Modelling Learning Behaviour of Intelligent Agents Using UML 2.0

by

Hossam Allam

A thesis submitted to the University of Plymouth

In partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing, Communications, and Electronics

Faculty of Technology

University of Plymouth

2005

Modelling Learning Behaviour of Intelligent Agents Using UML 2.0

Hossam Allam

Abstract

This thesis aims to explore and demonstrate the ability of the new standard of structural and behavioural components in Unified Modelling Language (UML 2.0 / 2004) to model the learning behaviour of Intelligent Agents. The thesis adopts the research direction that views agent-oriented systems as an extension to object-oriented systems. In view of the fact that UML has been the de facto standard for modelling object-oriented systems, this thesis concentrates on exploring such modelling potential with Intelligent Agent-oriented systems. Intelligent Agents are Agents that have the capability to learn and reach agreement with other Agents or users. The research focuses on modelling the learning behaviour of a single Intelligent Agent, as it is the core of multi-agent systems.

During the writing of the thesis, the only work done to use UML 2.0 to model structural components of Agents was from the Foundation for Intelligent Physical Agent (FIPA). The research builds upon, explores, and utilises this work and provides further development to model the structural components of learning behaviour of Intelligent Agents. The research also shows the ability of UML version 2.0 behaviour diagrams, namely activity diagrams and sequence diagrams, to model the learning behaviour of Intelligent Agents that use learning from observation and discovery as well as learning from examples of strategies. The research also evaluates if UML 2.0 state machine diagrams can model specific reinforcement learning algorithms, namely dynamic programming, Monte Carlo, and temporal difference algorithms. The thesis includes user guides of UML 2.0 activity, sequence, and state machine diagrams to allow researchers in agent-oriented systems to use the UML 2.0 diagrams in modelling the learning components of Intelligent Agents.

The capacity for learning is a crucial feature of Intelligent Agents. The research identifies different learning components required to model the learning behaviour of Intelligent Agents such as learning goals, learning strategies, and learning feedback methods. In recent years, the Agent-oriented research has been geared towards the agency dimension of Intelligent Agents. Thus, there is a need to conduct more research on the intelligence dimension of Intelligent Agents, such as negotiation and argumentation skills.

The research shows that behavioural components of UML 2.0 are capable of modelling the learning behaviour of Intelligent Agents while structural components of UML 2.0 need extension to cover structural requirements of Agents and Intelligent Agents. UML 2.0 has an extension mechanism to fulfil Agents and Intelligent Agents for such requirements. This thesis will lead to increasing interest in the intelligence dimension rather than the agency dimension of Intelligent Agents, and pave the way for object-oriented methodologies to shift more easily to paradigms of Intelligent Agent-oriented systems.

Table of Contents

Table of Contents 2

List of Figures 5

Acknowledgments 11

Chapter 1: Introduction 13

1.1 AGENT-ORIENTED RESEARCH DIRECTIONS 14

1.2 AGENTS AND OBJECTS 14

1.3 INTELLIGENT AGENTS 17

1.4 UNIFIED MODELLING LANGUAGE (UML 2.0) DIAGRAMS 17

1.5 OVERVIEW OF THE THESIS 29

Chapter 2: Learning Capabilities of Intelligent Agents 31

2.1 INTRODUCTION 31

2.2 LEARNING 32

2.3 LEARNING GOAL 33

2.4 KNOWLEDGE TYPES AND REPRESENTATIONS OF THE LEARNING PROCESS 36

2.4.1 Knowledge Types 36

2.4.2 Knowledge Representations 37

2.4.3 Background Knowledge 40

2.4.4 Relation between Background Knowledge and Input Knowledge 40

2.5 TYPES OF LEARNING STRATEGIES 43

2.5.1 Rote Learning 43

2.5.2 Learning from Instructions 44

2.5.3 Learning from Examples 44

2.5.4 Learning from Observation and Discovery 46

2.5.5 Learning by Analogy 47

2.6 LEARNING FEEDBACK METHODS 48

2.6.1 Supervised Learning 49

2.6.2 Unsupervised Learning 49

2.6.3 Reinforcement Learning 49

2.7 LEARNING LOCATION 52
2.8 LEARNING SCHEDULE AND DURATION 52
2.9 EXECUTING THE NEW HYPOTHESIS 53
2.10 LEARNING AND COMMUNICATION 53
2.11 CONCLUSION 54
Chapter 3: Research Directions of Extending UML to Model Agent-Oriented Systems 56
3.1 INTRODUCTION 56
3.2 AGENT UNIFIED MODELLING LANGUAGE (AUML) 57
3.2.1 Extending UML Behaviour Component 57
3.2.2 Extending UML Structural Component 68
3.3 AGENT GRAPH TRANSFORMATION MODELLING 72
3.4 MESSAGE / UML 75
3.5 AGENT-OBJECT RELATION MODELS 78
3.6 CONCLUSION 81
Chapter 4: Exploring UML 2.0 to Model Structural Components of Intelligent Agents 83
4.1 INTRODUCTION 83
4.2 MODELLING INTELLIGENT AGENTS 84
4.3 LEARNING COMPARTMENT ATTRIBUTES: 86
4.3.1 Learning Goal Attribute 88
4.3.2 Commitment Strategy Attribute 88
4.3.3 Background Knowledge Attribute 89
4.3.4 Learning Strategy Attribute 90
4.3.5 Learning Feedback Attribute 94
4.3.6 Learning Location Attribute 96
4.3.7 Learning Schedule Attribute 97
4.3.8 Knowledge Representation Attribute 97
4.4 CONCLUSION 97
Chapter 5: Exploring UML 2.0 to Model the Learning Behaviour of Intelligent Agents 99
5.1 INTRODUCTION 99
5.2 MODELLING LEARNING STRATEGIES OF INTELLIGENT AGENTS 100
5.2.1 Learning from Observation and Discovery Strategy 101

5.2.2 Learning from Examples Strategy *119*

5.3 MODELLING LEARNING FEEDBACK METHODS OF INTELLIGENT AGENT 128

5.3.1 Reinforcement Learning Feedback Methods *128*

5.4 CONCLUSION 151

Chapter 6: Conclusions and Future Research Directions **153**

6.1 THESIS CONCLUSIONS 153

6.2 FUTURE RESEARCH DIRECTIONS 156

References **158**

Further Reading **165**

Appendix A: UML 2.0 Activity Diagram User Guide **173**

SECTION A.1: ACTIVITY DIAGRAM COMPONENTS GUIDE 173

SECTION A.2: STEP BY STEP UML 2 ACTIVITY DIAGRAM DRAWING GUIDE 181

SECTION A.3: TABLE OF ACTIVITY DIAGRAM COMPONENTS 184

Appendix B: UML 2.0 Sequence Diagram User Guide **187**

SECTION B1: SEQUENCE DIAGRAM COMPONENTS GUIDE 187

SECTION B.2: STEP BY STEP UML 2 SEQUENCE DRAWING GUIDE: 195

SECTION B.3: TABLE OF SEQUENCE DIAGRAM COMPONENTS 198

Appendix C: UML 2.0 State Machine Diagram User Guide **202**

SECTION C.1 STATE MACHINE DIAGRAM COMPONENTS GUIDE 202

SECTION C.2: STEP-BY-STEP STATE MACHINE DIAGRAM DRAWING GUIDE 212

SECTION C.3: TABLE OF STATE MACHINE DIAGRAM COMPONENTS 214

List of Figures

Figure 1.1 UML structure and behaviour diagrams 18

Figure 1.2 Class notations 19

Figure 1.3 Class Diagram 20

Figure 1.4 Class Diagram 20

Figure 1.5 Object diagram 21

Figure 1.6 Example of component diagram 21

Figure 1.7 Example of Composite structure 22

Figure 1.8 Example of deployment diagram 23

Figure 1.9 Package diagram 23

Figure 1.10 Example of Use case and actors for an ATM system 24

Figure 1.11 Example of activity diagram of process order that utilizes multi-partitions 25

Figure 1.12 Example of state machine diagram for a telephone 26

Figure 1.13 Example of sequence diagram 27

Figure 1.14 Example of communication diagram for displaying seminar screen 28

Figure 1.15 Example of timing diagram 28

Figure 1.16 Example of compact timing diagram 29

Figure 2.1 Conceptual graph and frame descriptions of a hotel bed 38

Figure 3.1 English-Auction protocol for surplus flight ticket 58

Figure 3.2 And, XOR, and Or connectors 59

Figure 3.3 Full and abbreviated notation of XOR connection 60

Figure 3.4 A generic AIP expressed as a template package 61

Figure 3.5 Multiple techniques to express concurrent communication with an Agent playing multiple roles or responding to different CAs 62

Figure 3.6 Huget extensions to AUML protocol diagrams 63

Figure 3.7 Huget extensions to AUML protocol diagrams 64

Figure 3.8 Augmented Activity diagram 65

Figure 3.9 Synchronization Point 66

Figure 3.10 FIPA Request Interaction Protocol 67

Figure 3.11 Agent class diagram and its abbreviation 69

Figure 3.12 Different Kinds of Agent classes 69

Figure 3.13 using UML class diagrams to specify Agent behaviour and its abbreviations 70

Figure 3.14 incoming and outgoing messages 70

Figure 3.15 FIPA proposed Agent class abstract syntax 72

Figure 3.16 Use Case diagram for banking example 73

Figure 3.17 Graph transformation rule 74

Figure 3.18 Three rules specifying the possible result of each interaction 75

Figure 3.19 Agent centric MESSAGE concepts 76

Figure 3.20 The core elements of External AOR modelling 79

Figure 3.21 A comparison of some important concepts of ER, UML, and AORML 80

Figure 4.1 FIPA proposed Agent class abstract syntax 84

Figure 4.2 Active Class 85

Figure 4.3 : FIPA Agent notation (FIPA, 2004) and proposed Intelligent Agent notation 85

Figure 4.4 Agent class and Agent Physical Classifier 86

Figure 5.1 Activity diagram of scenario 1.1 102

Figure 5.2 Activity diagram of email filter Intelligent Agent 103

Figure 5.3 cluster-actions activity diagram 104

Figure 5.4 Sequence diagram of scenario 1.1 105

Figure 5.5 Activity diagram of scenario 1.2 107

Figure 5.6 Activity diagram of internet proxy server Intelligent Agent 108

Figure 5.7 UML Sequence diagram for scenario 1.2 109

Figure 5.8 UML activity diagram for Scenario 1.3 111

Figure 5.9 UML activity diagram for Internet Proxy Server 112

Figure 5.10 UML sequence diagram for scenario 1.3 113

Figure 5.11 UML activity diagram for Scenario 1.4 114

Figure 5.12 UML Sequence diagram for scenario 1.4 116

Figure 5.13 UML activity diagram for Scenario 1.5 117

Figure 5.14 UML sequence diagram for scenario 1.5 119

Figure 5.15 UML activity diagram for scenario 2.1 121

Figure 5.16 UML sequence diagram for scenario 2.1 122

Figure 5.17 UML activity diagram of scenario 2.2 123

Figure 5.18 UML sequence diagram for scenario 2.2 124

Figure 5.19 UML activity diagram for scenario 2.3 126

Figure 5.20 UML sequence diagram for scenario 2.3 127

Figure 5.21 Policy iteration algorithm 129

Figure 5.22 UML State machine diagram for policy iteration algorithm 130

Figure 5.23 Iterative policy evaluation algorithm 131

Figure 5.24 State Machine Diagram for policy evaluation algorithm 132

Figure 5.25 UML State machine diagram for policy improvement algorithm 133

Figure 5.26 Value iteration algorithm 134

Figure 5.27 UML state machine diagram for value evaluation algorithm 136

Figure 5.28 First-visit Monte Carlo method for V estimating V^{π} 137

Figure 5.29 UML state machine diagram for first-visit Monte Carlo policy evaluation algorithm 137

Figure 5.30 UML state machine diagram for Black Jack game 139

Figure 5.31 Monte Carlo control algorithm assuming exploring starts 140

Figure 5.32 UML state machine diagram for the Monte Carlo control algorithm assuming exploring starts 141

Figure 5.33 Tabular TD(0) for estimating V^{π} 142

Figure 5.34 UML state machine diagram for TD algorithm of figure 5.33 143

Figure 5.35 Sarsa: An on-policy TD control algorithm 144

Figure 5.36 UML state machine diagram for sarsa algorithm 145

Figure 5.37 Q-learning Algorithm 146

Figure 5.38 UML state machine diagram for q-learning algorithm 146

Figure 5.39 Tabular Sarsa (λ) Algorithm 147

Figure 5.40 UML state machine diagram for tabular Sarsa (λ) algorithm 148

Figure 5.41 Tabular version of Watkins's Q(λ) algorithm 149

Figure 5.42 UML state machine diagram for Watkins's Q(λ) algorithm 150

Figure A. 0.1 Example of activity diagram 173

Figure A.0.2 illustration of activity 174

Figure A.0.3 Illustration of Action 174

Figure A.0.4 Example of action constraints 175

Figure A.0.5 Illustration of control flow 175

Figure A.0.6 Illustration of initial node 175

Figure A.0.7 Illustration of final node 176

Figure A.0.8 Illustration of flow final node 176

Figure A.0.9 Example of an object 176

Figure A.0.10 Object flow 177

Figure A.0.11 Another illustration of object flow 177

Figure A.0.12 Data store example 177

Figure A.0.13 Example of decision and merge nodes 178

Figure A.0.14 Example of Fork and Join nodes 178

Figure A.0.15 Example of expansion region 179

Figure A.0.16 Illustration of exception 179

Figure A.0.17 Example of interruptible activity region 180

Figure A.0.18 Example of activity diagram with partition 180

Figure A.0.19 Example of activity diagram 181

Figure A.0.20 Example of decision node 182

Figure A. 0.21 Fork nodes 183

Figure A. 0.22 Notation of join and merge Nodes 183

Figure B.0.1 Example of lifelines 188

Figure 0.2 Example of actor lifeline 188

Figure B.0.3 Examples of message exchange 189

Figure B.0.4 Example of self message 189

Figure B.0.5 Example of lost and found messages 190

Figure B.0.6 Example of creating and destroying an object 190

Figure B.0.7 Example of duration and time constraints notation 191

Figure B.0.8 Example of loop fragment 193

Figure B.0.9 Example of a gate 193

Figure B.0.10 Example of part decomposition 194

Figure B.0.11 Example of state invariant 194

Figure B.0.12 Example of sequence diagram 195

Figure B.0.13 Example of lifelines 196

Figure B.0.14 Example of transferring message through a gate 196

Figure B.0.15 Example of start and end of lifline 197

Figure C.0.1 Example of state machine diagram for a door 202

Figure B.0.2 Notation of a state 203

Figure C.0.3 Example of initial and final states 203

Figure C.0.4 Notation of a transition 204

Figure C.0.5 Notation of state actions 204

Figure C.0.6 Example of self -transition 205

Figure C.0.7 Example of state machine with compound state 205

Figure C.0.8 Example of state machine diagram with composite state 206

Figure C.0.9 Example of entry points 206

Figure C.0.10 Example of entry point 207

Figure C.0.11 Example of exit point 207

Figure C.0.12 Example of choice pseudo-state 208

Figure C.0.13 Example of Junction Pesudo-state 209

Figure C.0.14 Example of terminate pseudo-state 209

Figure C.0.15 Example of history state 210

Figure C.0.16 Example of concurrent regions 211

Figure C.0.17 UML state machine diagram for Black Jack game 212

Acknowledgments

I owe my deep interest in Intelligent Agents to my Director of Studies, Prof. M. Denham of the University of Plymouth, UK, whose patience and enthusiastic supervision have been invaluable to me. I am extremely grateful to Prof. J. Odell who voluntarily provided me with his precious feedback about my research. I also owe a great deal to Dr. Amal Elhadary, Assistant Professor of English, Ain Shams University, who contributed her valuable time and experience in the proofreading process for this thesis.

My appreciation goes to the British Council which awarded me a Chevening Scholarship to study in the UK during the academic years,1997-1999. I would also like to thank the School of Computing, University of Plymouth which waived my tuition fees for the academic year, 1999-2000, and the Arab-British Chamber Charitable Foundation for their financial support towards my tuition fees during the academic years, 2000-2004 .

I am grateful to Mrs. Carole Watson, Research Administrator, University of Plymouth and Ms. Nehad Wahba, American University in Cairo, for their support and attention during my postgraduate study.

My Parents, Prof. Esmat Allam and Prof. Olfat Elbagoury, have been a constant source of support-emotional and moral. Thanks to my parents, I have become interested in pursuing my postgraduate studies.

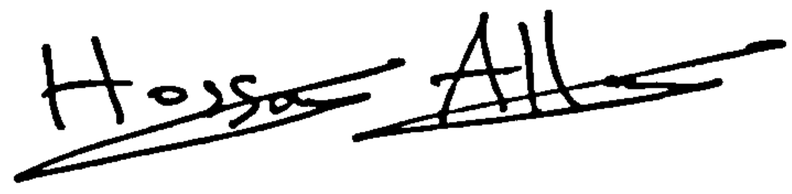
My wife Hanan has been, always, my pillar, my joy, and my guiding light, and I thank her.

AUTHOR'S DECLARATION

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award.

This research has been co-financed with the aid of a scholarship from the British Council, the University of Plymouth and the Arab-British Chamber Charitable Foundation.

The thesis contains 41,435 words, 1371 paragraphs, and 218 pages.

Signed: 

Date ... 11/11/05

Chapter 1: Introduction

This thesis explores the capability of the latest proposed version of the Unified Modelling Language, UML 2.0, to model the learning behaviour of Intelligent Agents. The thesis adopts the research direction that views agent-oriented systems as an extension to object-oriented systems. In view of the fact that UML has been the de facto standard for modelling object-oriented systems, this research concentrates on exploring such modelling potential with Intelligent Agent-oriented systems. The thesis also highlights that Intelligent Agents are Agents which have the capability to learn and to reach agreement with others. The research focuses on the learning capabilities of a single Agent, as it is the core of multi-agent systems. Advancement in information and communication technologies during the last two decades, especially the introduction of the Internet, have triggered the need for new software applications that are capable of performing new tasks that were not available in the past. The Agent-oriented paradigm has emerged as a solution to meet such requirements as working autonomously, acquiring adaptive behaviour, and assisting users in performing their complex tasks. Agent-oriented research has made a considerable effort to describe standard methods and languages for developing Agent-oriented systems; however, the agent-oriented research community did not adopt any of them as a global standard such as the Unified Modelling Language (UML) for object-oriented systems. There is a need to identify a standard method for developing Agent-oriented systems. It is expected that future information systems will contain objects, Agents, and Intelligent Agents.

Section 1.1 introduces Agent-oriented research directions. Section 1.2 highlights the differences between objects and Agents. Section 1.3 discusses the features of Intelligent Agents, such as their ability to learn and reach agreement with others. Section 1.4 introduces the Unified Modelling Language 2.0 structural and behavioural diagrams. Section 1.5 provides an overview of the Thesis.

1.1 Agent-oriented Research Directions

Currently there is no global standard definition for Agents; but there is global agreement on their capabilities. Software applications that have Agent-oriented features are marked by autonomy, proactivity, reactivity, and social ability (Wooldridge, 2000). Agent-oriented research has established two directions to develop Agent-oriented methods and languages. The first direction treats Agent-oriented systems as an extension to object-oriented systems. The second direction views Agent-oriented systems as a new software species with exceptional features. This thesis adopts the first research direction, as linking Agent-oriented standards to object-oriented ones will allow Agent-oriented systems to gain wide acceptance in the software industry (Bauer et al., 2001).

Most of the available literature does not differentiate between Agents and Intelligent Agents. This research highlights that Intelligent Agents are Agents with the ability to learn and to reach agreement with others. Intelligent Agents are best deployed in dynamic environments where they can adapt to new situations. The learning capabilities of Intelligent Agents are crucial for them to sustain successful performance.

1.2 Agents and Objects

This research adopts the first research direction that views Agent-oriented systems as an extension of object-oriented systems. Booch (1994) presents an interesting classification for actor, server, and Agents from the object-oriented point of view. An actor is an active object that uses other objects while a server is a passive object that is used by other objects. Agents are objects that can use other objects and be used by them. However, Booch's classification does not elucidate the main differences between objects and Agents; he only explains that the difference lies in the way objects interact with other objects. Wooldridge (2002) highlights the main differences between Agents and objects as being differing degrees of autonomous behaviour. Agents have a higher degree of

autonomous behaviour than objects. Objects exhibit autonomous behaviour over the state but do not exhibit control over their behaviour. Objects execute their method when they receive a message from another object without having the ability to refuse the execution of their method while Agents can refuse such requests based on their own decision. Agents do not invoke methods on each other like objects; rather, they initiate a request for action from other Agents. In object-oriented cases, the decision to execute depends on the object that invoked the method, while the Agent-oriented case decision to execute depends on the Agent that receives the request for action. Analysts can identify whether the entity under analysis has an Agent's feature by identifying whether the entity has the power of decision in executing its method or not. Odell (2002) points out that Agents employ some degree of unpredictable autonomy in their behaviour, while a conventional object tends toward a more predictable approach.

Brenner et al. (1998) highlight the fact that the internal structure of Agents is more complex than objects. Agents have mental states and concepts in addition to attributes and methods while objects only have attributes and methods. The belief, desire, and intentions of Agents can describe their mental states (Wooldridge, 2002). The belief of Agents corresponds with the information the Agents have about their environment. The desire of Agents represents the states of affairs the Agents wish to bring about. The intention of Agents represents the desire that the Agents are committed to achieve (Wooldridge, 2000).

