined using atomicity refinement. As a result, it is split into two events namely $\texttt{B}_1$ and $\texttt{B}_2$. It means that the older event `B` will be replaced by two new events $\texttt{B}_1$ and $\texttt{B}_2$ modelling a branching in the control flow. This can be shown as the regular expression

$$\texttt{A.}(\texttt{B}_1 + \texttt{B}_2)\texttt{.C}$$

As a CSP expression we can represent it as

$$\texttt{S}_1 = \texttt{A} \rightarrow ((\texttt{B}_1 \rightarrow \texttt{C} \rightarrow \texttt{SKIP}) \sqcap (\texttt{B}_2 \rightarrow \texttt{C} \rightarrow \texttt{SKIP}))$$

where $\sqcap$ is an internal choice operator in CSP.

Superposition refinement. Let us suppose we use superposition refinement to refine an event `C`. As a result, a new non-looping event `D` is introduced in the system. The new scenario can be expressed as a regular expression:

$$\texttt{A.B.D.C}$$

and as a CSP expression:

$$\texttt{S}_1 = \texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{D} \rightarrow \texttt{C} \rightarrow \texttt{SKIP}$$

Finally, let us suppose we again use superposition refinement to refine event `C`. However, this time a new looping event `D` is introduced into the system. The new scenario can be represented as a regular expression

$$\texttt{A.B.D}^*\texttt{.C}$$

and its corresponding CSP expression is given as

$$\texttt{S}_1 = \texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{D} \rightarrow \texttt{C} \rightarrow \texttt{SKIP}$$

where `D` is defined as

$$\texttt{D} = \texttt{D} \sqcap \texttt{SKIP}$$

In the next section, we outline how scenarios are unfolded into test cases.

3.3 From Scenarios to Test-cases

Unfolding of scenarios into test cases is a process that is very similar to system simulation. During this process, an Event-B model is initialised and executed, which being guided by the provided scenarios. For our approach, we use Pro-B model checker,which has the functionality to animate B specifications guided by the provided CSP expression. After the execution of each event, present in the scenario, information about the changed system state is stored.

In other words, the execution trace is represented by a sequence of pairs $< e, s >$, where e is an event and s is a post-state (the state after execution of event e). From now on we will refer to a single pair $< e, s >$ as an *ESPair.*

For a finite number of events $e_1, e_2.....e_n$, present both in the model M and the System Under Test (SUT), a test case t of length n consists of an initial state *INIT* and a sequence of *ESPairs*

$$t = INIT, \{< e_1, s_1 >, < e_2, s_2 >, ... < e_n, s_n >\}$$

Similarly, a scenario is formally defined as finite set of related test cases, i.e., scenario $S = \{t_1, t_2, .., t_n\}$ As mentioned earlier, *ESPair* relates an event with its post-state. This information is stored during test-case generation. For SUT these stored post-states become expected outputs of the system and act as a *verdict* for the testing. After execution of each event, the expected output is compared with the output of the SUT. This comparison is done with the help of probing functions. The probing functions are such functions of SUT that at a given point of their invocation, return state of the SUT. For a test-case to pass the test, each output should match the expected output of the respective event. Otherwise, we conclude that a test case has failed. In the same way, test cases from any refinement step can be used to test implementation as long as both the implementation and the respective test cases share the same events and signatures.

4 Testing Development of a Fault-Tolerant System

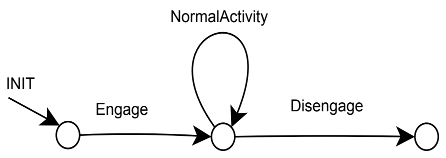
In this section, we show how our testing methodology can be used in the development of a fault-tolerant system . We consider an example of a mobile agent system [7], where an agent performs three basic tasks when connected to the server. These basic tasks are named as *Engage*, *NormalActivity* and *Disengage*. To incorporate the fault-tolerant behavior, the system is repeatedly refined using the basic refinement types described in Section 3.2. The introduction of fault-tolerance increases the complexity of the system. Our testing methodology can be applied to test the new scenarios that result from this complexity. The initial *Event-B* machine named *Cama* specify the three basic events, mentioned above.

```
MACHINE Cama
SETS Agents
VARIABLES agents
INVARIANT agents ⊆ Agents
INITIALISATION agents:= ∅
EVENTS
  Engage = ANY aa WHERE aa ∈ Agents ∧ aa ∉ agents
          THEN agents := agents ∪ {aa} END;
  NormalActivity = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN skip END ;
  Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
END
```

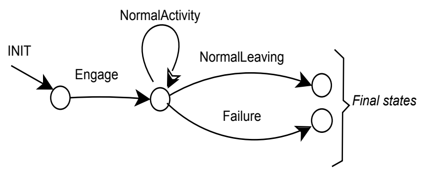
In the specification *Cama*, let us note that the event *NormalActivity* may happen zero or more times. The sequence of events, as determined by the specification, is shown in Fig.3(a).

In the next refinement machine *Cama1*, the event *Disengage* is refined into two new events in order to differentiate between leaving normally or because of a failure. This refinement step is atomicity refinement as discussed in Section 3.2. The other events of the specification remain the same. The execution graph for this refinement is shown in Fig.3(b).

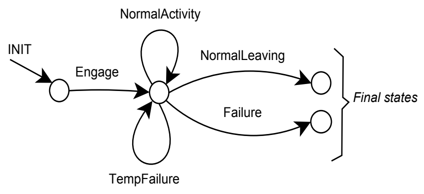
```
REFINEMENT Cama1 REFINES Cama
  ...
EVENTS
  ...
  NormalLeaving ref Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
  Failure ref Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
END
```



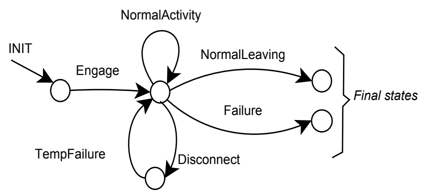
(a) Execution graph of Cama



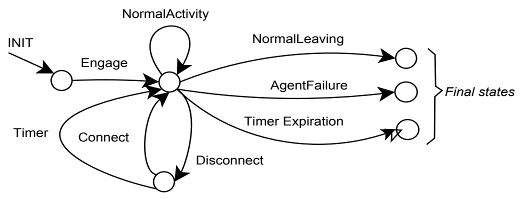
(b) Execution graph of Cama1



(c) Execution graph of Cama2



(d) Execution graph of Cama3



(e) Execution graph of Cama4

Fig. 3. All possible Event execution scenarios across refinements

In the next refinement machine *Cama2*, we introduce temporary loss of connection for our agents. This new event is called *TempFailure*. This refinement step introduces a looping event (see superposition refinement in Section 3.2). To guarantee termination of the new event, we introduce a new variable *disconn_limit*, which is used as a variant.

```
REFINEMENT Cama2 REFINES Cama1
  ...
VARIABLES agents, disconn_limit
INVARIANT disconn_limit ∈ NAT
VARIANT disconn_limit
EVENTS
  ...
  NormalActivity = ANY aa WHERE aa ∈ agents
          THEN disconn_limit := Disconn_limit END;
  TempFailure = ANY aa WHERE (aa ∈ agents)
          THEN disconn_limit := disconn_limit - 1 END;
END
```

The execution flow for *Cama2* is given in Fig.3(c). In next refinement machine *Cama3*, a new event *Disconnect* is introduced. It is the event that precedes (causes) *TempFailure*. This refinement is a superposition refinement introducing a non-looping event. A new variable *timers* is used to ensure order of execution.

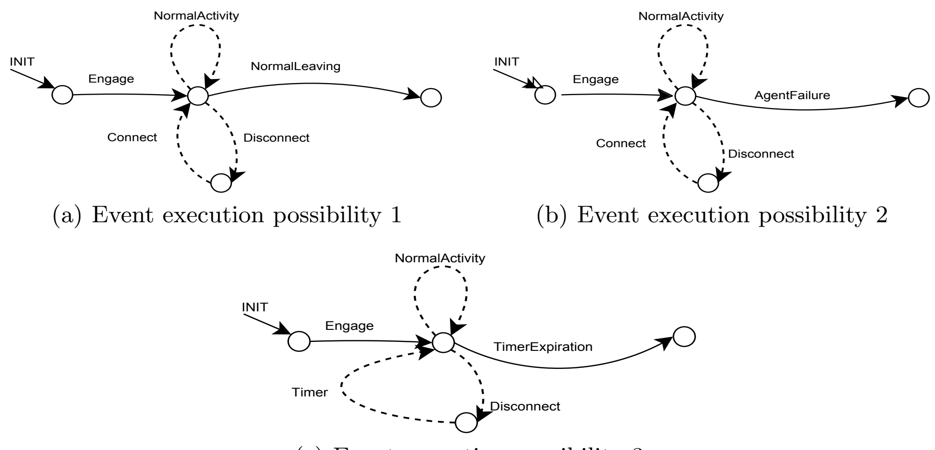
```
REFINEMENT Cama3 REFINES Cama2
  ...
EVENTS
  ...
  Disconnect = ANY aa WHERE aa ∈ agents
          THEN timers := timers ∪ {aa} END
  TempFailure = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
          THEN disconn_limit := disconn_limit - 1 || timers := timers - {aa} END;
END
```

The execution flow for *Cama*3 is shown in Fig.3(d). In the final refinement step, we elaborate on error recovery and time expiration by splitting the events *TempFailure* and *Failure* by atomicity refinement.

```
REFINEMENT Cama4 REFINES Cama3
  ...
EVENTS
  ...
  TimerExpiration ref Failure = ANY aa WHERE
            (aa ∈ agents) ∧ (aa ∈ ex_agents)
          THEN agents := agents - {aa} || ex_agents := ex_agents - {aa} END;
  AgentFailure ref Failure = ANY aa WHERE
            (aa ∈ agents) ∧ (aa ∉ timers) ∧ (aa ∉ ex_agents)
          THEN agents := agents - {aa} END;
  Connect ref TempFailure = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
          THEN disconn_limit := disconn_limit - 1 || timers := timers - {aa} END;
  Timer ref TempFailure = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
          THEN disconn_limit := disconn_limit - 1 || ex_agents := ex_agents ∪ {aa} ||
```

timers := *timers* - {*aa*} **END**
END

The execution graph for *Cama4* is shown in Fig.3(e). This graph shows all the possible events with their respective states but the order of execution is controlled by their guards. In addition, the Fig.4 shows all the possible scenarios based on the information derived from events' guards and bodies. The dashed



(a) Event execution possibility 1 (b) Event execution possibility 2

(c) Event execution possibility 3

Fig. 4. All possible Event execution scenarios

arrows represent possible loops of the event(s) during the execution. In order to generate concrete test cases from such models, the number of executions of an event in the loop can be restricted to some finite bound. The value for this bound depends on user's coverage criteria. The CSP representations of the *Cama* and *Cama4* machines are shown in the following.

```
Cama = Engage_Guard & Engage -> Node2;;
Node1 = NormalActivity_Guard & NormalActivity -> Node1;;
Node1 = Disengage_Guard & Disengage -> SKIP ;;
```

$\sqsubseteq$
....
$\sqsubseteq$

```
Cama4 = Engage_Guard & Engage -> Node1;;
Node1 = NormalActivity_Guard & NormalActivity -> Node1 ;;
Node1 = Disconnect_Guard & Disconnect -> Node2 ;;
Node1 = Failure_Guard & Failure -> SKIP ;;
Node1 = NormalLeaving_Guard & NormalLeaving -> SKIP ;;
Node1 = TimerExpiration_Guard & TimerExpiration -> SKIP ;;
```

```
Node2 = TempFailure_Guard & TempFailure -> Node1 ;;
Node2 = Timer_Guard & Timer -> Node1 ;;
```

These CSP expressions can be unfolded into test cases as described in Section 3.3.

5 Conclusions

In this paper, we presented a model-based testing approach based on automatic refinement of test scenarios. This work is being done as a possible extension (plug-in)for the RODIN open-source platform [1]. The EU project RODIN adopts systemic approach for development of complex systems in which fault-tolerance mechanisms are incorporated together with main system functionality. The scenario-based testing approach, presented in this paper, has been tried in several RODIN case-studies where fault-tolerance is the major concern. Our approach can also be used in formal software development process in general.

Acknowledgments

This work is supported by IST FP6 RODIN Project.

References

1. Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.
2. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
3. J.-R. Abrial. Event Driven Sequential Program Construction. 2000. Available at http://www.matisse.qinetiq.com.
4. J.-R. Abrial and L.Mussat. Introducing Dynamic Constraints in B. Second International B Conference, LNCS 1393, Springer-Verlag, April 1998.
5. Dalal S.R. et al. Model Based Testing in Practice . Proc. of the ICSE'99,Los Angeles,pp 285-294, 1999.
6. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.
7. Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Rigorous development of fault-tolerant agent systems. In *RODIN Book*, pages 241–260, 2006.
8. M. Leuschel and M. Butler. Prob: A model checker for b. Proc. of FME 2003, Springer-Verlag LNCS 2805, pages 855-874., 2003.
9. Leila Naslavsky, Thomas A. Alspaugh, Debra J. Richardson, and Hadar Ziv. Using Scenarios to support traceability. Proc of 3rd int. workshop on Traceability in emerging forms of software engineering, 2005.
10. Manoranjan Satpathy, Qaisar A. Malik, and Johan Lilius. Synthesis of scenario based test cases from b models. In *FATES/RV*, pages 133–147, 2006.

Formalizing UML-based Development of Fault Tolerant Control Systems*

Dubravka Ilić[1], Elena Troubitsyna[1], Linas Laibinis[1], and Colin Snook[2]

[1] Åbo Akademi University, Department of Information Technologies, 20520 Turku, Finland
{Dubravka.Ilic, Elena.Troubitsyna, Linas.Laibinis}@abo.fi
[2] School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
cfs@ecs.soton.ac.uk

Abstract. In this paper we demonstrate how to formalize UML-based development of protective wrappers for tolerating transient faults. In particular, we focus on the fault tolerance mechanisms common in the avionics domain and show the development of a protective wrapper, called Failure Management System. We demonstrate how to integrate the formal refinement approach proposed earlier into the UML-based development.

Keywords: Event-B, fault tolerance, refinement, statemachines, transient faults, UML-B.

1 Introduction

To guarantee dependability [1] of safety-critical software-intensive systems, we should ensure that they are not only fault-free but also tolerant to faults [2] of system components. This paper focuses on designing controlling software for tolerating transient faults [3]. Transient faults are temporal defects within the system. The mechanisms for tolerating this type of faults should ensure that the controlling software does not overreact on isolated faults yet does not allow the errors caused by these faults to propagate further into the system. These mechanisms constitute a large part of software in complex systems and hence they could be perceived as a separate subsystem dedicated to fault tolerance. In avionics, such a subsystem is traditionally called *Failure Management System* (FMS).

Earlier we proposed a generic formal pattern for specifying and developing the FMS [4] in the B Method [5, 6]. However, industrial engineers often perceive constructing a formal specification from informal requirements to be too complex to be done without an intermediate modeling stage. They usually use graphical modeling, mostly in UML [7], to facilitate this process. In this paper we

* This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

demonstrate how to integrate the formal approach proposed previously into the UML-based development. We use a subset of UML called UML-B [8] to specify and develop the FMS. To automate the process of obtaining a formal specification from UML models, we use the U2B tool [9], which translates UML-B models into Event-B [10]. Event-B is an extension of the B Method for developing reactive and distributed systems. We use the automated tool support for Event-B to verify the correctness of our development. Therefore, the proposed approach has a high degree of automation.

The paper is structured as follows. In Section 2 we shortly describe the FMS. Section 3 gives a brief introduction into our modeling frameworks – Event-B and UML-B. Section 4 demonstrates the process of developing the FMS in UML-B. We start from an abstract model of the FMS and obtain more detailed FMS models through a number of development phases. Moreover, we show how to translate these models into Event-B and verify their correctness. Finally, Section 5 concludes the paper.

2 Failure Management System

The Failure Management System (FMS) [11, 12] is a part of the embedded safety-critical control system as shown in Fig. 1. It can be perceived as a protective "wrapper" with the task to detect erroneous inputs from the sensors and prevent their propagation into the controller.

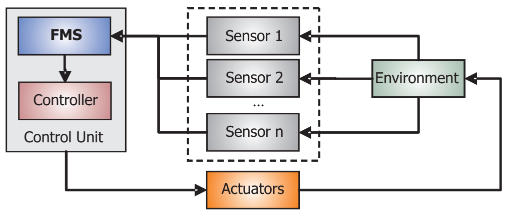


Fig. 1. Structure of an embedded control system

Based on sensor readings, the FMS calculates the output and forwards it to the controller. While calculating the output, the FMS has to ensure that only fault-free inputs received from the system environment are passed to the controller. This is achieved by considering the following pattern of the FMS behavior. We assume that initially the system is fault-free. Since control systems are usually cyclic, it is natural to describe the behavior of the FMS as cyclic as well. The FMS operating cycle starts by obtaining the readings from the monitored sensors as the inputs to the FMS. The FMS then tests the inputs by applying a certain error detection procedure. As a result, depending on whether the error was detected or not, the inputs are categorized as fault-free or faulty. Then the FMS takes the corresponding remedial actions that can be classified as healthy, temporary or confirmation [12]. An important part of these actions is input analysis, which distinguishes between recoverable and non-recoverable faulty inputs by assigning them different statuses.

To explain how the remedial actions work, for simplicity we consider a single sensor. *Healthy action* describes the "normal" FMS reaction when a received input is fault-free. In such a case, the input is assigned the status *ok* and it is forwarded unchanged to the controller. *Temporary action* describes the FMS reaction when a received input is faulty and recovering, meaning that the number of previously received faulty inputs has not yet reached some predefined limit. If this is the case, the input is assigned the status *suspected*. Then, the FMS calculates the output using the last good value of this input obtained in the previous FMS cycles. Finally, *confirmation action* describes the FMS reaction when a received input is faulty and it has failed to recover. Then, the input is assigned the status *confirmed failed* and the system proceeds with the control actions defined for freezing (stopping) the system or switching to a backup controller, if possible.

The pattern of the FMS behavior described above can be used in the product line development of the controlling software [13]. We use this pattern for developing the aircraft engine FMS in UML-B (and indirectly in Event-B). In the next section we introduce these modeling frameworks.

3 Modeling Frameworks – Event-B and UML-B

Event-B. The Event-B Method [10] is an approach for modeling dependable systems, which extends the B Method [5, 6]. In Event-B, a model of a system is described by *contexts* and *machines*. Contexts describe the static part of the system using *carrier sets*, *constants* and *axioms*. Machines describe system dynamics using *variables*, *invariants*, *theorems*, *events* and *variants*. Variables of the machine define the machine state. They are strongly typed by invariants and can be altered by events. Events are given in the form event=**WHEN** *guard* **THEN** *action* **END**, where *guard* is a state predicate on the variables, and *action* is a set of assignments, which simultaneously update the machine variables. If *guard* is satisfied, the event is enabled and the behavior of the event corresponds to the execution of its *action*. If *guard* is false, then the event is disabled, i.e., its execution is blocked.

The development methodology adopted by Event-B is based on stepwise refinement [14]. The result of a refinement step in Event-B is the machine that refines the state and events of an abstract machine. The invariant of this machine additionally contains the gluing invariant that describes the connection between the state spaces of the more abstract and refined machines.

To ensure correctness of a specification, we should verify that each event of the machine, including the initialisation, preserves the invariant. A high degree of automation in verifying correctness is provided by the available Event-B tool support [15].

UML-B. UML-B [8] is a specialisation of UML [7], which combines UML and Event-B to define a graphical formal modeling notation. UML is widely used graphical modeling language. However, it lacks precise semantics. Event-B, on the other hand, is a formal modeling framework, but it requires significant mathematical training from the users. The UML-B is developed as an alliance of these two modeling approaches. It contains a limited subset of UML entities which semantics is provided by their translation into Event-B using the U2B [9] translator tool. U2B converts a UML-B model into its equivalent Event-B model. We can then verify the model correctness by using the Event-B tool support.

In UML-B, a model of a system is described by *package*, *context*, *class*, and *state-machine diagrams*. A package diagram describes the abstract view on the system architecture. In other words, it describes the packages encapsulating the system on different levels of abstraction and the dependencies between them. In addition, it allows separating specification of the static and the dynamic parts of the system. This is achieved by defining two types of packages: *Context* and *Machine* package, which coincide with the concepts defined in Event-B. Each context has the associated context diagram defining the constants and properties of these constants (axioms). Each machine has the associated class diagram capturing the functional requirements of the modeled system. The classes of the class diagram define system components whose properties are specified as class attributes. The behavior of each component is defined by a statemachine diagram. Hence, on the abstract level the system is described by a set of class diagrams and statemachines encapsulated within the abstract machine package.

UML-B adopts the same approach to system development as Event-B, i.e., stepwise refinement. In particular, it uses superposition refinement [14], which allows us to extend the state space while preserving the existing data structures unchanged. The first step of refining a UML-B model is 'cloning' the current model in order to preserve the old class diagrams and statemachines. Then, we introduce new UML-B elements gradually by incorporating more details about the system structure and behavior. Specifically, more detailed behavior of the system is modeled with hierarchical states by adding sub-states and new transitions to the existing statemachines. Refinement of UML-B statemachines is described in detail in [16].

In general, while developing the system in a number of refinement steps, we create a chain of machine packages, where each subsequent package is a refinement of the previous package, i.e., of its class diagrams and statemachines. The refinement relation is established by adding the association *Refines* between the corresponding packages.

A more detailed description of UML-B entities is given in the following section, where we demonstrate how to specify and refine the FMS in UML-B. We also show how to obtain the Event-B models of FMS from their UML-B counterparts and verify their correctness.

4 Developing the FMS with specification and refinement templates in UML-B

The development of the FMS in UML-B is done in several phases. Each development phase corresponds to a refinement step. It is characterized by a set of

UML-B models (class and statemachine diagrams) representing the main structural and behavioral aspects of the FMS at the corresponding level of abstraction.

FMS abstract specification. At the highest level of abstraction, we consider a very simple FMS as shown in Fig. 2. In the class diagram FMS0, the fixed class SENSORS describes the set of n analogue sensors that are monitored by the FMS. Signals from each sensor are modeled as the class attribute Value. The output of the FMS is modeled as the machine variable Output. At this development phase, the FMS nondeterministically calculates the output using the last good sensor readings. Hence, we introduce an additional attribute to the class SENSORS – Last_Good_Value. Moreover, the subclass FAILED_SENSORS is introduced to model the sensors that have failed.

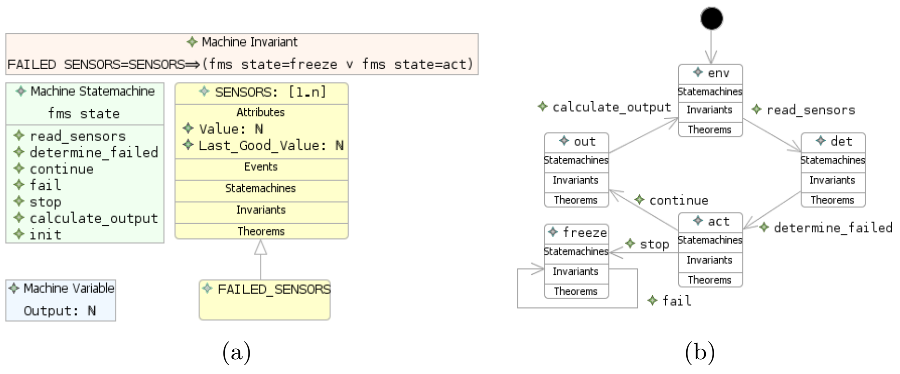


Fig. 2. (a) class diagram FMS0 and (b) statemachine fms_state for the 1st FMS development phase

The way in which the FMS behaves is described via the statemachine fms_state. The states env, det, act, out and freeze in this statemachine denote different stages of the FMS cycle. At this phase, we model the FMS cycle very abstractly: the FMS reads input values from the sensors, then it performs error detection, and either continues the cycle by calculating the output or fails. If the output is successfully calculated, the FMS cycle starts again. The FMS state changes are described by transitions between the states in the statemachine fms_state. For instance, the transition determine_failed simulates the error detection by nondeterministically choosing failed sensors, i.e., $\mathsf{FAILED\_SENSORS} :\in \{\mathsf{x} \mid \mathsf{x} \in \mathbb{P}(\mathsf{SENSORS})\}$. At the later development stages this transition will be refined to implement a more detailed error detection procedure.

To ensure that the FMS can proceed operating only with the sensors that have not failed, we define state invariants in the statemachine fms_state. Formally, the invariant $(\exists \mathsf{s} \cdot \mathsf{s} \in \mathsf{SENSORS} \wedge \mathsf{s} \notin \mathsf{FAILED\_SENSORS})$ is associated with the states env, det, and out. It means that, when the FMS is in these states, it processes readings from at least one operational (non-failed) sensor. The machine invariant, which is a part of the class diagram, additionally states the properties of the FMS when all the sensors have failed.

FMS refinement. The abstract FMS model is actually encapsulated in the machine package FMS0[1], as shown in Fig. 3. We further continue the FMS development by creating the refinement package FMSR1, which introduces changes into the abstract FMS model. At this development phase we refine the error detection from the abstract model by introducing sensor testing.

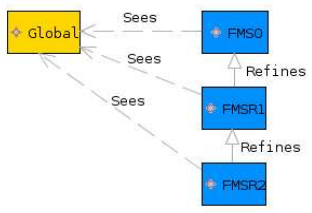


Fig. 3. FMS package diagram

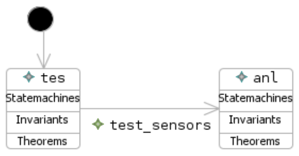


Fig. 4. sub-statemachine det_state

The diagrams from the previous phase remain the same. However, to introduce sensor testing, we refine the statemachine fms_state by creating the sub-statemachine det_state inside the state det, as shown in Fig. 4. The newly introduced sub-statemachine defines two new states tes and anl, designating the steps of the FMS error detection. Namely, after obtaining the sensor readings, the FMS performs testing the sensors (tes) and then analysis of inputs (anl) in order to detect errors, and then continues by determining which sensors have failed. The actual testing procedure is modeled as the transition test_sensors in the sub-statemachine det_state. It nondeterministically decides on the result of error detection. This result is modeled as a value assigned to a newly introduced attribute of the existing class SENSORS – Error_Detected. The FMS now uses this information to decide which sensors have failed. Hence, the transition determine_failed from the statemachine fms_state is refined as follows:

$$\begin{aligned}\mathsf{FAILED\_SENSORS} :\in \{&\mathsf{x} \mid \mathsf{x} \in \mathbb{P}(\mathsf{SENSORS}) \wedge \\ &(\forall \mathsf{s} \cdot \mathsf{s} \in \mathsf{x} \Rightarrow \mathsf{Error\_Detected}(\mathsf{s}) = \mathsf{TRUE})\}\end{aligned}$$

In addition, a new machine invariant is added to the existing class diagram. It describes in detail the properties of sensors: i.e., it requires that all failed sensors should be detected. The invariant is formally expressed as follows:

$$\mathsf{fms\_state} = \mathsf{act} \Rightarrow (\forall \mathsf{s} \cdot \mathsf{s} \in \mathsf{FAILED\_SENSORS} \Rightarrow \mathsf{Error\_Detected}(\mathsf{s}) = \mathsf{TRUE})$$

The way in which sensors are analyzed after testing is further refined in the 3rd development phase by creating the refinement package FMSR2. It contains all the class and statemachine diagrams from the previous phase. In addition, it contains a new sub-statemachine inside the state anl from the sub-statemachine det_state. This sub-statemachine defines more precisely the FMS behavior after performing tests on the sensors. Namely, the FMS decides about the status of each particular input before taking the corresponding remedial actions.

[1] FMS0 gets the access to the context Global by the association type Sees.

The structure of the FMS is refined as well, by introducing a new attribute Sensor_Status for modeling the result of this decision. The machine invariants can now be further strengthened to describe situations in which faulty sensors can recover. If they can not recover, the invariant guarantees that they will be considered as failed.