The method of communication is another major difference between Agents and objects: objects communicate at a low language level (Brenner et al., 1998). Object "A" sends a message to object "B"; so, "B" executes its method. In Agent-oriented systems, Agents use complex communicative language, protocols, and dialogue structures. Agent-oriented analysts would identify the skills of Agents as being able to interact with others. These skills can be negotiation, argumentation, and/or speech act skills. However, analysts do not consider such skills when designing object-oriented systems.

The number of threads in the Agent-oriented systems is different from object-oriented systems. Each Agent has its own thread; but there is only one thread for the whole system in the standard object-oriented systems. Only active objects may have a thread to exhibit some autonomous behaviour just like Agents (Flores-Mendez, 1999). Some researchers consider an active object as an Agent that does not exhibit flexible autonomous behaviour (Wooldridge, 2002). Active objects lack agency features such as the types of interaction as mentioned above. Thus, active objects communicate as objects and not as Agents.

Some of the computer science community consider Agent-oriented programming as a specialisation of object-oriented programming (Flores-Mendez, 1999). Object-oriented programming views systems as consisting of objects communicating with one another to perform internal computation, while Agent-oriented programming specialises this view to have Agents with internal structure based on beliefs, capabilities, and choices that communicate with each other using messages adopted from speech-act theory. Greneserth (1994) emphasises that the meaning of a message in object-oriented programming can vary from one object to another, but in Agent-based programming, Agents use a common language with Agent-independent semantics. Object-oriented programming lies in the fundamentals of object encapsulation, inheritance, and polymorphism, while Agent-oriented programming fundamentals concentrate on a goal-directed execution.

Clear understanding of the differences between agents and objects would allow the extension of object-oriented methods and languages to be used for developing Agent-oriented systems. Unified Modelling Language (UML) is the de facto standard for modelling object-oriented systems. UML unifies and formalizes the methods of developing the object-oriented software life cycle (Bauer, 2002). Object Management Group, an open consortium of companies, has proposed a new release of UML, UML 2.0, which has new behaviour components allowing smooth extension for UML to model Agent-oriented systems.

1.3 Intelligent Agents

This research argues that there are differences between Intelligent Agents and Agents. Intelligent Agents are Agents that have capabilities to learn and to reach agreement with others. To describe the learning behaviour of an Intelligent Agent, analysts should clearly define the learning goal, the knowledge type and representation, the learning strategy, the learning feedback method, the learning location, the learning schedule and duration, the relation between learning and communication, and when the new hypothesis executes. The ability to reach agreement would depend on Intelligent Agents' communication and interaction skills. Analysts should identify whether the Intelligent Agents use speech acts, negotiation, and /or argumentation skills.

This research focuses on the learning components with emphasis on modelling Intelligent Agents' learning behaviour. The research explores the ability of the latest proposed UML 2.0 structural and behavioural component to model the learning behaviour. It also explores the latest extension of UML 2.0 proposed by the Foundation for Intelligent Physical Agent (FIPA, 2004) to model structural components of agent-oriented systems if it is able to model the structural component of the learning behaviour of Intelligent Agents.

1.4 Unified Modelling Language (UML 2.0) Diagrams

Object Management Group[1] (OMG) has adopted UML to act as the standard modelling language for object-oriented systems. UML started by unifying three object-oriented methods developed by Grady Booch, Jim Rumbaugh, and Ivar Jackobson (Fowler, M., 2004). The UML development process started in 1997; The OMG has adopted UML

---

1 The Object Management Group (OMG) is an open membership, not-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

1.1, then revision 1.2, 1.3, 1.4, 1.5, and currently the latest version UML 2.0. UML contains graphic diagrams backed by a single meta-model that are collectively capable of modelling structural and behaviour components of object-oriented systems as shown in figure 1.1. UML is capable of providing sketches, blueprints, and programming language. UML allows specifying, visualising, and documenting models of software systems, including their structure and design. UML is language-independent and vendor-independent; these are the main factors that stimulated OMG to take a leading role in accelerating the development of object-oriented modelling standards. UML has an extensible mechanism that allows it to perform business modelling and modelling of other non-software systems (Object Management Group, 2005).

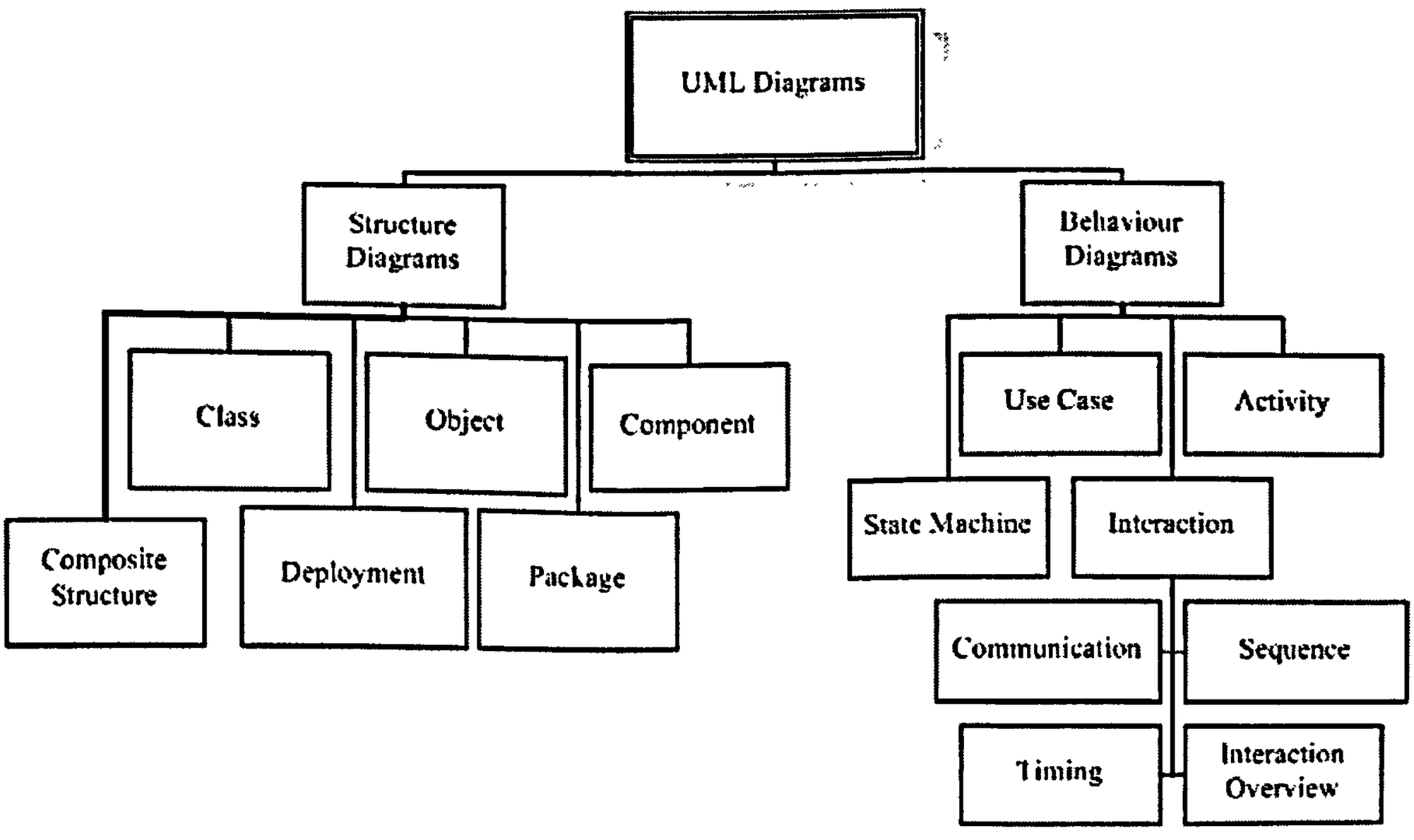


Figure 1.1 UML structure and behaviour diagrams

Structure diagrams depict the static structure components of systems; they show those elements with no consideration of time. The behaviour diagrams illustrate the dynamic behaviour of the system components including their methods, collaborations, activities, and their states history (Object Management Group, 2004). The dynamic behaviour describes the system components over time. Structure diagrams of UML 2.0

have six types of graphs; class, object, component, composite structure, deployment, and package diagrams. Behaviour diagrams have four types of graphs: use case, activity, state machine, and interaction diagrams. Interactions diagram have four types of graphs: sequence, communication, timing, and interaction overview diagrams. Analysts can use all or some of these diagrams to model systems.

Class diagrams describe the basic modelling concepts in UML and especially classes and their relationships. The notation of the class is a rectangle with three compartments; the first holds the name of the class, the second lists the attributes of the class, and operations of the class occupy the last compartment. Figure 1.2 illustrates different notations of classes, according to the level details, and figure 1.3 depicts an example of a class diagram.

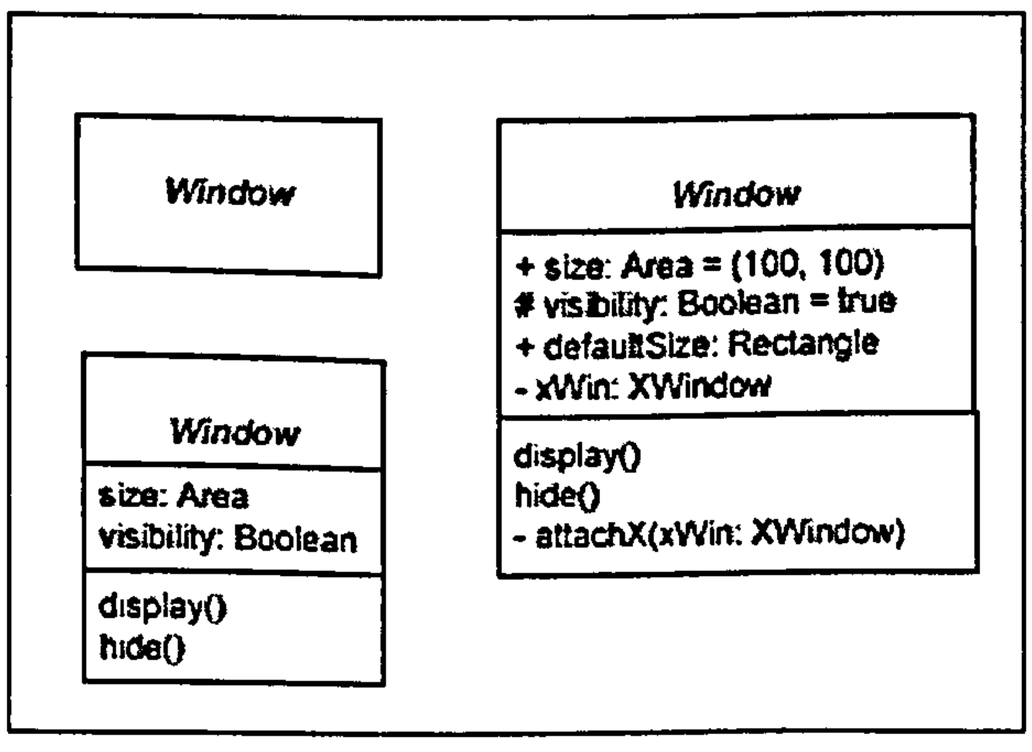


Figure 1.2 Class notations (Object Management Group, 2004)

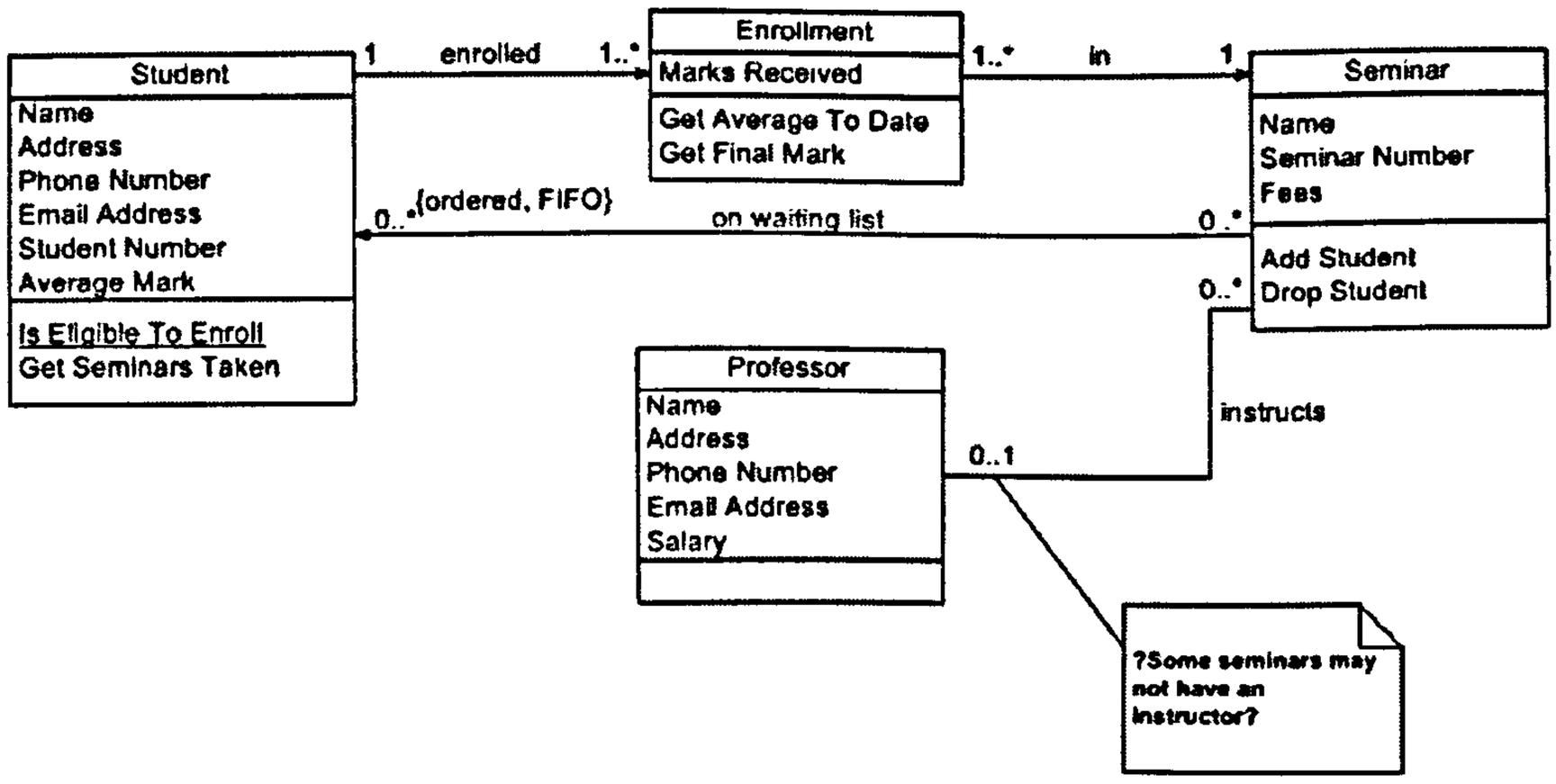


Figure 1.3 Class Diagram (Ambler, 2005b)

Object diagrams provide a snapshot of objects in a system at a point of time. The Object-oriented community often calls an object diagram an instance diagram as it shows instances of classes. They are useful for exploring examples of "real world" objects and the relationships between them (Ambler, 2005a). Object diagrams describe the static structure of a system at a particular time and they test the accuracy of class diagrams. The notation of the object is like a class, but its name is underlined. Figure 1.4 shows a class diagram for association between Plane class and Flight class, while figure 1.5 depicts the object diagram of this class diagram.

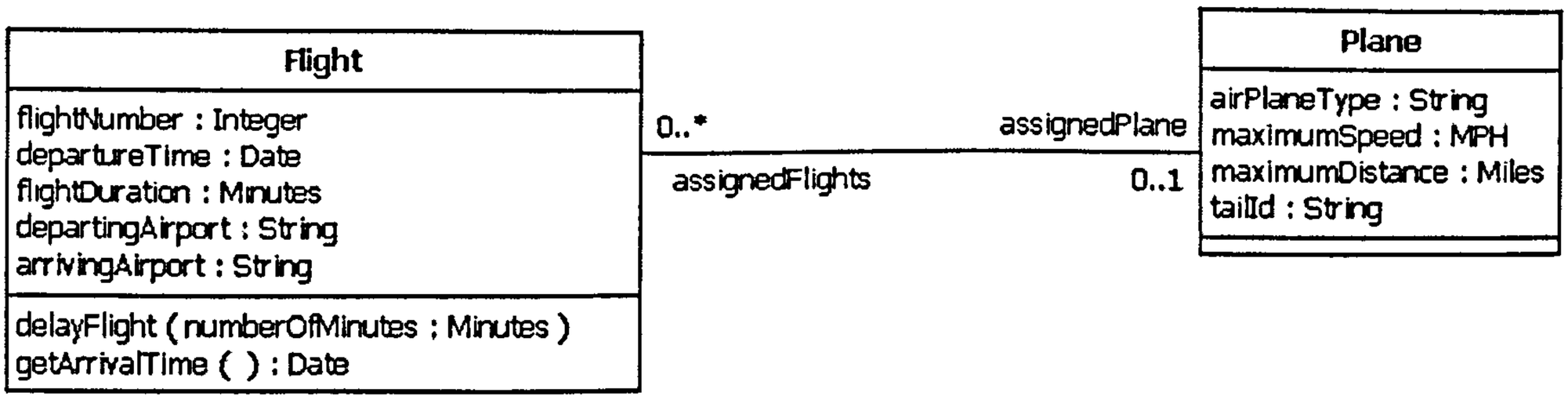


Figure 1.4 Class Diagram (Bell, 2004)

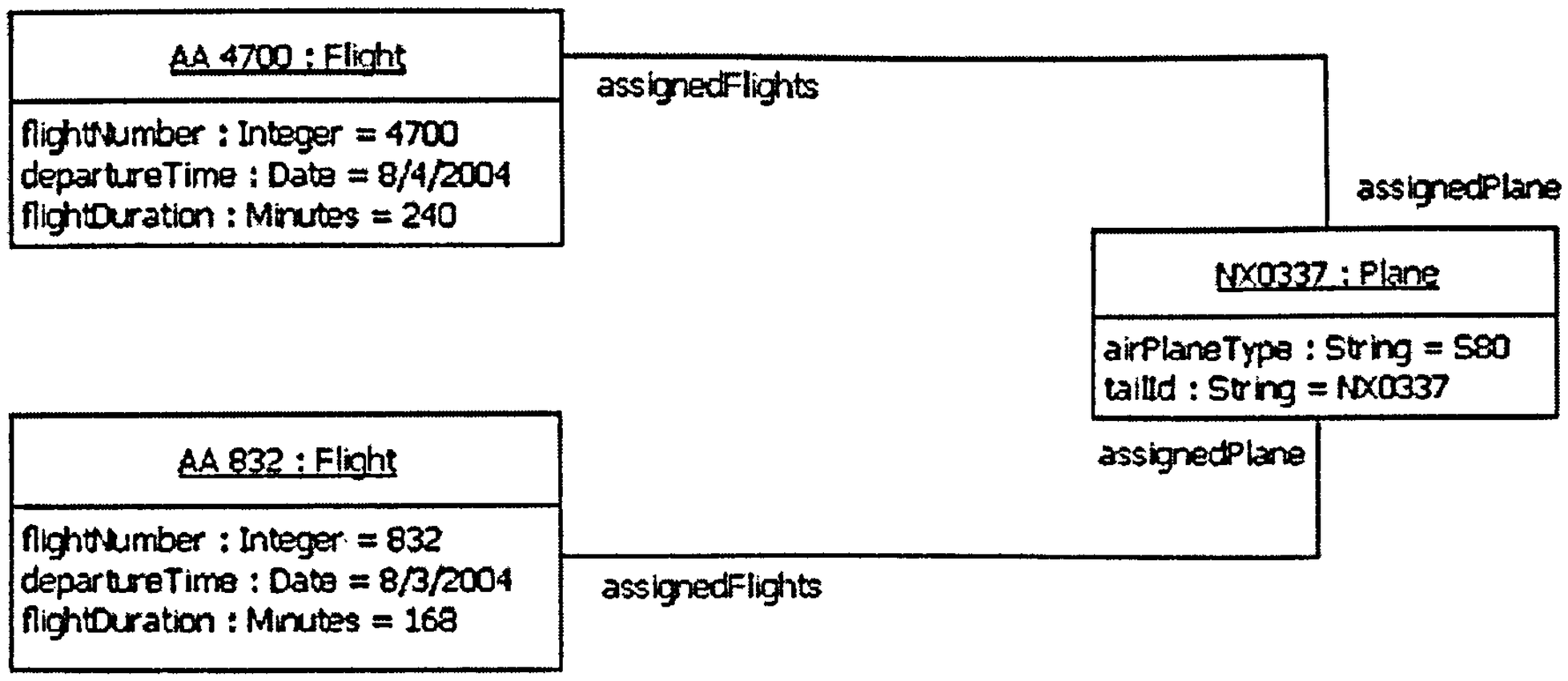


Figure 1.5 Object diagram (Bell, 2004)

Component diagrams illustrate the physical software components of the system. Coetzee (2005) has highlighted that component diagrams depict the relationship between software components, their dependencies, communication, location, and other conditions. OMG (2004, 170) define components as "a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment." Figure 1.6 shows an example of a component diagram.

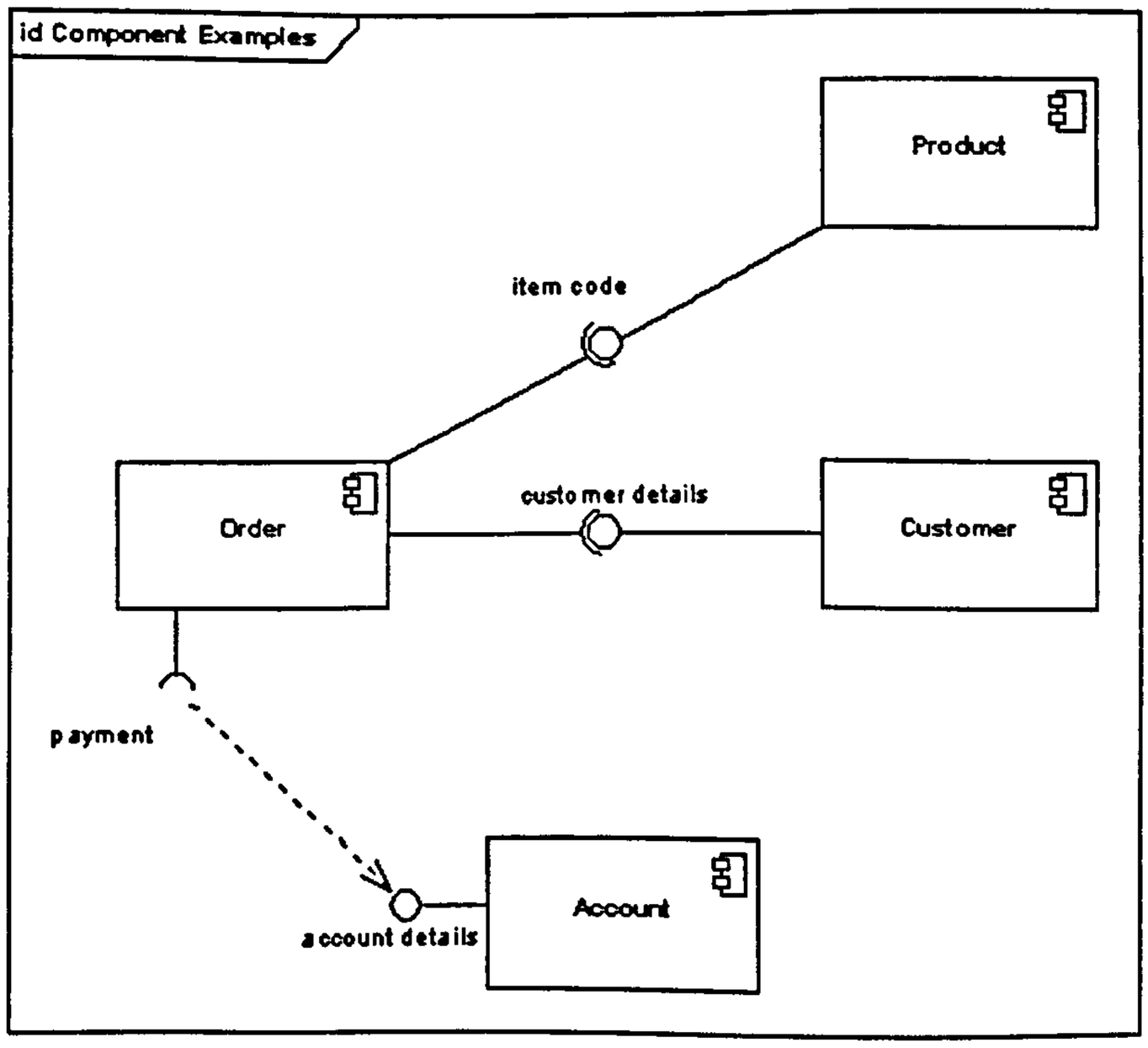


Figure 1.6 Example of component diagram (Coetzee, 2005).

Composite structure diagrams explore run-time instances of interconnected instances collaborating over communication links (Ambler, 2005). The composite structure diagrams assist in describing relationships between elements that collaborate within a classifier (Coetzee, 2005). It is similar to class diagrams but illustrates parts and connectors. Figure 1.7 depicts the composite structure of an invoice.

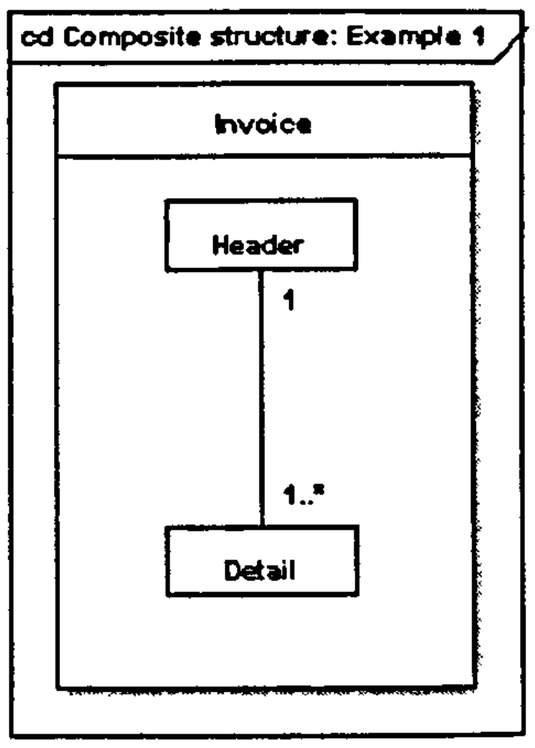


Figure 1.7 Example of Composite structure (Coetzee, 2005)

Deployment diagrams depict the physical layout of a system, indicating which pieces of software are executing, on which pieces of hardware (Fowler, 2004). The deployment diagram shows nodes, components, and connections. The nodes are the physical resources that execute code components. Nodes can be a hardware device or a software execution environment. They contain artifacts, which are usually physical files. Figure 1.8 shows a deployment diagram of financial application (Coetzee, 2005).

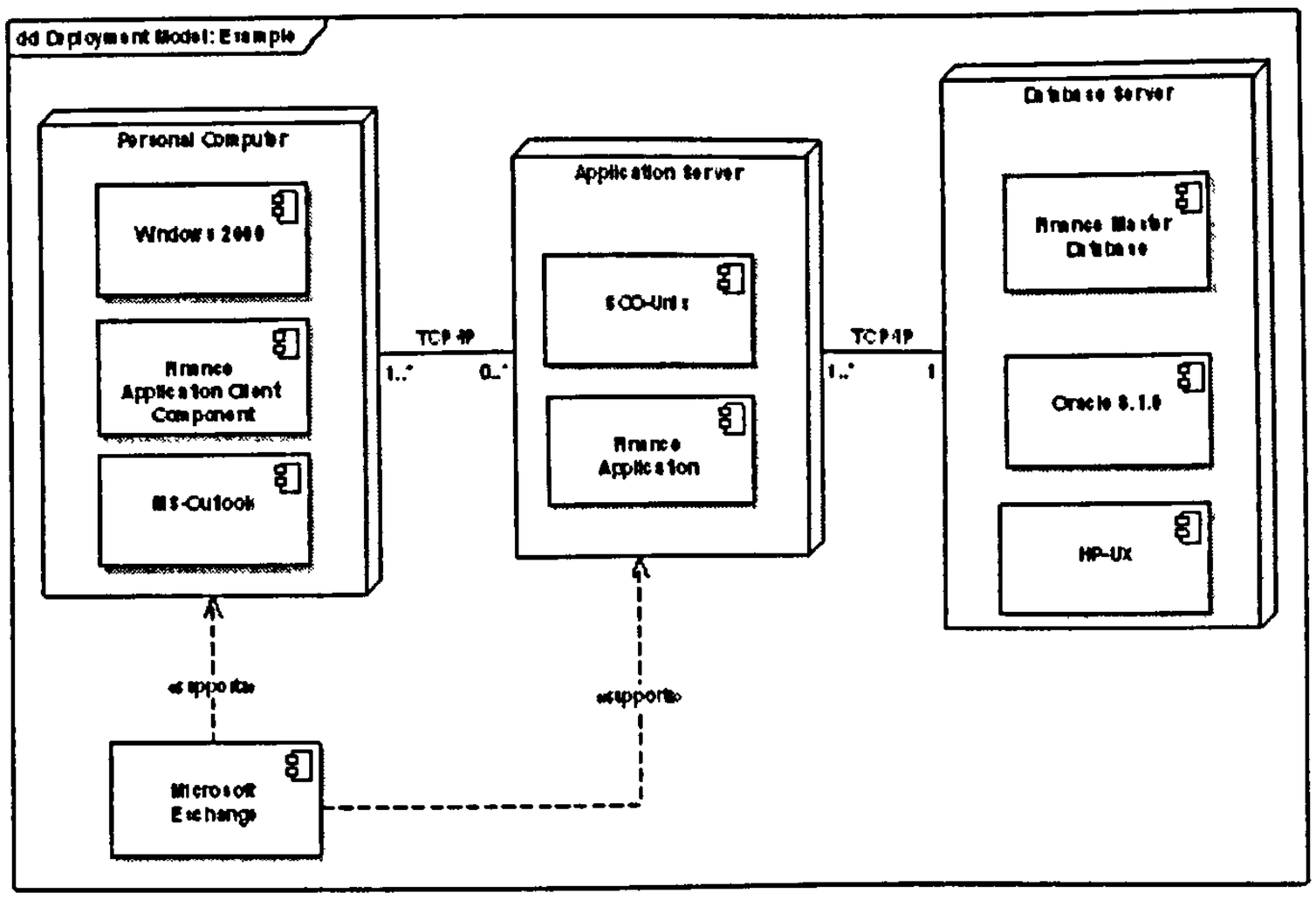


Figure 1.8 Example of deployment diagram (Coetzee, 2005)

Package diagrams depict packages of the system and their relationship. "A package is used to group elements, and provides a namespace for the grouped elements" (OMG, 2004), where elements can be use cases, classes, activities, states, or other packages. Packages enable analysts to organise model into groups, thus making UML diagrams easier to represent and to understand. The notation of a package is a large rectangle with a tab that shows the name of the package. Figure 1.9 shows an example of a package diagram for bank accounts.

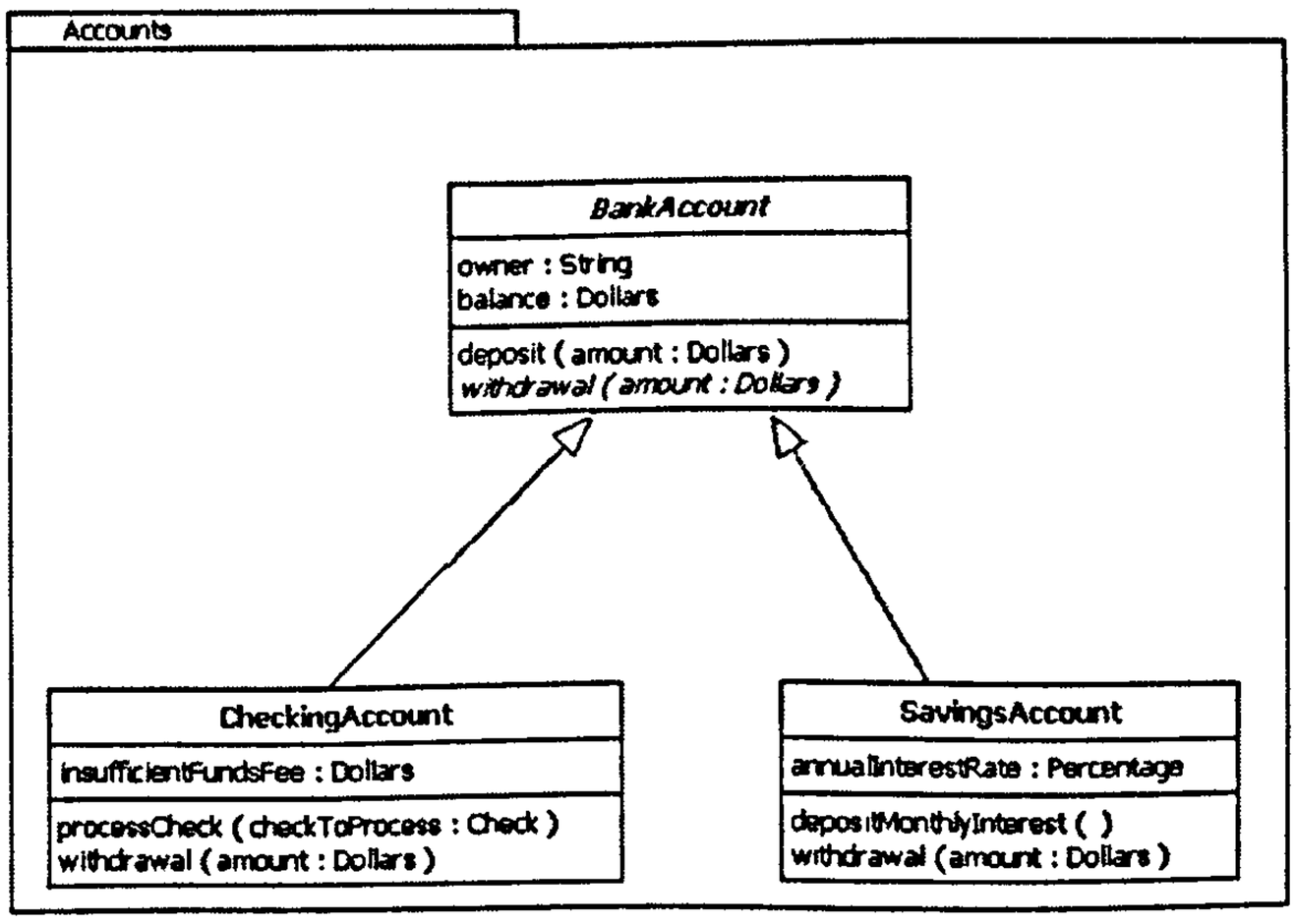


Figure 1.9 Package diagram (Bell, 2004)

Use case diagrams show the different functions available within a system. The diagram depicts the interaction between the users of a system and the system itself. A Use case diagram contains use cases, actors, and subject. The system under construction is the subject, users and other systems that are interacting with the system. Under construction are the actors. Analysts model the behaviour of the system under construction by one or more use cases. Actors are always entities outside the system (OMG, 2004). Figure 1.10 shows a use case diagram for ATM system that contains three actors and five use cases.

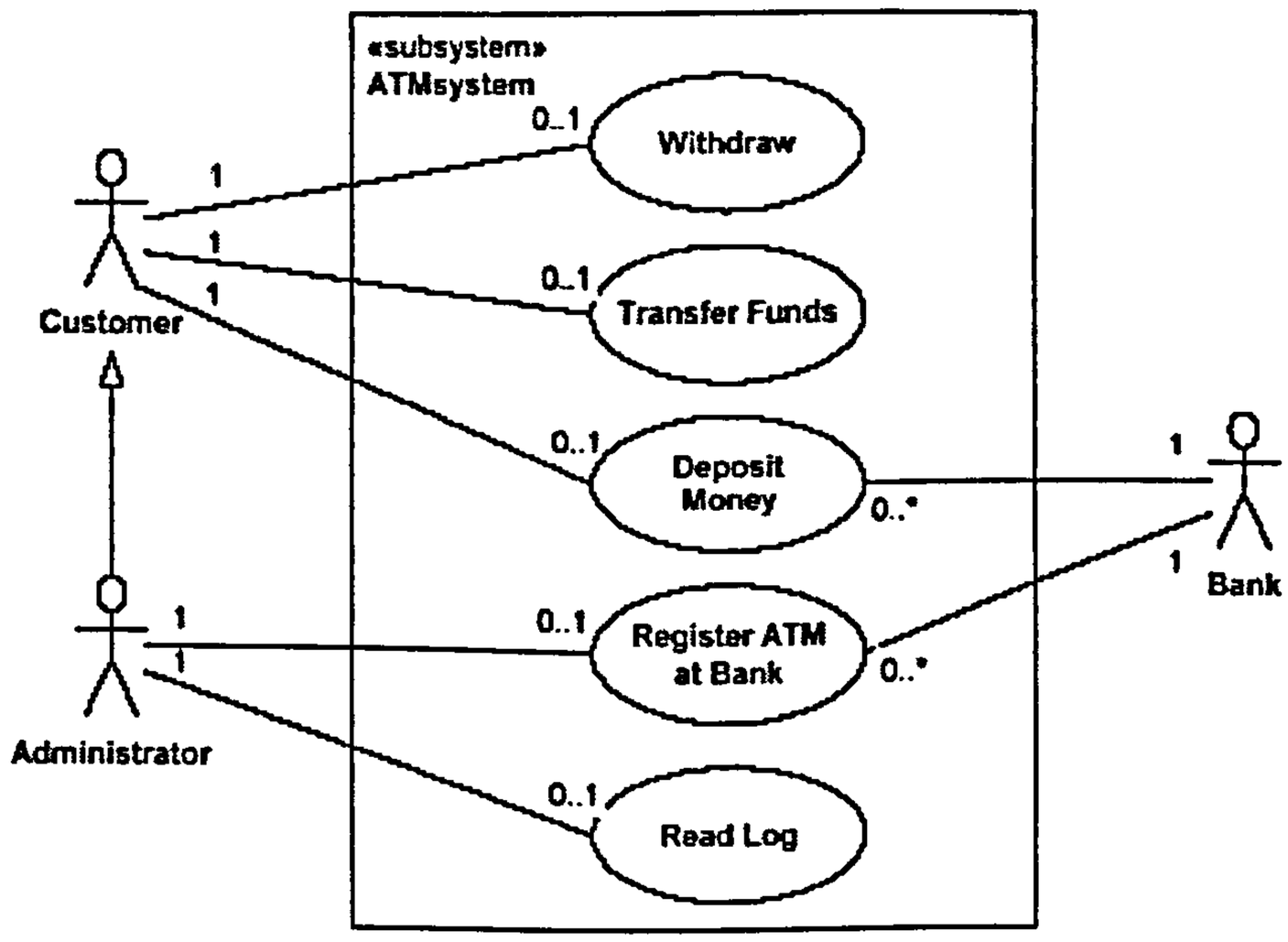


Figure 1.10 Example of Use case and actors for an ATM system (OMG, 2004)

Activity diagrams model behaviour of the system under construction. They are able to describe procedural logic, business process, and workflow (Fowler, 2004). They illustrate the flow of control from activity to activity. Activity diagrams show activities that can be broken down into actions. Activity diagrams can show concurrence behaviour of activities and actions by using the fork notation. They also use partitions to show who does what. Activity diagrams are a main UML behaviour diagram that allows designers to know in detail the sequence behaviour of all the activities and actions of the system under

construction. Figure 1.11 shows an example of an activity diagram of process order activity that utilises multidimensional partition (OMG, 2004).

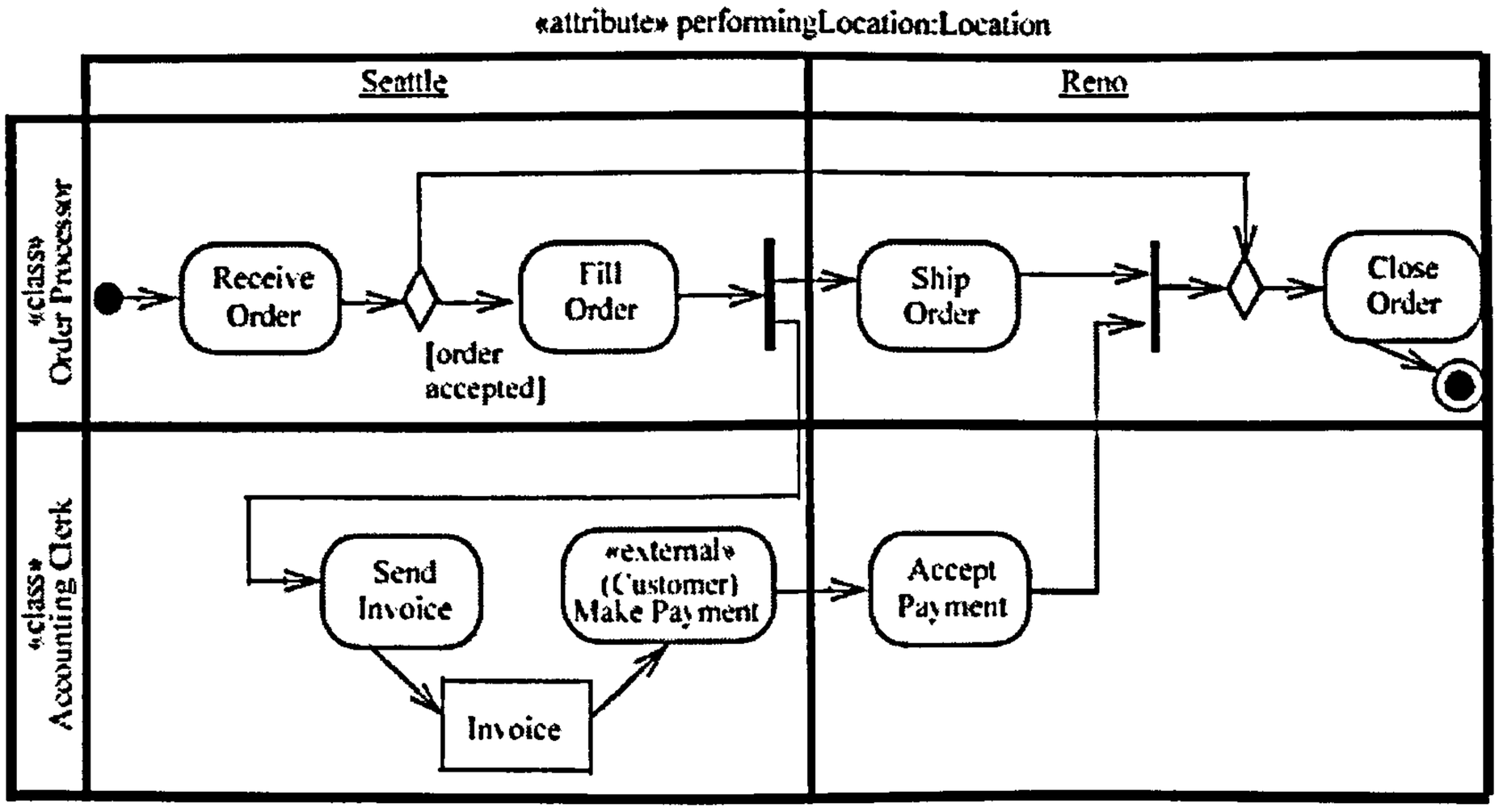


Figure 1.11 Example of activity diagram of process order that utilizes multi-partitions (OMG, 2004)

State machine diagrams depict the life cycle of objects, sub-systems, and systems (Coetzee, 2005). They are efficient in describing the behaviour of an object across several use cases (Fowler, 2004). The diagrams show different types of states, such as activity states, super states, sub states, or concurrent states. The notation of a state is a rounded rectangle with optional compartments for events, attributes, and internal activities. Figure 1.12 shows an example of state machine diagrams of a telephone (OMG, 2004). State flows or transitions connect states with a guard to identify when and where an object may transfer from one state to another.