The following FMS development phases continue to refine the structure and the behavior of the original system. Due to the lack of space, we only outline these further development phases and omit their detailed description: the 4th development phase introduces detailed analysis of inputs based on the results of error detection. The input analysis is further elaborated in the 5th development phase by specifying a customizable counting mechanism, which reevaluates the status of the analyzed inputs at each FMS cycle. In a similar way, the 6th development phase describes in detail the error detecting procedure performed on each sensor and continues by introducing error detection tests in the 7th development phase. The 8th development phase further elaborates on different types of these tests.

Creating FMS Event-B models from UML-B models. Using the U2B tool, the Event-B models are automatically generated from the above UML-B models. For instance, the machine package FMS0 containing the diagrams given in Fig. 2 corresponds to the FMS0 Event-B machine shown in Fig. 5.

```
MACHINE   FMS0
SEES Global, FMS0_implicitContext
VARIABLES
      fms_state, FAILED_SENSORS, Value, Output, Last_Good_Value
INVARIANTS
      fms_state∈fms_state_STATES ∧FAILED_SENSORS∈ℙ(SENSORS) ∧
      Value∈SENSORS⟶ℕ ∧…
EVENTS
INITIALISATION
      BEGIN fms_state≔env ‖FAILED_SENSORS≔∅ ‖Value≔SENSORS×{InitInput}‖… END
read_sensors ==
      WHEN fms_state=env THEN fms_state≔det ‖Value:∈SENSORS⟶ℕ END
determine_failed ==
      WHEN fms_state=det THEN fms_state≔act ‖
      FAILED_SENSORS:∈{xx|xx∈ℙ(SENSORS)} END
continue ==
      ANY yy WHERE yy∈ℙ(Value) ∧fms_state=act ∧FAILED_SENSORS≠SENSORS
      THEN fms_state≔out ‖Last_Good_Value≔Last_Good_Value⩤yy END
calculate_output ==
      ANY xx WHERE xx∈ℙ(Last_Good_Value) ∧fms_state=out ∧…
      THEN fms_state≔env ‖Output:∈ran(xx) END
stop ==
      WHEN fms_state=act ∧FAILED_SENSORS=SENSORS THEN fms_state≔freeze END
fail ==
      WHEN fms_state=freeze THEN skip END
END
```

Fig. 5. Excerpt from the Event-B abstract specification FMS0

Informally, the rules for mapping some frequently used UML-B concepts into Event-B can be summarized as follows:

- a fixed class is defined as a constant, e.g., the class SENSORS corresponds to a constant defined in the automatically generated context FMS0_implicitContext;
- a subclass is represented as a variable, which is typed as a subset of its superclass, e.g., the subclass FAILED_SENSORS is defined as a subset of SENSORS;
- the name of a statemachine corresponds to a variable, which type is defined by enumerating its states, e.g., the statemachine fms_state is defined as the variable fms_state of the type fms_state_STATES, where the type is the enumerated set {env,det,act,out,freeze} defined in FMS0_implicitContext;
- an attribute of a fixed class becomes a machine variable typed as a function from the constant designating that class to the given attribute type, e.g., $\mathsf{Value} \in \mathsf{SENSORS} \to \mathbb{N}$ is an array of input readings for n sensors;
- a machine variable and an invariant are equivalent to the same concepts in Event-B;
- the transitions from a statemachine correspond to the events defined using the transition properties stated in UML-B. The complete list of translation rules can be found elsewhere (e.g., [8, 9]).

Similarly to the translation of the package FMS0 into the Event-B machine FMS0, the package FMSR1 refining the abstract package FMS0 is translated into the corresponding refined Event-B machine as shown in Fig. 6.

```
MACHINE   FMSR1
REFINES   FMS0
SEES Global, FMSR1_implicitContext
VARIABLES
     …, det_state, Error_Detected
INVARIANTS
     … ∧ det_state∈det_state_STATES ∧ Error_Detected∈SENSORS⟶BOOL ∧
     fms_state=act⇒(∀s·s∈FAILED_SENSORS⇒Error_Detected(s)=TRUE)
EVENTS
INITIALISATION
     BEGIN … det_state≔tes ∥ Error_Detected≔SENSORS×{FALSE} END
read_sensors == …
test_sensors ==
     WHEN fms_state=det ∧ det_state=tes
     THEN det_state≔anl ∥ Error_Detected:∈SENSORS⟶BOOL END
determine_failed (refines determine_failed) ==
     WHEN fms_state=det ∧ det_state=anl
     THEN fms_state≔act ∥ det_state≔tes ∥
     FAILED_SENSORS:∈{x|x∈ℙ(SENSORS)∧(∀s·s∈x⇒Error_Detected(s)=TRUE)}
     END
continue == …
calculate_output == …
stop == …
fail == …
END
```

Fig. 6. Excerpt from the Event-B refinement FMSR1

Observe that the machine FMSR1 explicitly states which machine it refines. It obtains two additional variables: one for the newly introduced class attribute Error_Detected and another one modeling the current state in the sub-statemachine det_state. The invariant of the machine FSMR1 is strengthened by typing the newly introduced variables and adding the gluing invariant that connects the new variable Error_Detected with the existing variable FAILED_SENSORS. Furthermore, FMSR1 introduces the new event test_sensors corresponding to the new transition in the sub-statemachine det_state from Fig. 4. It describes how the new variable Error_Detected is changed during the FMS error detection procedure. Moreover, the event determine_failed from the machine FMS0 is refined in the machine FMSR1. The guard of this event is strengthened by adding the new predicate that specifies the event enabling state of the sub-statemachine det_state. Correspondingly, the actions of the event are refined as well in order to incorporate the knowledge of the newly introduced variables.

Formal verification of the obtained Event-B machines is done using the automatic tool support for Event-B [15].

5 Conclusion

In this paper we demonstrated how to integrate the classical refinement development of the FMS [4] with UML-based development. Moreover, we showed how to use the available tool support to automate modeling and verification. Our approach has been validated by a case study – modeling and verification of the engine FMS for tolerating transient faults. The approach has several phases. Each phase is characterized by the set of UML-B models (class diagrams and statemachines). The complete specification of the FMS is obtained through a series of gradually refined UML-B models. Using the U2B translator tool, we generated Event-B models from the overall set of previously developed UML-B models. By translating UML-B models into Event-B, we were able to use the Event-B proof tool support to verify the correctness of our development. The results showed that we were able to prove the correctness of models significantly faster, with a higher percentage of automatic proofs than in our previous B development [4].

In the future, we are planning to investigate instantiation of the developed templates by using the obtained FMS UML-B contexts.

References

1. J. C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, 1991.
2. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), 2004.
3. N. Storey. *Safety-critical computer systems.* Addison-Wesley, 1996.
4. D. Ilic, E. Troubitsyna, L. Laibinis, and C. Snook. Formal Development of Mechanisms for Tolerating Transient Faults. In *REFT 2005, LNCS 4157*, pages 189–209. Springer-Verlag, November 2006.
5. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

6. S. Schneider. *The B Method. An Introduction.* Palgrave, 2001.
7. J. Rumbaugh, I. Jakobson, and G. Booch. *The Unified Modelling Language Reference Manual.* Addison-Wesley, 1999.
8. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. pages 92–122. ACM Transactions on Software Engineering and Methodology, ACM Transactions on Software Engineering and Methodology, 15(1), 2006.
9. C. Snook and M. Butler. *U2B - A tool for translating UML-B models into B*, chapter 6. UML-B Specification for Proven Embedded Systems Design. Springer, 2004.
10. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. In *Fundamentae Informatic*, 2006.
11. I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pages 115–129. Lisbon, 2004.
12. I. Johnson and C. Snook. Rodin Project Case Study 2: Requirements Specification Document. In *Rigorous Open Development Environment for Complex Systems(RODIN), Deliverable D4 - Traceable Requirements Document for Case Studies*, pages 24–52, 2005.
13. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.
14. R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
15. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *Proceedings of ICFEM'06, LNCS 4260*, pages 588–605. Springer-Verlag, 2006.
16. C. Snook and M. Walden. Refinement of Statemachines Using Event B Semantics. In *B 2007, LNCS 4355*, pages 171–185. Springer-Verlag, 2006.

Graphical modelling for simulation and formal analysis of wireless network protocols

A. Fehnker[1], M. Fruth[2], and A. K. McIver[3]

[1] National ICT Australia, Sydney, Australia;* ansgar@nicta.com
[2] School of Computer Science, University of Birmingham, UK; **
m.fruth@cs.bham.ac.uk
[3] Dept. Computer Science, Macquarie University, NSW 2109 Australia, and National ICT Australia; anabel@ics.mq.edu.au

Abstract. The aim of this research is to enhance performance analysis of wireless networks based on simulation with formal performance analysis.
It is well-known that the performance of protocols for wireless networks, and their ability to tolerate faults arising due to the uncertainties underlying wireless communication, relies as much on the topology of the network as on the protocols' internal algorithms. Many general-purpose simulation tools however do not use realistic models of wireless communication, and indeed results of simulation experiments can differ widely between simulators and often bear scant relation to field experiments [7, 6].
On the other hand, whilst model checking can supply more robust and exhaustive measures of performance, as for simulation, it is similarly flawed in that the details of the wireless communication are often overly simplified.
In this paper we propose a graphical specification style, which eases the study of the effect of topologies in performance analysis by visualising both the spatial characteristics of the network as well as critical measures of performance that they imply. Unlike other graphical visualisation tools, our proposal integrates both simulation using the novel Castalia simulator [3] as well as probabilistic model checking using PRISM [8], where we capture the effect of the topology by using probabilistic abstractions to model reception rates.

Keywords: Graphical modelling, simulation, lossy communication channels, probabilistic model checking, wireless networks.

1 Introduction

Wireless networks comprise devices with limited computing power together with wireless communication. Protocols for organising large-scale activities over these

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.
** This work was in part supported by the EPSRC grant EP/D076625/1

networks must be tolerant to the random faults intrinsic to the wireless medium, and their effectiveness is judged by detailed performance evaluation. One of the major factors impacting on the accuracy of an evaluation method is the mathematical model for the "communication channels" and, especially important, is that it must account for the unexpected disturbances induced by noise and interference amongst close neighbours. Conventional analysis methods rely on simulators [1, 2] incorporating some measure of random faults, however simulation in this context suffers from a number of well-documented problems [7, 6] — most notable is that accurate channel models validated against physical data do not normally feature. This leads to unrealistic results of performance analyses, which can vary widely between different simulators.

An alternative to simulation is formal modelling and analysis, which is normally ideally suited to investigating complex protocols, and gives access to profiles of performance which exhaustively range over worst- and best-case behaviour. Inclusion of realistic models of wireless communication implies appeal to analytical formulae to determine the effect on performance of the spatial relationships between nodes, such as the distance and density of near neighbours. These context-dependent details however are not easily added to textual-style formal modelling languages, and indeed they militate against a clear and modular specification style.

In this paper we overcome these difficulties by proposing a simple graphical style of specification. We exploit the observations that (a) the distance between and the density of nodes in a network is the major factor impacting on the integrity of wireless communication (together with physical parameters such as transmission strength), and (b) the simplest way to express the crucial spatial relationships is graphically, so that the details of the formal model of communication are transparent to the user and are provided separately.

Besides its simplicity, the graphical style has other benefits in that it allows designers to visualise various performance indicators such as best- or worst-case signal strength between pairs of nodes, or the nodes' individual power consumption. Similarly the critical events occurring in a sample experiment may be "stepped through" in a typical debugging style. Finally — unlike other graphical visualisation tools — it acts as a "bridge" between formal analysis and the more conventional simulation, providing the option to investigate performance using probabilistic model checking, or to carry out more traditional system-wide simulation experiments. In both cases realistic models for wireless communication play a fundamental role.

Our specific contributions are

1. CaVi, a graphical user interface specialised for modelling networks comprising simple wireless nodes. The tool gives immediate access to crucial performance indicators such as signal strength between pairs of nodes;
2. A translation from a CaVi model to either a formal transition-style model suitable for model checking in the PRISM model checker [8] or as input to the recently-developed Castalia simulator [3]. Castalia is novel in that it incorporates an accurate wireless channel model. The PRISM models are the first

such formal models which take network topology into account. At present both Castalia and PRISM capture only flooding and gossiping protocols [4, 5].

3. The option to visualise the network-wide performance metrics calculated from Castalia simulation experiments.

In Sec. 2 we summarise the interference model of wireless communication, and in Sec. 3 we describe how the CaVi graphical tool can visualise the effects of the spatial relationships. We also describe a translation from the graphical representation to Castalia and PRISM.

2 Wireless networks and lossy communication

In this section we describe the context of wireless applications, and the challenges that arise for their formal modelling.

In abstract terms a wireless network is a collection of nodes, running one protocol or a combination of protocols that are deployed over a two-dimensional area. The behaviour of the network depends not only on the protocol, but also on the placement of the nodes in the network, and in particular on the interference patterns arising from neighbouring communications. Inspired by other graphical tools [1], we propose a graphical-style of specification whose novelty is that it acts as a uniform modelling language to combine simulation and model checking. In either case, the interference effects disturbing communication are accounted for by the spatial representation, and are converted to reception probabilities in the translation to formal models.

In the next section we summarise the characteristics of wireless communication, which will set the scene for the graphical style of network specification.

2.1 Interference in wireless networks

Standard formal modelling of networked systems features both the behaviour of the individual "processors" (in this case "wireless nodes") and an explicit description of the "communication medium" connecting them. The assumption is that if two nodes are "connected" then they are able to send and receive messages without loss.

In reality, whether two nodes can communicate effectively depends on a number of context-specific factors, including the physical distance between the nodes, the signal strength of the sending node, and the extent to which other neighbouring nodes' activities, and those of the receiver, interfere with the sent message.

This complex "interference model" has been studied in depth and analytical formulae have been developed and experimentally validated [9]. Here we are able to appeal to those formulae to define a convenient conceptual abstraction of communication in terms of the "probability" that a sent message is received, with the probability computed by taking the distance, signal strength and interference of other nodes into account.

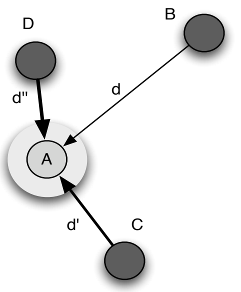
For example given two nodes A and B a distance d apart (see Fig. 1), the probability that A receives a message from B is given by

$$p_B^A(d, \iota_B) \mathrel{\hat{=}} (1 - e^{-\gamma_B(d,\iota_B)/(1.28)})^{8f} \ , \tag{1}$$

where f is the size of the message and γ_B is the *signal-to-noise ratio*. The latter is a function of the distance d, and the ambient noise ι_B. The signal-to-noise ratio is a measure of how much the background noise interferes with the wireless signal. The signal is dominant and the reception probability is high if $\gamma_B(d, \iota_B)$ is large.

The ambient ι_B includes effects due to the noise contributed by the signals from nearby nodes. For example, in Fig. 1 if nodes B, C and D all try to send to node A, then the mutual interference effects will produce a probability distribution over the message which A actually receives.

Because of the nature of the wireless communication, however, A will receive at most one message. The probability p_B^A at (1) is the probability that a message from B is received, and that either no message is received from either of the other nodes, or nothing is received at all. If all three nodes B, C and D send then the probability that A receives any message at all is given by the sum $p_B^A(d, \iota_A) + p_C^A(d', \iota_C) + p_D^A(d'', \iota_D)$. We note that this sum also takes into account the contribution to the ambient noise generated by each sender.



Sender D is closest to receiver A, so its signal is strongest; Sender B's is weakest. All reception probabilities are affected by the others' activities. Here d, d' and d'' are the distances from the senders to the receiver A.

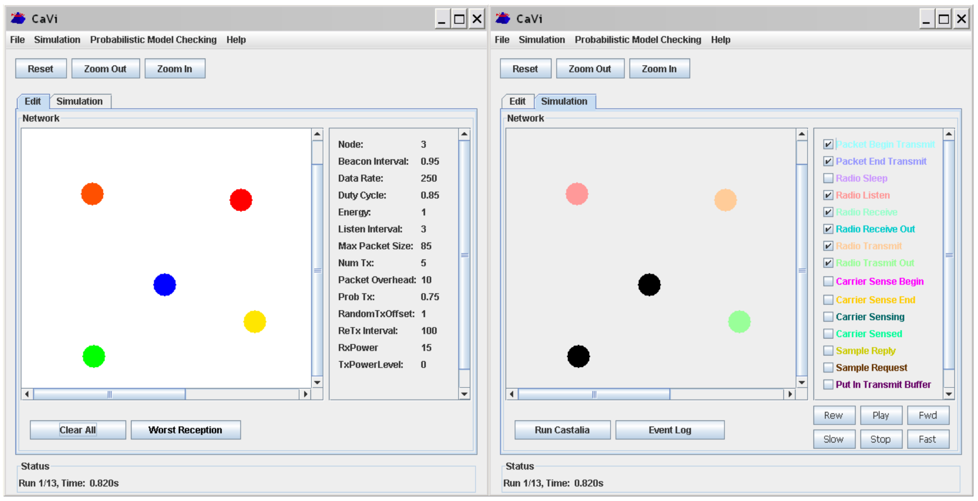
Fig. 1. Signal strength varying with distance and interference

3 CaVi: A graphical specification tool

CaVi is a tool which provides specification and analysis support optimised for studying wireless protocols. Its main feature is a graphical interface which eases the task of exploring the effect on performance of different topologies and network parameters. Nodes may be created in a "drag-and-drop" fashion, and the properties of individual nodes (such as the available power) may be tuned as necessary. Whilst the network is being created, a user can visualise the optimal "one-hop" signal strength between any pair of nodes, calculated from equation (1). In Fig. 2 we illustrate two examples of how the graphical interface may be used in the design and analysis. The figure shows two panes, with the left being the pane where designers may create and edit a network, and the pane on the right is for visualising the results of simulation experiments.

The pane on the left illustrates visualisation of "one-hop" signal strength by colour-coding the nodes according to probability thresholds calculated from (1). The user may indicate which node is the receiving node (in this case the central node), and the others are assumed to be senders. Colours then differentiate between nodes whose messages will be almost certainly lost (red), or have a good chance of succeeding (green), or merely a variable chance (yellow).

The pane on the right indicates how the events may be viewed as a result of a simulation experiment. The panel on the right gives a list of possible "colour-coded" events (*e.g.* transmitting, receiving etc.); users may select which events to observe, and the nodes assume the colour of the corresponding event as the simulation is "stepped through".



Node:	3
Beacon Interval:	0.95
Data Rate:	250
Duty Cycle:	0.85
Energy:	1
Listen Interval:	3
Max Packet Size:	85
Num Tx:	5
Packet Overhead:	10
Prob Tx:	0.75
RandomTxOffset:	1
ReTx Interval:	100
RxPower	15
TxPowerLevel:	0

Run 1/13, Time: 0.820s

- ☑ Packet Begin Transmit
- ☑ Packet End Transmit
- ☐ Radio Sleep
- ☑ Radio Listen
- ☑ Radio Receive
- ☑ Radio Receive Out
- ☑ Radio Transmit
- ☑ Radio Trasmit Out
- ☐ Carrier Sense Begin
- ☐ Carrier Sense End
- ☐ Carrier Sensing
- ☐ Carrier Sensed
- ☐ Sample Reply
- ☐ Sample Request
- ☐ Put In Transmit Buffer

Run 1/13, Time: 0.820s

Fig. 2. CaVi:Visualising network performance indicators

Once the network is specified, the graphical representation forms the basis for formal models which take account of the effect of the topology in terms of the reception probabilities. At present we have implemented the automated conversion to a textual format suitable for evaluation directly in the Castalia simulator [3], and as mentioned above the results of experiments may be visualised in various ways described at Fig. 2. We note that the Castalia simulator is a recently-developed simulator whose novelty is a realistic channel/radio model which builds on recent work done on modelling of the radio and the wireless channel based on empirically measured data [9].

3.1 Formal model checking

One of the major outcomes of this work has been to introduce realistic channel behaviour into formal models for more detailed analysis via probabilistic model checking. We use an abstraction of signal strength in terms of the probability of reception, computed from a formula based on on (1). In Fig. 3 we illustrate the formal template model of a simple node whose only capabilities are that it can receive or send a message, or "do nothing". If in receiving mode (*recv*=1) then the chance that it receives a message is p_r which is computed from (1) based on the states of the surrounding nodes. A network is made up of a collection of similar nodes in parallel.

This abstraction of the wireless communication using reception probabilities has been implemented in the PRISM model checker for simple flooding protocols, where the use of the formula rather than expanding the size of the resulting model leads to exceedingly compact models making probabilistic model checking a viable option.

At present we do not have an automated generation of PRISM models from CaVi — that remains a topic for future research.

$$Node \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var}\ \mathit{send},\ \mathit{recv}\colon \{0,1\} \\ \textbf{tick}:\ (\mathit{recv}=1) \to\ \mathit{send},\ \mathit{recv}\!:=\ 1,0;\ \ {}_{\mathsf{p_r}}\!\oplus\ \ \textbf{skip}; \\ \textbf{tick}:\ (\mathit{send}=1) \to\ \mathit{send},\ \mathit{recv}\!:=\ 0,0; \\ \textbf{tick}:\ (\mathit{send}=0 \wedge \mathit{recv}=0) \to\ \textbf{skip}; \end{array}\right)$$

The probability p_r is computed as a function of the state dependent on the neighbouring nodes. Here **tick** is a named event, each "guarded" by a Boolean-function of the state; if any one of the guards is true, then the variables are updated according to the assignments on the right-hand side of the arrow. The probabilistic choice operator ${}_{p_r}\!\oplus$ means that the left-hand side of the operator is executed with probability p_r, and the right-hand side with probability $1-p_r$.

Fig. 3. A template for a node with parameterised reception probability.

4 Conclusions and future work

In this paper we have described a prototype tool which supports a uniform modelling approach optimised for specifying wireless protocols. Its main features include the capabilities to take account of the topology and other parameters of the network which, experiments have shown, have a major impact on the integrity of the communication. The CaVi tool allows the specification of a network via a graphical interface, and the automated generation to a format for simulation. Detailed performance indicators may be visualised during specification of the network, as well as the results of subsequent simulation experiments.

The principal difference between CaVi and other specification tools is the link it provides between simulation and formal model checking. To simplify the details related to the topology in the formal specification task, we use a translation directly to reception probabilities. Those probabilities are calculated according to a validated analytic formula.

An understanding of realistic channel behaviour has suggested some novel approaches to formal verification of wireless protocols, and in the future we hope to incorporate such detailed analyses within the CaVi tool.

For the future we would like to automate the translation from CaVi to PRISM, making CaVi a truly uniform interface between simulation and model checking. Whilst we do not envisage a translation from a CaVi model of an arbitrary protocol to PRISM, we aim rather to provide a library of templates for certain classes of protocol whose precise behaviour can be defined by a number of parameters, in the same way that models are defined in Castalia.

One of the benefits would be a single "top-level" graphical model for simulation and model checking and the ability to visualise the results obtained from both in a uniform way. Such a "bridging language" would allow "counterexamples" computed via model checking to be validated in the simulator, for example.

In the longer term we would like to expand the repertoire of protocols, and to build up a repository of well-studied templates for Castalia and PRISM patterns.

References

1. OPNET.
   `http://www.opnet.com/`.
2. The network simulator ns-2.
   `http://www.isi.edu/nsnam/ns/`.
3. A. Boulis. Castalia: A simulator for wireless sensor networks.
   `http://castalia.npc.nicta.com.au`.
4. A.Fehnker and P. Gao. Formal verification and simulation for performance analysis of probabilistic broadcast protocols. In *5'th International Conference, ADHOC-NOW*, volume 4104 of *LNCS*, pages 128–141. Springer, 2006.
5. A.Fehnker and A. McIver. Formal analysis of wireless protocols. In *Proc 2nd International Symposium in Leveraging applications in formal methods, verification and validation*, 2006.
6. D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of manet simulators. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 38–43. ACM Press, 2002.
7. K. Kotz, C. Newport, R.S.Gray, J. Liu, Y. Yuan, and C. Elliott. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 78–82. ACM Press, 2004.
8. PRISM. Probabilistic symbolic model checker.
   `www.cs.bham.ac.uk/~dxp/prism`.

9. M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. In *First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON)*, pages 517–526. IEEE, 2004.

Temporal Verification of Fault-Tolerant Protocols

Michael Fisher, Boris Konev, and Alexei Lisitsa

Department of Computer Science, University of Liverpool, Liverpool, United Kingdom
{M.Fisher, B.Konev, A.Lisitsa}@csc.liv.ac.uk

1 Introduction

The automated verification of concurrent and distributed systems is a vibrant and successful area within Computer Science. Over the last 30 years, *temporal logic* [8, 16] has been shown to provide a clear, concise and intuitive description of many such systems, and automata-theoretic techniques such as *model checking* [5] have been shown to be very useful in practical verification. In recent years, the verification of *infinite-state* systems, particularly parametrised systems comprising arbitrary numbers of identical processes, has become increasingly important. Practical problems of an open, distributed nature often fit into this model. However, once we move beyond finite-state systems, which we do when we consider systems with *arbitrary* numbers of components, problems occur. Although temporal logic still retains its ability to express such complex systems, verification techniques such as model checking must be modified. In particular, *abstraction* techniques are typically used to reduce an infinite-state problem down to a finite-state variant suitable for application of standard model checking techniques. However, it is clear that such abstraction techniques are not always easy to apply and that more sophisticated verification approaches must be developed. In assessing reliability of these systems, formal verification is clearly desirable and so several new approaches have been developed:

1. *model checking for parametrised and infinite state-systems* [1, 2];
2. *constraint based verification using counting abstractions* [7, 9]; and
3. *deductive verification in first-order temporal logics* [10, 6].

The last of these approaches is particularly appealing, often being both complete (unlike (1)) and decidable (unlike (2)), able to verify both safety *and* liveness properties, and adaptable to more sophisticated systems involving asynchronous processes or communication delays.

Now we come to the problem of verifying fault tolerance in protocols involving an arbitrary number of processes. What if some of the processes develop faults? Will the protocol still work? And how many processes must fail before the protocol fails? Rather than specifying *exactly* how many processes will fail, which reduces the problem to a simpler version, we wish to say that there is *some* number of faulty processes, and that failure can occur at any time. Again we can capture this using temporal logics. If we allow there to be an infinite number of failures, then the specification and verification problem again becomes easier; however, such scenarios appear unrealistic. So, we are left with the core problem: *can we develop deductive temporal techniques for the verification of parametrised systems where a* ***finite*** *number of failures can occur?* This question is exactly what we address here.

We proceed as follows. Section 2 gives a brief review of *first-order temporal logic* (FOTL) and its properties. In Section 3, we propose two mechanisms for adapting deductive techniques for FOTL to the problem of finite numbers of failures in infinite-state systems, and in Section 4 we outline a case study. Finally, in Section 5, we provide concluding remarks.

2 Monodic First-Order Temporal Logics

First-order linear time temporal logic (FOTL) is a very powerful and expressive formalism in which the specification of many algorithms, protocols and computational systems can be given at the natural level of abstraction [16]. Unfortunately, this power also means that, over many natural time flows, this logic is highly undecidable (non recursively enumerable). Even with incomplete proof systems, or with proof systems complete only for restricted fragments, FOTL is interesting for the case of parametrised verification: one proof may certify correctness of an algorithm for infinitely many possible inputs, or correctness of a system with infinitely many states.

FOTL is an extension of classical first-order logic by temporal operators for a discrete linear model of time (isomorphic to $\mathbb{N}$, being the most commonly used model of time). Formulae of this logic are interpreted over structures that associate with each element n of $\mathbb{N}$, representing a moment in time, a first-order structure $\mathfrak{M}_n = (D, I_n)$ with the same non-empty domain D.

The *truth* relation $\mathfrak{M}_n \models^{\mathfrak{a}} \phi$ in the structure $\mathfrak{M}$ and a variable assignment $\mathfrak{a}$ is defined inductively in the usual way under for the following (sample) temporal operators:

$\mathfrak{M}_n \models^{\mathfrak{a}} \bigcirc\phi$ iff $\mathfrak{M}_{n+1} \models^{\mathfrak{a}} \phi$;
$\mathfrak{M}_n \models^{\mathfrak{a}} \Diamond\phi$ iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;
$\mathfrak{M}_n \models^{\mathfrak{a}} \Box\phi$ iff for all $m \geq n$, $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;
$\mathfrak{M}_n \models^{\mathfrak{a}} \blacklozenge\phi$ iff there exists $0 \leq m < n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$.