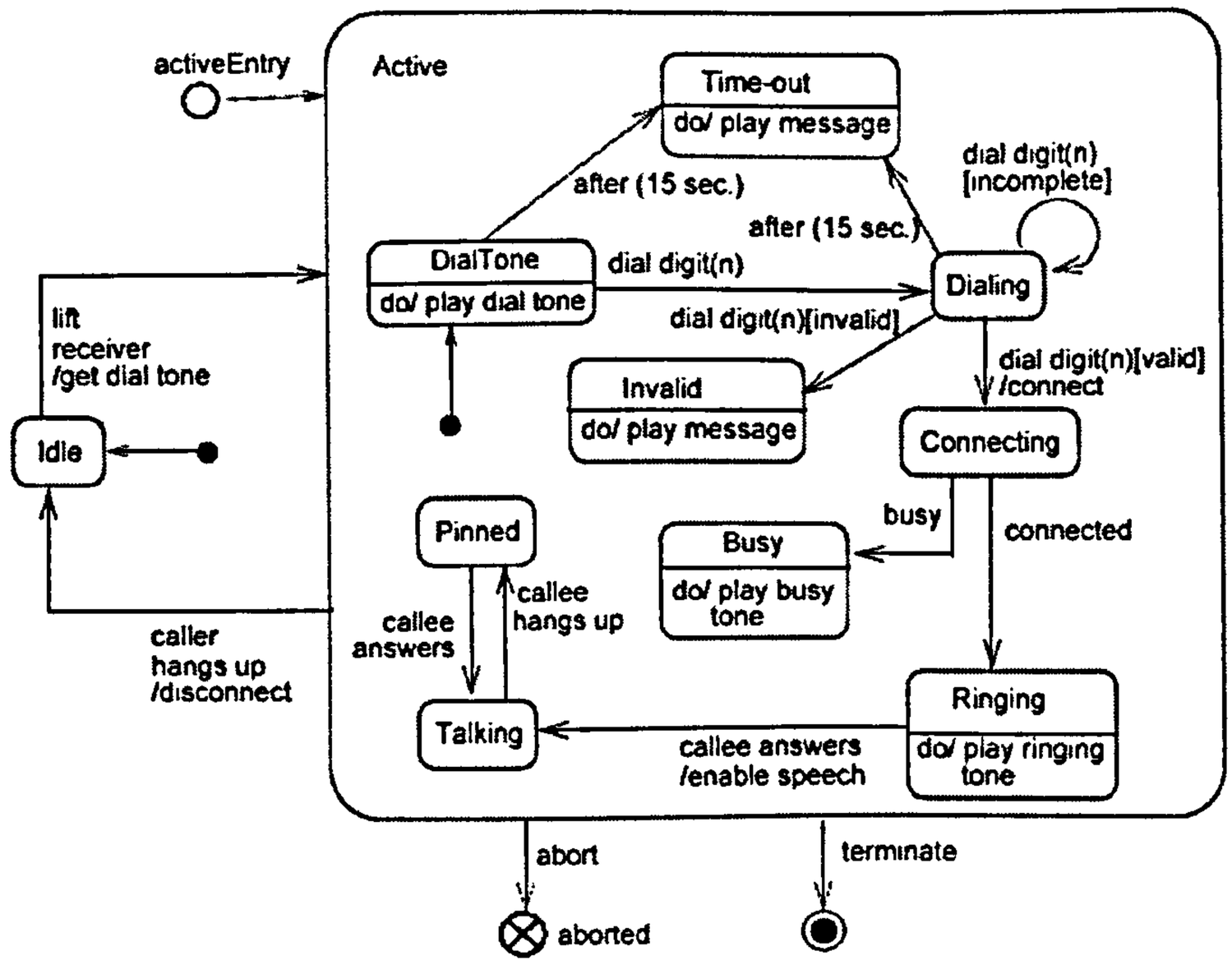


Figure 1.12 Example of state machine diagram for a telephone (OMG, 2004)

Interaction diagrams have four types; communication, sequence, timing, and interaction overview diagrams. Sequence diagrams are the best known interaction diagram used. They show message interchange between lifelines such as objects and actors. The sequence diagrams are able to model different behaviours such as concurrency, synchronous, and asynchronous behaviour. Figure 1.13 shows an example of a sequence diagram that depicts an example of the 'if-then' rule by the alternative frame labelled alt.

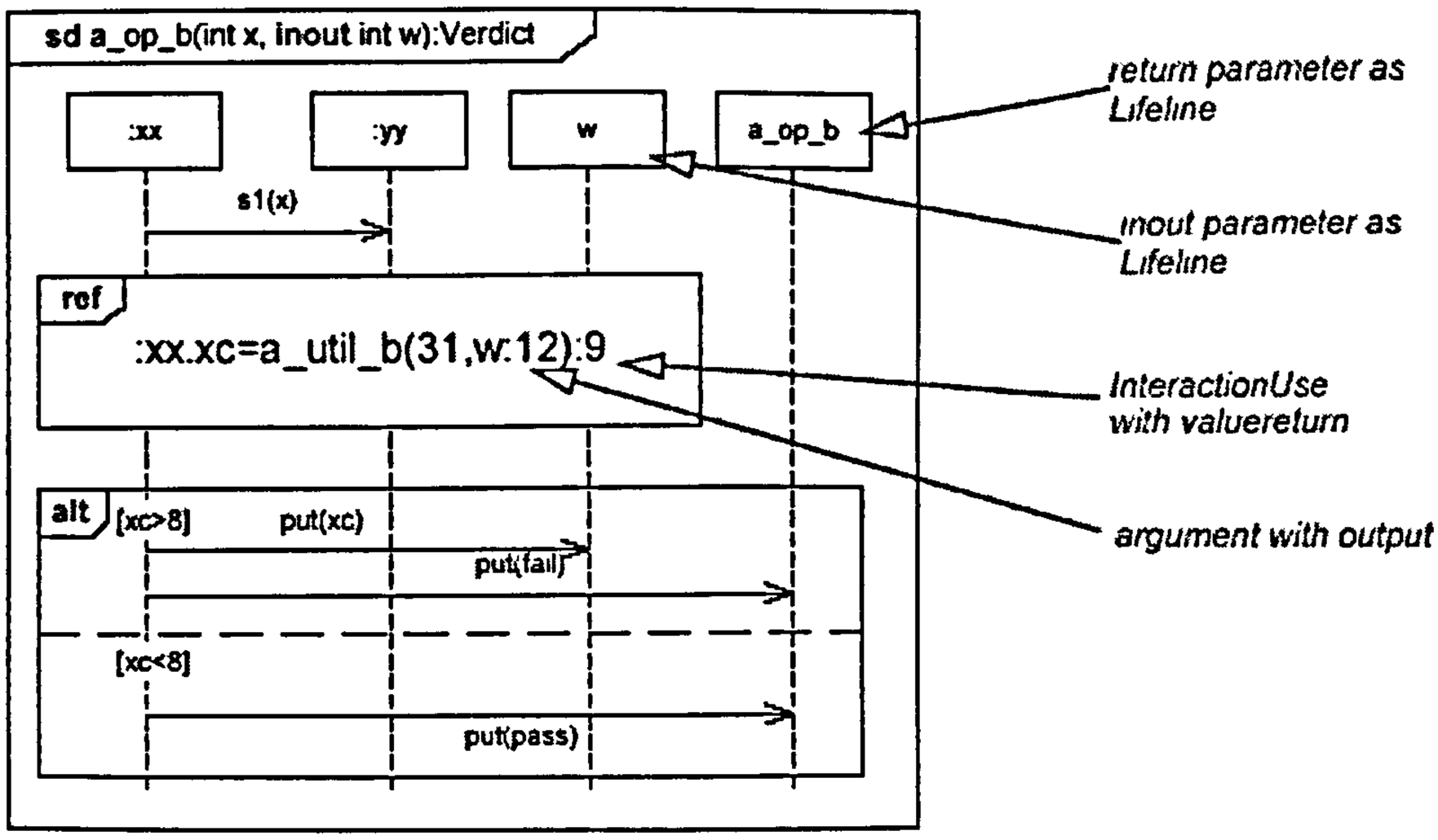


Figure 1.13 Example of sequence diagram (OMG, 2004)

Communication diagrams focus on the interactions among entities, they are similar to sequence diagrams but they emphasise the data links between the various participants (Fowler, 2004). Analysts use communication diagrams when the demonstration of the context is important rather than the sequence of events (Coetzee, 2005). Communication diagrams show a number of objects along with the relationships between them. Message arrows are drawn between them to show the flow of messages between the objects. The sequencing of Messages is given through a sequence-numbering scheme (OMG, 2004). Figure 1.14 shows an example of a communication diagram for seminar displaying screen (Ambler, 2005).

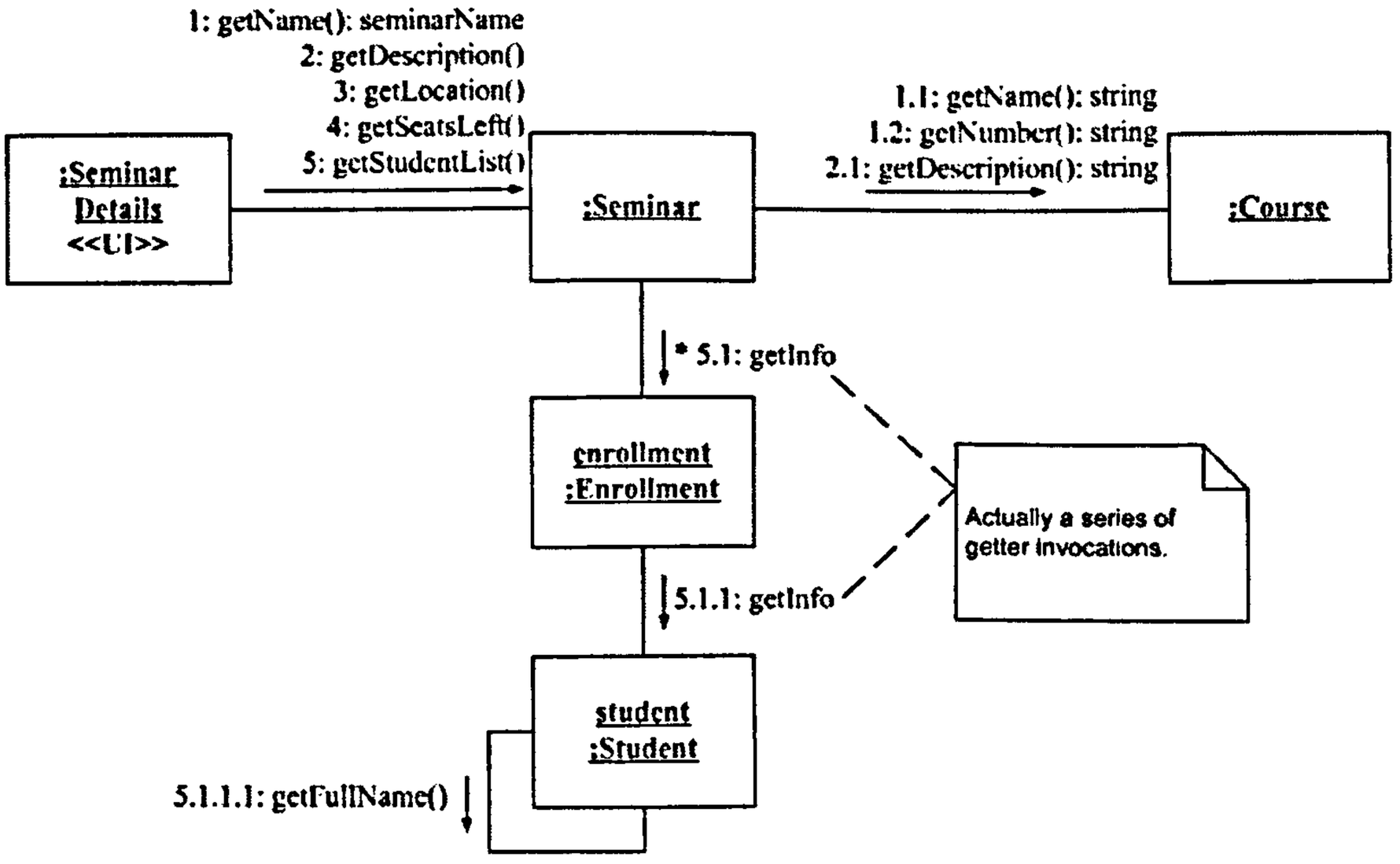


Figure 1.14 Example of communication diagram for displaying seminar screen (Ambler, 2005)

Timing diagrams are the fourth type of interaction diagram with the primary role of reasoning about time (OMG, 2004). The diagrams provide a visual representation of entities changing states and interacting over time. Timing diagrams show the behaviour of one or more entity during a specific duration (Ambler, 2005). There are two notations for timing diagrams; one is time-based as shown in figure 1.15 and the other is value-based as shown in figure 1.16 (Coetzee, 2005).

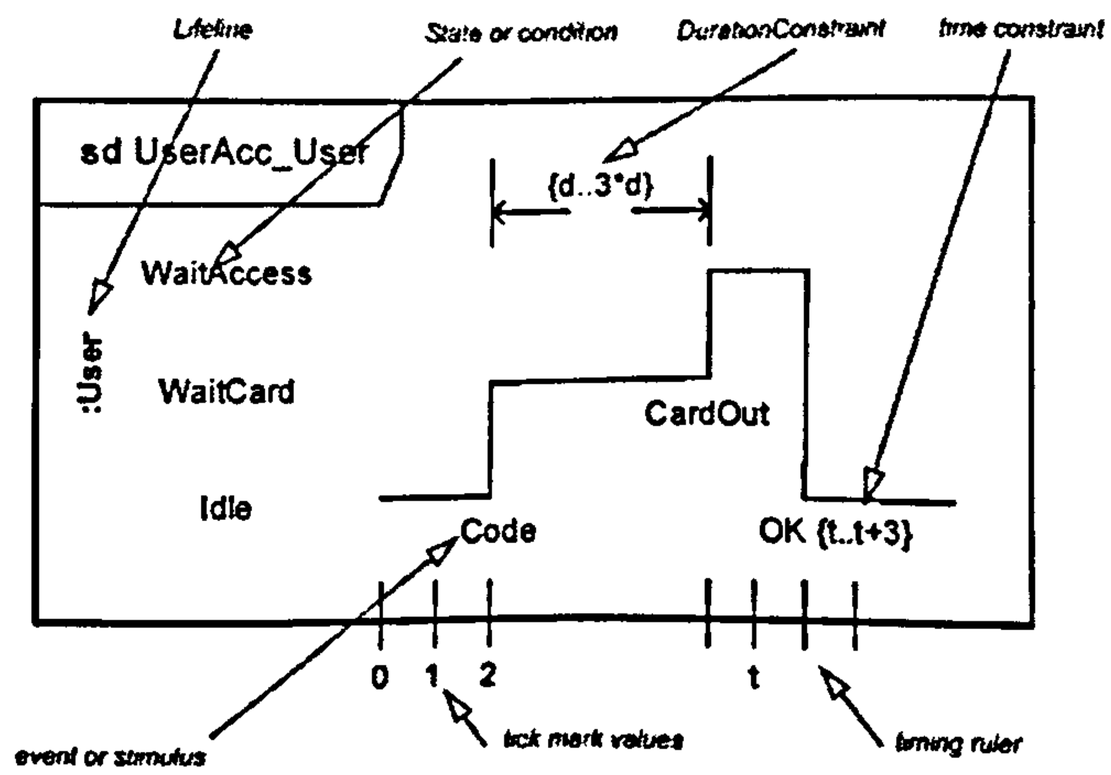


Figure 1.15 Example of timing diagram (OMG, 2004)

The available diagrams of UML 2.0 are able to model structural and behavioural features of object-oriented systems, this research focuses on exploring the capability of some of these diagrams to model the structural and behavioural learning characteristics of Intelligent Agents.

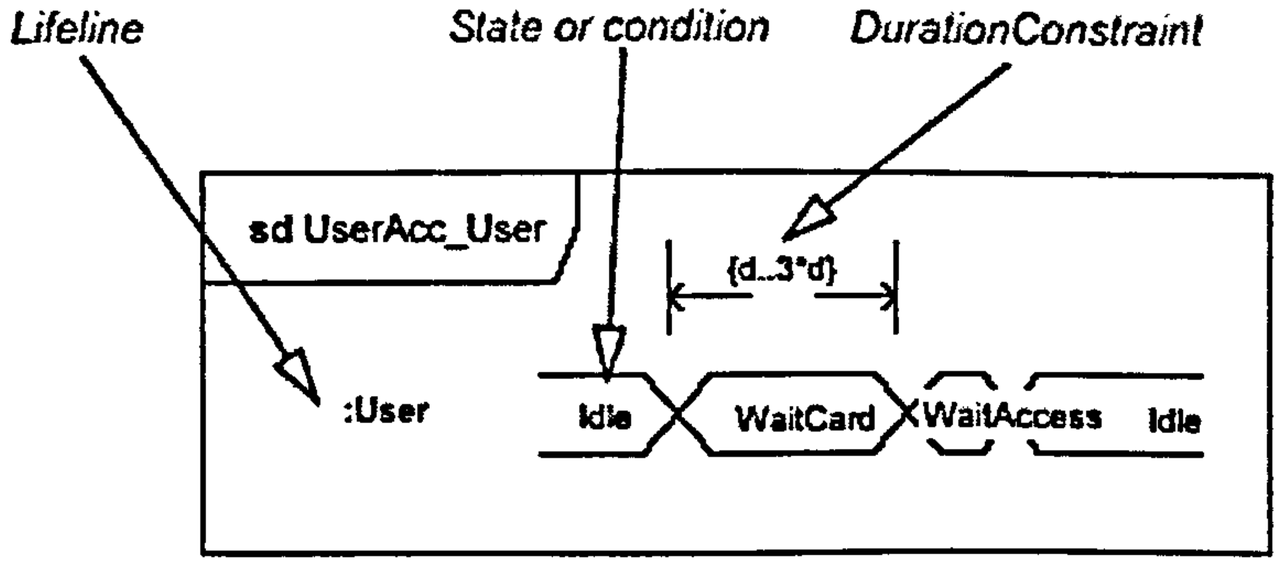


Figure 1.16 Example of compact timing diagram (OMG, 2004)

1.5 Overview of the Thesis

This thesis consists of six chapters. Chapter 1 is an introduction. Chapter 2 discusses the components required to describe the learning behaviour of Intelligent Agents. It also discusses the effects of each component of the learning capabilities on the performance of Intelligent Agents and the requirements for these capabilities.

Chapter 3 reviews four research directions that use UML to model Agent-oriented systems. Two possible approaches are available to allow UML to fit some of the Agent-oriented software requirements: either to extend UML with new structural elements and diagrams or to use UML base language to describe Agent-specific aspects of the software systems. The first approach provides new structural elements and diagrams to enhance the expressive power of the base language while the second approach adheres to the boundaries of the original language, and uses only those extension mechanisms that UML admits. The four research directions include examples from both approaches.

Chapter 4 explores the capability of UML 2.0 to model structural features of Intelligent Agents. This chapter utilises FIPA Agent Class Superstructure Meta-model to model Intelligent Agents structural components. Chapter 4 discusses two scenarios to model Intelligent Agent structural features by using the FIPA Agent Class Superstructure Meta-model. It also proposes the learning compartment that describes the learning features of Intelligent Agents.

Chapter 5 investigates the capability of the behaviour components of Unified Modelling Language (UML) version 2.0 to model learning behaviour of Intelligent Agents. Chapter 5 uses UML 2.0 activity and sequence diagrams to model different scenarios of learning strategies, namely learning from observation and learning from examples. It also evaluates if UML 2.0 state machine diagrams can model specific reinforcement learning algorithms, namely dynamic programming, Monte Carlo, and temporal difference algorithms. Chapter 6 provides the conclusion of this thesis and future research directions. Appendices A, B, and C provide user guides of UML 2.0 activity, sequence, and state machine diagrams to allow researchers in agent-oriented systems to use these diagrams in modelling the learning components of Intelligent Agents.

The thesis aims to explore the ability of the structural and behavioural components of the Unified Modelling Language (UML 2.0 / 2004) to model the learning behaviour of Intelligent Agents. The research shows that the behavioural components of UML 2.0 are capable of modelling the learning behaviour of Intelligent Agents while structural components need to be extended to cover the structural requirements of Agents and Intelligent Agents. This thesis will lead to increasing interest in the intelligence dimension rather than the agency dimension of Intelligent Agents, and pave the way for object-oriented methodologies to migrate more easily to paradigms of Intelligent Agent-oriented systems.

Chapter 2: Learning Capabilities of Intelligent Agents

This chapter presents an overview of the required components to describe the learning behaviour of Intelligent Agents. It also discusses the effects of each component of the learning capabilities on the performance of Intelligent Agents and the requirement for these capabilities. Section 2.2 discusses the description of learning behaviour. Section 2.3 outlines learning goal properties and types. Section 2.4 provides an overview of knowledge types and representations that Intelligent Agents use during the learning process. It further outlines the forms of background knowledge and their relation with the input knowledge. Section 2.4 highlights how different types of learning use background knowledge. Section 2.5 explains the different types of learning strategies that Intelligent Agents can use during the learning process. Section 2.6 discusses learning feedback methods, through which Intelligent Agents receive feedback about their performance. Section 2.7 highlights the importance for identifying the mobility feature of Intelligent Agents, that is whether Intelligent Agents are static or mobile. Section 2.8 discusses the identification of scheduling for the learning process. Section 2.9 outlines the relation between learning and communication that is whether the Intelligent Agents learn to communicate or communicate to learn.

2.1 Introduction

Computer science community uses the phrase "Intelligent Agents" or "Agents" for the same entities while such usage indicates different meanings. What distinguishes Intelligent Agents from Agents is their ability to learn and to reach agreement. Sein and Weiss (1999) define learning as "the acquisition of new knowledge, motor, and cognitive skills and the incorporation of the acquired knowledge and skills in future system activities provided that this acquisition and incorporation is conducted by the system itself and leads

to improvement in the performance." In addition, Intelligent Agents should exhibit negotiation skills and/or argumentation skills, as they are vital to achieve agreement with other Agents or users (Wooldridge, 2002). The lack of ability to learn or to reach agreement would reduce the efficiency of Intelligent Agents especially in dynamic multi-Agent environments. Analysts should provide unambiguous description about learning goals, learning strategy, appropriate knowledge the Intelligent Agents need for learning, knowledge representation used by the Intelligent Agents, learning location, learning schedule, and when the Intelligent Agents use the new hypothesis.

2.2 Learning

Learning is a key aspect of intelligence; systems that are capable of learning deserve to be classified as intelligent systems (Sein and Weiss, 1999). Schank and Kass (2000) claim that a machine is intelligent if it has the capability to explain, to learn, and to be creative. Michalski (1993) introduces inferential theory of learning that identifies learning as a process that carries out inference from the information provided, the learner's background knowledge, and subsequently memorizing useful results. As Michalski (1993) puts it: "Learning = Inferencing + Memorizing." Michalski (1993) highlights three components that describe a learning task: types of information acquired, background knowledge relevant to the learning goal, and aim of learning.

Brenner et al. (1998) highlight the need to identify six aspects to model learning capabilities of Reactive Agents[2] that are as follows:

- Identify the mechanism that Intelligent Agents use to select the action to perform;

---

[2] Reactive Agents: Agents that "possess no explicit internal knowledge representation and react to specific, standard events, by using sensors to monitor their environment and to look for specific situations that agree with their internal recognition patterns" (Brenner, 1998, 40).

- Identify which principle Intelligent Agents use for the actual learning activity;
- Identify the method that Intelligent Agents use to decide which information it saves and which it destroys;
- Identify previous situations in which the Intelligent Agents gain knowledge;
- Identify when the Intelligent Agents attempt to learn, and
- Define how Intelligent Agents evaluate the results of actions.

The minimum required features that an analyst should identify to model the learning behaviour of Intelligent Agents are learning goals, type of knowledge acquired, knowledge representations used, type of learning strategy, learning feedback mechanism, learning location, learning schedule, learning triggers, and when the new hypothesis should take effect.

Learning is a central feature of intelligence; it involves inference and memorizing capability. Agents that exhibit capabilities to learn and to reach agreement with other Agents or users deserve to be called Intelligent Agent. To model learning behaviour of Intelligent Agents analysts should identify features relevant to the learning process, such as learning goals, the use of background knowledge, and the type of knowledge the Intelligent Agent uses during the learning process.

2.3 Learning Goal

The learning goal of Intelligent Agents is the first type of information that analysts identify to model their learning behaviour. Michalski (1994) explains that the Inferential Theory of Learning "assumes that learning is a goal-guided process of improving the learner's knowledge by exploring the learner's experience and prior knowledge." The learning goal has two types: either to increase the amount of learner's knowledge (synthetic

learning) or to increase the effectiveness of the knowledge already possessed (analytic learning). The Intelligent Agents may use one or both types through the learning process (Michalski, 1993). Synthetic learning aims to acquire new knowledge that is not available within the current hypothesis while analytic learning transfers knowledge, already available in the current hypothesis, into the form that is most desirable and/or effective for achieving the learning goal. Synthetic learning employs induction and/or analogy as the primary inference while analytic learning employs deduction as the primary inference.

The learning goal can be either a domain-independent or a domain-dependent goal. Domain-independent learning goal describes a certain generic type of learning activity, independent of the topic of discourse while domain-dependent goal acquires specific knowledge (Michalski, 1993). Different domains can use Intelligent Agents with domain-independent learning goal while specific domains can only use domain-dependent Intelligent Agents.

In multi-Agent systems, analysts should identify if a learning goal requires the improvement of a single Intelligent Agent or a group of Intelligent Agents. In addition, they should determine the compatibility of learning goals among Intelligent Agents working in multi-Agent environments. Learning goals of Intelligent Agents can complement or conflict with each other (Sein and Weiss, 1999). Synchronisation of learning is an important issue in multi-Agent environments when the learning goals of Intelligent Agents complement each other. Analysts should clearly identify the sequence of learning among the different Intelligent Agents. On the other hand, with conflict-based learning Intelligent Agents, analysts should identify as much as possible what constraints affect the learning requirements of Intelligent Agents and what scenarios occur to overcome such constraints are.

Commitment strategy is the mechanism that Agents use to determine when and how to drop their intensions (Wooldridge, 2000). Commitment strategy of Intelligent Agents affects their behaviour to achieve the learning goal. There are three types of

commitment strategies: blind, single-minded, and open-minded commitment strategies. Intelligent Agents which use blind commitment strategy will continue to maintain an intention to achieve their learning goal until they believe that they have achieved the learning goal. Intelligent Agents which use single-minded commitment strategies terminate trials to achieve their learning goal either when they believe that they achieved the learning goal or when it is no longer possible to achieve the learning goal. However, Intelligent Agents using open-minded commitment strategy will continue to achieve their learning goal as long as they believe that their learning goal has not changed.

Intelligent Agents that use blind commitment strategy consume their efforts without realising that they cannot achieve their learning goal. Static environment is the most appropriate for using such strategy. Analysts can produce predicates for all or most of the states that will face Intelligent Agents working in static environment. For single-minded commitment strategy, analysts can choose such type for Intelligent Agents working in dynamic environment such as the Internet. Intelligent Agents stop the process of learning when they believe that they have accomplished the learning goal, or it is no longer possible for them to achieve the learning goal. Intelligent Agents with open-minded commitment strategy continue to try to achieve that learning goal as long as they believe that the learning goal has not changed. When Intelligent Agents realise that the learning goal has changed, they start to achieve the new learning goal. Analysts may identify constraints for Intelligent Agents using open-minded commitment strategy such as the duration after which the learning goal is no longer relevant; so, the Intelligent Agent can reconsider its learning goal.

The identification of the learning goal of intelligent Agents is a key issue to model their learning behaviour. Learning can be either synthetic or analytic learning. The learning goal can be domain-dependent used by specific domains or domain-independent that could be utilised by any domain area. In multi-Agent environments, analysts need to identify if the leaning goals of different Agents complement each other or conflict with

each other. Commitment strategy of Intelligent Agents has a direct impact on their performance to achieve the required learning goal.

2.4 Knowledge Types and Representations of the Learning Process

Knowledge is "facts or conditions of knowing something with familiarity gained through experience or associations" (Bigus, 2001). Knowledge can be simple facts, complex relationships, mathematical formulas, rules for natural language syntax, associations between related concepts, and inheritance hierarchies between classes of objects (Bigus, 2001). Knowledge is a key issue that should be clearly analysed and described. Analysts should define four issues to describe knowledge essential for the learning process: knowledge type, knowledge representation, the level of using background knowledge, and relation between input knowledge and background knowledge.

2.4.1 Knowledge Types

According to Bigus (2001), there are three types of Knowledge: procedural, relational, and hierarchal. Procedural knowledge is a popular technique for representing knowledge in computers. Knowledge analysts encode facts and definitions of the sequence of operation which use and manipulate such facts. Relational knowledge defines a record of information about an item such as relational databases. Each record contains a set of attributes and values for different items. Data definition language defines the format of tables, parameters, and relationships while data manipulation language manipulates the values of the parameters. Structured query language is the most popular language for manipulating relational data. Hierarchal knowledge type depends on the ability to inherit knowledge that focuses on relationships and shared attributes between classes of objects.

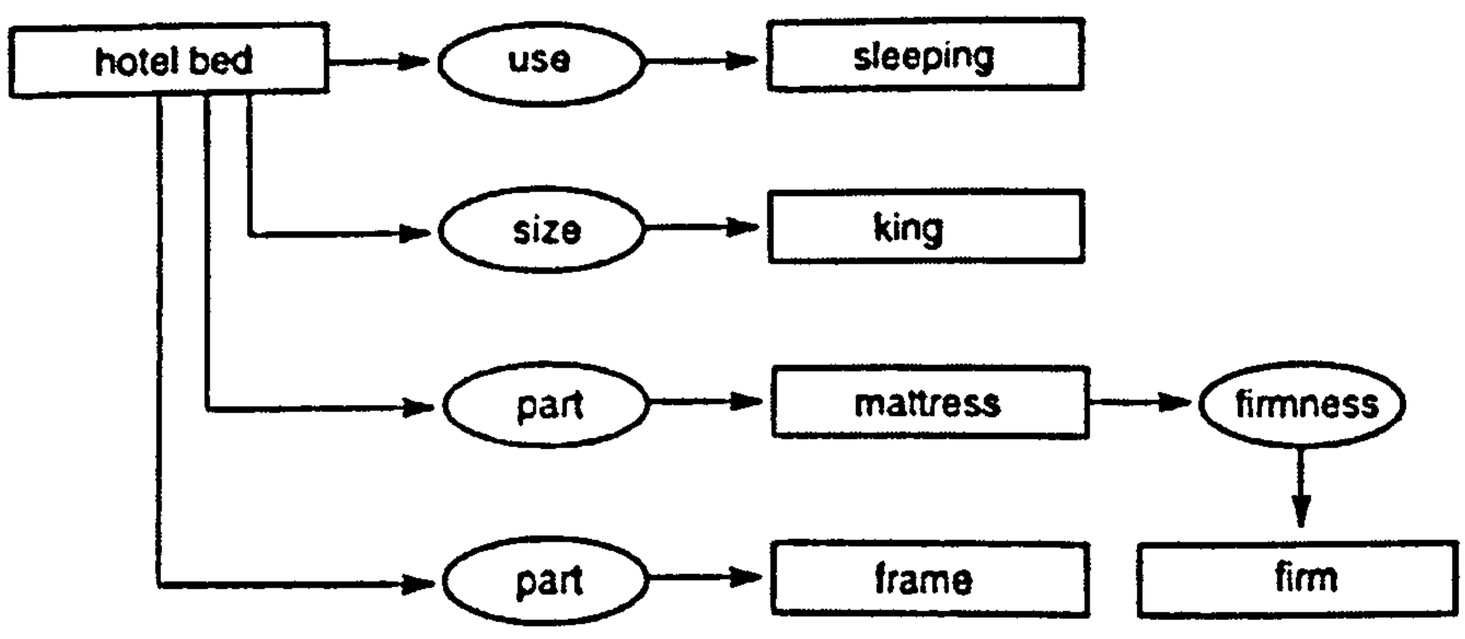
2.4.2 Knowledge Representations

Knowledge can have different representations, and Intelligent Agents should be capable of manipulating and reasoning such different types. Knowledge representations can be parameters in algebraic expressions, decision trees, formal grammars, production rules, formal logic-based expressions, predicate logic, graphs and networks, frames and schemas, computer programs, taxonomies, Bayesian networks, or multiple representations (Carbonell et al., 1983). Knowledge representation is a very complex issue; each of the above-mentioned types can have different variants. Analysts should be capable of understanding the properties of each representation and its variants.

Parameters in algebraic expressions representation obtain the desired performance by adjusting numerical parameters or coefficients in algebraic expression of a fixed function. Decision trees discriminate among classes and objects. The nodes in a decision tree correspond with selected object attributes while the edges correspond with a predetermined alternative value for these attributes. Leaves of the tree correspond with sets of objects with identical classification. In learning to recognise a particular language, the sequence of expression in the language induces formal grammars. These grammars represent regular expressions, finite-state automata, context-free grammar rules, or transformation rules. Production rules representation is a condition-pair {C => A} where C is a set of conditions and A is a sequence of actions. When production rule satisfies all conditions, a sequence of actions will be executed. In formal logic-based expressions representation, the components can be propositions, arbitrary predicates, finite-value variables, statements restricting ranges of variables, or embedded logical expressions (Carbonell et al., 1983).

Formal logic is a language with its own syntax and corresponding semantics. The language syntax defines how to make sentences, and the corresponding semantics describes the meaning of the sentences (Bigus, 2001). Predicate logic allows predicating on objects to define attributes and relations between objects. Predicate logic introduces the

concepts of quantifiers that allow refereeing a set of objects. Two types of quantifiers exist: existential and universal. Existential qualifiers define that some objects of certain types have specified attributes, while universal quantifiers define that all objects of this type have these attributes (Bigus, 2001). Intelligent Agent use resolution and unification techniques to process predicate statements to confirm whether a particular statement is true or not, depending on other known facts. Resolution is an algorithm for proving that facts are true or false by virtue of contradiction, while unification is a technique for taking two sentences in predicate logic, and finding a substitution that makes them look the same. Carbonell et al. (1983) consider graphs and networks representations more suitable and efficient than logical expressions in many domains. Graph-matching and graph-transformation schemes are being used by some learning techniques to compare and to index knowledge efficiently. Frames and schemas representations provide larger expressions or production rules. Frames and schemas representations collect labelled entities ("slots"); each slot plays a certain prescribed role in the representations. Figure 2.1 shows example of describing Hotel beds by using graphs and frames representations.



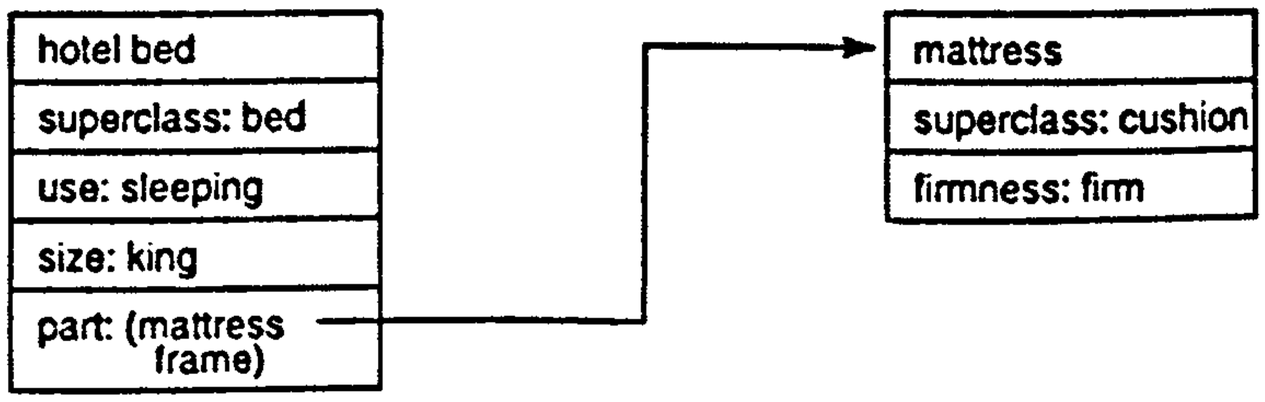


Figure 2.1 Conceptual graph and frame descriptions of a hotel bed (Luger and Stubblefield, 1993)

Computer programs and procedural encoding representation are more concerned about acquiring an ability to carry out a specific process efficiently rather than to reason about the internal structure of the process. Procedural encodings include human motor skills, instruction sequence to robot manipulators, and other "compiled" human or machine skills. Such representation type is different from logical description, networks, or frames representations in terms of the detailed internal structure of the resultant procedural encodings. Procedural encodings do not need to be comprehensive to a human or to an automated reasoning system. The external behaviour of the acquired procedural skills becomes directly available to the reasoning system. Learning from observation may result in global structuring of domain objects into a hierarchy or taxonomy representation. Clustering object descriptions into newly proposed categories and framing hierarchical classifications require the system to formulate relevant criteria for classification.

Bayesian network is data that represents dependence between variables. Each variable has a corresponding node with a conditional probability table defining the relationships between its parent nodes (Bigus, 2001). The primary use of Bayesian networks is to utilize probability theory to reason with uncertainty. Some knowledge acquisition systems may use multiple representations for newly acquired knowledge such as discovery and theory-formation systems. Discovery and theory-formation acquire concepts, operations on those concepts, and heuristic rules for a new domain. Such systems should select appropriate combinations of representation schemes applicable to the different forms of knowledge acquired.

Analysts should identify the knowledge representation that Intelligent Agents will use during learning. Knowledge representation types highlight major aspects such as the space of storing the newly acquired knowledge, the type of inference, the source of knowledge from which the Intelligent Agents will receive their knowledge such as users, Agents, or databases as well as accessibility to the source of knowledge.

2.4.3 Background Knowledge

Russell and Norvig (1995) emphasise that the use of prior knowledge in learning improves Intelligent Agents learning ability. They highlight that background knowledge assists in developing consistent hypotheses. Analysts should identify the level of using background knowledge during the learning process (Michalski, 1993). The level of using background knowledge can range from the need to use intensive background knowledge to the use of none of the available background knowledge. The form of background knowledge can be declarative, procedural, or both. The declarative form is a collection of statements representing conceptual knowledge while the procedural form is a sequence of instructions for performing some skills (Michalski, 1993).

Empirical learning methods rely on relatively small amounts of background knowledge while constructive learning methods are knowledge intensive learning methods that use background knowledge and / or search techniques to create new attribute terms or predicates that are more relevant to the learning task (Michalski, 1993). Constructive learning techniques use the new attributes or predicate to drive characterization of the input where such characterisations can be generalised, explained or both.

2.4.4 Relation between Background Knowledge and Input Knowledge

Michalski (1993) introduces five relationships between the input knowledge and background knowledge as follows:

- The first relationship exists when the input knowledge is a new knowledge that has no relationship with the current background knowledge.
- The second relationship exists when the background knowledge accounts for the input knowledge or is a special case of it.
- The third relationship exists when Intelligent Agents identify that part of the background knowledge contradicts with part or all of the input knowledge.

- The fourth relationship exists when the input knowledge does not match any background fact or rule exactly or when it is not related to any part of the background knowledge; however, there is a similarity between the background fact and some parts of the background knowledge at some level of abstraction.
- The fifth and last relationship exists when some parts of the background knowledge match exactly the input knowledge.

The Intelligent Agents act according to the identified relationship between the input knowledge and the background knowledge. For the first case where the input is new information, Intelligent Agents try to identify parts of the background knowledge that are sibling of the input under the same node in some hierarchy. If the Intelligent Agents succeed in such process, the Intelligent Agents generalise the related knowledge components and evaluate the input facts in terms of their importance to the learning goal. The Intelligent Agents store the input facts that pass threshold criteria. If the Intelligent Agents fail in the identification process, they store the input facts and pass control to case 4 where the input invokes an analogy to part of the background knowledge. This case involves some forms of synthetic learning (Michalski, 1994).

The second case occurs when the background knowledge accounts for the input or a special case of the input. Intelligent Agents create a deviational explanatory structure that links the input with the invoked part of background knowledge. Based on the learning task, Intelligent Agents use this structure to create new knowledge that is more adequate for future handling of such cases. Intelligent Agents store the new knowledge for future use if it passes an "importance criterion." The second case handles situations requiring some forms of analytic learning (Michalski, 1994). The third case exists when the input knowledge contradicts the current background knowledge. Intelligent Agents identify the part of the background knowledge that contradicts the input knowledge and then attempts to classify this part in terms of specialisation. Intelligent Agents make no changes to this part of the background knowledge if the specialisation involves too much restructuring, or

the confidence in the input is low. This requires Intelligent Agents to set "input confidence parameter." The value of such parameter provides information about the noise in the input knowledge that ranges from "no noise" to "noisy" information. The Intelligent Agents restructure some parts of the background knowledge to accommodate the input, and it stores the input if it passes "importance criterion." This case requires some form of synthetic learning (Michalski, 1994).