$\mathfrak{M}$ is a *model* for a formula ϕ (or ϕ is *true* in $\mathfrak{M}$) if there exists an assignment $\mathfrak{a}$ such that $\mathfrak{M}_0 \models^{\mathfrak{a}} \phi$. A formula is *satisfiable* if it has a model. A formula is *valid* if it is satisfiable in any temporal structure under any assignment. The set of valid formulae of this logic is not recursively enumerable. Thus, there was a need for an approach that could tackle the temporal verification of parametrised systems in a *complete* and *decidable* way. This was achieved for a wide class of parametrised systems using *monodic temporal logic*.

Definition 1. *A FOTL formula is said to be* **monodic** *if, and only if, any subformula with its main connective being a temporal operator has at most one free variable.*

Thus, ϕ is called *monodic* if any subformula of ϕ of the form $\bigcirc\psi$, $\Box\psi$, $\Diamond\psi$, $\blacklozenge\psi$, etc., contains at most one free variable. For example, the formulae $\forall x \Box \exists y P(x, y)$ and $\forall x \Box P(x, c)$ are monodic, while $\forall x, y(P(x, y) \Rightarrow \Box P(x, y))$ is not monodic.

The monodic fragment of FOTL has appealing properties: it is axiomatisable [17] and many of its subfragments, such as the two-variable or monadic cases, are decidable. This fragment has a wide range of applications, for example in spatio-temporal logics [11] and temporal description logics [3]. A practical approach to proving monodic temporal formulae is to use *fine-grained temporal resolution* [14], which has been implemented in the theorem prover TeMP [13]. It was also used for deductive verification of parametrised systems [10]. One can see that in many cases temporal specifications fit into the even narrower, and decidable, monodic *monadic* fragment. A formula is monadic if all its predicates are *unary*.

3 Incorporating Finiteness

When modelling parametrised systems in temporal logic, informally, elements of the domain correspond to processes, and predicates to states of such processes. For example $idle(x)$ means that a process x is in the idle state. For many protocols, especially when fault tolerance is concerned, it is essential that the number of processes is finite. Although decidability of monodic fragments holds also for the case of semantics where only temporal structures over *finite domains* are allowed [12], the proof is model-theoretic and no practical procedure is known.

We here examine two approaches that allow us to handle the problem of finiteness within temporal specification. First, we consider proof principles which can be used to establish correctness of some parametrised protocols; then we prove that, for a wide class of protocols, decision procedures that do not assume the finiteness of a domain can still be used.

3.1 Formalising Principles of Finiteness

The language of FOTL is very powerful and one might ask if a form of finiteness can be defined inside the logic. We have found the following principles (which are valid over finite domains, though not in general) useful when analysing the proofs of correctness of various protocols and algorithms specified in FOTL (recall: $\blacklozenge\varphi$ means φ *was* true in the past):

Fin_1: $\Diamond(\forall\, x.(P(x) \lor \Diamond P(x) \to \blacklozenge P(x)))$ **(deadline axiom)**
Fin_2: $[\forall x.\Box(P(x) \to \bigcirc\Box\neg P(x)] \Rightarrow [\Diamond\Box(\forall x.\neg P(x))]$ **(finite clock axiom)**
Fin_3: $[\Box(\forall x.(P(x) \to \bigcirc P(x))] \Rightarrow [\Diamond\Box(\forall x.(\bigcirc P(x) \to P(x))]$ **(stabilisation axiom)**

Actually the Fin_1 principle is a (more applicable) variant of the intuitively clearer principle $[\forall x.\, \Diamond P(x)] \Rightarrow [\Diamond\forall x.\, \blacklozenge P(x)]$ which is also valid over finite domains.

Consider now Fin_i for $i = 1, 2, 3$ as axiom schemes which can be added to Ax_{FOTL} in order to capture, at least partially, "finite reasoning". We show that all these three principles are actually equivalent modulo any reasonable Ax_{FOTL} (i.e. they can be mutually derived).

The principle (an axiom scheme) F_1 is said to be derivable from F_2 if, for every instance α of F_1, we have $Ax_{FOTL} + F_2 \vdash \alpha$. We will denote it simply $Ax_{FOTL} + F_2 \vdash F_1$.

Theorem 1 *The principles Fin_1, Fin_2 and Fin_3 are mutually derivable.*

3.2 Eventually Stable Protocols

In the previous section we highlighted some deduction principles capturing the finiteness of the domain. Alternatively, we can consider a family of protocols which terminate after a certain (but unknown) number of steps. For example, if every process sends only a finite number of messages, such protocol will eventually terminate. Consensus protocols [15], distributed commit protocols [4], and some other protocols fit into this class. Temporal models of specifications of such terminating protocols will eventually stabilise, that is, the interpretations I_n will be the same for sufficiently large n. We show that for these *eventually stable* specifications satisfiability over finite domains coincides with satisfiability over arbitrary domains.

Let $\mathcal{P}$ be a set of unary predicates. The *stabilisation principle w.r.t.* $\mathcal{P}$ is the formula:

$$\mathsf{Stab}_{\mathcal{P}} = \Box(\forall x \bigwedge_{P \in \mathcal{P}} [P(x) \equiv \bigcirc P(x)]).$$

Informally, if $\mathsf{Stab}_{\mathcal{P}}$ is true at some moment of time, from this moment the interpretation of predicates in $\mathcal{P}$ does not change. Let ϕ be a monodic temporal formula. Let $\mathcal{P}$ be the set of unary predicates occurring in ϕ. Then the formula

$$\phi_{\mathsf{Stab}} = \phi \wedge \Diamond \mathsf{Stab}$$

is called an *eventually stable formula*. We formulate the following proposition for monodic monadic formulae; it can be extended to other monodic classes obtained by *temporalisation by renaming* [6] of first-order classes with the finite model property.

Proposition 1. *Let ϕ be a monodic monadic formula. The eventually stable formula ϕ_{Stab} is satisfiable in a model with a finite domain if, and only if, ϕ_{Stab} is satisfiable in a model with an arbitrary domain.*

This proposition implies that if a protocol is such that it can be faithfully represented by an eventually stable formula, correctness of such protocol can be established by a procedure that does *not* assume the finiteness of the domain.

4 Case Study: FloodSet Protocol

Next, we provide an example of how both methods described in previous section (explicit finiteness principles, and stabilisation principle for protocols with finite change) can be used for the proof of correctness of a protocol specified in monodic FOTL.

The setting is as follows. There are n processes, each having an *input bit* and an *output bit*. The processes work synchronously, run the same algorithm and use *broadcast* for communication. Some processes may fail and, from that point onward, such processes do not send any further messages. Note, however, that the messages sent by a process *in the moment of failure* may be delivered to *an arbitrary subset* of the processes. Crucially, there is a *finite* bound, f, on the number of processes that may fail.

The goal of the algorithm is to eventually reach an agreement, i.e. to produce an output bit, which would be the same for all non-faulty processes. It is required also that if all processes have the same input bit, that bit should be produced as an output bit.

This is a variant of *FloodSet algorithm with alternative decision rule* (in terms of [15], p.105) designed for solution of the Consensus problem in the presence of crash (or fail-stop) failures, and the basic elements of the protocol (adapted from [15][1]) are as follows.

- In the first round of computations, every process broadcasts its input bit.
- In every later round, a process broadcasts any value *the first time it sees it*.
- In every round the (tentative) output bit is set to the minimum value seen so far.

The correctness criterion for this protocol is that, eventually (actually, no later than in $f + 2$ rounds) the output bits of all non-faulty processes will be the same.

Claim. The above FloodSet algorithm and its correctness conditions can be specified (naturally) within monodic monadic temporal logic without equality and its correctness can be proved in monodic monadic temporal logic, using the above **finite clock axiom**.

We do not include the whole proof here, but will reproduce sample formulae to give the reader a flavour of the specification and proof.

1. Each process (s) must be categorised as one of the above types:
   $\Box(\forall x(Normal(x) \mid Failure(x) \mid Faulty(x)))$
2. If we see a '0' (the process has this already, or receives a message with this value) then we output '0':
   $\Box(\forall x(\neg Faulty(x) \wedge Seen(x, 0) \rightarrow \bigcirc Output(x) = 0))$
3. If we have not seen a '0' but *have* seen a '1', then we output '1':
   $\Box(\forall x(\neg Faulty(x) \wedge \neg Seen(x, 0) \wedge Seen(x, 1) \rightarrow \bigcirc Output(x) = 1))$
4. The condition to be verified, namely that eventually all (non faulty) processes agree on the bit '0', or eventually all agree on the bit '1':

$$\Diamond((\forall x \neg Faulty(x) \Rightarrow Output(x) = 0) \;\vee\; (\forall x \neg Faulty(x) \Rightarrow Output(x) = 1))$$

Notice that the temporal specification uses among others the predicates $Normal(\_)$ to denote normal operating processes, $Failure(\_)$ to denote processes, experiencing failure (at some point of time), $Faulty(\_)$ for the processes already failed. There are also predicates such as $Seen(\_,\_)$ specifying the effect of communications. Having these, it is straightforward to write down the temporal formulae describing the above protocol and correctness condition (i.e. (4) above). In the proof of correctness the **finite clock axiom** has to be instantiated to the $Failure(x)$ predicate (i.e. replace P by $Failure$ in Fin_2).

One may also verify the *FloodSet* protocol using the eventual stabilisation principle from Section 3.2. To establish the applicability of the principle one may use the following arguments: every process can broadcast at most twice, and taking into account

[1] In [15], every process *knows* the bound f in advance and stops the execution of the protocol after $f + 2$ rounds, producing the appropriate output bit. We consider the version where the processes do not know f in advance and produce a *tentative output bit* at every round.

finiteness of both the numbers of processes and of failures, one may conclude that eventually the protocol stabilises. Note that such an analysis only allows us to conclude that the protocol stabilises, but its properties still need to be proved. Let ϕ be a temporal specification of the protocol. Taking into account the stabilisation property, the protocol is correct iff $(\phi \wedge \neg\psi)_{\mathsf{Stab}}$ is not satisfiable over finite domains. By Proposition 1, there is no difference in satisfiability over finite and general domains for such formulae and so one may use theorem proving methods developed for monadic monodic temporal logics over general models to establish this fact.

5 Concluding Remarks

In this paper we have studied two approaches to handling the finiteness of the domain in temporal reasoning.

The first approach uses explicit finiteness principles as axioms (or proof rules), and has potentially wider applicability, not being restricted to protocols with the stabilisation property. On the other hand, the automation of temporal proof search with finiteness principles appears to be more difficult and it is still largely an open problem.

In the approach based on the stabilisation principle, all "finiteness reasoning" is done at the meta-level and essentially this is used to reduce the problem formulated for finite domains to the general (not necessarily finite) case. When applicable, this method is more straightforward for implementation and potentially more efficient. Applicability, however, is restricted to the protocols which have stabilisation property (and this property should be demonstrated in advance as a pre-condition).

Finally, we briefly mention some future work. Automated proof techniques for monadic monodic FOTL have been developed [6, 14] and implemented in the TeMP system [13], yet currently proof search involving the finiteness principles requires improvement. Once this has been completed, larger case studies will be tackled. The techniques themselves would also benefit from extension involving probabilistic, real-time and equational reasoning.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular Model Checking for LTL(MSO). In *Proc. 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 348–360. Springer, 2004.
2. P. A. Abdulla, B. Jonsson, A. Rezine, and M. Saksena. Proving Liveness by Backwards Reachability. In *Proc. 17th International Conference on Concurrency Theory (CONCUR)*, volume 4137 of *LNCS*, pages 95–109. Springer, 2006.
3. A. Artale, E. Franconi, F. Wolter, and M. Zakharyaschev. A Temporal Description Logic for Reasoning over Conceptual Schemas and Queries. In *Proc. European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *LNCS*, pages 98–110. Springer, 2002.
4. D. Chkliaev, P. van der Stock, and J. Hooman. Mechanical Verification of a Non-Blocking Atomic Commitment Protocol. In *Proc. ICDCS Workshop on Distributed System Validation and Verification*, pages 96–103, IEEE, 2000.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
6. A. Degtyarev, M. Fisher, and B. Konev. Monodic Temporal Resolution. *ACM Transactions on Computational Logic*, 7(1):108–150, January 2006.
7. G. Delzanno. Constraint-based Verification of Parametrized Cache Coherence Protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
8. E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.
9. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS)*, pages 352–359. IEEE CS Press, 1999.
10. M. Fisher, B. Konev, and A. Lisitsa. Practical Infinite-state Verification with Temporal Reasoning. In *Verification of Infinite State Systems and Security*, volume 1 of *NATO Security through Science Series: Information and Communication*. IOS Press, January 2006.
11. D. Gabelaia, R. Kontchakov, A. Kurucz, F. Wolter, and M. Zakharyaschev. On the Computational Complexity of Spatio-Temporal Logics. In *Proc. 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 460–464. AAAI Press, 2003.
12. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable Fragments of First-order Temporal Logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.
13. U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov. TeMP: A Temporal Monodic Prover. In *Proc. 2nd Second International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *LNAI*, pages 326–330. Springer, 2004.
14. B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Mechanising First-order Temporal Resolution. *Information and Computation*, 199(1-2):55–86, 2005.
15. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
16. Z. Manna and A. Pnueli. *Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.
17. F. Wolter and M. Zakharyaschev. Axiomatizing the Monodic Fragment of First-order Temporal Logic. *Annals of Pure and Applied Logic*, 118(1-2):133–145, 2002.

Interaction Analysis for Fault-Tolerance in Aspect-Oriented Programming[⋆]

Nathan Weston, Francois Taiani, Awais Rashid

Computing Department, InfoLab21, Lancaster University, UK.
{westonn,f.taiani,marash}@comp.lancs.ac.uk

Abstract. The key contribution of Aspect-Oriented Programming (AOP) is the encapsulation of crosscutting concerns in aspects, which facilities modular reasoning. However, common methods of introducing aspects into the system, incorporating features such as implicit control-flow, mean that the ability to discover interactions between aspects can be compromised. This has profound implications for developers working on fault-tolerant systems. We present an analysis for aspects which can reveal these interactions, thus providing insight into positioning of error detection mechanisms and outlining candidate containment units. We also present Aida, an implementation of this analysis for the AspectJ language.

1 Introduction and Problem Statement

The key contribution of Aspect-Oriented Programming (AOP) is the encapsulation of crosscutting concerns in *aspects*, which are then introduced into the system using *advice* - code which applies at a particular *joinpoint* in the system, be that in the base program or within aspect advice. Advice is then *woven* into the system, either statically at compile-time or dynamically later on. This crucial feature of AOP supports modularity and evolvability of otherwise scattered and tangled code, as well as offering the possibility of aspect reuse.

However, it also raises a potential difficulty for developers working with aspects in a fault-tolerant context. For example, the usual method of implementing aspects - such as the popular AspectJ[1] compiler - allows a developer to code a piece of advice which applies implicitly at multiple points in the code. This can cause problems in determining how faults might propagate through the system, as it is not immediately clear from the code how advice code interacts with the base system, and especially how aspects interact with one another.

To see this, let us consider the example of a version control system which includes the following code:

[⋆] This work is supported by European Commission Framework 6 Grant: AOSD-Europe: European Network of Excellence on Aspect-Oriented Software Development (IST-2-004349).

```
void commitChanges(String username, String server) {
    ...
    sendFile(username, file , server );
    ...
}
```

The system has two aspects - one which logs all calls to the `sendFile()` method, and one which encrypts usernames at calls to the `commitChanges()` method:

```
void aspect LogFileSends {
    after(): call (void sendFile(String, String, String)) && args(uname, file, serv) {
        printToFile("Sent to server, sender is "+uname);
    }
}

void aspect Encrypt {
    around(): call(void commitChanges(String, String)) ∧ args(uname, serv) {
        encrypt(uname);
        proceed(uname, serv);
    }
}
```

Although these two advices apply at different joinpoints in the system, they have an indirect interaction with one another - the `Encrypt` aspect modifies the `username` variable, which the `LogFileSends` aspect reads. Therefore there is a potential coupling between the two advices. Knowing this could impact the strategy for making this system fault-tolerant - for example, if the `LogFileSends` aspect is considered particularly crucial to the system, this might require the `Encrypt` aspect to be hardened with additional fault-detection mechanisms.

As well as this *indirect* interaction, we must also consider the possibility of *transitive* interactions between aspects. By contrast to indirect interactions, which are based on shared accesses of a single variable (for example, the `uname` variable in the above example), transitive interactions occur when a chain of variable accesses link advices. That is, consider the system in Fig. 1. The system has an aspect A which modifies a variable x. The variable is then passed to a method f, which uses x to define a variable y. Subsequently, the value of y is used in aspect B to determine its behaviour. Hence there is a transitive interaction between the two aspects, even though they do not access a shared variable.

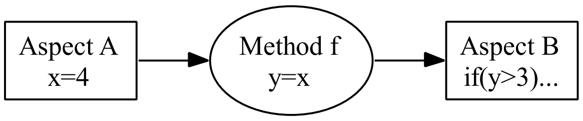


Fig. 1: Transitive interaction

In general, the ability to identify and trace these interactions can help developers understand how error might propagate, and thus decide where to position error detection mechanisms (such as assertions or acceptance tests) in order to maximise error detection coverage while minimising overhead. It can also provide insight into error handling by identifying parts of the system which might simultaneously become corrupted, thus outlining candidate containment units.

This kind of interaction analysis also has applications should AO techniques be used to implement fault-tolerance itself. As has been noted, AOP can be an extremely helpful tool to aid developers in building fault-tolerant systems. For example, transaction mechanisms can be implemented as aspects in order to perform rollback in the event of failure[5]; similarly, the mechanisms for switching between versions in N-version programming can be encapsulated in an aspect. Contract enforcement[3] can also be modularised in this way. In this formulation, then, the interactions between fault-tolerance aspects and others can shed light on potential problems which could cause the system to be intolerant to faults - for example, if it can be seen that the presence of another aspect causes a contract enforcement aspect to be bypassed.

In this article we investigate how Data-Flow Analysis can be adapted to be applied to aspect-oriented programs in order to help developers with this problem. We also present Aida, an implementation of our analysis for use with the AspectJ language. Section 2 gives some background in Data-Flow Analysis, and we discuss possible modifications with respect to AO programs in Section 3. Section 4 presents Aida, and Section 5 concludes and looks to potential future work.

2 Data-Flow Analysis

Data-Flow Analysis (DFA)[6] is an ideal tool in determining this kind of indirect interference between aspects. DFA gives us the ability to see which data aspect advice modifies, and (crucially) trace the effects of that modification throughout the program, including its effect on other aspects. In this section we present the basic tenets of DFA which are necessary in order to understand our approach.

The classical *Definition-Use* analysis forms the basis of our approach. The idea behind this analysis is to find *Definition-Use chains* or *du-chains*, associations between an assignment to a variable and all its uses in a program. The def-use analysis is based on a classical data-flow analysis called Reaching Definitions Analysis[6]. Given a program point, the analysis returns the definitions which may have been made and not re-defined when the execution of the program reaches that point. Comparing these definitions with uses of variables at the program point enables us to determine du-chains, which are candidates for error propagation paths. Our approach to performing this inter-procedural analysis is an extension to the functional approach proposed by Shahir and Pnueli[7], which has an acceptable tradeoff between efficiency and accuracy.

The approach operates on an Inter-procedural Control Flow Graph (ICFG), which contains a control-flow graph (representing code statements as nodes and potential flow as edges) for each of the methods and aspect advices in the program. From this, an intra-procedural analysis computes a *transfer function* for each program point within a method, which represents the effect of the Reaching Definitions Analysis up to that point. As this happens within the method, it models which definitions reach the program point based on abstract initial values at the start of the method. For example, in Fig. 2, the transfer function at program point n_2 in method A is $\rho_2 = f_1$.

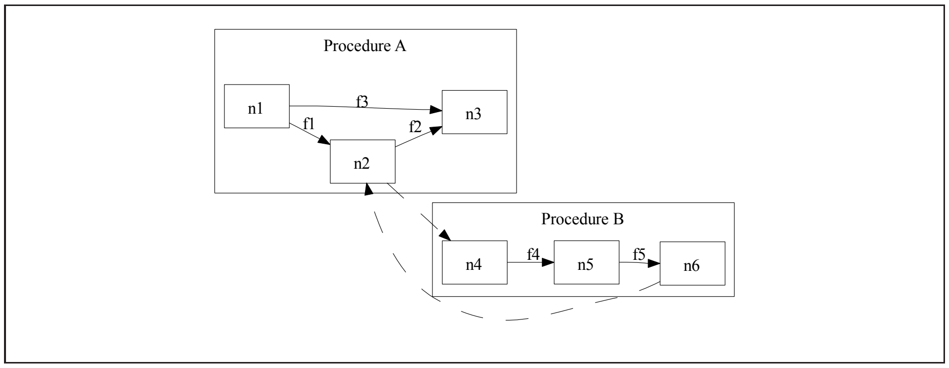


Fig. 2: Computing transfer functions

In order to model method calls, a special instance of a transfer function is created which defines the summary of calling a method - effectively the conjunction of the transfer functions at the exit points of each method. In Fig. 2, the summary transfer function ϕ_B modelling the effect of calling method B is $\phi_B = \rho_6 = f_5 \circ f_4$. This information is propagated bottom-up using a fixpoint calculation to determine the transfer functions for each method, taking calls into account. For example, the final transfer function at node n_3 is:

$$\begin{aligned}\rho_3 &= f_3 \wedge (f_2 \circ \phi_B \circ f_1) = f_3 \wedge (f_2 \circ \rho_6 \circ f_1) \\ &= f_3 \wedge (f_2 \circ f_5 \circ f_4 \circ f_1)\end{aligned}$$

The next step is to propagate real data-flow information in a top-down fashion, starting from `main()` methods with an empty set of reaching definitions. At this stage, information is only propagated to entry points of procedures and to call sites, which is possible because transfer functions based on abstract initial values have already been computed at these points. Again a fixpoint calculation is used to resolve any circular dependencies. In the above example, the real solution S_4 at node n_4 is $S_4 = \rho_4(\rho_2(\eta))$, where η is the data-flow information present at the beginning of procedure A.

Once this is done, it is trivial to calculate the results of the reaching definitions analysis on-demand, as both the concrete data-flow information at the beginning of each procedure and transfer functions for each program point within that procedure are available. Therefore, the analysis result at node n_5 is the result of the function $\rho_5(S_4)$, both elements of which have previously computed.

3 Transfer functions for advice

One consideration in applying this technique to AO programs is that of computing transfer functions for advice. In languages such as AspectJ, advice which applies *around* a joinpoint can be difficult to reason about independently of the base system, mainly due to the inability to discover what a `proceed()` statement could refer two. We present two main options:

Advices as methods Perhaps the simplest option is to perform the analysis after the aspect advice has been woven, and treat aspect advices identically to method calls. An *around* advice's `proceed()` statement would therefore be transformed to another call back to the advised procedure, and computing the transfer functions would proceed as normal.

Binding functions The disadvantage of the first option is that, as we perform the analysis after the advice has been woven, it is difficult to consider the effect of the advice independently of its binding. Therefore, there could be no partial analysis results associated with library aspects which include *around* advice, as it would be impossible to know the effect of a `proceed()` statement before weaving. One solution is to transform the *around* advice into $before$ and $after$ advice and treat it as two separate advices - however, real-world advices may well have multiple `proceed()` statements or have control-flow paths which bypass the `proceed()` instruction altogether (see Fig. 3).

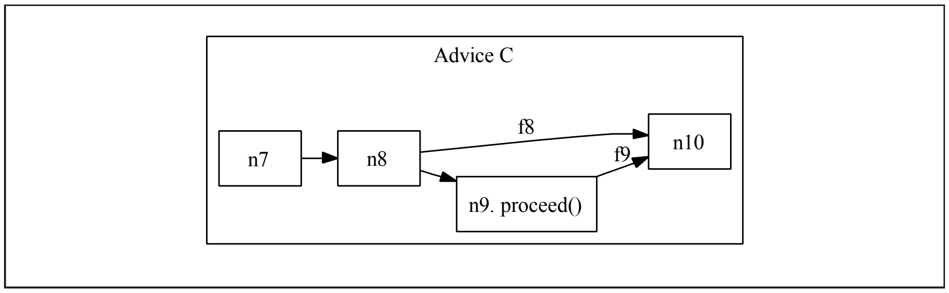


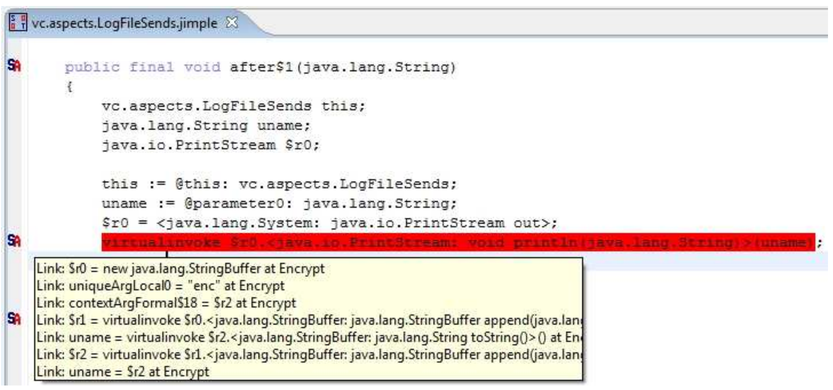
Fig. 3: Computing transfer functions for advice

Instead, then, we can compute a partial transfer function based on unknown bindings using a binding function ψ, which represents the effect of the `proceed()` statement. So for the advice in Fig. 3, the summary transfer function for the whole advice would be $\phi_C = \rho_{10} = f_9 \wedge (f_8 \circ \psi_8)$.

Using this formulation, then, we compute the summary transfer functions for methods without considering advices first. We then introduce binding information based on the possible joinpoints of each advice, and at each binding point, propagate the values of the binding function - namely, the call to the method which is being advised - to the advice such that a summary transfer function can be calculated. The fixpoint calculation is then re-run to propagate the effect of the advice to the rest of the system.

The main advantage of this approach is that partial analysis results can be pre-computed for aspect advice, which means that when an advice is used in a different context - as a library advice, for example, or at a different joinpoint - we no longer have to start from scratch in our analysis. As well as this, we may be able to infer generic properties of the advice - in the form of a categorisation of its behaviour, for example - which could then inform our later analysis.

4 Aida



```
public final void after$1(java.lang.String)
{
    vc.aspects.LogFileSends this;
    java.lang.String uname;
    java.io.PrintStream $r0;

    this := @this: vc.aspects.LogFileSends;
    uname := @parameter0: java.lang.String;
    $r0 = <java.lang.System: java.io.PrintStream out>;
    virtualinvoke $r0.<java.io.PrintStream: void println(java.lang.String)>(uname);
```

```
Link: $r0 = new java.lang.StringBuffer at Encrypt
Link: uniqueArgLocal0 = "enc" at Encrypt
Link: contextArgFormal$18 = $r2 at Encrypt
Link: $r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lan
Link: uname = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.String toString()> () at En
Link: $r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lan
Link: uname = $r2 at Encrypt
```

Fig. 4: Aida in Eclipse

We have implemented the simpler version of the algorithm presented above - namely, that which treats advices as method calls - as an extension to the abc compiler[2] for AspectJ called Aida - Aspect Interference Detection Analysis. The goal of Aida is to provide developers with a summary and visualisation of the potential interactions within the system, such that they can develop and evolve a strategy for fault-tolerance.

abc is built on the Soot[8] framework, and so transforms the woven bytecode into an intermediate representation called Jimple, which allows inspection and analysis of the code. On running the analysis, Aida produces annotated Jimple code which shows the links between advices in terms of which statements are directly affected by other advice being present in the system (see Fig. 4 for how this looks to the user in Eclipse[4]). The analysis also works at any specified depth of transitivity - that is, it can chain any given number of definition-use chains together to find even more subtle interactions. Aida also produces a visual representation of interactions by means of an interaction graph, shown in Fig. 5. Here we have trimmed the graph to show the results of analysing the example

Version Control system presented in Section 1, and the interaction between the `Encrypt` and `LogFileSends` aspects is clearly shown by tracing the red arrows.