The fourth case exists when the input neither matches any background fact or rule exactly, or when it is not related to any part of the background knowledge in the sense of case 1; however, there is a similarity between the fact and some part of background knowledge at some level of abstraction. In this case, Intelligent Agents conduct a matching process at this level of abstraction by using generalised attributes or relations (Michalski, 1994). If the new facts pass "importance criterion," Intelligent Agents save the input with an indication of a similarity to a background knowledge component. This case uses a combination of synthetic and analytic learning. The last relation between the input and background knowledge exists when the input matches exactly some part of the background knowledge. Intelligent Agents can update a measure of confidence and a frequency parameter associated with this part. In a dynamic environment, such as the internet, it is difficult for analysts to predict which case will occur. Analysts should provide a description for the actions of Intelligent Agents in each of the five cases. In addition, Intelligent Agents should be able to update procedures according to results obtained from executing them.

Knowledge is a key aspect to describe the learning features of Intelligent Agents. Analysts should identify type(s) of knowledge used during the learning process, knowledge representation, the level of using background knowledge, and the relation between input knowledge and background knowledge. Knowledge can be procedural, relational, and hierarchical. Intelligent Agents can manipulate single or multiple representations of knowledge, required during the learning process. Intelligent Agents can also use

background knowledge during the learning process, or depend only on the new acquired knowledge. Relations between acquired knowledge and background knowledge can provide designers with scenarios for Intelligent Agents' behaviour.

2.5 Types of Learning Strategies

Intelligent Agent uses either mono-strategy learning or act like human beings in using multi-strategy learning. The type of learning strategy used by Intelligent Agents is a key aspect for designing their learning capabilities. Mono-strategy learning Intelligent Agents are intrinsically limited to solving only certain classes of learning problems, defined by the type of input information they can learn from, the type of operations they are able to perform on the given knowledge, and the type of output knowledge they can produce (Michalski, 1993). Multi-strategy learning Intelligent Agents are more competent and have a greater ability to solve diverse learning problems than mono-strategy Intelligent Agents; but they are more complex. The choice of creating a mono-strategy or a multi-strategy Intelligent Agents is crucial. Analysts should investigate different aspects before deciding which strategy to adopt, such as the type of input, the role of the background knowledge in learning, the mobility of the Intelligent Agent, and the available processing power of the host computer. Sein and Weiss (1999) highlight five main learning strategies that Intelligent Agents can use: rote learning, learning from instruction, learning from examples, learning by observation, and learning by analogy.

2.5.1 Rote Learning

Rote learning is a direct implantation of knowledge and skill without further inference or transformation from the learner (Sein and Weiss, 1999). Rote learning corresponds to storing each observed training example in memory. Intelligent Agents classify subsequent instances by looking them up in memory. If Intelligent Agents finds

instance in memory, the memory returns the stored classification; otherwise, the system refuses to classify the new instances (Mitchell, 1997). Carbonell et al. (1983) emphasise two main variants of rote learning: the first variant is learning by being programmed, constructed, or modified by an external entity such as the usual style of computer programming. The second variant is learning by memorising given facts and data with no inference drawn from the incoming information. Analysts should consider three main issues when designing rote learning Intelligent Agents: Memory-searching mechanism, indexing used to retrieve the memorised data, and the capacity to memorise the acquired knowledge (Michalski et al., 1986).

2.5.2 Learning from Instructions

Learning from instruction transforms new information, like an instruction or advice, which is not directly executable by the learner, into an internal representation, and integrate new information with prior knowledge and skill. The learner acquires the new knowledge from a teacher or other organised source requiring that the learner transforms the knowledge from the input language to an internally usable representation. The Intelligent Agent integrates new information with prior knowledge for effective use (Carbonell et al., 1983). Learning from instructions can be unidirectional or interactive. Unidirectional learning occurs when a tutor provides Intelligent Agents with knowledge in a sequential series of instructions. In interactive learning, the tutor provides Intelligent Agents with knowledge when Intelligent Agents lack the knowledge or request instructions (Lemon et al., 2004).

2.5.3 Learning from Examples

Learning from examples exists when there is extraction and refinement of knowledge and skills like a general concept and a standardised pattern of motion either

from positive and negative examples or from practical experience (Sein and Weiss, 1999). Michalski et al. (1986) highlight that the main task of learning from examples is to determine a general description, explain all positive examples, and exclude all negative examples of the target concept. Michalski (1993) identifies two types of learning from examples: instance–to-class and part-to-whole. Instance-to-class type examples are independent entities that represent a given class or a set of concepts. The goal of instance-to-class generalization is to induce a general description of the class. The part-to-whole examples are independent components that have to be investigated together to generate a concept description. The goal of part-to-whole is to hypothesize a description of a whole object by giving selected parts.

Carbonell et al. (1983) identify two main categories to classify learning from examples: classification according to the type of examples available to the learner and rate of supplying the examples. The type of examples can be either positive examples only or positive and negative examples. The rate of supplying the examples can be incremental learning rate or a non-incremental learning one. For sources of knowledge that provide positive examples, Intelligent Agents acquire instances of a concept but they do not acquire information for preventing overgeneralization of the inferred concept. Intelligent Agents can avoid overgeneralisations by considering only the minimal generalisation necessary, or by relying upon prior domain knowledge to constrain the overgeneralized concept to be inferred. Intelligent Agents that use positive and negative examples for learning act in two parallel ways: they use positive examples to force generalisation and negative examples to prevent overgeneralisations (Carbonell et al., 1983). Non-incremental learning is like training salesmen new skills in a private workshop and they go back to work after finishing the workshop. On the other hand, incremental learning allows Intelligent Agents to form one or more hypotheses of the concept, which are consistent with the available data, and subsequently refines the hypotheses after considering additional examples. In this type of

learning Intelligent Agents involve in on-the-job training; they acquire the new skills while they are working.

Analysts should define when the new hypothesis should be executable. In addition, they should identify if the incremental learning will be concurrent with work or sequential. Unlike sequential learning, Concurrent incremental learning requires the Intelligent Agents to work on a multi-processor host computer.

Learning from examples is a synthetic learning type that aims to create new knowledge and use induction as the primary inference (Michalski, 1993). Learning from examples is not adequate for Intelligent Agents that are working in dynamic environment such as the internet; it will be very difficult to provide examples for all states that might happen in the dynamic environment.

2.5.4 Learning from Observation and Discovery

In this type of learning, Intelligent Agents gather new knowledge and skills by conducting observations, experimentations, as well as generating and testing hypotheses, or theories based on observational and experimental results (Sein and Weiss, 1999). Michalski (1993) highlights that learning from observation occurs when the input to the synthetic learning method includes facts that Intelligent Agents use to describe or to organise into knowledge structure without the benefit of advice from a source of knowledge. Neither Intelligent Agents receives a set of instances of particular concepts nor are they given access to Oracle that can classify internally generated instances as positive or negative instances of any given concept. Carbonell et al. (1983) classify learning from observation according to its degree of interaction with the environment; the two extremes are passive observation and active experimentation. Passive-observer Intelligent Agents classify taxonomies observations of multiple aspects of the environment while active-experimentation Intelligent Agents act on stimulating the environment to

observe the results of its stimulation. Experimentation may be random, dynamically focused according to general criteria of importance, or strongly guided by theoretical constraints. Such interaction with the environment would confirm or disconfirm the Intelligent Agents' hypothesis. Learning from observation and discovery includes different types such as conceptual clustering, constructing classifications, fitting equations to data, discovering laws explaining a set of observations, and formulating theories accounting for the behaviour of a system (Michalski et al., 1986).

2.5.5 Learning by Analogy

Intelligent Agents that learn by analogy perform solution-preserving transformation of knowledge and skills from a solved problem to a similar but unsolved problem (Sein and Weiss, 1999). Carbonell et al. (1983) identify learning by analogy as "acquiring new facts or skills by transforming and augmenting existing knowledge that bears strong similarity to a desired new concept or skill into a form effectively useful in the new situation." Carbonell (1986) classifies learning by analogy as transformation analogy and derivational analogy. Transformational analogy exists when Intelligent Agents find a particular solution to work on a problem similar to the one at hand which they might use with minor modification. Derivational analogy formulates plans and solves problems where a considerable amount of intermediate information is available to the resultant plan or specific solution. The main difference between transformation and derivational analogy lies in the learner's background knowledge. In transformation analogy a person receives C++ code of a program, and transforms it to C# while in the derivational analogy the same person is the one who programmes the C++ version. The more experience Intelligent Agents have, the better understanding for conducting learning by analogy they develop.

Learning by analogy is a combination of deductive and inductive learning. Learning from examples and learning by observation is an example of inductive learning.

Deduction learning includes knowledge reformulation, knowledge compilation, caching, chunking, equivalence preserving operations and another truth-preserving transformation (Michalski et al., 1986). Learning by analogy is a multi-strategy method which increases the load of inference conduct by Intelligent Agents. Human learning is intrinsically multiple learning strategies where people can learn from a great variety of inputs, engage any kind of prior knowledge relevant to the problem, and perform all types of inference, using a multitude of knowledge representation. It is a challenge to develop Intelligent Agents that can mimic the learning capabilities of their users (Michalski, 1994).

Learning strategy is the essence of describing the learning mechanism that Intelligent Agents use. Intelligent Agents can have mono-strategy or multiple strategies like humans. Intelligent Agents that use multiple learning strategies are more complex than those who use mono-learning strategies. Learning strategies can be rote learning, learning from observations, learning from instructions, learning from examples, learning from observations, and learning from analogy. Each type of these strategies employs inference as inductive, deductive or analogy. Designers should choose the right strategies for their Intelligent Agents according to their learning goals, sources of knowledge, and working environments features.

2.6 Learning Feedback Methods

The feedback that Intelligent Agents receive on their performance, allow Intelligent Agents to update their hypothesis. Analysts should identify a feedback method to Intelligent Agents' performance and identify when Intelligent Agents receive the feedback. Sein and Weiss (1999) classify learning feedback methods as supervised, unsupervised, and reinforcement learning. The source of knowledge, which provides feedback about the performance of the Intelligent Agent, can be a user, an environment, or an Agent. In

addition, Intelligent Agents can receive the feedback immediately or after a period of delay.

2.6.1 Supervised Learning

Michalski (1993) classifies learning from examples and instructions as supervised learning where the feedback specifies the desired activity of the Intelligent Agents. For dynamic environment, it will be an impractical task for the source of knowledge to provide all the desired behaviour of an Intelligent Agent for all situations. Analysts should identify the method of communication between the Intelligent Agent and the source of knowledge that is providing the feedback. If the source is a user, communication can be through user-friendly interface. If the environment is the critic then the Intelligent Agent should be able to access the environment whereas if the source is another Agent, the Intelligent Agent should be able to communicate with other Agents by using a specific Agent communication language.

2.6.2 Unsupervised Learning

Michalski (1993) classifies Learning from observation as unsupervised learning where there is no source of knowledge to criticise the performance of the Intelligent Agent. The Intelligent Agent receives no explicit feedback and the objective is to find useful and desired activities based on trial-and-error and self-organisation (Send and Weiss, 1999). Unlike supervised learning, the environment or the Agent that is providing feedback is acting as observer.

2.6.3 Reinforcement Learning

Sutton and Barto(1998) highlight that Intelligent Agents use reinforcement learning to learn what to do and how to map situations to actions by maximising a numerical reward

signal. Intelligent Agents do not receive which actions to take; instead, they should discover which actions yield the most reward by trying them. Reinforcement learning is adequate for learning from interaction and dynamic environment. Intelligent Agents receive neither the correct action nor the source of knowledge provides which rewards are given which actions (Russell and Norvig, 1995). They only receive some evaluation of their actions

Reinforcement learning has four components: policy, reward function, value function, and environment model (Sutton and Barto, 1998). The policy defines the Intelligent Agent's behaviour at a given time, and maps perceived states of the environment to actions. A reward function defines the goal in a reinforcement-learning problem. It maps each perceived state of the environment to a single number, that is a reward, indicating the intrinsic desirability of that state. The Agent's sole objective is to maximise the reward it receives on the long run. The value function specifies what is good on the long run. The value of a state is the total amount of rewards an Intelligent Agent can expect to accumulate over the future, starting from that state. The model of the environment mimics the behaviour of the environment. The working environment has a direct impact on the required skills for Intelligent Agent that uses reinforcement learning feedback method. The environment can be accessible or inaccessible (Russell and Norvig, 1995). Intelligent Agents working in an accessible environment can preset different states of the environment while Intelligent Agents working in an inaccessible environment need to maintain some internal state to try to keep track of the environment. Background knowledge about the working environment is an important issue for Intelligent Agents to perform efficiently. If Intelligent Agents do not have knowledge about the working environment model, they need to learn the model of the environment as well as the utility function-state histories.

Analysts should identify when Intelligent Agents receive rewards; Intelligent Agents can receive rewards at the end of accomplishing the task, at a terminal state, or at

any state. While the Intelligent Agents that receive their reward at the end of the task or at terminal state executing their tasks, they do not need to monitor their source of knowledge concurrently. Rewards can be component of the actual utility that Agents are trying to maximise, or they can be hints as to the actual utility (Russell and Norvig, 1995).

Intelligent Agents that utilise a reinforcement learning feedback method can be passive or active learners (Russell and Norvig, 1995). A passive learner simply watches the world going, and tries to learn the utility in various states while an active learner should act using the learned information, and can use its problem generator to suggest exploration of unknown portions of the environment. Passive-learning Intelligent Agents have a fixed policy, and they do not need to worry about which actions to take. Active Intelligent Agents should consider what actions to take, what their outcome may be and how they will affect the rewards received. Intelligent Agents have to exploit what it already knows in order to obtain rewards; but it also has to explore in order to make a better selection of actions. The dilemma is whether Intelligent Agents pursue exploration or exploitation exclusively without deteriorating the task.

Intelligent Agents should take a trade-off between their immediate good as reflected in their current utility estimates and their long-term well-being. An Intelligent Agent that simply chooses to maximise its rewards on the current sequence can easily be stuck in a nut. At the other end of the extreme, continually acting to improve its knowledge is useless if it never practices the knowledge (Russell and Norvig, 1995). Reinforcement learning is an appropriate learning feedback method for Intelligent Agents working in dynamic environment. One of the main features of Intelligent Agents is their capabilities to work in a dynamic environment as the Internet.

Learning feedback methods has a direct impact on the performance of Intelligent Agents. The methods can be supervised, unsupervised or based on reinforcement learning feedback method. In supervised learning methods Intelligent Agents receive the correct answer from a source of knowledge about their performance while in unsupervised

learning method no source of knowledge is available to guide the learning behaviour. For reinforcement learning the source of knowledge provides some feedback about the performance of Intelligent Agents without providing what is the right answer. Designers should identify which method will be available to Intelligent Agents in order to develop their learning features.

2.7 Learning Location

Intelligent Agents can be static or mobile. Static Intelligent Agents work in a single host while mobile Intelligent Agents travel between different hosts. Analysts should identify at which host Intelligent Agents will conduct learning. Mobile Intelligent Agents should be able to access the host where they will conduct learning. Static Intelligent Agents will learn at the same place where they live.

2.8 Learning Schedule and Duration

Analysts should identify the duration and schedule of Intelligent Agents learning activities. Failure to do so might allow the Intelligent Agents to start learning at the wrong time where needed resources may not be available to conduct the learning activity. Intelligent Agents have two methods of executing learning: concurrently or sequentially with the execution of their tasks. In concurrent learning, Intelligent Agents will need a multi-processor host to conduct learning. On the other hand, in sequential learning Intelligent Agents can conduct learning on a single processor host. In addition, analysts should identify learning schedule and constraints. The learning schedule will identify when learning will start, end, and trigger. Learning triggers can be categorised as schedule triggers, hardware triggers, or software triggers. In schedule triggers, learning initiates at a specific time or date. For hardware triggers, learning starts when a hardware resource is available such as specific processor, host etc... For software triggers, learning initiates

when a specific variable exceeds the trigger threshold such as a negative value of reinforcement learning reward function. Analysts should identify learning constraints such as the availability of specific resource, internet access or the learning location where the Intelligent Agents execute learning only at a specific host.

2.9 Executing the New Hypothesis

Another important issue is the timing for the new hypothesis to execute. The new hypothesis can execute immediately or after specific conditions. New hypothesis can immediately execute when such immediate execution does not have a hazardous effect on the performance of the Intelligent Agents or the environment. For non-immediate hypothesis, analysts should clearly define the conditions when the new hypothesis should take place.

2.10 Learning and Communication

There are two types of relationships between learning and communication: learning to communicate and communicating to learn (Sein and Weiss, 1999). For the first type, learning is a method for reducing the load of communication among individual Agents while in the second type communication is a method for exchanging information that allows Agents to continue or refine their learning activities. Learning can be low-level communication which is relatively a simple-query-and-answer interaction for exchanging missing pieces of information that result in shared information. On the other hand, learning can be high-level communication which is a complex communicative interaction.

Negotiation and mutual explanation is a complex communicative interaction with the purpose of combining and synthesizing pieces of information that result in shared understanding or agreement. The ability to reach agreement and the ability to learn are the main features of Intelligent Agents that allow such Agent to be intelligent. Intelligent

Agents with a limited low ability to communicate with users and Intelligent Agents will have a limited ability to benefit from the knowledge and experiences of others. This might allow Intelligent Agents to make the same mistakes that other Agents do.

2.11 Conclusion

Intelligent Agent is an Agent that can learn and reach agreement with other Agents or users. To describe learning behaviour of an Intelligent Agent, analysts should clearly define the learning goal, the knowledge type and representation, the learning strategy, the learning feedback method, the learning location, the learning schedule and duration, the relation between learning and communication, and when the new hypothesis executes.

The identification of the learning goals of Intelligent Agents is a key issue to model their learning behaviour. Learning can be either synthetic or analytic. The learning goal can be domain-dependent used by specific domains or domain-independent utilised by any domain area. In multi-Agent environment analysts need to identify if the leaning goals of different Agents complement or conflict with each other. Commitment strategy of Intelligent Agents has a direct impact on their performance to achieve the required learning goal.

Knowledge is a key aspect to describe the learning features of Intelligent Agents. Analysts should identify type(s) of knowledge used during the learning process, knowledge representation, the level of using background knowledge, and the relation between input knowledge and background knowledge. Knowledge can be procedural, relational, and hierarchical. Intelligent Agents can manipulate single or multiple representations of knowledge required during the learning process. In addition, Intelligent Agents can use background knowledge during the learning process or depend only on the new acquired knowledge. Relations between acquired knowledge and background knowledge can provide designers with scenarios for Intelligent Agents behaviour.

Learning feedback methods have a direct impact on the performance of Intelligent Agents. The methods can be supervised, unsupervised or based on reinforcement learning feedback. Intelligent Agents receive the correct answer from a source of knowledge about their performance while no source of knowledge is available for unsupervised method to guide the learning behaviour. For reinforcement learning, the source of knowledge provides some feedback about the performance of Intelligent Agents without providing what the right answer is. Designers should identify which method will be available to Intelligent Agents as to develop their learning features.

Analysts should identify where and when Intelligent Agents will conduct learning to allow designers to provide learning requirements at the learning location and time. Such identification is crucial for mobile Intelligent Agents that work in a multi-Agent environment. Analysts should also identify whether Intelligent Agents use learning to reduce the load of communication among individual Agents or they communicate to exchange information to refine learning.

The clear identification of the learning features described above has a crucial impact on the performance of Intelligent Agents. Failure to do so would allow Intelligent Agents to consume available resources and create difficult working environment for other Agents to perform efficiently.

Chapter 3: Research Directions of Extending UML to Model Agent-Oriented Systems

This chapter reviews four research directions that use UML to model Agent-oriented systems. Section 3.2 describes the first direction, Agent Unified Modelling Language (AUML) that extends UML base language to describe behaviour and structural aspects of Agents, especially interaction among Agents and the description of Agent class diagrams. Section 3.3 introduces the second direction, Agent Graph Transformation Modelling, which is an Agent-oriented modelling technique that uses UML notation, and integrates the theory of graph transformation in the proposed modelling technique. Section 3.4 reviews the third direction, Methodology for Engineering Software Agent (Message), which has a modelling language related to UML by sharing a common Meta modelling language and Meta Object Facility (MOF). Section 3.5 describes the fourth direction, Agent-object relation models (AOR), which extends UML, and provides a framework for the existence of Agents and objects in the same system under development.

3.1 Introduction

Wooldridge and Cinancaini (2001) recognise software Agents as complex systems that exhibit autonomy, reactivity, pro-activity, and social ability capabilities. Interaction is one of the most important characteristics of software Agents. Bauer et al. (2001) highlight that linking Agent-oriented standards to object-oriented ones will allow Agent-oriented systems to gain wide acceptance in the software industry. Unified Modelling Language (UML) is the de facto standard for modelling object-oriented systems. UML unifies and formalizes the methods of developing object-oriented software life cycle (Bauer, 2002). Lind (2002) identifies two possible approaches to allow UML fits some of the Agent-oriented software requirements: either to extend UML with new structural elements and diagrams or to use UML base language to describe Agent-specific aspects of the software

systems. The first approach provides new structural elements and diagrams to enhance the expressive power of the base language while the second approach adheres to the boundaries of the original language, and uses only those extension mechanisms that UML admits.