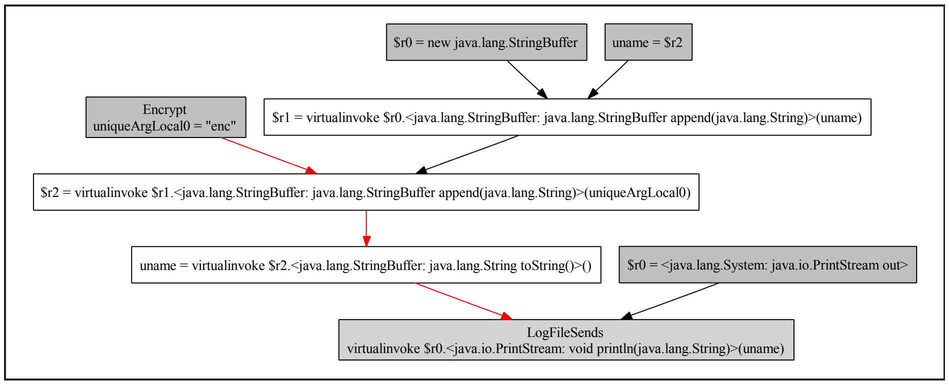

$r0 = new java.lang.StringBuffer
uname = $r2
Encrypt
uniqueArgLocal0 = "enc"
$r1 = virtualinvoke $r0.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>(uname)
$r2 = virtualinvoke $r1.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>(uniqueArgLocal0)
uname = virtualinvoke $r2.<java.lang.StringBuffer: java.lang.String toString()>()
$r0 = <java.lang.System: java.io.PrintStream out>
LogFileSends
virtualinvoke $r0.<java.io.PrintStream: void println(java.lang.String)>(uname)

Fig. 5: AIDA-generated interaction graph

5 Conclusion and future work

In this article we have motivated the adaptation of summary-based DFA for discovery of interactions between aspects. We have motivated the importance of this interaction analysis with respect to its application to fault-tolerant systems - it can reveal error-propagation paths and outline containment units; give hints to the developer on where modules need hardening with assertions and the like; and show how aspects for fault-tolerance can be impacted by other aspects in the system. We have presented AIDA, an implementation of this analysis which works on AspectJ programs. We are currently extending our implementation to incorporate the more AO-specific algorithm described above, which treats advices differently to methods and is able to pre-compute some analysis results such that they can be reused. We have also developed a categorisation schema which describes the interactions between aspects in a more meaningful way, and we hope to incorporate all of this into an Eclipse plugin.

References

1. AspectJ. Home page of the AspectJ project. `http://eclipse.org/aspectj`.
2. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.
3. Contract4Java. Homepage of the C4J project. `http://www.contract4j.org`.

4. Eclipse. Homepade of the Eclipse project. `http://eclipse.org`.
5. Jorg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.
6. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*, chapter 2, pages 35–135. Springer, 2nd edition, 2005.
7. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
8. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

Rigorous Development of Ambient Campus Applications that can Recover from Errors

Budi Arief, Alexei Iliasov, and Alexander Romanovsky

School of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne NE1 7RU, England.
{L.B.Arief, Alexei.Iliasov, Alexander.Romanovsky}@newcastle.ac.uk

Abstract. In this paper, we discuss a new method for developing fault-tolerant ambient applications. It supports stepwise rigorous development producing a well structured design and resulting in disciplined integration of error recovery measures into the resulting implementation.

1 Introduction

Ambient campus is a loosely defined term to describe *ambient intelligence* (AmI)[1] systems in an educational (university campus) setting. As such, ambient campus applications are tailored to support activities typically found in the campus domain, including – among others, delivering lectures, organising meetings, and promoting collaborations among researches and students.

This paper presents our work in the development of the ambient campus case study within the RODIN project [1]. This EU-funded project, led by the School of Computing Science of Newcastle University, has an objective to create a methodology and supporting open tool platform for the cost-effective rigorous development of dependable complex software systems and services. In the RODIN project, the ambient campus case study acts as one of the research drivers, where we are investigating how to use formal methods combined with advanced fault-tolerance techniques in developing highly dependable AmI applications. In particular, we are developing modelling and design templates for fault-tolerant, adaptable and reconfigurable software. This case study consists of several working ambient applications (referred to as *scenarios*) for supporting various educational and research activities.

Software developed for AmI applications need to operate in an unstable environment susceptible to various errors and unexpected changes (such as network disconnection and re-connection) as well as delivering context-aware services. These applications tend to rely on the *mobile agent paradigm*, which supports system-structuring using decentralised and distributed entities (*agents*) working together in order to achieve their individual aims. Multi-agent applications pose

[1] Ambient intelligence is a concept developed by the Information Society Technologies Advisory Group (ISTAG) to the European Commission's DG Information Society and the Media, where humans are surrounded by unobtrusive computing and networking technology to assist them in their activities.

many challenges due to their openness, the inherent autonomy of their components (i.e. the agents), the asynchrony and anonymity of the agent communication, and the specific types of faults they need to be resilient to. To address these issues, we developed a framework called CAMA *(Context-Aware Mobile Agents)* [2], which encourages disciplined development of open fault-tolerant mobile agent applications by supporting a set of abstractions ensuring exception handling, system structuring and openness. These abstractions are backed by an effective and easy-to-use middleware allowing high system scalability and guaranteeing agent compatibility. More details on CAMA and its abstractions can be found in [2–4]. The rest of this paper outlines our case study scenarios (Section 2), discusses important fault-tolerance issues in AmI systems (Section 3), and describes our design approach (Section 4).

2 Case Study Scenarios

We have so far implemented two scenarios for our ambient campus case study using the CAMA framework as the core component of the applications. The first scenario (*ambient lecture*) deals with the activities carried out by the teacher and the students during a lecture – such as questions and answers, and group work among the students – through various mobile devices (PDAs and smartphones). The second scenario (*presentation assistant*) covers the activities involved in giving or attending a presentation/seminar. The presenter uses a PDA to control the slides during their presentation and they may receive 'quiet' questions on the topic displayed on the slide from the audience. Each member of the audience will have the current slide displayed on their PDA, which also provides a feature to type in a question relevant to that slide.

We are now working on a more challenging scenario which involves greater agent mobility as well as location specific services. Agents may move physically among multiple locations (rooms), and depending on the location, different services will be provided.

In this scenario – we call it *student induction assistant* scenario – we have new students visiting the university campus for the first time. They need to register to various university departments and services, which are spread on many locations on campus, but they do not want to spend too much time looking for offices and standing in queues. They much prefer spending their time getting to know other students and socialising. So they delegate the registration process to their personalised software agent, which then visits virtual offices of various university departments and institutions, obtains the necessary information for the registration, and makes decisions based on the student's preferences. The agent also records pieces of information collected during this process so that the students can have all the details about their registration.

Unfortunately, not all the registration stages can be handled automatically. Certain steps require personal involvement of the student, for example, signing paperwork in the financial department and manually handling the registration in some of the departments which do not provide fully-featured agent able to handle the registration automatically. To help the students to go through the rest of registration process, their software agent creates an optimal plan for visiting different university departments and even arranges appointments when needed.

Walking around on the university campus, these new students pass through *ambients* – special locations providing context-sensitive services (see Figure 1). An ambient has sensors detecting the presence of a student and a means of communicating to the student. An ambient gets additional information about students nearby by talking to their software agent. Ambients help students to navigate within the campus, provide information on campus events and activities, and assist them with the registration process. The ambients infrastructure can also be used to guide students to safety in case of emergency, such as fire.

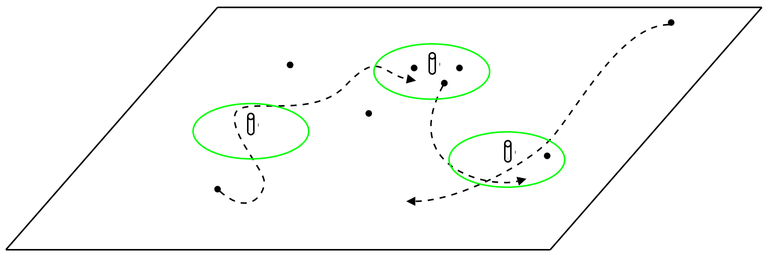


Fig. 1. *Student induction assistant* scenario: the dots represent free roaming student agents; the cylinders are static infrastructure agents (equipped with detection sensors); and the ovals represent *ambients* – areas where roaming agents can get connection and location-specific services.

3 Challenges in Developing Fault-Tolerance Ambient Intelligence Systems

Developing fault-tolerant ambient intelligence systems is not a trivial task. There are many challenging factors to consider; some of the most important ones are:

- *Decentralisation and homogeneity*
  Multi-agent systems are composed of a number of independent computing nodes. However, while traditional distributed systems are *orchestrated* – explicitly, by a dedicated entity, or implicitly, through an implemented algorithm – in order to solve a common task, agents in an ambient system decide *independently* to *collaborate* in order to achieve their individual goals. In other words, ambient systems do not have inherent hierarchical organisation. Typically, individual agents are not linked by any relations and they may not have the same privileges, rights or capabilities.
- *Weak Communication Mechanisms*
  Agent systems commonly employ communication mechanisms which provide

very weak, if any, delivery and ordering guarantees. This is important from the implementation point of view as agent systems are often deployed on wearable computing platforms with limited processing power, and they tend to use unreliable wireless networks for communication means. This makes it difficult to distinguish between a crash of an agent, a delay in a message delivery and other similar problems caused by network delay. Thus, a recovery mechanism should not attempt to make a distinction between network failures and agent crashes unless there is a support for this from the communication mechanism.

– *Autonomy*
  During its lifetime, an agent usually communicates with a large number of other agents, which are developed in a decentralised manner by independent developers. This is very different from the situation in classical distributed system where all the system components are part of a closed system and thus fully trusted. Each agent participating in a multi-agent application tries to achieve its own goal. This may lead to a situation where some agents may have conflicting goals. From recovery viewpoint, this means that no single agent should be given an unfair advantage. Any scenarios where an agent controls or prescribes a recovery process to another agent must be avoided.
– *Anonymity*
  Most agent systems employ anonymous communication where agents do not have to disclose their names or identity to other agents. This has a number of benefits: agents do not have to learn the names of other agents prior to communication; there is no need to create fresh names nor to ensure naming consistency in the presence of migration; and it is easy to implement group communication. Anonymity is also an important security feature - no one can sense an agent's presence until it produces a message or an event. It is also harder to tell which messages are produced by which agent. For a recovery mechanism, anonymity means that we are not able to explicitly address agents which must be involved in the recovery. It may even be impossible to discover the number of agents that must be involved. Even though it is straightforward to implement an exchange for agents names, its impact on agent security and the cost of maintaining consistency usually outweigh the benefits of having named-agents.
– *Message Context*
  In sequential systems, recovery actions are attached to certain regions, objects or classes which define a context for a recovery procedure. There is no obvious counterpart for these structuring units in asynchronously communicating agents. Agent produces messages in a certain order, each being a result of some calculations. When the data sent along with a message cause an exception in an agent, the agent may want to notify the original message producer, for example, by sending an exception. When an exception arrives at the message producer (which is believed to be the source of the problem), it is possible that the agent has proceeded with other calculations and the context in which the message was produced is already destroyed. In addition, an agent can disappear due to migration or termination.

– *Message Semantics*
  In a distributed system developed in a centralised manner, semantics of values passed between system components is fixed at the time of the system design and implementation. In an open agent system, implementation is decentralised and thus the message semantics must be defined at the stage of a multi-agent application design. If an agent is allowed to send exceptions, the list of exceptions and their semantics must also be defined at the level of an abstract application model. For a recovery mechanism, this means that each agent has to deal only with the exception types it can understand, which usually means having a list of predefined exceptions.

We have to take these issues into account when designing and developing fault-tolerant AmI systems. In the following section, we outline the design approach that can be used for constructing ambient campus case study scenarios.

4 Design Approach

In our previous work, we have developed several case study scenarios [5, 6, 4]. In these scenarios, we focused on the implementation aspects and the general problems of applying formal methods in ambients systems. In our new case study scenario (*student induction assistant* scenario – see Section 2), we shift our focus from implementation to design to validate our formal development approach.

The new scenario will be modelled using Event-B formalism [7]. Our intention is to have a fairly detailed model which covers issues such as communication, networking failures, proactive recovery, liveness, termination, and migration.

4.1 Overview

We are using *design patterns*, *refinement patterns*, and *mobility modelling* in developing the student induction assistant scenario.

Design patterns are described using a natural language and act as guide in formal development. Typically, design patterns are narrowly focused and can be applied only within a given problem domain. We have developed a set of design patterns that are specific for ambient systems [8]. These patterns are inspired by the architecture of the CAMA system and help us to formally design a system which can be implemented on top of the CAMA framework.

Refinement patterns are rules describing a transformation of an abstract model into a more concrete one. These patterns are specified in an unambiguous form and can be mechanically applied using a special tool, which is a plugin to the RODIN Event-B platform [7, 1]. Some refinement patterns can be seen as a possible implementation of the design patterns. Others are simply useful for the transformation steps, which we believe are common in this kind of systems.

We are planning to apply the Mobility Plugin [9] for verification of dynamic properties, such as liveness, termination, and mobility-related properties. Using this plugin, we are able to extend the Event-B model with process algebraic description of dynamic behaviour, agent composition and communication scenarios. The plugin handles additional proof-obligation using a built-in model checker. It also includes an animator for visual interactive execution of formal models.

4.2 Application to the Scenario

To proceed any further, we need to agree on same major design principles, identify major challenges and outline the strategy for finding the solution.

To better understand the scenario, we apply the *agent metaphor*. The agent metaphor is a way to reason about systems (not necessarily information systems) by decomposing it into agents and agent subsystems. In this paper, we use term *agent* to refer to a component with independent thread of control and state and the term *agent system* to refer to a system of cooperative agents.

From agent systems' viewpoint, the scenario is composed of the following three major parts: physical university campus, virtual university campus and ambients. In physical university campus, there are students and university employees. Virtual campus is populated with student agents and university agents. Ambients typically have a single controlling agent and a number of visiting agents. These systems are not isolated, they interact in a complex manner and information can flow from one part into another.

However, since we are building a distributed system, it is important to get an implementation as a set of independent but cooperative components (agents). To achieve this, we apply the following design patterns:

agent decomposition During the design, we will gradually introduce more agents by replacing abstract agents with two or more concrete agents.

super agent It is often hard to make a transition from an abstract agent to a set of autonomous agents. What before was a simple centralised algorithm in a set of agents must now be implemented in a distributed manner. To aid this transition, we use *super agent* abstraction, which controls some aspects of the behaviour of the associated agents. Super agent must be gradually removed during refinement as it is unimplementable.

scoping Our system has three clearly distinguishable parts: physical campus, virtual campus and ambients. We want to isolate these subsystems as much as possible. To do this, we use the scoping mechanism, which temporarily isolates cooperating agents. This is a way to achieve the required system decomposition. The isolation properties of the scoping mechanism also make it possible to attempt autonomous recovery of a subsystem.

orthogonal composition As mentioned above, the different parts of our scenario are actually interlinked in a complex manner. To model this connections, we use the *orthogonal composition* pattern. In orthogonal composition, two systems are connected by one or more shared agents. Hence, information from one system into another can flow only through the agent states. We will try to constrain this flow as much as possible in order to obtain to a more robust system.

locations definition To help students and student agents navigate within the physical campus and the virtual campus, we define location as places associated with a particular agent type.

decomposition into roles The end results of system design is a set of agent roles. To obtain role specifications, we decompose scopes into a set of roles.

4.3 Fault-Tolerance Mechanism

We are going to address the fault-tolerance issues at three levels: architectural, modelling and implementation.

At the architectural level, we use the agent metaphor to introduce fault-tolerance properties such as redundancy (spawning an agent copy to survive an agent crash) and diversity (having the same service provided by independently implemented agents).

At the modelling level, we apply the assumptions mechanism [10] along with FT-specific design patterns. The assumptions mechanism helps us to build more robust agent applications and it has two different styles. In the first one, it is assumed that certain undesirable events are not going to happen during some activity. If such an event happens, the whole activity is aborted. In the second style, through negotiations, agents agree to temporarily restrict their behaviour and cooperate in a simpler environment. Design patterns help developers to introduce some common fault-tolerance techniques when modelling an agent system. These techniques range from abstract system-level patterns to very specific agent-level patterns dealing with specific faults.

Early on, we have extended the blackboard communication pattern [11] with nested scopes and exception propagation [12]. These two extensions are essentially the implementation techniques for recovery actions introduced during the modelling stage. We also rely extensively on reactive agent architecture. This has two immediate benefits: its implementation style matches the modelling style of Event-B, and the recovery of multi-threaded agents becomes similar to that of the asynchronous reactive architecture.

5 Conclusion

This paper provides an outline of the work that we have carried out in developing fault-tolerant ambient applications. We use design patterns, refinement patterns, and mobility modelling during the design process. By using these techniques, we aim to validate the formal development approach that we take in developing more robust and fault-tolerant ambient applications.

We plan to demonstrate our approach through an ambient campus *student induction assistant* scenario. We are currently developing an agent-based system for this scenario, using the CAMA framework and middleware [2] that we have previously developed as the centre piece of the system. This will be augmented with sensors for providing location specific services, as well as fault-tolerance mechanism at the architectural, modelling and implementation levels.

6 Acknowledgements

This work is supported by the IST RODIN Project [1]. A. Iliasov is partially supported by the ORS award (UK).

References

1. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ (Last accessed: 17 May 2007)
2. Arief, B., Iliasov, A., Romanovsky, A.: On Developing Open Mobile Fault Tolerant Agent Systems. In Choren, R., et al., eds.: SELMAS 2006, LNCS 4408. Springer-Verlag (2007) 21–40
3. Iliasov, A.: Implementation of Cama Middleware. http://sourceforge.net/projects/cama (Last accessed: 17 May 2007)
4. Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. Technical report, CS-TR-993, School of Computing Science, Newcastle University (Dec 2006)
5. Arief, B., Coleman, J., Hall, A., Hilton, A., Iliasov, A., Johnson, I., Jones, C., Laibinis, L., Leppanen, S., Oliver, I., Romanovsky, A., Snook, C., Troubitsyna, E., Ziegler, J.: Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
6. Troubitsyna, E., ed.: Rodin Deliverable D8: Initial Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2005)
7. Metayer, C., Abrial, J.R., Voisin, L.: Rodin Deliverable 3.2: Event-B Language. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
8. Laibinis, L., Iliasov, A., Troubitsyna, E., Romanovsky, A.: Formal Approach to Ensuring Interoperability of Mobile Agents. Technical report, CS-TR-989, School of Computing Science, Newcastle University, UK. October (2006)
9. Butler, M., ed.: Rodin Deliverable D11: Definition of Plug-in Tools. Project IST-511599, School of Computing Science, University of Newcastle (2005)
10. Iliasov, A., Romanovsky, A.: Choosing Application Structuring and Fault Tolerance Using Assumptions, To be presented at DSN 2007 (2007)
11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System Of Patterns. West Sussex, England: John Wiley & Sons Ltd. (1996)
12. Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: LNCS 4119. (2006) 181–199

SERENITY: A Generic framework for Dependability Construction

A Patterns Oriented Approach

Ayda Saidane
[1] University of Trento, via sommarive 14 Povo, Italy
ayda.saidane@unitn.it

Abstract. The applications are becoming more sophisticated and guaranteeing their correct functioning is an everyday more difficult task. The apparition of new environments characterized by the combination of heterogeneity, mobility and dynamism makes it even more difficult the development of trustworthy and dependable systems. We present in this paper the *SERENITY* Framework as a generic construction framework for dependable and fault tolerant systems.

Keywords: dependability, fault tolerance, patterns, construction framework.

1 Introduction

The combination of heterogeneity, mobility, dynamism, sheer number of devices, along with the growing demands for dependability and trustworthiness, is going to make the dependability provision for complex systems increasingly difficult to achieve with existing solutions, engineering approaches and tools.

Applications are becoming more and more complex. Their development and deployment need to be rigorously monitored considering both functional requirements and dependability requirements. This complexity makes it impossible, even for the most experienced and knowledgeable dependability experts, to foresee all possible situations and interactions, which may arise, and therefore create suitable solutions to address the users' dependability requirements. Additionally dependability experts will be faced with pieces of software, communication infrastructures and hardware devices not under their control. Thus, approaches based on the application-level dependability will not be sufficient to fulfill the requirements of the new applications.

The SERENITY project [1] aims to develop a framework to support the automated integration, configuration, monitoring and adaptation of security and dependability (S&D) in Ambient Intelligence (AmI) ecosystems. The purpose of this paper is to demonstrate the efficiency of the SERENITY approach during the development and deployment of dependable and fault tolerant systems.

The paper is structured as follows. Section 2 presents some basic definitions of dependability and fault tolerance. Section 3 describes the SERENITY Framework and Section 4 explains the development of fault tolerant systems using SERENITY. Section 5 concludes the paper and discusses some future work.

2 Definitions

A dependable system [2] is defined as one that is able to deliver a service that can justifiably be trusted; attributes of dependability include availability (readiness for correct service), reliability (continuity of correct service), confidentiality (prevention of unauthorized disclosure of information), and integrity (absence of improper system state alterations). There exist four mechanisms for developing dependable systems: 1) fault prevention; 2) fault tolerance; 3) fault removal; 4) fault forecasting.

Fault Tolerance (Fig.1) is carried out via error detection and system recovery, which aims at transforming a system state that contains errors into a state without detected errors or faults that can be activated.. It consists of two steps: error handling and fault handling. There are two main strategies for fault tolerance: 1) detection and recovery or 2) fault masking. The first category corresponds to the deployment of error handling on demand followed by fault handling. Fault masking results from the systematic usage of compensation.

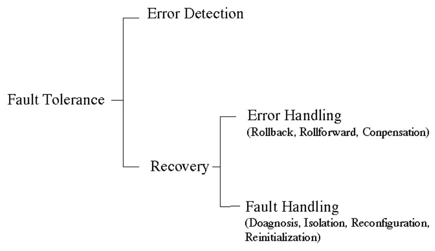

Fault Tolerance
Error Detection
Recovery
Error Handling
(Rollback, Rollforward, Conpensation)
Fault Handling
(Doagnosis, Isolation, Reconfiguration, Reinitialization)

Figure 1. Fault Tolerance Techniques

Current systems are so complex that it is impossible to identify and correct all their faults before they are put in operation. Thus, the deployment of fault tolerance techniques is becoming more and more required in the construction of dependable and survivable systems. Actually, the application designers are not generally familiar with foundation and techniques of dependability and in particular fault tolerance. In the same way, dependability experts are not necessary familiar with software engineering best practices. Thus, separating the two issues would benefit on the trustworthiness of the system-to-be. The approach would be to create a repository of dependability solutions that can be used by application designers to architect the system.

3 SERENITY Framework

The framework was developed in the context of the SERENITY project [1]. The framework consists of two parts: design time framework and runtime framework. In this section we are going to present an overview of the SERENITY approach [3,4] that will be customized for the development of adaptive fault tolerant systems.

3.1 Basic Concepts

Before presenting the SERENITY framework, it is necessary to define the basic concepts. In particular, we introduce the notions of: S&D patterns, S&D integration schemes and S&D implementation.

- *S&D pattern*: A self-contained description of an S&D Solution, meaning that it does not refer to (or depend on) other S&D Solutions.
- *S&D integration scheme*: it is a special type of S&D Patterns that is used to represent ways of correctly combining other S&D Patterns.
- *S&D implementation*: it represents working S&D Solutions. These solutions are made accessible to applications thanks to the SERENITY Runtime Framework.
- *Context*: it is defined as a set of elements that are recorded and tracked by the SERENITY Runtime Framework in order to evaluate the state of the framework and assist the S&D Manager in choosing the appropriate patterns or undertake pre-active or pro-active actions.

3.2 SERENITY Design time Framework

The SDF consists of a set of modeling and validation tools intended to help security engineers to design generic and correct solutions and to help application designers to select and customize the appropriate security solutions for their applications.

The framework includes the following elements:

- *S&D Patterns' Library*: it contains organizational, workflow and network patterns that describe, at different levels of abstraction, security solutions that solve specific security problems. The patterns not only hold the description of the solution but also how to use it, the conditions needed for its application and how to monitor the correctness of the process.
- *Modeling and verification tools*: there are 5 tools for specifying solutions at different levels of abstraction: Business & Organizational Tool, Workflow Static Analysis Tool, Net Pattern Static Analysis Tool, *S&D Pattern* Specification Tool, S&D Pattern Management Tool and

3.3 SERENITY Runtime Framework

The Run-Time Framework has been designed to allow different security requirements to be fulfilled through a number of available patterns.

The SRF architecture includes the following components:

- S&D Patterns' Library
- *S&D Manager*: it implements the logic of S&D patterns by combining application requirement, available S&D patterns and current system context in order to choose the appropriate implementation that needs to be activated. The S&D Manager is the component responsible for activating and deactivating pattern implementations and will also be accountable for taking necessary actions (based on the monitoring rules) when informed by the Monitor Service of a violation.
- *Event Manager*: it is responsible for collecting events from the Event Collector,
- *Context Manager*: it is in charge of the context, as previously defined for the SRF environment

- *Monitor Service*: it is in charge of analyzing events and mapping them onto the monitoring rules, in order to identify any violations and consequently inform the S&D Manager.

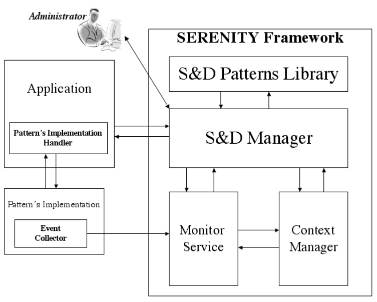


Figure 2. An overview on the SERENITY Architecture

4 SERENITY framework for the development of dependable and fault tolerant systems

As presented in section 2, the provision of fault tolerance requires three elements: 1/ error detection; 2/ errors handling and 3/ fault handling. This section aims to illustrate how the SERENITY framework facilitates the development of fault tolerant systems.

4.1 Patterns' Repository

As mentioned before, the main interest of SERENITY at design time is the provision of validated dependability solutions and mechanisms to combine them. Thus, we are going to adopt a component-based development process where the crucial point is choosing the appropriate components fulfilling the dependability requirements of the system-to-be.

The patterns included in the repository, are characterized by different information that are used for selecting the appropriate pattern at both design time and runtime. The patterns' description includes the dependability properties provided by the solution, the fault model, the validation process, the pattern's provider and the monitoring rules used at runtime to monitor the status of the component. The patterns that are intended for runtime use, one or more implementations should be defined. They will be activated for reconfiguration purposes when a failure is detected on one of the components.

The SERENITY patterns are specified according to Patterns Specification Language [3] but we are going to present some examples using the natural language description for more clarity. The proposed description includes:

- context description,
- dependability properties,
- a high level solution description.

The knowledge of dependability experts will be captured in the patterns and used by the application designers to develop dependable and fault tolerant applications. The applications' designers have only to express properly their dependability requirements and do not need to have advance knowledge on dependability principles to be able to create high quality dependable systems.

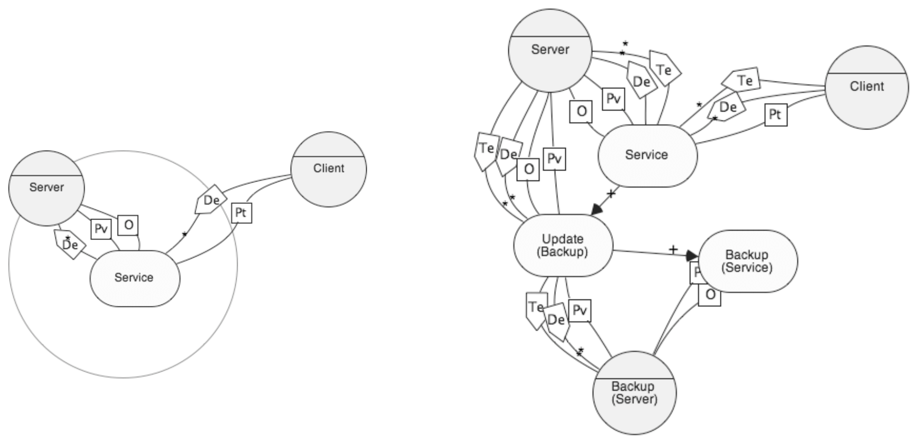
In the following, we present some examples of dependability patterns. The examples are described here in natural language for more clarity. In this section we present two patterns: 1) Rollback through Primary Backup Redundancy; 2) Fault masking through active redundancy.

Example 1: Rollback through Primary Backup Redundancy

Context: At organizational level, the context is presented as the set of agents, their goal and their resources. It is also needed to describe the relations between the agents such as delegation and trust. Figure 3 presents the pattern's context modeled with Tropos [5]. It consists of two agents a server and a client. The client delegates on execution the fulfillment of the goal service. There is no trust between the client and the server that can be justified by the unsatisfactory service failure's rate.

Provided Properties: Availability (Server,Client,Service) : *Server* must provide acceptable *Service* to *Client*. The property can be specified qualitatively or quantitatively.

Solution: In order to establish the trust relation between the client and the server, the latter has to improve the availability of the service. The proposed solution is to adopt the rollback technique by adding a backup that is updated after each request (Fig. 3).



Context **Solution**

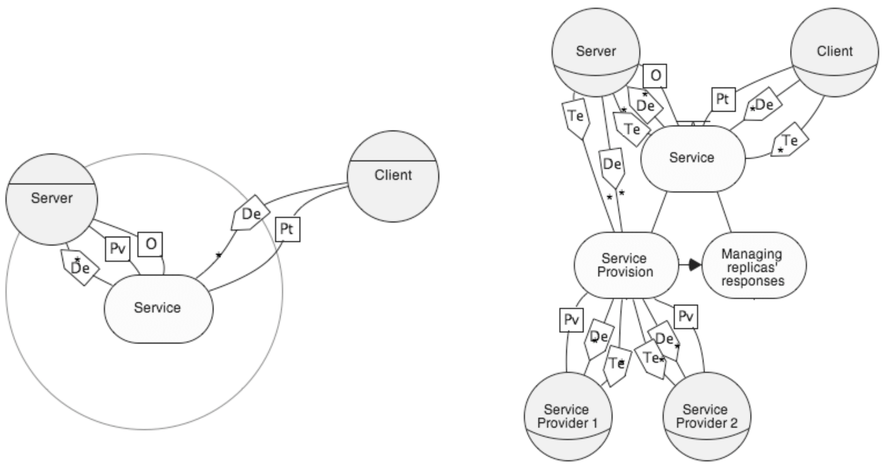
Figure 3. Tropos models of Context/Solution for Primary Backup pattern

Example 2: Fault masking through Active Redundancy

Context: it corresponds to the same context as the previous pattern.

Provided Properties: Availability (*Server*,*Client*,*Service*), Reliability (*Server*,*Client*, *Service*),: *Server* must improve the quality of *Service* provided to *Client*.

Solution: In order to establish the missing trust relation between the client and the server, the latter has to improve the availability and reliability of the service. Active redundancy constitutes an appropriate technique providing the requested (Fig. 4).



Context **Solution**

Figure 4. Tropos models of Context/Solution for active redundancy pattern

4.2 SERENITY Runtime

The SERENITY framework enables the creation of adaptive fault tolerant systems through the patterns' repository and the runtime framework. When analyzing the properties provided by the runtime serenity framework, we realize that all serenity enabled systems will be adaptive and fault tolerant. In fact, the framework itself implements both detection and recovery mechanisms.

- Detection: the *Monitoring Manager* (MM) implements the error detection step. In fact, it is in charge of detecting the failures of the active patterns and to raise alerts to the *Security Manager* (SM). In fact, the patterns' implementations include *event captures* that capture the events referenced in the monitoring rules and notify consequently the MM.
- Recovery: the SM is responsible for ensuring the recovery of the system when receiving MM's alerts. In order, to bring back the system to a correct state, the SM selects new patterns to be applied as counter-measurements to the detected failures. The error handling is realized according to the architecting patterns chosen at design time but it is also possible to change from an error handling technique to another at runtime under certain conditions and if the necessary resources for such reconfiguration are available. The fault handling is implemented by the application of the new patterns. This operation allows the reconfiguration of the system avoiding the reactivation of the detected faults.

6 Conclusions

We have presented in this paper the SERENITY approach for developing dependable and fault tolerant applications. The key issue is to capture the knowledge of dependability experts into validated patterns to be used by application designers when developing the system. The runtime framework provides monitoring and recovery mechanisms facilitating the development of fault tolerant systems.

Acknowledgments. This work has been partly supported by the project EU-IST-IP SERENITY- 27587.

References

1. http://www.serenity-project.org
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE TDSC. 1(1) (2004)
3. SERENITY Project, A5 Deliverable – A5.D2.1 – Patterns and Integration Schemes Languages. http://www.serenity-project.org
4. SERENITY Project, A6 Deliverable – A6.D3.1 – Specification of SERENITY Architecture http://www.serenity-project.org
5. Giorgini P., Massacci F. and Zannone N. (2005). Security and Trust Requirements Engineering. In Foundations of Security Analysis and Design III - Tutorial Lectures, LNCS 3655, pages 237-272. Springer-Verlag GmbH.

Documenting the Progress of the System Development*

Marta Pląska[1], Marina Waldén[1] and Colin Snook[2]

[1] Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] University of Southampton, Southampton, SO17 1BJ, UK

Abstract. While UML gives an intuitive image of the system, formal methods provide the proof of its correctness. We can benefit from both aspects by combining UML and formal methods. Even for the combined method we need consistent and compact description of the changes made during the system development. In the development process certain design patterns can be applied. In this paper we introduce progress diagrams to document the design decisions and detailing of the system in successive refinement steps. A case study illustrates the use of the progress diagrams.

Keywords: Progress diagram, Statemachines, Stepwise development, Refinement, UML, Event-B, Action Systems, Graphical representation.

1. Introduction

For complex systems the stepwise development approach of formal methods is beneficial, especially considering issues of ensuring the correctness of the system. However, formal methods are often difficult for industrial practitioners to use. Therefore, they need to be supported by a more approachable platform. The Unified Modelling Language (UML) is commonly used within the computer industry but, currently, mature formal proof tools are not available. Hence, we use formal methods in combination with the semi-formal UML.

For a formal top-down approach we use the Event B formalism [10] and associated proof tool to develop the system and prove its correctness. Event-B is based on Action Systems [4] as well as the B Method [1], and is related to B Action Systems [17]. With the Event-B formalism we have tool support for proving the correctness of the development. In order to translate UML models into Event B, the UML-B tool [14] is used. UML-B is a specialisation of UML that defines a formal modelling notation combining UML and B.

The first phase of the design approach is to state the functional requirements of the system using natural language illustrated by various UML diagrams, such as statechart diagrams and sequence diagrams that depict the behaviour of the system. The system is built up gradually in small steps using superposition refinement [3, 9]. We rely on patterns in the refinement process, since these are the cornerstones for creating *reusable* and *robust* software [2, 7]. UML diagrams and corresponding Event B code are developed for each step simultaneously. To get a better overview of the design process, we introduce the *progress diagram*, which illustrates only the refinement-affected parts of the system and is based on statechart diagrams. Progress diagrams support the construction of large software systems in an incremental and layered fashion. Moreover, they help to master the

* Work done within the RODIN-project, IST-511599

complexity of the project and to reason about the properties of the system. We illustrate the use of the diagrams with a case study.

Design patterns in UML and B have been studied previously. Chan et al. [6] work on identifying patterns at the specification level, while we are interested in refinement patterns. The refinement approach on design patterns was presented by Ilič et al. [8]. They focused on using design patterns for integrating requirements into the system models via model transformation. This was done with strong support of the Model Driven Architecture methodology, which we do not consider in this paper. Instead we provide an overview of the development from the patterns.

The rest of the paper is organised as follows. In Section 2 we give an overview of our case study, Memento, from a general and functional perspective. An abstract specification is presented as a graphical, as well as a formal representation in Section 3. Section 4 describes stepwise refinement of the system and introduces the idea of progress diagrams. The system development is analysed and illustrated with the progress diagrams relying on the case study. We conclude with some general remarks in Section 5.

2. Case study – Memento application

The Memento application [13] that is used as a case study in this paper is a commercial application developed by Unforgiven.pl. It is an organiser and reminder system that has lately evolved into an internet-based application. Memento is designed to be a framework for running different modules that interact with each other.

In the distributed version of Memento every user of the application must have its own, unique identifier, and all communication is done via a central application server. In addition to its basic reminder and address book functions, Memento can be configured with other function modules, such as a simple chat module. Centralisation via the use of a server allows the application to store its data independently of the physical user location, which means that the user is able to use his own Memento data on any computer that has access to the network.

The design combines the web-based approach of internet communicators and an open architecture without the need for installation at client machines. During its start-up the client application attempts to *connect to a central server*. When the connection is established, the *preparation phase* begins. In this phase the user provides his/her unique identifier and password for authorisation. On successful login the server responds by sending the data for the account including a list of contacts, news, personal files etc. Subsequently the *application searches for modules* in a working folder and attempts to initialise them, so that the user is free to run any of them at any time. During execution of the application, *commands from the server and the user are processed* at once. Memento translates the requested actions of the user to internal commands and then handles them either locally or via the server. Upon a termination command Memento *finalises all the modules*, saves the needed data on the server, logs out the user and *closes the connection*. To minimise the risk of losing data, in case of fatal error, this termination procedure is also part of the fatal exception handling routine.

3. Abstract specification

3.1. UML-models

We use the Unified Modelling Language™ (UML) [5], as a way of modelling not only the application structure, behaviour, and architecture of a system, but also its data structure. UML can be used to overcome the barrier between the informal industry world and the formal one of the researchers. It provides a graphical interface and documentation for every stage of the (formal) development process. Although UML offers miscellaneous diagrams for different purposes, we focus on two types of these in our paper: sequence diagrams and statechart diagrams.

The sequence diagram can be used within the development of the system to show the interactions between objects and in which order these interactions occur. The diagram can be derived directly from the requirements. Furthermore, it can give information on the transitions of the statemachines. The interaction between entities in the sequence diagram can be mapped to self-transitions on the statechart diagram to model communication between the modelled entity and its external entities.

In our case study the external entities are the server and the users interacting with the modelled entity Memento. An example of a sequence diagram for the application is given in Fig. 1, where part of the requirements (the emphasized text in Section 2) concerning the server connection and the program preparation phase is shown. In the diagram we describe the initialisation phase of the system, which consists of establishing a connection (in the connection phase) and then preparing the program (in the preparation phase). The first of these actions requires the interaction with the server via an internet connection. The second action requires user interaction as well. The described interaction (in Fig. 1) is transferred to a statechart diagram as transition *tryInit* (to later be refined to the transitions *tryConn* and *tryPrep* as in section 4.1).

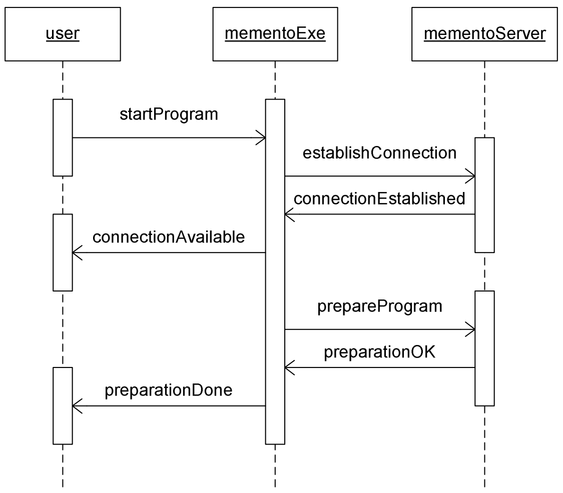


Fig. 1. Sequence diagram presenting the object interaction in the *initialisation phase*

In statechart diagrams objects consist of states and behaviours (transitions). The state of an object depends on the previous transition and its condition (guard). A statechart diagram provides graphical description of the transitions of an object from one state to another in response to events [12, 11]. The diagram can be used to illustrate the behaviour of instances of a model element. In other words, a statechart diagram shows the possible states of the object and the transitions between them.

The statemachine depicting the abstract behaviour of Memento is shown in Fig. 2. The first phase is to initialise the system by communicating with the server, which is modelled with the event *tryInit*. When initialisation has been successfully completed, the transition goReady brings the system to the state ready, where it awaits and processes the user and server commands. Upon the command *close*, the system enters the finalisation phase, which leads to the system cleanup and proper termination.

The detection of errors in each phase is taken into consideration. In the model, the errors are captured by transitions targeting the suspended state (*susp*), where error handling (rollback) takes place. The system may return to the state where the error was detected, if the error happens to be recoverable. If the error is non-recoverable, the fatal termination action is taken and the system operation finishes. Any error detected during or after finalisation phase is always non-recoverable.

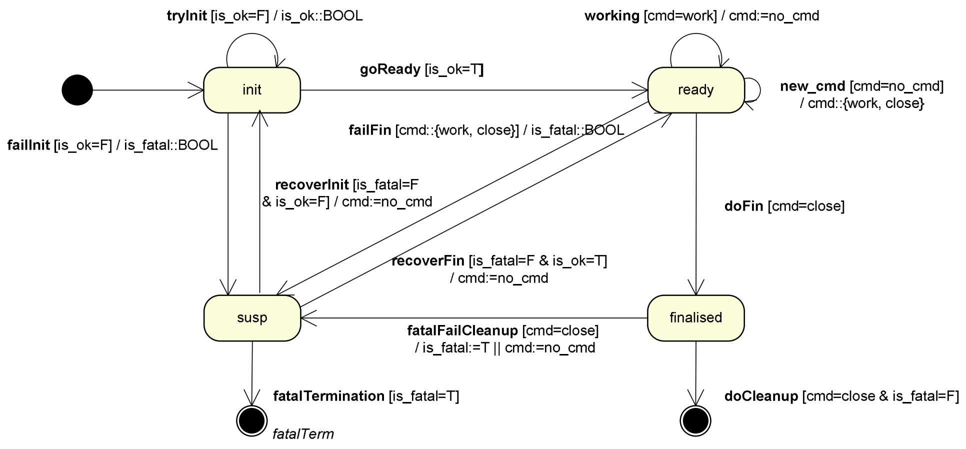

tryInit [is_ok=F] / is_ok::BOOL
working [cmd=work] / cmd:=no_cmd
goReady [is_ok=T]
init
ready
new_cmd [cmd=no_cmd] / cmd::{work, close}
failFin [cmd::{work, close}] / is_fatal::BOOL
failInit [is_ok=F] / is_fatal::BOOL
recoverInit [is_fatal=F & is_ok=F] / cmd:=no_cmd
doFin [cmd=close]
recoverFin [is_fatal=F & is_ok=T] / cmd:=no_cmd
susp
finalised
fatalFailCleanup [cmd=close] / is_fatal:=T || cmd:=no_cmd
fatalTermination [is_fatal=T]
doCleanup [cmd=close & is_fatal=F]
fatalTerm

Fig. 2. The abstract statemachine of Memento

3.2. Formal specification

In order to be able to reason formally about the abstract specification, we translate it to the formal language Event B [10]. An Event-B specification consists of a model and its context that depict the dynamic and the static part of the specification, respectively. They are both identified by unique names. The context contains the sets and constants of the model with their properties and is accessed by the model through the SEES relationship [1]. The dynamic model, on the other hand, defines the state variables, as well as the operations on these. Types and properties of the variables are given in the invariant. All the variables are assigned an initial value according to the invariant. The operations on the variables are given as events of the form **WHEN** guard **THEN** substitution **END** in the Event-B specification. When the guard evaluates to true the event is said to be enabled. The events are considered to be atomic, and hence, only their pre and post states are of interest. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible [10].

Each transition of a statechart diagram is translated to an event in Event-B. Below we show the Event B-translation of the statemachine concerning the initialisation (state *init*) of the cooperation with the server in Fig. 2:

```
MODEL           Memento
SEES            Data
VARIABLES       is_fatal, is_ok, cmd, state
INVARIANT       is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
                (state=init ⇒ cmd=no_cmd) ∧ ...
INITIALISATION  is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init
EVENTS
    tryInit =     WHEN state=init ∧ is_ok=FALSE  THEN is_ok :∈ BOOL END;
    failInit =    WHEN state=init ∧ is_ok=FALSE  THEN state:=susp || is_fatal :∈ BOOL END;
    recoverInit=  WHEN state=susp ∧ is_ok=FALSE ∧ is_fatal=FALSE THEN state:=init || cmd:=no_cmd END;
    goReady =     WHEN state=init ∧ is_ok=TRUE   THEN state:=ready END;
    ...
END
```

The variables model a proper initialisation (*is_ok*), occurrence of a fatal error (*is_fatal*), as well as the command (*cmd*) and the state of the system (*state*). Initially no command is given and the initialisation phase is marked as not completed (*is_ok := FALSE*). The guards of the transitions in the statechart diagram in Fig. 2 are transformed to the guards of the events in the Event B model above, whereas the substitutions in the transitions are given as the substitutions of the events. The feasibility and the consistency of the specification is then proved using the Event-B prover tool.

4. Modelling refinement steps

It is convenient not to handle all the implementation issues at the same time, but to introduce details of the system to the specification in a stepwise manner. Stepwise refinement of a specification is supported by the Event-B formalism. In the refinement process an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. We use the superposition refinement technique [3, 9, 17], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. The variables are added gradually to the specification with their conditions and properties. The computation concerning the new variables is introduced in the existing events by strengthening their guards and adding new substitutions on these variables. New events, assigning the new variables, may also be introduced.

System *C* is said to be a correct refinement of *A* if the following proof obligations are satisfied [10, 15, 17]:

1. The initialisation in *C* should be a refinement of the initialisation in *A*, and it should establish the invariant in *C*.

2. Each old event in *C* should refine an event in *A*, and preserve the invariant of *C*.

3. Each new event in *C* (that does not refine an event in *A*) should only concern the new variables, and preserve the invariant.

4. The new events in *C* should eventually all be disabled, if they are executed in isolation, so that one of the old events is executed (non-divergence).

5. Whenever an event in *A* is enabled, either the corresponding event in *C* or one of the new events in *C* should be enabled (strong relative deadlock freeness).

6. Whenever an error detection event (event leading to the state *susp)* in *A* is enabled, an error detection event in *C* should be enabled (partitioning an abstract representation of an error type into distinct concrete errors during the refinement process [16]).

The tool support provided by Event-B allows us to prove that the concrete specification *C* is a refinement of the abstract specification *A* according to the proof obligations (1) - (6) given above.

In order to guide the refinement process and make it more controllable, refinement patterns [11] can be used. The size of the system grows during the development making it difficult to get an overview of the refinement process. In this paper we introduce progress diagrams to give an abstraction and graphical-descriptive view documenting the applied patterns in each step.

4.1. Progress diagrams

We introduce the idea of progress diagram in the form of a table that is divided into a description part and a diagram part. With this type of table we can point out the design patterns derived from the most important features and changes done in the refinement step. It provides compact information about each refinement step, thereby indicating and documenting the progress of the development. The tabular part briefly describes the relevant features or design patterns of the system in the development step. Moreover, it depicts how states and transitions (initiated, refined or anticipated) are refined, as well as new variables that are added with respect to these features. The diagram part gives a supplementary view of the current refinement step and is in fact a fragment of the statechart diagram.

During the development we benefit from the progress diagram, as we concentrate only on the refined part of the system. The combination of descriptive and visual approaches to show the development of the system gives a compact overview of the part that is the current scope of development. This enables us to focus on the details we are most interested in, and provides a legible picture of the (possibly complex) systems development. The visualisation helps us to better understand the refinement steps and proofs that need to be performed. Progress diagrams do not involve any mathematical notation and are, therefore, useful for communicating the development steps to non-formal methods colleagues. We will illustrate the use of progress diagrams with our case study Memento.

Fig. 3 depicts the progress diagram of the *first refinement step*, where states are partitioned into substates and transitions are added with respect to these. Partitioning the state *init* indicates that the initialisation phase is divided into a connection (state *conn*) and a preparation (state *prep*) phase, that both need the cooperation with the server. The state *susp* is treated in a similar way. Namely, the hierarchical substates *sc*, *sp*, *sr* and *sf* are created, implying that there are in fact various ways of handling the errors, corresponding to the states *conn*, *prep*, *ready* and *finalised*. Thereby, more elaborate information about conditions of error occurrence is added. Note that introducing hierarchical substates corresponds not only to a more detailed model in the structural sense, but also in the functional sense. The transitions (events) *tryInit*, *failInit* and *recoverInit* are refined to more detailed ones taking into account the partitioning of the initialisation phase. The self-transition *tryInit* is refined by two events, *tryConn* and *tryPrep*, which remain self-transitions for the states *conn* and *prep,* respectively. The error handling is refined by events: *failConn* and *recoverConn* for the substate *conn*, and *failPrep* and *recoverPrep* for the substate *prep*. The anticipating transition *cont* is added between the new substates *conn* and *prep*. The new variables are introduced to control the system execution flow. Note that for the substates *sr* and *sf* there are separate diagram parts.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in states *int* and *susp*) • adding new transitions concerning the substates	*init*	*conn* *prep*	*tryInit*	*tryConn, tryPrep*	*is_conn* *is_prep* *wwaited*
			-	*Cont*	
	susp	*sc, sp, sr, sf*	*failInit*	*failConn, failPrep*	
			recoverInit	*recoverConn, recoverPrep*	

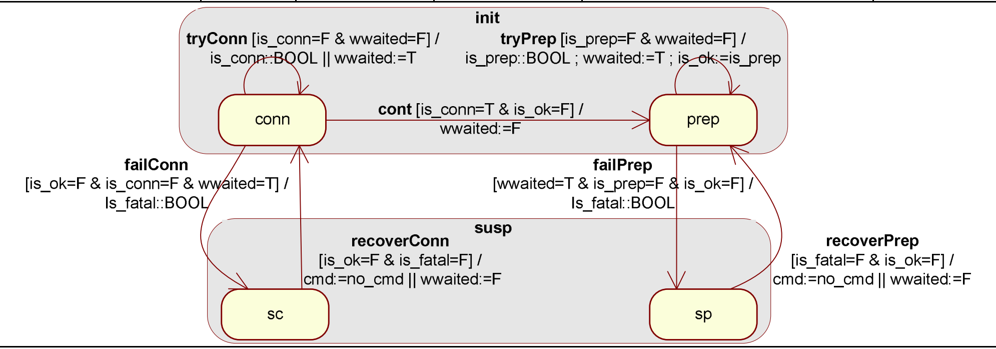

init
tryConn [is_conn=F & wwaited=F] / is_conn::BOOL || wwaited:=T
tryPrep [is_prep=F & wwaited=F] / is_prep::BOOL ; wwaited:=T ; is_ok:=is_prep
conn
cont [is_conn=T & is_ok=F] / wwaited:=F
prep
failConn [is_ok=F & is_conn=F & wwaited=T] / Is_fatal::BOOL
failPrep [wwaited=T & is_prep=F & is_ok=F] / Is_fatal::BOOL
susp
recoverConn [is_ok=F & is_fatal=F] / cmd:=no_cmd || wwaited:=F
recoverPrep [is_fatal=F & is_ok=F] / cmd:=no_cmd || wwaited:=F
sc
sp

Fig. 3. Progress diagram of the first refinement step of Memento

As the refined specification is translated to Event B for proving its correctness, the progress diagram can provide an overview of the proof obligations needed for the refinement step concerning the refined and the anticipating events. In the progress diagram the refined events are the ones given in the column "Refined Transitions" that have a corresponding event in the column "Transitions" (Proof Obligation (2)). For example in Fig. 3 events *tryConn* and *tryPrep* refine *tryInit*. Also the anticipating events are given in the column "Refined Transitions" (event *cont* in Fig. 3). However, they do not have a corresponding event in the column "Transitions". They may only assign the variables in column "New Variables" according to the invariant (Proof Obligation (3)). Furthermore, the non-divergence of the anticipating transitions (Proof Obligation (4)) is indicated in the diagram part by the fact that these transitions do not form a loop [15]. From the columns "Transitions" and "Refined Transitions" also partitioning of the error detection events is indicated (Proof Obligation (6)). In Fig. 3 the error detection event *failinit* is partitioned into *failConn* and *failPrep*.

The result of the first refinement step is shown in the statechart diagram in Fig. 4. When comparing this diagram to the one in Fig. 3, it is worth mentioning that even if the former shows the complete system, the diagram is more difficult to read with all its details. As we focus on the development of a certain part of the system, we particularly want to concentrate on visualising that part. This is of high importance especially when the system develops into a significant sized one. Hence, the progress diagram shows the relevant changes in a more legible way.

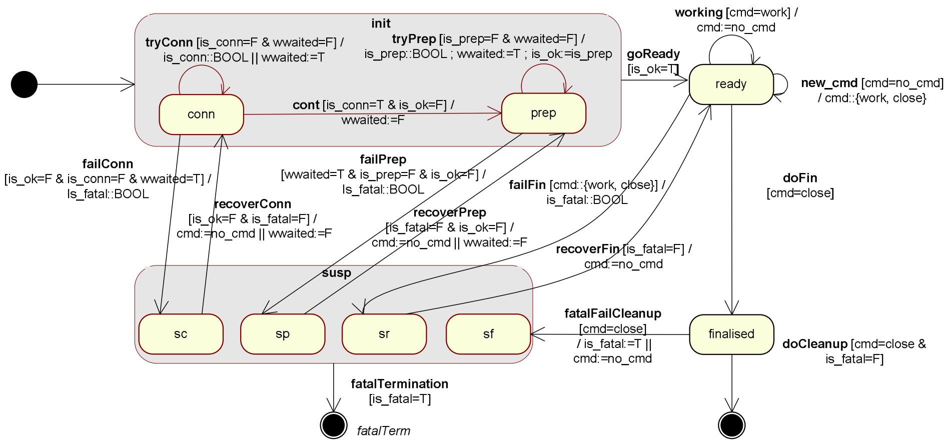

init
tryConn [is_conn=F & wwaited=F] / is_conn::BOOL || wwaited:=T
tryPrep [is_prep=F & wwaited=F] / is_prep::BOOL ; wwaited:=T ; is_ok:=is_prep
conn
cont [is_conn=T & is_ok=F] / wwaited:=F
prep
goReady [is_ok=T]
working [cmd=work] / cmd:=no_cmd
ready
new_cmd [cmd=no_cmd] / cmd::{work, close}
failConn [is_ok=F & is_conn=F & wwaited=T] / Is_fatal::BOOL
recoverConn [is_ok=F & is_fatal=F] / cmd:=no_cmd || wwaited:=F
failPrep [wwaited=T & is_prep=F & is_ok=F] / Is_fatal::BOOL
recoverPrep [is_fatal=F & is_ok=F] / cmd:=no_cmd || wwaited:=F
failFin [cmd::{work, close}] / is_fatal::BOOL
recoverFin [is_fatal=F] / cmd:=no_cmd
doFin [cmd=close]
susp
sc
sp
sr
sf
fatalFailCleanup [cmd=close] / is_fatal:=T || cmd:=no_cmd
finalised
doCleanup [cmd=close & is_fatal=F]
fatalTermination [is_fatal=T]
fatalTerm

Fig. 4. Statechart diagram of the first refinement step of Memento

In *the second refinement step* (not shown) new hierarchical substates are added in the state *prep* along with new transitions that make use of them. These hierarchical substates indicate that the preparation phase is actually composed of two phases (program as well as module preparation). This step is similar to the one above and is not further described here.