3.2 Agent Unified Modelling Language (AUML)

AUML, one of the intensive research directions, uses UML to model Agent-oriented systems comparable to the other above-mentioned directions. The early AUML research mostly concentrates on interaction among Agents. Currently AUML research extends UML 2.0 structural and behavioural diagrams to model Agent-oriented systems. The Foundation for Intelligent Physical Agent (FIPA) is an international organisation that promotes the industry of Intelligent Agents by openly developing specifications and supporting interoperability among Agents as well as Agent-based applications. FIPA establishes a Modelling technical committee (TC) to develop vendor-neutral common semantics, meta-model, and abstract syntax for Agent-based methodologies. While this research study was still in process, FIPA issued two main documents for AUML Agent class metamodel (2003a, 2004) and AUML interaction diagrams (2003b). AUML uses in both documents an extension to UML 2.0 that Object Management Group (OMG) proposes. Currently, the FIPA modelling TC is the only approach that uses the new proposed UML 2.0 standards.

3.2.1 Extending UML Behaviour Component

Recent research on extending UML concentrates on modelling interaction among Agents. Bauer et al. (2001) have developed one of the first approaches to extend UML by providing new structural elements and diagrams that enhance the expressive power of the base language. They have introduced Agents as an extension of active objects. They view

Agents as an object that can initiate tasks, and refuse requests from other objects or Agents. Bauer et al. (2001) have proposed to extend UML with a new diagram called protocol diagram. Figure 3.1 represents an interaction protocol diagram of English-auction protocol for surplus flight tickets. The Protocol diagram combines UML sequence diagram and notation of UML state diagrams to provide the specification of interaction protocols among Agents. Protocol diagrams extend UML sequence diagrams by introducing Agent roles, multithreaded lifelines, extended message semantics, parameterised nested protocols, and protocol templates.

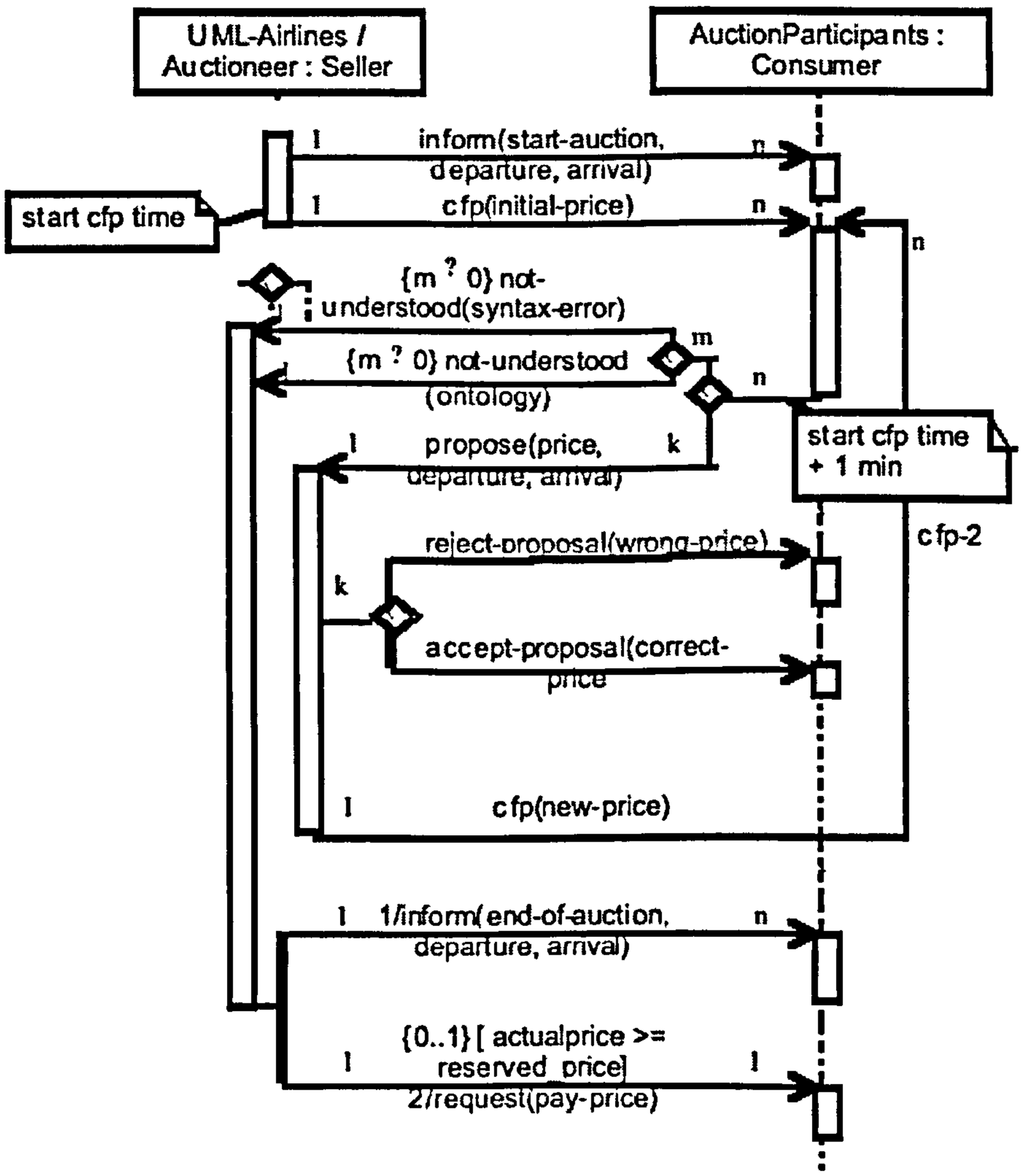


Figure 3.1 English-Auction protocol for surplus flight ticket (Bauer et al, 2001)

Bauer et al. (2001) identify Agent interaction protocol (AIP) as a communication pattern with an allowed sequence of messages between Agents that have different roles and constraints on the content of the messages. The messages semantics are consistent with the communicative (speech) acts (CAs) within a communication pattern. Speech acts are certain class of natural language utterances that have the characteristics of actions where they change the state of the world in a way analogous to physical actions, such as accept-proposal performative from the FIPA Agent communication language protocol that allows an Agent to accept a proposal made by another Agent (Wooldridge, 2002).

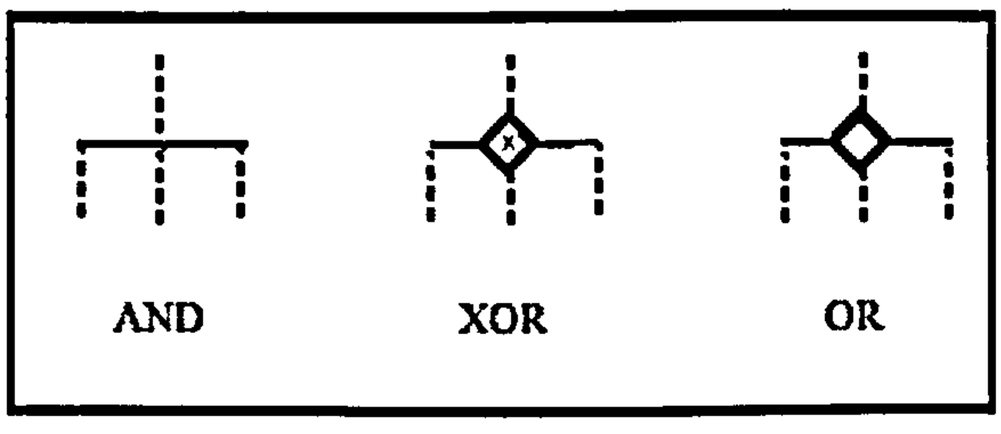


Figure 3.2 And, XOR, and Or connectors (Bauer et al, 2001)

Bauer et al. (2001) propose Agent roles as one of the extensions to UML. Agent roles are a set of Agents that satisfy distinguished properties, interfaces, service description, or have a distinguished behaviour. Agent roles can perform various roles within one interaction protocol. The general form for describing Agent roles in Agent UML is "instance-1... instance-n / role-1.....role-m:class" that distinguish a set of Agent instances instance-1,...,instance-n satisfying the Agent roles role-1,..,role-m, n, m >= 0 and the class it belongs to. Bauer et al. (2001) propose new logical connectors for the protocol diagram, AND, OR, and XOR connectors, that can represent parallelism and decisions corresponding to branches in message flows. Figure 3.2 shows the proposed logical connectors. The XOR connector interrupts the threads of interaction by splitting it up into different threads of interaction; figure 3.3 shows different usage of XOR connector. The AND connector combines the inputs while the OR chooses either one or nothing from the input.

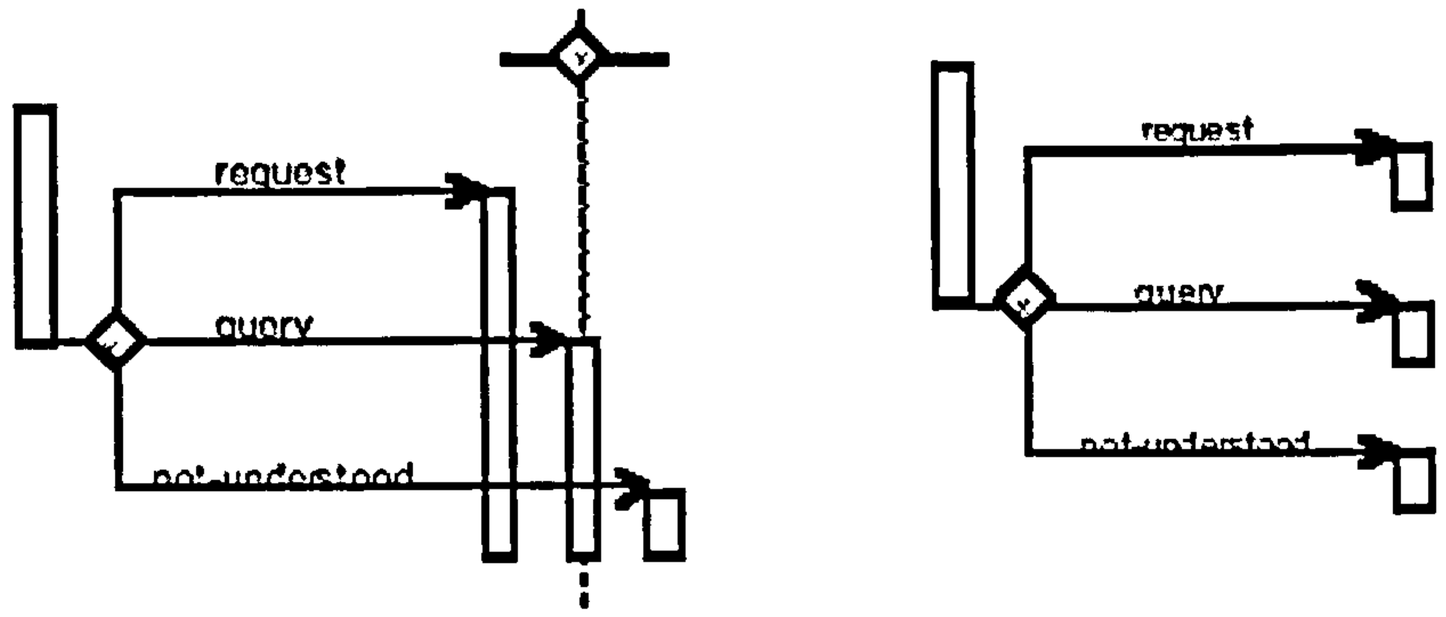


Figure 3.3 Full and abbreviated notation of XOR connection (Bauer et al. , 2001)

In addition, Bauer et al. (2001) identify that protocols do not only codify as recognizable patterns of Agents' interaction that can be reusable modules of processing; rather, they are also treated as first-class notations. Protocol diagrams can allow nesting and interleaving of protocols. Bauer et al. (2001) extend the semantics of UML messages by introducing the repetition of a part of a protocol. An arrow, or one of its variations, that is usually marked with some guards or constraints, represents a repetition part. Bauer et al. (2001) emphasise that AUML provides the user who is familiar with object-oriented software development with an easy method to understand multi-Agent systems. Analysts view such systems like a society of Agents rather than a distributed collection of objects. In addition, designers spend much less time for designing Agent-oriented system by a minor amount that grows with the number of Agent-based projects.

Odell et al. (2001) further edit and explain how AUML is useful for representing Agent interaction protocols by providing three-layers approach. The first layer represents the protocol by extending sequence diagrams to act as protocol diagrams. Protocol diagrams indicate with the proposed XOR connector the Agent's feature of the freedom to choose to respond or not to the received request. Furthermore, Odell et al. (2001) highlight that analysts can codify protocols as recognisable patterns of Agent interaction. Protocols become reusable modules of processing that analysts can treat them as first-class notations.

Odell et al. (2001) use UML package diagram to treat protocols as templates for customisation of analogous problem domains (figure 3.4).

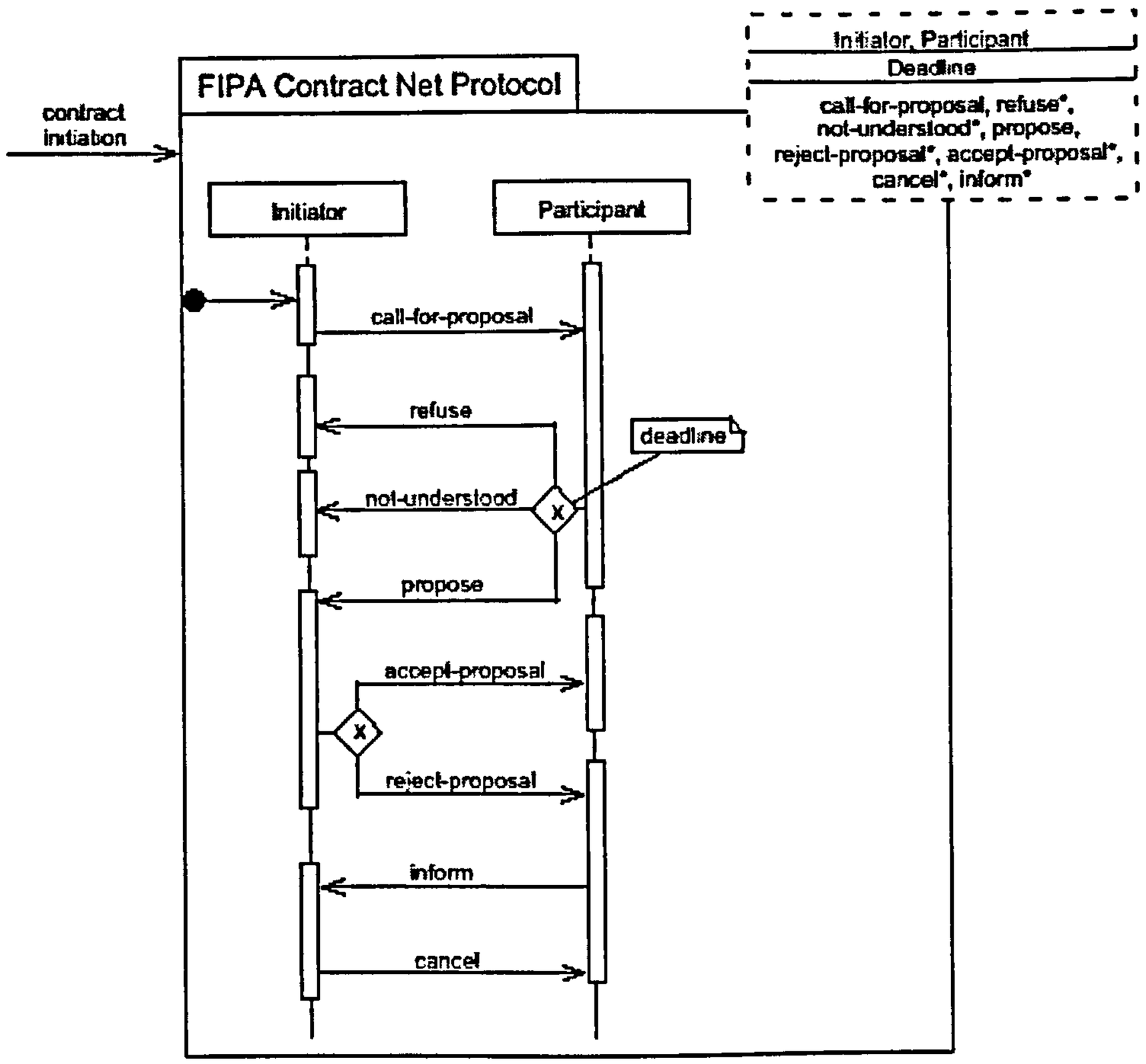


Figure 3.4 A generic AIP expressed as a template package (Odell et al, 2001).

Odell et al. (2001) describe that the second layer represents interaction among Agents. They extend sequence diagrams to represent Agents or roles of Agents. Similar to objects in UML, they use the Agent-name/role:class syntax to describe Agents. They extend UML to support concurrent threads of interactions similar to those identified by Bauer et al. (2001). Although the connectors are similar to those described by Bauer et al. (2001), they provide more examples for using concurrent communications as shown in figure 3.5.

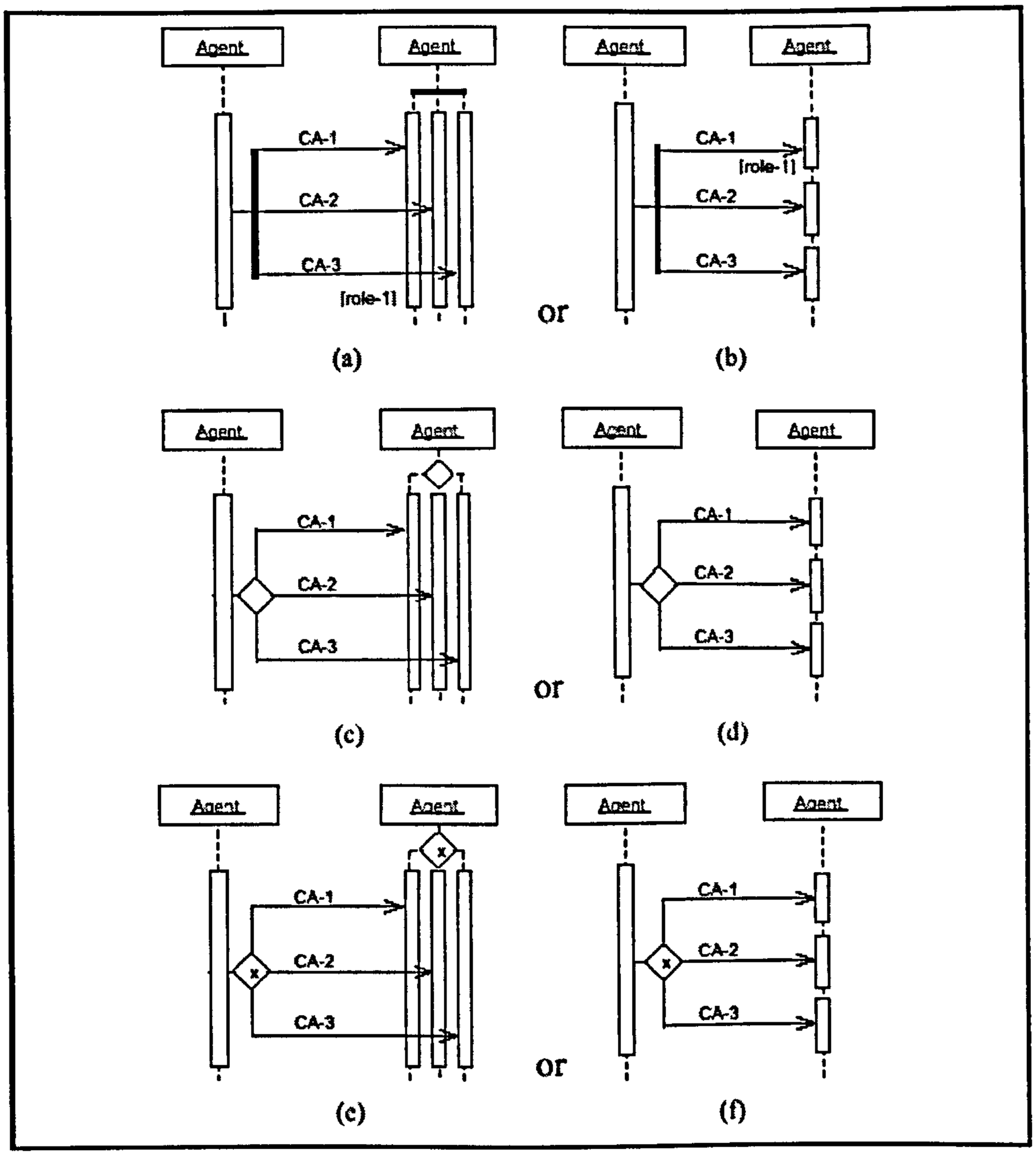


Figure 3.5 Multiple techniques to express concurrent communication with an Agent playing multiple roles or responding to different CAs (Odell et al., 2001).

Odell et al. (2001) emphasise that activity diagrams provide simple graphical representations to visualise processes and concurrent asynchronous processing being an important feature for some Agents. Odell et al. (2001) use state chart diagrams to represent state–centric views that emphasise permissible states more prominently than the transitions Agent processing does. The third layer represents internal Agent processing Odell et al. (2001) point out that activity diagrams and statechart diagrams can describe the internal processing of a single Agent. Their version of AUML does not only concentrate on interaction among Agents by specifying protocols as a whole but also

conveys the interaction pattern within a protocol, and expresses the internal behaviour of Agents.