The third refinement step (Fig. 5) strengthens the guards of the transitions (events) (using the choice symbol - salmiakki [15]) and shows a more detailed failure management. In fact we split transitions into alternative paths using the choice points. Each path represents a separate transition whose guard is the conjunction of all the segment guards of that path [15]. Hence, new variables, concerning communication with the server, are introduced to express the details of the program preparation phase. These variables represent sending the identification data (*idDataSent*), reading the response (*respRead*), and checking whether the values for response and user are valid (*respValid* and *userValid*). Furthermore, new failure transitions *nIdDS*, *nRR*, *nRV* and *nUV* corresponding to these variables refine the old general failure transition.

Description	States	Ref. States	Transitions	Ref. Trans.	New Var.
3rd step – adding alternative paths to transitions	-	-	*FailPrepPr*	*nIdDS, nRR, nRV, nUV*	*idDataSent, respRead, respValid, userValid*

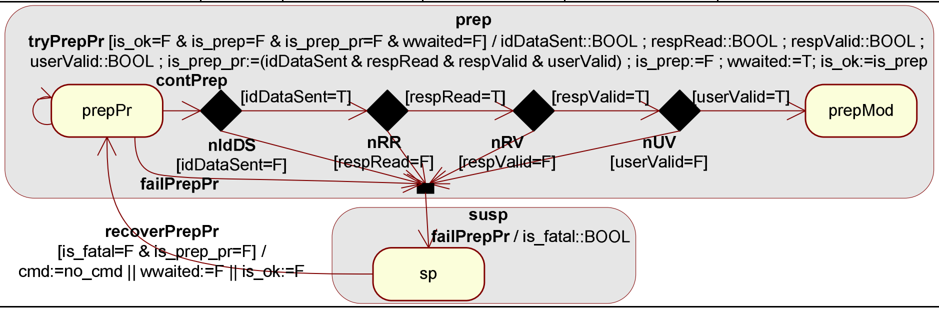

prep
tryPrepPr [is_ok=F & is_prep=F & is_prep_pr=F & wwaited=F] / idDataSent::BOOL ; respRead::BOOL ; respValid::BOOL ; userValid::BOOL ; is_prep_pr:=(idDataSent & respRead & respValid & userValid) ; is_prep:=F ; wwaited:=T; is_ok:=is_prep
contPrep
prepPr
[idDataSent=T]
[respRead=T]
[respValid=T]
[userValid=T]
prepMod
nIdDS [idDataSent=F]
nRR [respRead=F]
nRV [respValid=F]
nUV [userValid=F]
failPrepPr
susp
failPrepPr / is_fatal::BOOL
recoverPrepPr [is_fatal=F & is_prep_pr=F] / cmd:=no_cmd || wwaited:=F || is_ok:=F
sp

Fig. 5. Third refinement step

Here, the progress diagram also gives an intuitive representation of the proof obligations, now concerning strengthening the guards of the old events (Proof Obligation (2)). This is indicated by the transitions between the *salmiakki* symbols [15] in the diagram part of the progress diagram. Moreover, the outgoing transitions of these symbols illustrate intuitively that the relative deadlock freeness (Proof Obligation (5)) is preserved. Again the partitioning of the error detection event *failPrepPr* in the columns "Transitions" and "Refined Transitions" visualises Proof Obligation (6).

5. Conclusion

This paper presents a new approach to documentation of the stepwise refinement of a system. Since the specification for each step becomes more and more complex and a clear overview of the development is lacking, we focus our approach on illustrating the development steps. This kind of documentation is not only helpful for the developers, but also for those that later will try to reuse the exploited features. The documentation is also useful for communicating the development to stakeholders outside of the development team. Thus, a clear and compact form of progress diagrams is appropriate both for industry developers and researchers.

Formal methods and verification techniques are used in the general design of the Memento application to ensure that the development is correct. Our approach uses the B Method as a formal framework and allows us to address modelling at different levels of abstraction. The progress diagrams give an overview of the refinement steps and the needed proofs. Furthermore, the use of progress diagrams during the incremental construction of large software systems helps to manage their complexity and provides legible and accessible documentation.

In future work we will further explore the link between the progress diagrams and patterns. We will investigate how suitable the progress diagrams are for identifying and differentiating patterns used in the refinement steps. Although progress diagrams already appear to be a viable graphical view of the system development, further experimentation on other case studies is envisaged leading to possible enhancements of the progress diagrams. Tool support will be developed for drawing progress diagrams and linking their analysis with the refined models.

Acknowledgements
We would like to thank Dr Linas Laibinis and Dubravka Ilič for the fruitful discussions on the use of the tools supporting the research.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley*,* 2004.

[3] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In: *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[4] R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools* 17, pp. 26-39, 1996.

[5] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a Reference Manual*. Addison-Wesley, 1998.

[6] E. Chan, K. Robinson and B. Welch. Patterns for B: Bridging Formal and Informal Development. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, LNCS 4355, pp. 125-139, 2007. Springer.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[8] D. Ilič and E. Troubitsyna. A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland, February 2005.

[9] S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.

[10] C. Metayer, J.R. Abrial and L. Voisin. *Event-B Language*, RODIN Deliverable 3.2 (D7), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf (May 2005)

[11] Object Management Group. *Unified Modelling Language Specification - Complete UML 1.4 specification*, September 2001. http://www.omg.org/docs/formal/01-09-67.pdf

[12] Object Management Group Systems Engineering Domain Special Interest Group (SE DSIG). S. A. Friedenthal and R. Burkhart. *Extending UML™ from Software to Systems*. (accessed 04.05.2007) http://www.syseng.omg.org/

[13] M. Olszewski and M. Pląska. *Memento system*. http://memento.unforgiven.pl, 2006.

[14] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[15] C. Snook and M. Waldén. Refinement of Statemachines using Event B semantics. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, Besançon, France, LNCS 4355, January 2007, pp. 171-185. Springer.

[16] E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No. 29. June 2000.

[17] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design 13(5-35),* 1998. Kluwer Academic Publishers.

Recording Accurate Process Documentation in the Presence of Failures

Zheng Chen, Luc Moreau

School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
zc05r@ecs.soton.ac.uk, L.Moreau@ecs.soton.ac.uk

Abstract. Scientific and business communities present unprecedented requirements on *provenance*, where the provenance of some data item is the process that led to that data item. Previous work has conceived a computer-based representation of past executions for determining provenance, termed *process documentation*, and has developed a protocol, PReP, to record process documentation in service oriented architectures. However, PReP assumes a failure free environment. The presence of failures may lead to inaccurate process documentation, which does not reflect reality and hence cannot be trustable and utilised. This paper outlines our solution, F_PReP, a protocol for recording accurate process documentation in the presence of failures.

1 Introduction

In scientific and business communities, a wide variety of applications have presented unprecedented requirements [11] for knowing the provenance of their data products, e.g., where they originated from and what has happened to them since creation. In chemistry experiments, provenance is used to detail the procedure by which a material is generated, allowing the material to be patented. In healthcare applications, in order to audit if the proper decisions were made and the proper procedures were followed for a given patient, there is a need to trace back the origins of these decisions and procedures. In engineering manufacturing, keeping track of the history of generated data in simulations is important for users to analyze the derivation of their data products. In finance business, the provenance of some data item establishes the origin and authenticity of the data item produced by financial transactions, enabling users, reviewers and auditors to verify if these transactions are compliant with specific financial regulations.

To meet these requirements, Groth et al. [7] have proposed an open architecture to record and access a computer-based representation of past executions, termed *process documentation*, which can be used for determining the provenance of data. A generic recording protocol, PReP [8], has been developed to provide interoperable means for recording process documentation in the context of service oriented architectures. In their work, process documentation is modelled as a set of assertions (termed *p-assertions*) made by *actors* (i.e., either clients or services) involved in a process (i.e., the execution of a workflow). Each p-assertion documents some steps of the process, e.g., a client invoked a service or the amount of CPU an actor used in a computation. A dedicated repository, termed *provenance store*, is used to maintain p-assertions. For scalability reason, multiple provenance stores may be employed and process documentation may end up distributed, linked by pointers recorded along with p-assertions in each store. Using the pointer chain, distributed process documentation can be retrieved from one store to another.

Process documentation serves as evidence for what actually happened in computer systems [9]. Users interpretate such a documentation as statements made by actors about what they have observed. Therefore, process documentation should in nature be accurate, i.e., it must document events that happened in a process and must not be based on inferences. Otherwise, users would not trust and utilize it when deriving the provenance of their data products.

Recording process documentation in the presence of failures is an issue that has been lacking attention. PReP assumes a system in which no failure occurs. However, large scale, open distributed systems are not failure-free [4]. For example, a service may not be available and network connection may be broken. Failures can lead to inaccurate process documentation: documentation may fail to describe events that occurred, it may describe events that did not happen, or the pointer chain may be broken. Inaccurate process documentation can have disastrous consequences. For example, in a provenance-based service billing system, if a user invoked a service, but documentation fails to describe this invocation, or if a user failed to invoke a service, but its recorded documentation reveals that the invocation occurred, then the user will be charged too little or too much, respectively, which is not acceptable. Also, process documentation distributed in multiple provenance stores may not be retrievable in its entirety because the pointer chain may be broken due to failures.

To address these problems, we have designed a protocol, F_PReP [3], to record accurate process documentation in the presence of failures. It consists of three phases: Exchanging, Recording and Updating. In Exchanging phase, two actors exchange an application message and produce documentation describing the exchange of that message. Asynchronously, in Recording phase, both actors submit their documentation to their respective provenance store. F_PReP provides guaranteed recording of documentation in the presence of failures through the use of redundant provenance stores. If the pointer chain is broken in the two phases, the Updating phase begins. A novel component, Recovery Coordinator, is introduced to fix any broken pointer chain. The protocol has been formalised as an abstract state machine and its correctness has been proved. In this paper, we outline F_PReP, and introduce its formalisation and proof.

2 F_PReP Overview

Terminology and Requirements Process documentation describes a process that led to a result. Such a process is modelled as a set of interactions between actors involved in that process [7]. Each *interaction* is concerned with one application message exchanged between two actors, i.e., the sender and the receiver. Each actor documents the interaction using p-assertions and records them in a provenance store. Since the two *asserting actors* in an interaction may use different stores, they must also record a pointer, termed *viewlink*, indicating where the other actor records its p-assertions. After repeating these actions for all interactions of a process, the documentation of that process is obtained resulting in a bidirectional pointer chain, connecting all the stores hosting the documentation of that process. Therefore, to record *accurate* process documentation, we need to ensure that each *interaction record*, i.e., the documentation of an interaction, is accurate preserving the following properties: (1) Each actor's p-assertions describe what actually happened in that interaction from that actor's viewpoint (*Assertion Accuracy*); (2) Each actor's viewlink points to the provenance store where the other actor recorded p-assertions in that interaction (*ViewLink Accuracy*). In addition, the protocol needs to enforce that (3) all p-assertions produced by each actor in an interaction and the actor's viewlink must be recorded in a provenance store in the presence of failures (*Documentation Availability*).

Assumptions Our failure assumptions are the following: asserting actors, provenance stores may crash, i.e., they halt and stop any further execution, and never recover. However, we assume that there is no failure in a provenance store's persistent storage. We assume that each asserting actor keeps a list of provenance store addresses and at least one store is available. Communication channels can lose and reorder, but not duplicate messages. Process documentation may be inaccurate when an actor is maliciously recording incorrect information. However, we assume this case does not happen.

To ensure separation of concerns, each actor employs a Communication Agent (CA) to send/receive messages to/from other actors. We assume that the sender CA can use acknowledgement, timeout and retransmission to reliably deliver a message, and report the delivery outcome to the sending actor: *success* or *failure*. It reports *success* notification if the message is acknowledged, indicating that the receiver CA has received that message. It also reports *failure* notification if the message fails to be delivered or it is not acknowledged even after retry attempts. In this way, our protocol does not have to handle communication details but can focus on actions in response to the notifications (in sending actors) and messages (in receiving actors) provided by CA.

Exchanging Phase In the exchanging phase, two actors, the sender S and the receiver R, exchange an application message *app* and document the interaction, as demonstrated in Figure 1. To facilitate creating and recording interaction records, each actor employs a Recording Manager (RM). In order to form a pointer chain, the two actors also exchange a pointer to their respective provenance store. For this purpose, S embeds its pointer in *app*, while R informs S with its pointer in a separate message *linkr*. Meanwhile, each actor creates an interaction record which includes the p-assertions describing the interaction, and a viewlink, i.e., the other actor's pointer. The created interaction record is accumulated in RM before being sent to a provenance store. This buffering of interaction records is designed to reduce the performance penalty upon the application by allowing the actor to send interaction records when convenient.

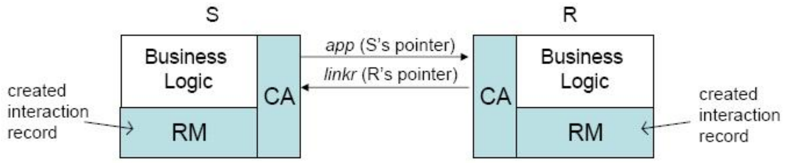


Fig. 1. Exchanging Phase

In order to create an *accurate* interaction record, an actor must only assert facts that it can observe. Hence, we specify some rules for asserting actors to follow. (1) S must assert that an interaction occurred if and only if it receives SUCCESS notification from its CA for delivering *app* message; (2) S must assert failure information when receiving FAILURE notification from its CA for delivering *app* message. One reason for this rule is that failure information provides evidence that an interaction was attempted even if that interaction failed. Without such information, there would be no record of the attempted interaction; (3) S must record R's pointer as its viewlink if it receives the pointer; (4) R must assert that an interaction occurred after it receives *app* message; (5) R must record S's pointer as its viewlink after it receives the pointer; (6) R must assert failure information when receiving a FAILURE notification from its CA for delivering *linkr*. This is because S may not receive the pointer, disconnecting the chain. In this case, S takes no action and the assertion made by R will be used to fix the broken chain in Updating phase; (7) S and R may generate application specific p-assertions.

Recording Phase F_PReP provides guaranteed recording of interaction record in the presence of failures through the use of redundant provenance stores. Figure 2 illustrates this phase. In Step 1, an actor's RM submits an interaction record to a provenance store PS. In Step 2, PS stores the interaction record that it receives in its persistent storage. After successfully recording the interaction record, it replies the submitting actor with an acknowledgement (Step 3). We have assumed that there is no failure in persistent storage; hence any interaction records stored in a provenance store's persistent storage remain available forever. If the actor's RM receives FAILURE notification from CA for delivering the interaction record or it does not receive the acknowledgement before a timeout, then it can imply that failures may have occurred, e.g., PS has crashed. In the two cases, the RM may resend the interaction record to PS. Since a crashing provenance store can no longer be used for further recording, the RM needs to use an alternative store after retry attempts also fail. We have assumed that each asserting actor keeps a list of provenance store addresses and at least one store is available, therefore, the use of redundant stores ensures that an actor's interaction record is eventually recorded. Only after the acknowledgement is received, can the RM eliminate the accumulated interaction record. The use of an alternative store would result in a broken pointer chain if an actor's original pointer has been sent to the other actor, which now does not point to a correct location. Hence, the RM needs to add an assertion documenting the use of an alternative store in its interaction record so that actions can be taken to fix any broken chain in the next phase.

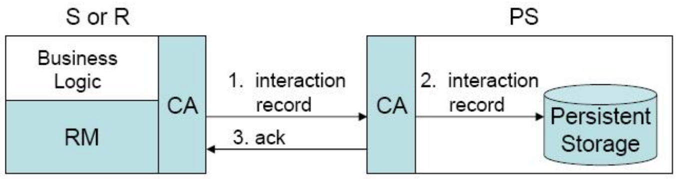


Fig. 2. Recording Phase

Updating Phase In this phase, the protocol updates an actor's viewlink in order to fix a potentially broken pointer chain. A pointer chain may be broken in two situations, as demonstrated in Figure 3. (1) R gets a FAILURE notification when sending *linkr* to S in Step 2, hence S may not know R's pointer; (2) If an actor, say S, does not successfully record its interaction record in Step 3 and selects an alternative store, say $PS1'$, to submit the record, then S's pointer sent to R in Step 1 does not point to the correct location, $PS1'$. In either case, an actor has made an assertion documenting failure information when delivering *linkr* or the use of an alternative store, as described in the previous two phases. We use BROKEN to denote any of the two assertions in Step 4, since either assertion documents a fact that may cause a broken pointer chain. Upon receipt of a BROKEN, a provenance store requests a novel component, Recovery Coordinator, to facilitate repairing the broken chain (Step 5). By taking remedial actions, the Recovery Coordinator updates the viewlink in a destination store (Step 6) with any broken pointer chain fixed. In the example of Figure 3, when the protocol terminates, $PS1'$ has a viewlink to $PS2$ and vice versa.

We assume that *the Recovery Coordinator does not fail.* There is only one Recovery Coordinator, so we can use traditional fault-tolerant mechanisms such as replication to ensure its reliability and availability. Recovery Coordinator is necessary to fix a broken pointer chain, since both actors in an interaction may each report a BROKEN, as shown in Figure 3. In this case, direct communication between two actors' provenance stores does not help, because at that moment, one does not know which store the other actor is using. For example, in the figure, R does not know that S is using $PS1'$ and S does not know where R's p-assertions are stored. Assuming that the pointer chain is not broken frequently,

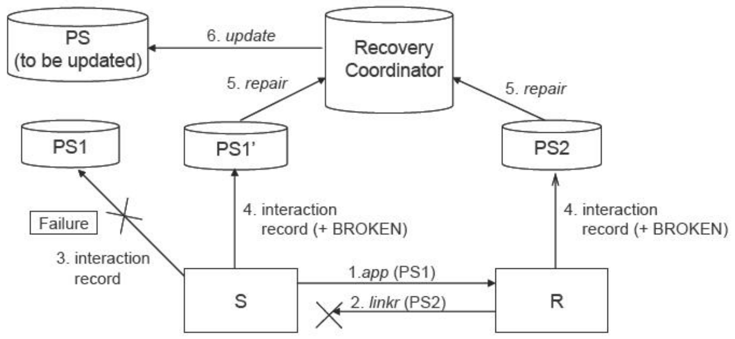


Fig. 3. Updating Phase

the Recovery Coordinator is not involved in each interaction and hence does not affect the system's scalability, despite being centralised.

3 Protocol Analysis and Formalisation

Protocol Analysis The agreement on interaction occurrence may not be reached by the sender and receiver of an application message. If the sender gets a *failure* notification, it is impossible for it to decide whether the receiver has received that message or not [10]. If the receiver happens to receive the application message, an inconsistency occurs, i.e., the sender asserts failure information while the receiver documents that the interaction has occurred. Such an inconsistency reflects the difference between the sender and receiver's knowledge of an interaction, which does not contradict the *Assertion Accuracy* requirement. Therefore, our protocol does not need to remove such an inconsistency.

Concurrency is a major concern to the correctness of the protocol. The protocol specifies actions for asserting actors, provenance stores, and Recovery Coordinator, which may co-operate with one another. On receiving messages from different components, the receiver has to respond properly against all the possible message arrival orders. For example, a provenance store may concurrently receive an interaction record from an asserting actor and an *update* message from Recovery Coordinator; Recovery Coordinator may receive two *repair* messages from two provenance stores in any order. The concurrency issue hence requires us to rigorously design the protocol.

Formalisation F_PReP has been formalised through the use of an abstract state machine (ASM). The machine's behaviour is described by states and transitions between those states. Such a formalisation provides a precise, implementation-independent means of describing the system.

Figure 4 shows the system state space. We identify specific subsets of actors in the system, namely, senders, receivers, provenance stores, and Recovery

Coordinator. Protocol messages are sent over communication channels, denoted by $\mathcal{K}$. Since no assumption is made about message order in channels, channels are represented as bags of messages between pairs of actors. The set of each of protocol messages is defined formally as an inductive type. For example, the set of Application Messages is defined by an inductive type whose constructor is app and whose parameters are from the set of $\textsc{Data}$, ID and OL[1]. The set of all messages $\mathcal{M}$ is defined as the union of these message sets. CA's notifications are modelled as two messages: $\mathsf{failure}(m)$ and $\mathsf{success}(m)$. The power set notation (P) denotes that there can be more than one of a given element. We specify several notations representing p-assertions. The notation $\mathsf{Occurrence}$ stands for an assertion which documents the occurrence of an interaction. The notation $\mathsf{FailureInfo}(pa)$ denotes any assertions describing failure information during the delivery of an application message, while $\mathsf{Broken}(pa)$ represents any assertions documenting a fact that may cause a broken pointer chain.

The internal functionality of each kind of processes is modelled as follows. (1) The set ACTOR models all the asserting actors, each identified by an actor identity. Informally, each asserting actor contains a table ($actor\_T$) that maps an interaction identity (id) and the actor's view kind (v), i.e., Sender or Receiver, to a tuple: the state of its ownlink (stl), the state of an interaction record (st), the actor's ownlink (ol) and its viewlink (vl). An asserting actor's p-assertions are accumulated in a message queue before being sent to a provenance store. The queue is modelled by a table ($queue\_T$). The table $timer\_T$ maintains timing information such as timer status, current time, timing interval and timeout, which is used by an asserting actor's RM in Recording phase. The notation LC defines a function that maps a sender identifier to a natural number, used to distinguish interactions between the sender and receiver. The sender needs to ensure that the natural number is locally unique in each interaction. (2) The set PS models provenance stores, each identified by an actor identity. Each provenance store contains a table ($store\_T$) that maps an interaction identity (id) and a view kind (v) to a tuple: recorded p-assertions (pas) and a viewlink (vl). (3) The set C models coordinators. There is only one coordinator, identified by a_c, and it also keeps a table ($coord\_T$). For each interaction (id) and for each view kind (v) in the interaction, the table stores a tuple: the destination provenance store (a_{dps}) to be updated and the ownlink (ol) of the asserting actor with this view kind.

Given the state space, the ASM is described by an initial state and a set of transitions. Figure 4 contains the initial state, which can be summarised as empty channels and empty tables in all processes. The ASM proceeds from this state through its execution by going through transitions that lead to new states.

The permissible transitions in the ASM are described through rules. Rules are identified by their name and a number of parameters that the rule operates over. Once a rule's conditions are met, the rule fires. The execution of a rule is atomic, so that no other rule may interrupt or interleave with an executing

[1] An asserting actor's *ownlink*, OL, refers to the provenance store where the asserting actor records its *own* p-assertions.

$$
\begin{array}{rcll}
A & = & \{a_1, a_2, \ldots, a_n\} & \text{(Set of Actor Identities)} \\
\mathrm{SID} & \subset & A & \text{(Sender Identities)} \\
\mathrm{RID} & \subset & A & \text{(Receiver Identities)} \\
\mathrm{PID} & \subset & A & \text{(Provenance Store Identities)} \\
a_c & \subseteq & A & \text{(Coordinator Identity)} \\
 & & & \\
\mathcal{M} & = & \mathsf{app} : \textsc{Data} \times \mathrm{ID} \times \mathrm{OL} \rightarrow \mathcal{M} & \text{(Application Messages)} \\
 & & \mid\ \mathsf{linkr} : \mathrm{ID} \times \mathrm{VK} \times \mathrm{OL} \rightarrow \mathcal{M} & \text{(R's Ownlink Messages)} \\
 & & \mid\ \mathsf{record} : \mathrm{ID} \times \mathrm{VK} \times \mathrm{VL} \times P\,(\mathrm{P}\textsc{Assertion}) \rightarrow \mathcal{M} & \text{(Interaction Record Messages)} \\
 & & \mid\ \mathsf{ack} : \mathrm{ID} \times \mathrm{VK} \rightarrow \mathcal{M} & \text{(Record Ack Messages)} \\
 & & \mid\ \mathsf{repair} : \mathrm{ID} \times \mathrm{VK} \times \textsc{Dest}\mathrm{PS} \times \mathrm{OL} \rightarrow \mathcal{M} & \text{(Repair Messages)} \\
 & & \mid\ \mathsf{update} : \mathrm{ID} \times \mathrm{VK} \times \mathrm{OL} \rightarrow \mathcal{M} & \text{(Update Messages)} \\
 & & \mid\ \mathsf{failure} : \mathcal{M} \rightarrow \mathcal{M} & \text{(Failure Notifications)} \\
 & & \mid\ \mathsf{success} : \mathcal{M} \rightarrow \mathcal{M} & \text{(Success Notifications)} \\
 & & & \\
\mathrm{ID} & = & \{id_1, id_2, \ldots, id_n\} & \text{(Set of Interaction Identifiers)} \\
\mathrm{VK} & = & \{\mathrm{S}, \mathrm{R}\} & \text{(Set of ViewKinds)} \\
\mathrm{OL} & = & \mathrm{PID} & \text{(Set of an Actor's Ownlinks)} \\
\mathrm{VL} & = & \mathrm{PID} & \text{(Set of an Actor's Viewlinks)} \\
\textsc{Dest}\mathrm{PS} & = & \mathrm{PID} & \text{(Set of Destination Stores)} \\
\mathrm{P}\textsc{Assertion} & = & \{\mathsf{Occurrence}, \mathsf{FailureInfo}(pa), \mathsf{Broken}(pa), pa_1, \ldots, pa_n\} & \text{(Set of P-Assertions)} \\
 & & & \\
\mathrm{ACTOR} & = & A \rightarrow \mathrm{ID} \times \mathrm{VK} \rightarrow \mathrm{STL} \times \mathrm{STR} \times \mathrm{OL} \times \mathrm{VL} & \text{(Set of Asserting Actors)} \\
\mathrm{STL} & = & \{\bot, \mathsf{SENT}, \mathsf{F}, \mathsf{OK}\} & \text{(States of an OwnLink)} \\
\mathrm{STR} & = & \{\bot, \mathsf{SENT}, \mathsf{OK}\} & \text{(States of Interaction Record)} \\
 & & & \\
\mathrm{QUEUE} & = & A \times \mathrm{ID} \rightarrow Bag(\mathrm{P}\textsc{Assertion}) & \text{(Set of Quened P-Assertions)} \\
 & & & \\
\mathrm{TIMER} & = & A \times \mathrm{ID} \rightarrow \textsc{Status} \times \textsc{CurrentTime} \times & \\
 & & \textsc{Interval} \times \textsc{Timeout} & \text{(Set of Timers)} \\
\textsc{Status} & = & \{\bot, \mathsf{Enabled}, \mathsf{Disabled}\} & \text{(Set of Timer Statuses)} \\
\textsc{CurrentTime} & = & \{t_1, t_2, \ldots, t_n\} & \text{(Set of Current Times)} \\
\textsc{Interval} & = & \{\varDelta_1, \varDelta_2, \ldots, \varDelta_n\} & \text{(Set of Timing Intervals)} \\
\textsc{Timeout} & = & \{to_1, to_2, \ldots, to_n\} & \text{(Set of Timeouts)} \\
 & & & \\
\mathrm{LC} & = & \mathrm{SID} \rightarrow N & \text{(Sender's Local Counts)} \\
 & & & \\
\mathrm{PS} & = & A \rightarrow \mathrm{ID} \times \mathrm{VK} \rightarrow \mathrm{VL} \times P\,(\mathrm{P}\textsc{Assertion}) & \text{(Set of Provenance Stores)} \\
 & & & \\
\mathrm{C} & = & A \rightarrow \mathrm{ID} \times \mathrm{VK} \rightarrow \textsc{Dest}\mathrm{PS} \times \mathrm{OL} & \text{(Set of Coordinators)} \\
 & & & \\
\mathcal{K} & = & A \times A \rightarrow Bag(\mathcal{M}) & \text{(Set of Channels)}
\end{array}
$$

Characteristic Variables:
$a \in A$, $a_s \in \mathrm{SID}$, $a_r \in \mathrm{RID}$, $a_{ps} \in \mathrm{PID}$, $m \in \mathcal{M}$, $d \in \textsc{Data}$, $pa \in \mathrm{P}\textsc{Assertion}$, $pas \in P\,(\mathrm{P}\textsc{Assertion})$, $id \in \mathrm{ID}$, $v \in \mathrm{VK}$, $a_{dps} \in \textsc{Dest}\mathrm{PS}$, $ol \in \mathrm{OL}$, $vl \in \mathrm{VL}$, $stl \in \mathrm{STL}$, $str \in \mathrm{STR}$, $status \in \textsc{Status}$, $t \in \textsc{CurrentTime}$, $\varDelta \in \textsc{Interval}$, $to \in \textsc{Timeout}$, $actor\_T \in \mathrm{ACTOR}$, $queue\_T \in \mathrm{QUEUE}$, $timer\_T \in \mathrm{TIMER}$, $lc \in \mathrm{LC}$, $store\_T \in \mathrm{PS}$, $coord\_T \in \mathrm{C}$, $k \in \mathcal{K}$

Configuration: $c = \langle actor\_T, queue\_T, timer\_T, store\_T, coord\_T, k \rangle$

Initial State: $c_i = \langle actor\_T_i, queue\_T_i, timer\_T, store\_T_i, coord\_T_i, k_i \rangle$
where:

$$
\begin{array}{ll}
actor\_T_i = \lambda a \lambda idv \cdot \langle \bot, \bot, \bot, \bot \rangle, & queue\_T_i = \lambda aid \cdot \emptyset, \\
timer\_T_i = \lambda a \lambda idv \cdot \langle \bot, \bot, \bot, \bot \rangle, & store\_T_i = \lambda a \lambda idv \cdot \langle \bot, \emptyset \rangle, \\
coord\_T_i = \lambda a \lambda idv \cdot \langle \bot, \bot \rangle, & k_i = \lambda a_i a_j \cdot \emptyset
\end{array}
$$

Fig. 4. System State Space

```
$send\_\mathsf{app}(a_s, a_r, id, ls, d)$ :
    $id = newIdentifier(a_s, a_r)$
$\rightarrow$ {
        $send(\mathsf{app}(d, id, ls), a_s, a_r)$;
        $actor\_T(a_s)(id, S) := \langle \mathsf{SENT}, \bot, ls, \bot \rangle$ ;
    }

$failure\_\mathsf{app}(a_s, a_r, id, ls, d)$ :
    $\mathsf{failure}(\mathsf{app}(d, id, ls)) \in k(a_s, a_r) \wedge$
    $\{\mathsf{FailureInfo}(pa), pa_2, \ldots, pa_n\} = createPA()$
$\rightarrow$ {
        $receive(\mathsf{failure}(\mathsf{app}(d, id, ls)), a_s, a_r)$;
        $queue\_T(a_s, id) := queue\_T(a_s, id) \oplus$
        $\{\mathsf{FailureInfo}(pa), pa_2, \ldots, pa_n\}$ ;
        $actor\_T(a_s)(id, S).stl := \mathsf{F}$;
    }

$success\_\mathsf{app}(a_s, a_r, id, ls, d)$ :
    $\mathsf{success}(\mathsf{app}(d, id, ls)) \in k(a_s, a_r) \wedge$
    $\{\mathsf{Occurrence}, pa_2, \ldots, pa_n\} = createPA()$
$\rightarrow$ {
        $receive(\mathsf{success}(\mathsf{app}(d, id, ls)), a_s, a_r)$;
        $queue\_T(a_s, id) := queue\_T(a_s, id) \oplus$
        $\{\mathsf{Occurrence}, pa_2, \ldots, pa_n\}$ ;
        $actor\_T(a_s)(id, S).stl := \mathsf{OK}$;
    }

$receive\_\mathsf{linkr}(a_s, a_r, id, lr)$ :
    $\mathsf{linkr}(id, R, lr) \in k(a_s, a_r)$
$\rightarrow$ {
        $receive(\mathsf{linkr}(id, R, lr), a_r, a_s)$;
        $actor\_T(a_s)(id, S).vl := lr$;
    }
```

Fig. 5. The Sender's rules in Exchanging phase

```
$receive\_\mathsf{app}(a_s, a_r, id, d, ls, lr)$ :
    $\mathsf{app}(d, id, ls) \in k(a_s, a_r) \wedge$
    $\{\mathsf{Occurrence}, pa_2, \ldots, pa_n\} = createPA()$
$\rightarrow$ {
        $receive(\mathsf{app}(d, id, ls), a_s, a_r)$;
        $queue\_T(a_r, id) := queue\_T(a_r, id) \oplus$
        $\{\mathsf{Occurrence}, pa_2, \ldots, pa_n\}$ ;
        $send(\mathsf{linkr}(id, R, lr), a_r, a_s)$;
        $actor\_T(a_r)(id, R) := \langle \mathsf{SENT}, \bot, lr, ls \rangle$ ;
        // business logic
    }

$failure\_\mathsf{linkr}(a_s, a_r, id, lr)$ :
    $\mathsf{failure}(\mathsf{linkr}(id, R, lr)) \in k(a_s, a_r) \wedge$
    $\{\mathsf{Broken}(pa)\} = createPA()$
$\rightarrow$ {
        $receive(\mathsf{failure}(\mathsf{linkr}(id, R, lr)), a_r, a_s)$;
        $queue\_T(a_r, id) := queue\_T(a_r, id) \oplus$
        $\{\mathsf{Broken}(pa)\}$ ;
        $actor\_T(a_r)(id, R).stl := \mathsf{F}$;
    }

$success\_\mathsf{linkr}(a_s, a_r, id, lr)$ :
    $\mathsf{success}(\mathsf{linkr}(id, R, lr)) \in k(a_s, a_r)$
$\rightarrow$ {
        $receive(\mathsf{success}(\mathsf{linkr}(id, R, lr)), a_r, a_s)$;
        $actor\_T(a_r)(id, R).stl := \mathsf{OK}$;
    }
```

Fig. 6. The Receiver's rules in Exchanging phase

rule. This maintains the consistency of the ASM. A new state is achieved after applying all the rule's pseudo-statements to the state that met the conditions of the rule. A rule's pseudo-statements consist of a set of *send*, *receive* and table update operations. Informally, $send(m, a_1, a_2)$ inserts a message m into the channel from actor a_1 to actor a_2, and $receive(m, a_1, a_2)$ removes m from the channel between a_1 and a_2. A table update operation places a message into a table or changes the state of a table field.

Due to space restriction, we only give the Sender and Receiver's rules in Exchanging Phase in Figure 5 and 6. These rules precisely specify an asserting actor's behaviour described in Section 2. The whole set of rules can be found at [3]. The function $newIdentifier(a_s, a_r)$ creates a globally unique interaction identifier, which can be a tuple consisting of the sender and receiver's identity plus a locally unique number managed by the sender, expressed as $\langle a_s, a_r, lc(a_s) \rangle$. The function $createPA()$ generates a set of p-assertions and the operator $\oplus$ denotes union on bags.

The properties *Documentation Availability*, *Assertion Accuracy* and *ViewLink Accuracy* have been formalised as the following invariants. The notations v and $\overline{v}$ stand for the two views in an interaction and $actor\_T(a_s, id, S).stl = \mathsf{OK}$ marks the occurrence of an interaction.

Documentation Availability:
(P1) *If* $actor\_T(a_s)(id, S).stl = \mathsf{OK}$, *then*
$\forall v \in \mathrm{VK}$, $\mathsf{Occurrence} \in store\_T(a_{ps})(id, v).pas \wedge store\_T(a_{ps})(id, v).vl \neq \bot$,
such that $a_{ps} = actor\_T(a_v)(id, v).ol$.
(P2) *If* $actor\_T(a_s)(id, S).stl = \mathsf{F}$, *then*
$\mathsf{FailureInfo}(pa) \in store\_T(a_{ps})(id, S).pas$, *such that* $a_{ps} = actor\_T(a_s)(id, S).ol$.
Assertion Accuracy:
(P3) *If* $\forall v \in \mathrm{VK}$, $\mathsf{Occurrence} \in store\_T(a_{ps})(id, v).pas$, *then*
$actor\_T(a_s)(id, S).stl = \mathsf{OK}$, *such that* $a_{ps} = actor\_T(a_v)(id, v).ol$.
(P4) *If* $\mathsf{FailureInfo}(pa) \in store\_T(a_{ps})(id, S).pas$, *then*
$actor\_T(a_s)(id, S).stl = \mathsf{F}$, *such that* $a_{ps} = actor\_T(a_s)(id, S).ol$.
Viewlink Accuracy:
(P5) *If* $actor\_T(a_s)(id, S).stl = \mathsf{OK}$, *then*
$\forall v \in \mathrm{VK}$, $store\_T(a_{ps})(id, v).vl = actor\_T(a_{\overline{v}})(id, \overline{v}).ol$,
such that $a_{ps} = actor\_T(a_v)(id, v).ol$.

We have proved that these invariants hold when the protocol terminates. Given an arbitrary valid configuration of the ASM, our proofs typically proceed by induction on the length of the transitions that lead to the configuration, and by a case analysis on the kind of transitions. We show that a property is true in the initial configuration of the machine and remains true for every possible transition. This kind of proof is systematic, less error prone and can be easily encoded in a mechanical theorem prover.

4 Related Work and Conclusion

Much research has been seen to support recording process documentation, e.g., Chimera [5], myGrid [12], PReP [8, 7] and Kepler [1]. From an analysis of these works, only PReP provides an application-independent solution to recording process documentation. However, all the surveyed systems either assume a failure-free execution environment or do not discuss this issue.

Redundancy has long been used as a means to provide fault tolerance in distributed systems [2]. Key components may be replicated (replication in space) or re-executed (replication in time) to protect against hardware malfunctions or transient system faults. Our work adopts this mechanism through the use of redundant provenance stores and retransmission of messages.

Distributed transactions typically requires *all-or-nothing* atomicity to maintain system consistency [6]. The *all-or-nothing* property is not applicable to our work. Assume that the asserting actors and provenance stores are the participants in a transaction. If any participant fails after the interaction took place, then the recording action is aborted and hence the documentation about that interaction cannot be recorded. This is not desired since process documentation must reflect reality and document events that happened in a process. As long as the interaction has occurred, its documentation must be recorded in a provenance store.

In conclusion, we have presented a protocol F_PReP for recording accurate process documentation in the presence of failures. Compared with PReP, F_PReP not only keeps the application-independent nature, but also guarantees that process documentation is accurate and available in a provenance store in the presence of failures. Also, it enables distributed process documentation to be still retrievable in large scale distributed environments where failures may occur. The protocol is being implemented and its performance will be evaluated in future work.

References

1. Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 118–132, 2006.
2. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
3. Zheng Chen. Accurately recording process documentation in the presence of failures. Mini thesis, School of Electronics and Computer Science, University of Southampton, UK, http://www.ecs.soton.ac.uk/~zc05r/protocol, 2007.
4. Ewa Deelman and et. al. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *Proceedings of the e-Science 2006 Conference in Amsterdam, the Netherlands*, December 2006.
5. Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *CIDR*, 2003.
6. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.
7. Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An architecture for provenance systems. Technical Report D3.1.1, University of Southampton, February 2006.
8. Paul Groth, Michael Luck, and Luc Moreau. A protocol for recording provenance in service-oriented grids. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04)*, France, 2004.
9. Paul Groth, Simon Miles, and Steve Munroe. Principles of high quality documentation for provenance: A philosophical discussion. In *Proceedings of Third International Provenance and Annotation Workshop, Chicago*, 2006.
10. Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1997.
11. Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.
12. Jun Zhao, Chris Wroe, Carole A. Goble, Robert Stevens, Dennis Quan, and R. Mark Greenwood:. Using semantic web technologies for representing e-science provenance. *International Semantic Web Conference*, pages 92–106, 2004.

Vertical and Horizontal Composition in Service-Oriented Architecture

Anatoliy Gorbenko[1], Vyacheslav Kharchenko[1], Alexander Romanovsky[2]

[1] Department of Computer Systems and Networks, National Aerospace University, Kharkiv, Ukraine
A.Gorbenko@csac.khai.edu, V.Kharchenko@khai.edu
[2] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
Alexander.Romanovsky@newcastle.ac.uk

Abstract. Achieving high dependability in the Service-Oriented Architecture (SOA) is an open problem. One of the possible solutions for this problem is employing service diversity represented by a number of component web services with the identical or similar functionality at the each level of the composite system hierarchy during service composition. It is clear that such redundancy can improve web service reliability (trustworthiness) and availability. However to apply this approach we need to solve a number of problems. The paper proposes several solutions for ensuring dependable services composition when natural service redundancy and diversity are used.

1 Introduction

The Web Services (WS) architecture [1] based on the SOAP, WSDL and UDDI specifications is rapidly becoming a de facto standard technology for organization of global distributed computing and achieving interoperability between different software applications running on various platforms. It is now extensively used in developing numerous business-critical applications for banking, auctions, Internet shopping, hotel/car/flight/train reservation and booking, e-business, e-science, GRID-systems, etc. That is why analysis and dependability ensuring of this architecture are emerging areas of research and development [1–3]. The WS architecture is in effect a further step in the evolution of the well-known component-based system development with off-the-shelf (OTS) components. The main advances enabling this architecture have been made by the standardisation of the integration process (a set of interrelated standards such as SOAP, WSDL, UDDI, etc.). WSs are the OTS components for which a standard way of advertising their functionality has been widely adopted.

In the paper we analyse possible Web Services composition modes and also propose solutions making use of the natural redundancy and diversity which exists in such systems and guaranteeing that the overall dependability (availability, correctness and responsiveness) of the composite system is improving.

2 Web Services Composition

Web service composition is currently an active area of research, with many languages being proposed by academic and industrial research groups. IBM's Web Service Flow Language (WSFL) [4] and Microsoft's XLANG [5] were two of the earliest languages to define standards for Web services composition. Both languages extended W3C's Web Service Description Language (WSDL) [6], which is the standard language used for describing the syntactic aspects of a Web service. Business Process Execution Language for Web Services (BPEL4WS) [7] is a recently proposed specification that represents the merging of WSFL and XLANG. BPEL4WS combines the graph oriented process representation of WSFL and the structural construct based processes of XLANG into a unified standard for Web services composition.

In addition to these commercial XML based standards, there have been work on a unique Web service composition language called Web Ontology Language for Services (www.daml.org/services) OWL-S (previously known as DAML-S) [8], which provides for a richer (more semantic) description of Web service compositions.

In our work we focus on the general patterns (types) of this composition and identify two typical blueprints of composing WSs: i) "vertical" composition for functionality extension, and ii) "horizontal" composition for dependability improvement.

The first type of service composition ("vertical") is used for building the Work-Flow (WF) of the systems and is already supported by BPEL, BPML, XPDL, JPDL and other WF languages.

The second one ("horizontal") deals with a set of redundant (and possible diverse) Web Services with identical or similar functionality and is used for improvement of various dependability attributes (availability, trustworthiness, responsibility, etc.).

Bellow we show some illustrative examples of two different types of WSs composition and discuss the way in which the "horizontal" composition improves dependability of SOA.

2.1 Vertical Composition for Functionality Extension

Vertical composition (Fig. 1-a) extends the Web Services functionality. New Composite Web Service is composed out of several target services which provide different functions. For example, "Travel Agency (TA) Service" can be composed of "Flight Service", "Car Rental Service", "Hotel Service", etc.

The composite Web Service can invoke a set of target (composed) services concurrently to reduces the mean execution time or sequentially (if execution of one service depends on the result of another one). In case when some of the target (composed) services fails or cannot satisfy user the request all other services will have to be rolled back and their results should be cancelled. To improve dependability of such composite system various means of fault-tolerance and error recovery should be implemented (redundancy, exception handling, forward error recovery, etc.).

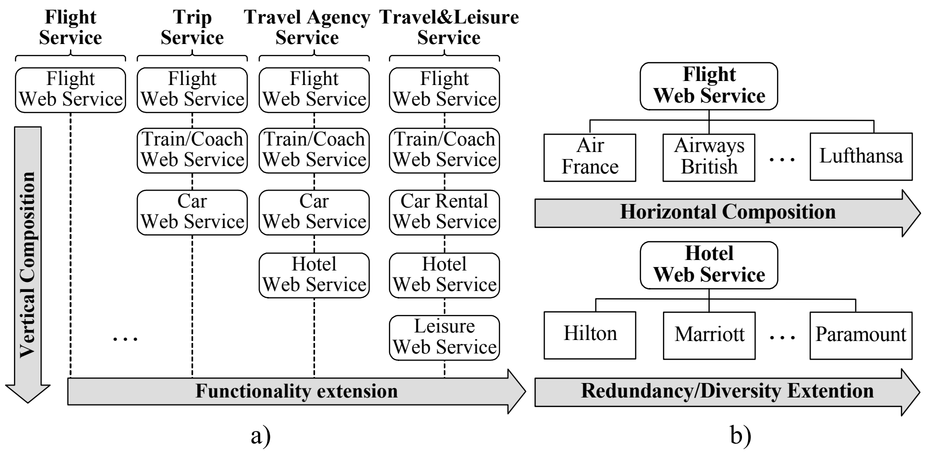


Fig. 1. Web Services Composition: a) vertical; b) horizontal

2.2 Horizontal Composition for Dependability Improvement

Horizontal composition (Fig. 1-b) uses several alternative (diverse) Web Services with the identical (or similar) functionality or several operational releases of the same service (Table 1). Such kind of redundancy which based on natural diversity improves reliability (correctness and trustworthiness) of Web Service.

Table 1. An example of existed alternative (diverse) Web Services

Diverse Stock Quotes Web Services
stock_wsx.GetQuote:
http://www.webservicex.com/stockquote.asmx?WSDL
stock_gama.GetLatestStockDailyValue:
http://www.gama-system.com/webservices/stockquotes.asmx?wsdl
stock_xmethods.getQuote:
http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
stock_sm.GetStockQuotes:
http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx?WSDL
Diverse Currency Exchange Web Services
currency_exchange.getRate:
http://www.xmethods.net/sd/CurrencyExchangeService.wsdl
currency_convert.ConversionRate:
http://www.webservicex.com/CurrencyConvertor.asmx?wsdl

Architecture with horizontal composition includes a “Service Resolver” which adjudicates the responses from the all diverse Web Services and returns an adjudicated response to the consumer. In the simplest case “Service Resolver” is a “Voter” (i.e. performing majority voting using the responses from diverse Web

Services). Papers [10-13] introduce special components (called "Mediator", "Proxy", "Service Container" or "Wrapper") performing similar functionality.

Combined vertical and horizontal composition supports two possible operating modes:

1. Concurrent-alternative execution (Fig. 2-a) with voting at the each level.
2. Sequential-alternative execution (Fig. 2-b) with voting at the top level.

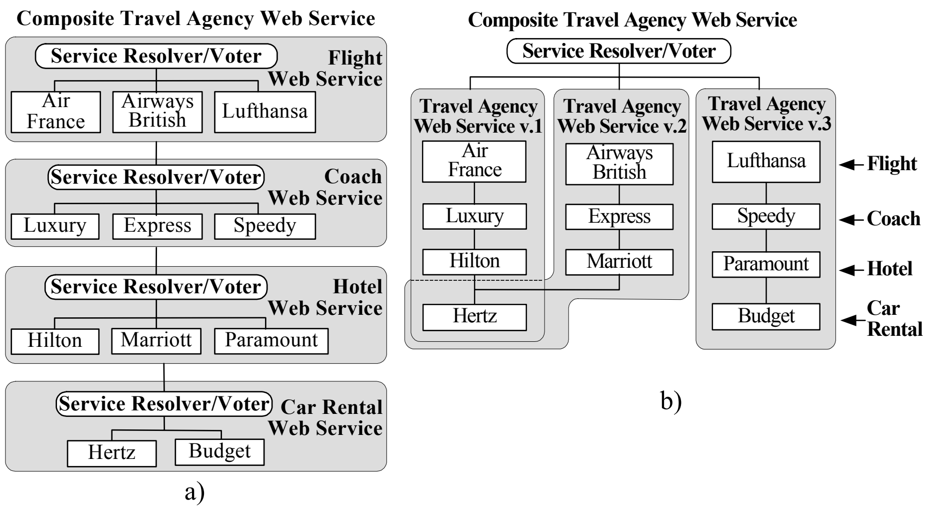


Fig. 2. Combined vertical and horizontal Web Service composition: a) concurrent-alternative execution; b) sequential-alternative execution

3 Middleware-based Architecture Supporting Horizontal Composition

In [9] we proposed an architecture which uses a middleware for a managed dependable upgrade and "horizontal" composition of Web Services. The middleware runs several diverse WSs (releases). It intercepts the consumer requests coming through the WS interface, relays them to all the releases and collects the responses from the releases. It is also responsible for 'publishing' the confidence associated with the each target WS.

There are some possible operating modes of the diverse Web Services with several operational releases:

1. *Parallel execution for maximum reliability* (Fig. 3). All available diverse WSs (or releases) are executed concurrently and their responses are used by the middleware to produce an adjudicated response to the consumer of the WS.
2. *Parallel execution for maximum responsiveness* (Fig. 4). All available diverse WSs (releases) are executed concurrently and the fastest non-evidently incorrect response is returned to the consumer of the service as an adjudicated response.

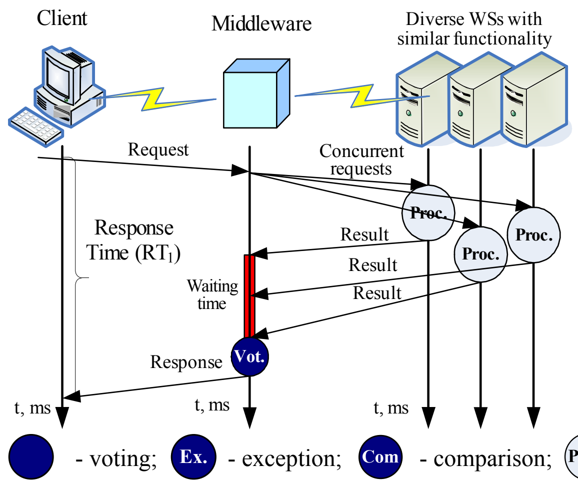
3. *Configured parallel execution* (Fig. 5). All available diverse WSs (releases) are executed concurrently. The middleware may be dynamically configured to wait for up to a certain number of responses to be collected from the deployed releases.
4. *Sequential execution for minimal service capacity* (Fig. 6). The diverse WSs (releases) are executed sequentially. The subsequent releases are only executed if the responses received from the previous releases are evidently incorrect.

Effectiveness of the different operating modes depends on the probability of service unavailability, occurrence of evident (exceptions raised) and non-evident (erroneous results returned) failures. The probability of service unavailability due to different reasons (service overload, network failures, and congestions) is several orders greater than probability of failure occurrence. Moreover, the different exceptions arise during service invocation more frequently than non-evident failures occur.

To analyse effectiveness of the proposed operating modes we developed a simulation model running in MATLAB 6.0 environment and used such initial values:

	Case 1	*Case 2*	*Case 3*
P(exception), P(ex)	*0,09*	*0,05*	*0,09*
P(non-evident failure), P(nf)	*0,01*	*0,05*	*0,01*
P(service unavailability), P(ua)	*0,20*	*0,20*	*0,30*
P(correct response), P(cr)	*0,70*	*0,70*	*0,60*

During simulation we also set *Mean Response Time* (MRT) equals 200 ms and *Maximum Waiting Time* (time-out) equals 500 ms.

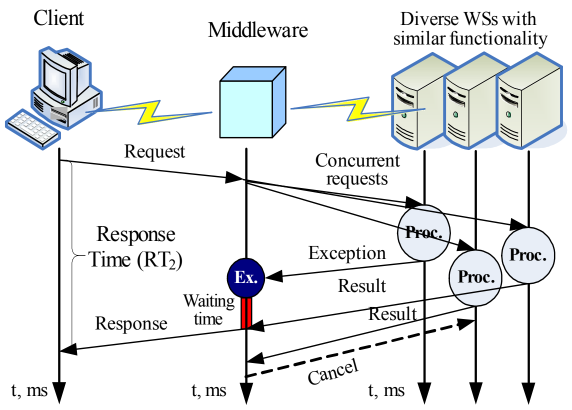


Simulation results:

	Case1	Case2	Case3
P(cr)	0,966	0,936	0,928
P(nf)	0,010	0,048	0,013
P(ex)	0,016	0,008	0,032
P(ua)	0,008	0,008	0,027
MRT	397,60	397,60	431,40

Fig. 3. Simulation results for mode *"Parallel execution for maximum reliability"*

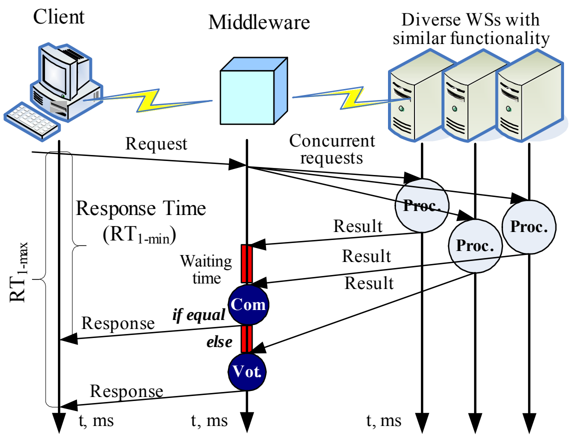
A practical application of the horizontal composition needs to reply on developing new workflow patterns and languages constructs, supporting different operating modes and procedures of multiple results resolving and voting.



Simulation results:

	Case1	Case2	Case3
P(cr)	0,961	0,915	0,922
P(nf)	0,003	0,017	0,005
P(ex)	0,029	0,060	0,046
P(ua)	0,008	0,008	0,027
MRT	141,10	133,85	163,82

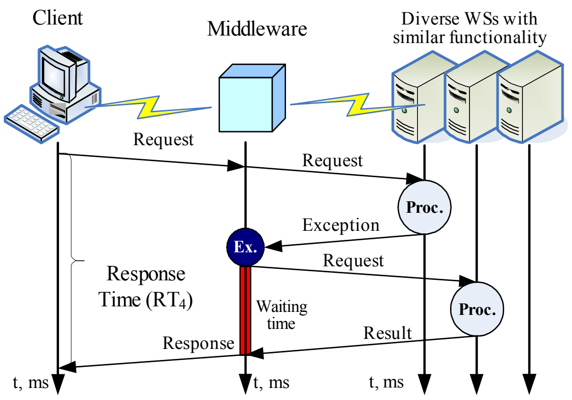
Fig. 4. Simulation results for mode *“Parallel execution for maximum responsiveness”*



Simulation results:

	Case1	Case2	Case3
P(cr)	0,966	0,936	0,928
P(nf)	0,010	0,048	0,013
P(ex)	0,016	0,008	0,032
P(ua)	0,008	0,008	0,027
MRT	289,72	289,08	330,52

Fig. 5. Simulation results for mode *“Configured parallel execution”*



Simulation results:

	Case1	Case2	Case3
P(cr)	0,961	0,915	0,922
P(nf)	0,003	0,017	0,005
P(ex)	0,029	0,060	0,046
P(ua)	0,008	0,008	0,027
MRT	283,99	266,25	387,86

Fig. 6. Simulation results for mode *“Sequential execution for minimal service capacity”*

4 Work-Flow Patterns Supporting Web Services Composition

The workflow patterns capture typical control flow dependencies encountered during workflow modelling. There are more then 20 typical patterns used for description different workflow constructions of "vertical" composition [14]. The basic ones are: Sequence, Exclusive Choice, Simple Merge, Parallel Split, Synchronization, Discriminator, Regular Cycle, etc.

Each WF language describes a set of elements (activities) used for implementing different WF patterns. For example, BPEL4WS defines both primitive (invoke, receive, reply, wait, assign, throw, terminate, empty) and structured (sequence, switch, while, flow, scope) activities. The first ones are used for intercommunication and invoking operations on some web service. Structured activities present of complex workflow structures and can be nested and combined in arbitrary ways.

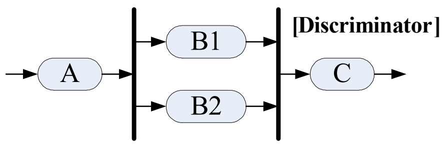


Fig. 7. Workflow pattern "Discriminator"

In fact, the only one of the basic WF pattern - "discriminator" fits for implementing the parallel execution mode for maximum responsiveness. Discriminator (see Fig. 7) is a point in the workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. The first one that comes up with the result should proceed the workflow. The other results will be ignored.

However, the only BPML language supports such WF pattern [15, 16] (see Fig. 8).

```
<process name="PatternDiscriminator">
  <sequence> <context>
      <signal name="completed_B"/>
      <process name="B1">
        …
        <raise signal="completed_B"/>
      </process>
      <process name="B2">
        …
        <raise signal=" completed_B"/>
      </process>
    </context>
    <action name="A" …>
      …
    </action>
    <all>
      <spawn process="B1"/>
      <spawn process="B2"/>
    </all>
    <synch signal="completed_B"/>
    <action name="C" …>
      …
    </action>
  </sequence> </process>
```

Fig. 8. BPML implementation of the pattern "Discriminator"

To support different execution modes (workflows) within the “horizontal” composition the additional WF patterns need to be developed and implemented for different WF languages. The new activities allowing a business process to support redundancy and perform *voting* procedure should be also developed. This is a motivation of our further work.

5 Conclusions

We have addressed different aspects of a Web Services composition which extend functionality (vertical composition) or improve dependability (vertical composition). Vertical composition uses redundancy which based on natural diversity of existed Web Services with the identical or similar functionality deployed by third parties. We discussed middleware-based architecture that provides dependable “horizontal” composition of Web Services. For the best result middleware have to implement on-line monitoring and flexible control. However, questions like “How many composition level will provide the maximal improvement?”, “When middleware should be placed?” are yet unsolved. The technologies supporting WS composition also should be improved.

“Horizontal” composition, which uses redundancy in combination with diversity, is one of the basic means of enhancing service availability and providing fault-tolerance. Different operational modes are applicable here. “Configured parallel execution” mode gives maximal reliability (as well as “Parallel execution for maximum reliability” mode) and acceptable response time. “Parallel execution for maximum responsiveness” mode provides minimal response time (even less than MRT of target WS) and also improves service availability. An unexpected result was that “Sequential execution for minimal service capacity” mode both improves availability and provides rather good response time.

Applying in practice techniques of horizontal composition and other means of improving SOA dependability require developing new workflow patterns and implementing them in different WF languages. In our future work we are going to deal with BPEL4WS which is a modern and popular language for the specification of business processes and business interaction protocols. It supports extensibility by allowing namespace-qualified attributes to appear on any standard element and by allowing elements from other namespaces to appear within BPEL4WS defined elements.

Novel approaches for *custom-oriented quality and dependability control* in service oriented architectures implies that service customer will be able to choice necessary operating modes for each particular request according to one’s needs. It can be done explicitly or implicitly by using extended WSDL and SOAP tags.

Acknowledgments. Alexander Romanovsky is particularly supported by the IST RODIN project.

References

1. W3C Working Group. “Web Services Architecture”, http://www.w3.org/TR/ws-arch/ (2004)
2. Ferguson, D.F., Storey, T., Lovering, B., Shewchuk, J. “Secure, Reliable, Transacted Web Services: Architecture and Composition”. Microsoft and IBM Technical Report, http://www-106.ibm.com/developerworks/webservices/library/ws-securtrans (2003)
3. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N. “Dependability in the Web Service Architecture”. In: Architecting Dependable Systems. Springer-Verlag (2003) 89–108.
4. Leymann, F. “Web Services Flow Language”. Technical report, IBM (2001)
5. Thatte, S. “XLANG: Web Services for Business Process Design”. Technical report, Microsoft (2001).
6. Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. “WSDL: Web services description language”, http://www.w3.org/TR/wsdl (2001)
7. Andrews, T., F. Cubera, H. Dholakia, et all. “Business Process Execution Language for Web Services Version 1.1”. OASIS, http://ifr.sap.com/bpel4ws (2003).
8. Ankolekar et all. “Ontology Web Language for Services (OWL-S)”, http://www.daml.org/services (2002)

9 Gorbenko, A., Kharchenko, V., Popov, P., Romanovsky, A. “Dependable Composite Web Services with Components Upgraded Online”. In R. de Lemos (Eds.) et al. Architecting Dependable Systems III, LNCS 3549. Berlin, Heidelberg: Springer-Verlag (2005) 92–121

10. Hall, S., Dobson, G. and Sommerville, I. “A Container-Based Approach to Fault Tolerance in Service-Oriented Architectures”, http://digs.sourceforge.net/papers/2005-icse-paper.pdf (2005)
11. Maheshwari, P., Erradi, A. “Architectural Styles for Reliable and Manageable Web Services”, http://mercury.it.swin.edu.au/ctg/AWSA05/Papers/erradi.pdf (2005)
12. Chen, Yu., Romanovsky, A., Li, P. “Web Services Dependability and Performance Monitoring”. Proc. 21st Annual UK Performance Engineering Workshop (UKPEW'2005), http://www.staff.ncl.ac.uk/nigel.thomas/UKPEW2005/ukpew-proceedings.pdf (2005)
13. Townend, P. Groth, P. Xu, J. “A Provenance-Aware Weighted Fault Tolerance Scheme for Service-Based Applications”. Proc. of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (2005)
14. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede “Pattern-Based Analysis of BPEL4WS”. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane (Australia), http://is.tm.tue.nl/staff/wvdaalst/publications/p175.pdf (2002)
15. W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. “Pattern-Based Analysis of BPML (and WSCI)”. QUT Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane (Australia), http://is.tm.tue.nl/research/patterns/download/qut_bpml_rep.pdf (2002)
16. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. “Analysis of Web Service Composition Languages: The Case of BPEL4WS”. Proc. of the 22nd Int. Conf. on Conceptual Modeling (ER), Chicago, USA (2003)

A Method of Multiversion Technologies Choice on Development of Fault-Tolerant Software Systems

Vladimir Sklyar and Vyacheslav Kharchenko

National Aerospace University “KhAI”, Computer Systems and Networks Department,

Chkalova str. 17, Kharkiv, 61070, Ukraine

V.Kharchenko@khai.edu

Abstract. A problem of optimal multiversion technologies (MVT) choice on development of fault-tolerant software systems is formalised by use of a graph life cycle model of multiversion software systems (MSS). There are four main variants of target setting according to criteria “diversity-cost” which look like problems of searching the shortest/maximum path in oriented graph or like problems of dynamic programming. Results of implementing method on development of MSS for Nuclear Power Plant (NPP) reactor protection systems are discussed.

Keywords: optimal choice, a multiversion technology, diversity metrics.

1 Introduction

Diversity is an efficient method for defence from design defects and for development fault-tolerant software systems [1,2]. There are following types of diversity: human, design, functional, signal, software, hardware [3].

There are the following types of models for MSS formal description and development:

– a structural-functional model describing a structure of version that are applied for realisation of MSS functions [4];

– a Bayesian model based on calculation of conditional probabilities of MSS version failures [5];

– an automata model describing a process of MSS input data transformation into output data on the base of automata theory [6];

– a theoretical-set model which presents MSS software and hardware as a composition of common for the all version cores and of distributed shells that permits to descript of faults appearance processes [7].

A problem of MVT choice i.e. diversity types choice for MSS life cycle stages taking into account their compatibility remains undecided in spite of variety of models. High price of MVT realisation raises a problem of resources distribution for receiving of maximum effect for probability of common case failures decreasing. Peculiarities of cost assessment of MVT realisation are committed in proceedings [3,4]. It is expedient to use diversity metrics [7,8] which permits to assess versions multiplicity as well as to estimate probability of common case failures and MSS dependability as a whole for assessment of an effect from MVT using.

The paper's objective is to formalise and to decide a problem of MVT optimal choice in a general form. For that a graph life cycle model of MSS is proposed in section 2. Problems setting and peculiarities of optimal MVT choice and results of method implementation are described in sections 3 and 4 accordingly.

2 A graph model of life cycle of multiversion software systems

MSS life cycle is represented as a sequence of N stages. It is possible to apply M_i diversity types for every stage of MSS life cycle. A choice of MVT consists in sequential choice of MSS diversity types (means of version redundancy implementation) $j = 1,...,M_i$ for every life cycle stage $i = 1,...,N$. Moreover for some life cycle stages can be chosen one version technology of development when diversity is not used. Every MSS diversity type $j = 1,...,M_i$ is characterised by two indicators: by a diversity metric d_{ij} (diversity degree) and by a cost c_{ij} (a cost increment in comparison with one version life cycle stage).

Thus a possible solution space for MVT choice is described by to matrixes: by a matrix of diversity metric values $MD = \| d_{ij} \|$ and by a matrix of cost values $MC = \| c_{ij} \|$, $i = 1,...,N$; $j = 1,...,M_i$. Matrixes MD and MC should be enlarged by null-rows with N elements $D_0 = \| 0,...,0 \|$ and $C_0 = \| 0,...,0 \|$ taking into account a possibility of one version implementation of MSS life cycle stages.

Hence a model of MSS life cycle can be presented as a N-levels graph $G = \langle V,U \rangle$, where $V = \{v_{ij}\}$ – a set of nodes, $U = \{(v_{i1j1}, v_{i2j2})\}$ – a set of edges, every edge is defined by a couple of joint nodes. If a graph G is enlarged by the initial node V_S and by the terminal node V_F, it will look like a bipolar network (Fig. 1).

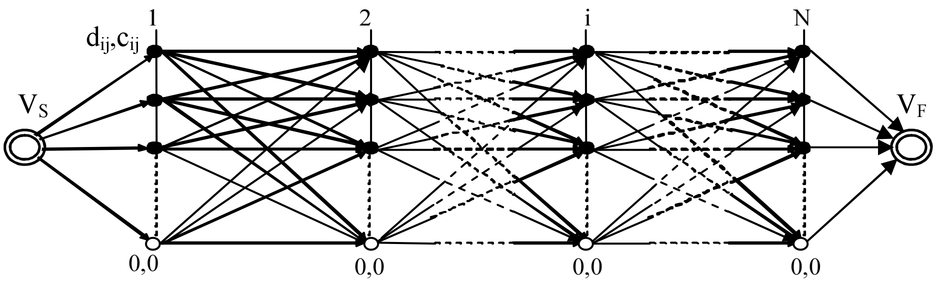


Fig. 1. A graph model of life cycle of multiversion software systems.

Formally MVT of MSS development can be described as a way in a graph G $MVT = \{V_S, v_{1C}, v_{2C},..., v_{iC}, v_{NC}, V_F\}$ (when v_{iC} are nodes chosen for life cycle stages $i = 1,...,N$) which contains at least one non-null node $v_{iC} \neq v_{i0}$. MVT is characterised by integral diversity metric $D=\sum_{i=1}^{N} d_{iC}$ and by integral cost (additional in respect of one version technology) $C=\sum_{i=1}^{N} c_{iC}$, when d_{iC} and c_{iC} are diversity metric and cost values for diversity types chosen in life cycle stages $i = 1,...,N$.

3 Optimal choice of multiversion technologies

A problem of MVT optimal choice can have four variants of setting. Algorithms of decision have been received for every variant.

The problem 1 is a problem of MVT choice with anyone diversity level ($D \neq 0$) and with the minimal cost C_{min}. This problem is a problem of search of the shortest way through graph G nodes which are labelled by values c_{ij}. In the common case the shortest way is the way through nodes v_{i0} for which $c_{i0} = 0$. However with that the condition $D \neq 0$ is not met inasmuch as it is one version technology. Therefore the shortest way met the condition $D \neq 0$ is the way which includes one node $v_{iC}(c_{ij\,min})$ and all other nodes $v_{iC}(c_{i0}) = 0$.

<u>The problem 2</u> is a problem of MVT choice with the maximal diversity level D_{max} without a cost limits. This problem is a problem of search of the longest way through graph G nodes which are labelled by values d_{ij}. The longest way is the way which includes nodes $v_{iC}(d_{i\,max})$ for every life cycle stage $i = 1,...,N$.

<u>The problem 3</u> is a problem of MVT choice with the minimal cost and with a required diversity level ($C \rightarrow \min$ and $D \geq D_{req}$). This problem is a problem of dynamic programming. An objective function is $f(C)=\sum_{i=1}^{N} c_{iC} / d_{iC} \rightarrow \min$ with limitation: $\sum_{i=1}^{N} d_{iC} \geq D_{req}$. A problem solution is a way in a graph G $MVT(C \rightarrow \min) = \{V_S,\ v_{1C}\left(c_{1j}/d_{1j}=\min\right),\ v_{2C}\left(c_{2j}/d_{2j}=\min\right), ...,\ v_{iC}\left(c_{ij}/d_{ij}=\min\right),...,\ v_{NC}\left(c_{Nj}/d_{Nj}=\min\right), V_F\}$.

<u>The problem 4</u> is a problem of MVT choice with the maximal diversity level and with a limited cost ($D \rightarrow \max$ and $C \leq C_{lim}$). This problem is a problem of dynamic programming. An objective function is $f(D)=\sum_{i=1}^{N} d_{iC}/c_{iC} \rightarrow \max$ with limitation: $\sum_{i=1}^{N} c_{iC} \leq C_{lim}$. A problem solution is a way in a graph G $MVT(D \rightarrow \max) = \{V_S,\ v_{1C}\left(d_{1j}/c_{1j}=\max\right),\ v_{2C}\left(d_{2j}/c_{2j}=\max\right),...,\ v_{iC}\left(d_{ij}/c_{ij}=\max\right),...,\ v_{NC}\left(d_{Nj}/c_{Nj}=\max\right), V_F\}$.

As an illustration an algorithm of the problem 3 decision is presented on the Fig. 2. A matrix with elements c_{ij}/d_{ij} corresponding nodes of graph G should be formed for decision of a dynamic programming problem. After that an iterative procedure is realised. A diversity type with the minimal values of relation "cost / diversity" should be chosen for every step of an iterative procedure. Chosen diversity type is included in MVT for corresponding life cycle stage and the corresponding column is deleted from the base matrix. If $D \geq D_{req}$ then an iterative procedure is stopped else an iterative procedure is continued.

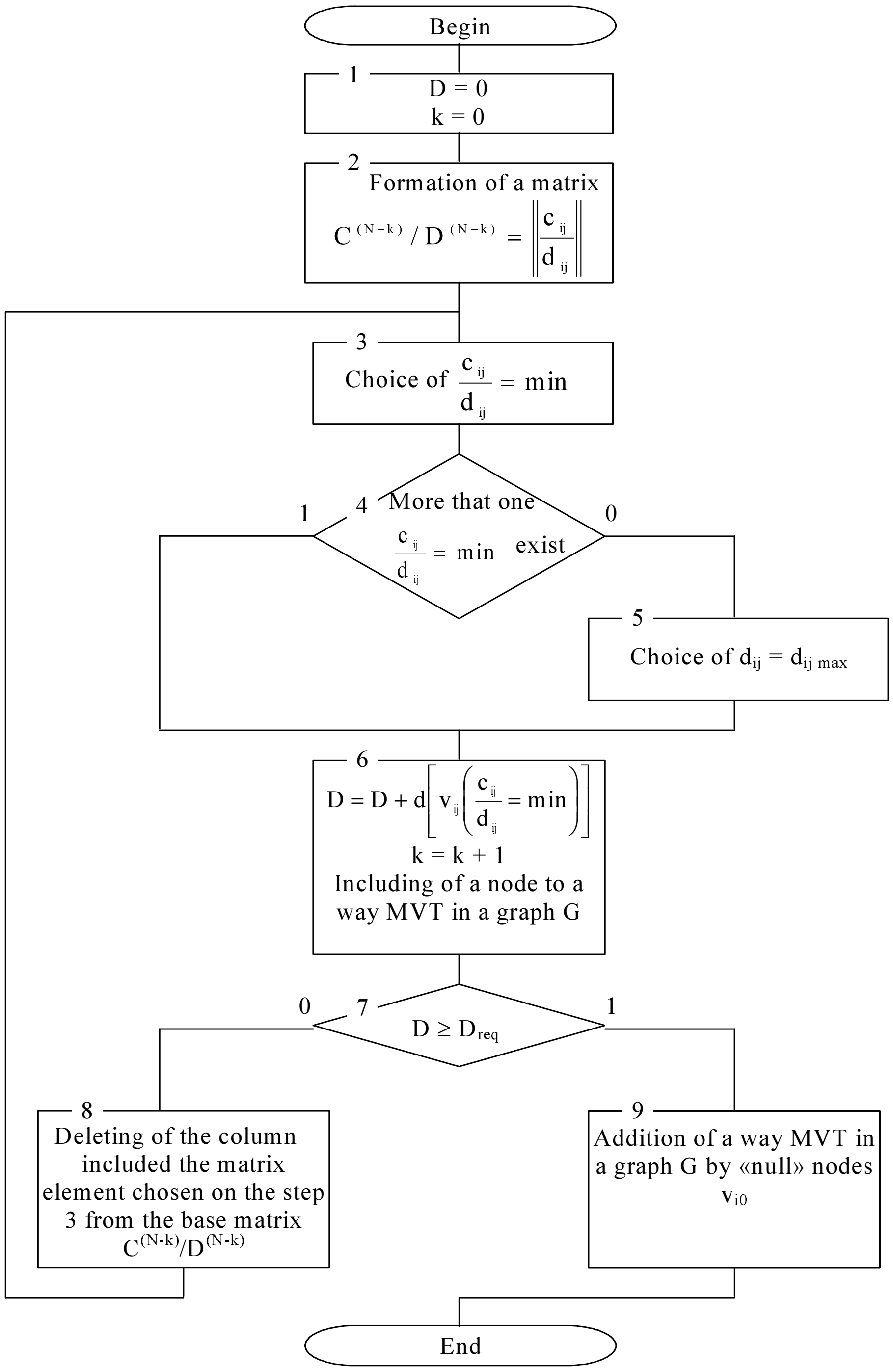

Begin
1
D = 0
k = 0
2
Formation of a matrix
$C^{(N-k)} / D^{(N-k)} = \left\| \frac{c_{ij}}{d_{ij}} \right\|$
3
Choice of $\frac{c_{ij}}{d_{ij}} = \min$
4
More that one $\frac{c_{ij}}{d_{ij}} = \min$ exist
1
0
5
Choice of $d_{ij} = d_{ij\,max}$
6
$D = D + d\left[v_{ij}\left(\frac{c_{ij}}{d_{ij}} = \min \right) \right]$
k = k + 1
Including of a node to a way MVT in a graph G
7
$D \geq D_{req}$
0
1
8
Deleting of the column included the matrix element chosen on the step 3 from the base matrix $C^{(N-k)}/D^{(N-k)}$
9
Addition of a way MVT in a graph G by «null» nodes v_{i0}
End

Fig. 2. An algorithm of the problem 3 decision.

4 Definition of input data for a method implementation

Input data for a method implementation a defined for Field Programmable Gates Arrays based (FPGA-based) systems which can be considered as a kind of software systems. Life cycle stages i = 1,...,4 with possibilities of diversity implementation for FPGA-based systems are the following (Fig. 3): 1) development of schemes of control algorithms; 2) development of programme models of control algorithms in CASE-tools area; 3) integration of programme models of control algorithms in CASE-tools area; 4) implementation of integrated programme model into FPGA.

Some variants of implementation have proposed for every diversity type (Table 1). Diversity metrics values have established as ranking from the minimum difference between diverse variants to the maximum difference.

Table 1. Variants of diversity type's implementation and diversity metrics values

Diversity types	Variants of diversity type's implementation	Diversity metrics values
A Diversity of programmable components	Diversity of firm-developers of FPGAs (A1)	4
	Diversity of technologies of FPGAs producing (A2)	3
	Diversity of FPGAs families (A3)	2
	Diversity of FPGAs from the same family (A4)	1
B Diversity of CASE-tools	Diversity of developers of CASE-tools (B1)	3
	Diversity of CASE-tools (B2)	2
	Diversity of configurations of CASE-tools (B3)	1
C. Diversity of languages of FPGA projects development	Diversity on the base of graphical language and hardware description language (C1)	2
	Diversity of hardware description languages (C2)	1
D. Diversity of specification of schemes	Diversity of FPGAs specification languages (D1)	1

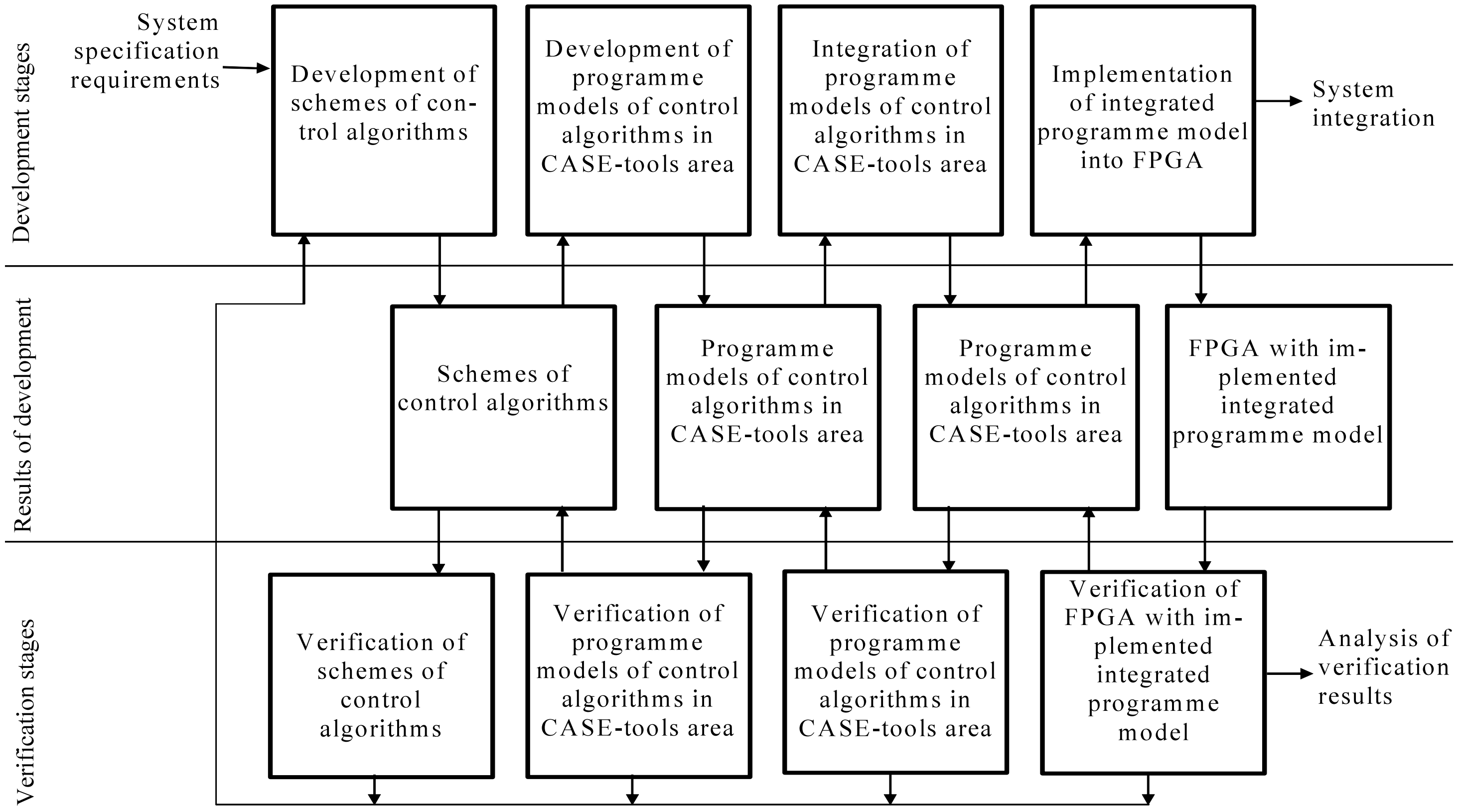

Development stages
Results of development
Verification stages
System specification requirements
Development of schemes of control algorithms
Development of programme models of control algorithms in CASE-tools area
Integration of programme models of control algorithms in CASE-tools area
Implementation of integrated programme model into FPGA
System integration
Schemes of control algorithms
Programme models of control algorithms in CASE-tools area
Programme models of control algorithms in CASE-tools area
FPGA with implemented integrated programme model
Verification of schemes of control algorithms
Verification of programme models of control algorithms in CASE-tools area
Verification of programme models of control algorithms in CASE-tools area
Verification of FPGA with implemented integrated programme model
Analysis of verification results

Fig. 3. A structure of life cycle of FPGA-based systems.

Results of compatibility analysis of diversity types with life cycle stages are given in the Table 2. Possible diversity types for every life cycle stage are signed by “+”.

Table 2. Variants of diversity type’s implementation and diversity metrics values

Diversity types	Life cycle stages			
	Development of schemes of control algorithms	Development of programme models of control algorithms in CASE-tools area	Integration of programme models of control algorithms in CASE-tools area	Implementation of integrated programme model into FPGA
A Diversity of programmable components	–	–	–	+
B Diversity of CASE-tools	+	+	+	+
C. Diversity of languages of FPGA projects development	–	+	+	–
D. Diversity of specification of schemes	+	–	–	–

5 Implementation of a method

The developed method has been applied for design of a two-version reactor protection system for NPPs. One from the peculiarities of a reactor protection system is development of a software part in FPGAs. Such projects of FPGAs include three type of software: 1) graphical schemes; 2) program code in language VHDL; 3) software in language C, which operates in environment of emulators of microprocessor cores.

Diversity types for different life cycle stages have been taken into account for values definition of d_{ij} and c_{ij} for a FPGA-based reactor protection system. Diversity metrics which take into account the following aspects have been proposed: differences for every diversity type; integral differences for every life cycle stage; integral differences for MSS life cycle as a hole.

Values of diversity metrics are established as the set $d_{ij} = \{0,1,2,3\}$ (see Table 2) and are determined according to appropriate diversity type influence to probability of common case failures. Such probabilities in turn can be defined operation and development experience as well as on the base of expert assessment. Diversity metrics are calculated by additive formulas for taking into account integral difference between MSS versions. The final results of diversity type choice and diversity metrics calculation for life cycle stages are given in the Table 3.

Table 3. Results of using a method of multiversion technologies choice

Life cycle stages	Chosen diversity types	Diversity metrics values
Development of schemes of control algorithms	Diversity of CASE-tools	1
Development of programme models of control algorithms in CASE-tools area	Diversity of languages	3
Integration of programme models of control algorithms in CASE-tools area	Diversity of languages	3
Implementation of integrated programme model into FPGA	Diversity of programmable components	3

Use of diversity types like in table 1 permits ensure a value of integral diversity metric D = 10. A cost of development of diverse programmable components increases twice as much. However, using diversity permits decrease common case failures intensity half in comparison with one version reactor protection system.

6 Conclusion

Proposed method of optimal MVT choice is one of the practical solutions allowing to realise a cost-effective approach to developing safety-critical systems for which use of diversity are normative requirement.

Application of this method can ensure required dependability of MSS and minimal cost due to directed selection of diversity kinds.

Next steps of research may be connected with working out in detail of the algorithms choice taking into account different technologies of MSS development and dependence of diversity metrics values for different life cycle stages.

References

1. Avizienis A., Laprie J.-C., Randell B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. on Dependable and Secure Computing 1 (2004)
2. Lyu, M.: Software Fault Tolerance. Wiley (1995)
3. Preckshot, G.: Method for Performing Diversity and Defense-in-Depth Analyses of Reactor Protection Systems. Lawrence Livermore National Laboratory (1994)
4. Kharchenko, V.: Multiversion Systems: Models, Reliability, Design, Technologies. Proc. by 10th European Conf. on Safety and Reliability. Munich (1999).
5. Littlewood, B., Strigini, L.: A discussion of practices for enhancing diversity in software designs. Technical report. Centre for Software Reliability, London (2000)
6. Kharchenko, V., Sklyar, V., Golovir, V.: Automata models of multiversion instrumentation and control systems. Radio-electronic and Computer Systems 4 (2007) (In Russian)
7. Kharchenko, V., Yastrebenetsky, M., Sklyar, V.: Diversity Assessment of Nuclear Power Plants Instrumentation and Control Systems. Proc. by 7th International Conf. on Probabilistic Safety Assessment and Management and European Safety and Reliability Conf. "PSAM 7 – ESREL ´04". Berlin (2004)
8. Vogs, U.: Software Diversity. Reliability Engineering and System Safety. 43 (1994)