Huget (2003) proposes to extend AUML protocol diagrams with features that are useful to work in the domain of reliability, electronic commerce, and supply chain management that previous AUML features lack. He provides extensions for AUML protocol diagrams to describe broadcasting messages, synchronization, triggering actions, exception handling, time management, atomic transactions, and message sending until the receiver acknowledges receipt features. Figures 3.6 and 3.7 depict the different extensions that Huget proposes for AUML protocol diagram that can fulfil the features he indicates.

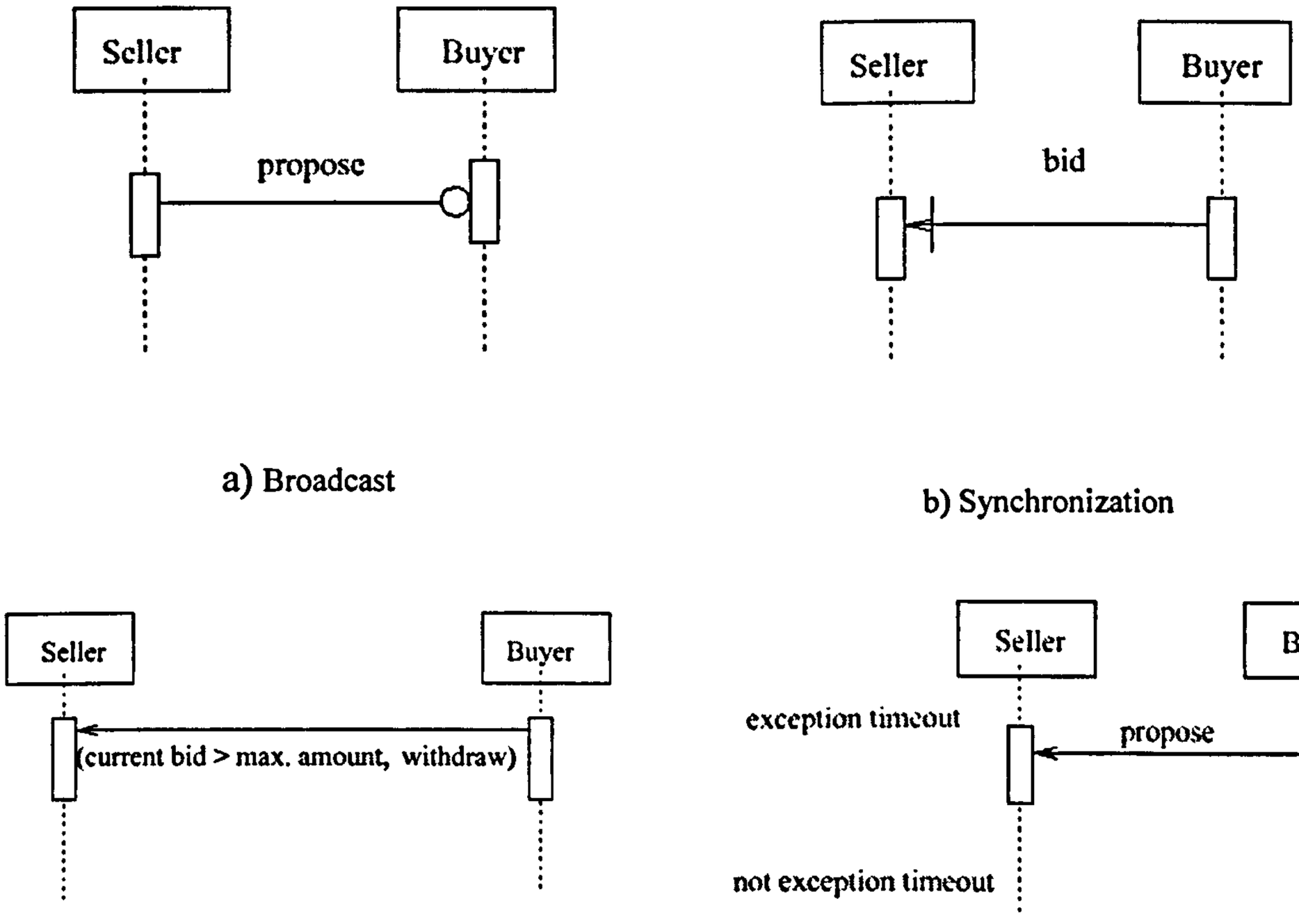
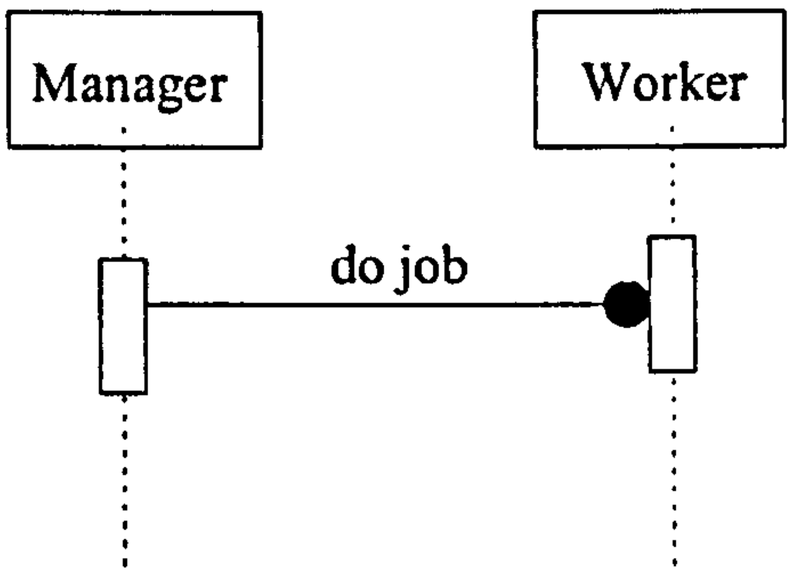
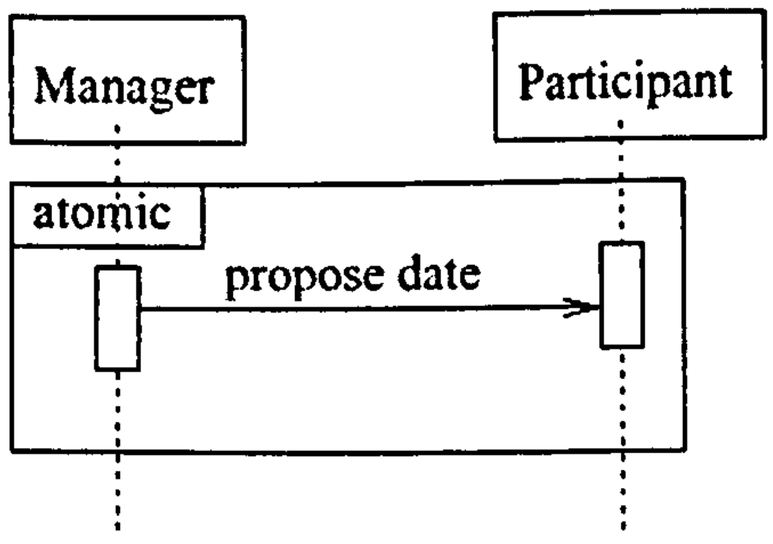


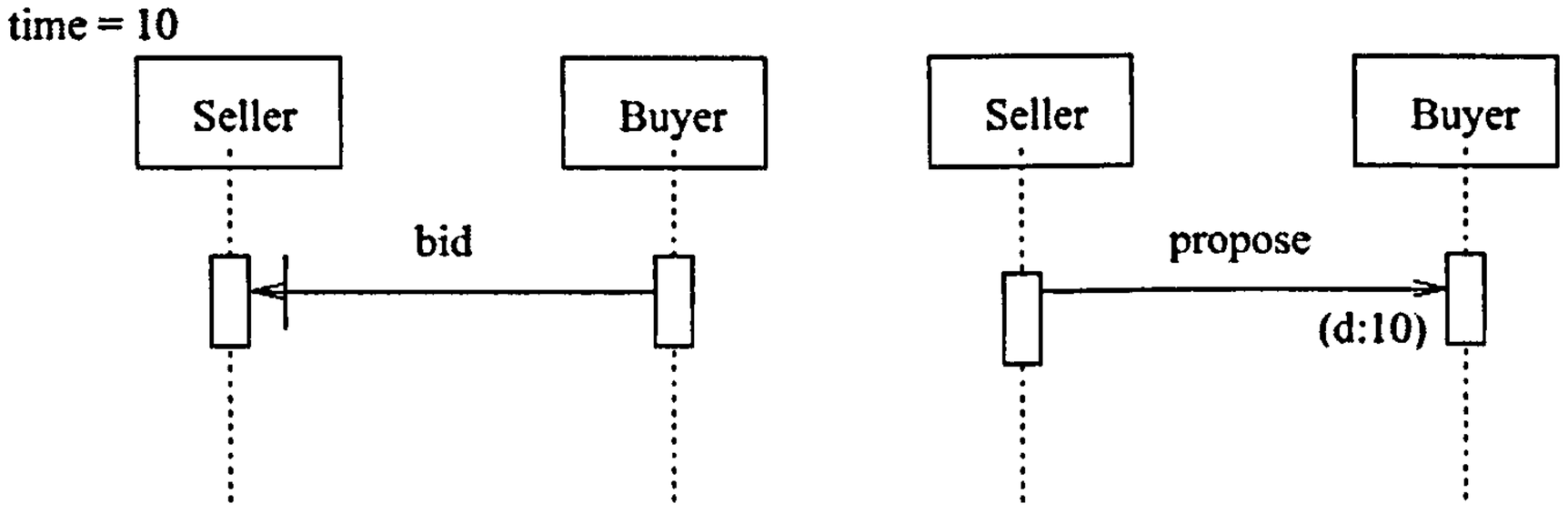
Figure 3.6 Huget extensions to AUML protocol diagrams (Huget, 2003)



a) Sending messages until delivery



b) Atomic transactions



c) Time management

Figure 3.7 Huget extensions to AUML protocol diagrams (Huget, 2003)

Lind (2002) uses the second approach, adhering to the boundaries of UML, to model Agent-oriented systems. He proposes a notation for an Agent-interaction protocol, based on the basic elements of UML activity diagram. He highlights that extending UML by new structural elements and diagrams would provide a major risk, as the new extensions will not be universally understandable. He extends the idea of swim lanes[3] of activity diagrams as a means to describe the roles as <<stereotypes>> that occur within the

[3] Swim lanes: Activity diagram can be divided into partitions, referred to as swim lanes, to show who does which action.

application. Figure 3.8 depicts an example of augmented activity diagram. The roles within the diagrams link each other in explicit communication channels (<<channels>>) that manage the message exchange between two roles.

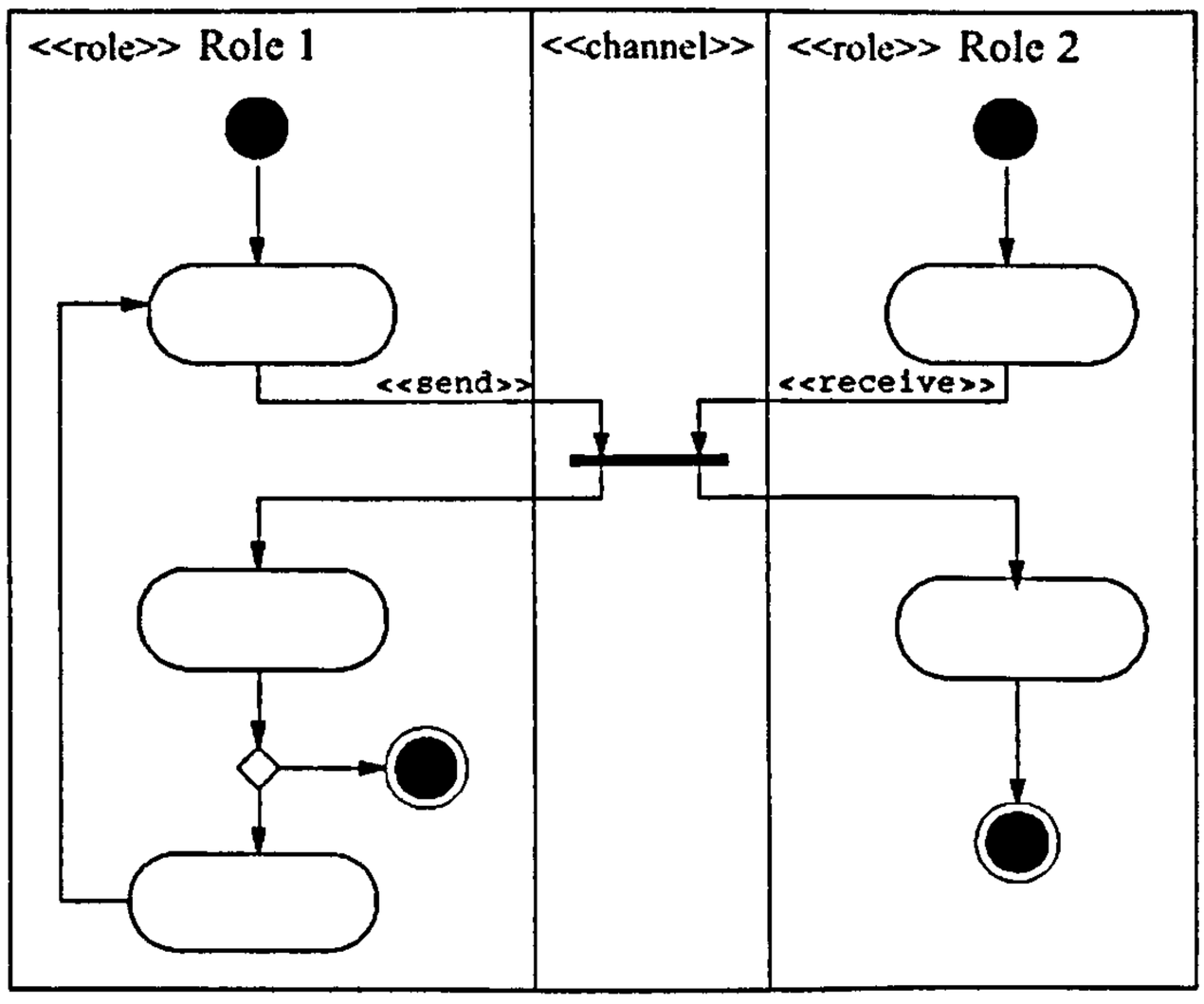


Figure 3.8 Augmented Activity diagram (Lind, 2002).

Synchronisation points (figure 3.9) illustrate message exchange that models sending and receiving of messages. The <<synchronization point>> stereotype includes a semantic extension of the UML, <<timeout>>. Lind (2002) highlights the importance of <<timeout>> stereotype, as it protects either the sender or receiver from infinite blocking.

As AUML is gaining more recognition within the Agent-oriented research community, Koning and Romero-Hernandez (2003) introduce a textual notation for the AUML modelling specification, and show how one could translate it in order to generate both an extended finite state machine and a specification that model-checker can directly process.

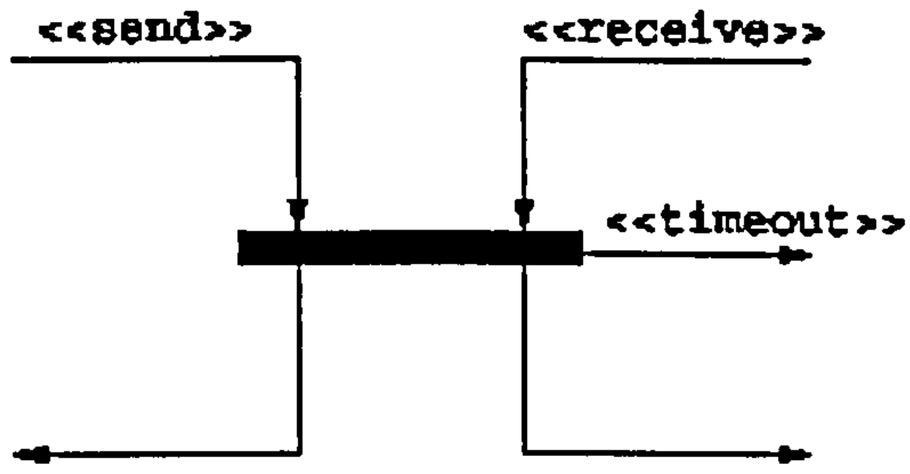


Figure 3.9 Synchronization Point (Lind, 2002).

Koning and Romero-Hernandez (2003) present the Atos system that takes AUML sequence (protocol) diagram specifications from text files as input and translates them into a single Promela specification, C-like process modelling language. Next, a SPIN[4] model checker tests them. The Atos system policy uses "as is" policy regarding translation of input specification, and allows the developer to change the input specifications, the final Promela specifications, or even the intermediate specifications. The author tries to use Atos for developing Agent-oriented CASE tool.

3.2.1.1 FIPA Agent Interaction Diagrams

FIPA Modelling TC uses UML 2.0 interaction diagrams, namely sequence diagram, interaction overview diagram, communication diagram, and timing diagram to model Agent-oriented interactions (FIPA, 2003b). Through this AUML version, FIPA extends UML sequence diagram to act as AUML sequence diagram. In a previous version, FIPA Modelling TC calls the AUML extension of sequence diagram a protocol diagram. AUML sequence diagrams consider two parts: the first is a frame, which delimits the sequence and the second is a message flow between roles through a set of lifelines and

[4] Spin is a generic verification system that supports the design and verification of asynchronous process system (Holzmann, 1997).

messages. FIPA modelling TC tries as much as possible to use UML 2.0 notations while it does add new structural or behavioural components. AUML connectors, for AND, OR, and XOR proposed in previous versions, are eliminated with the use of the standard UML 2.0 notations of splitting or merging. Unlike UML, FIPA modelling TC extends UML lifelines to represent several Agents. Figure 3.10 depicts an example of AUML sequence diagram for FIPA request Interaction protocol.

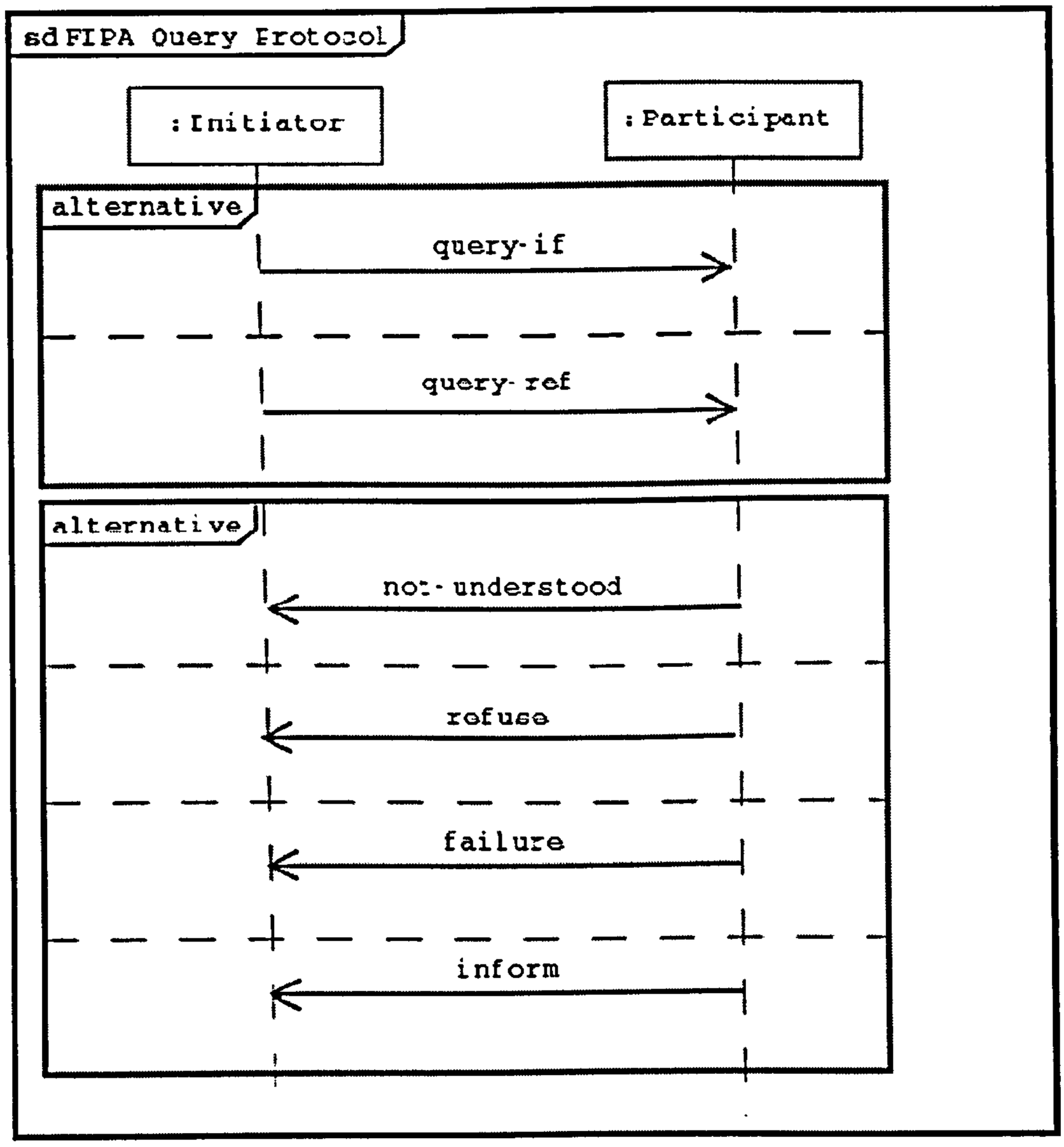


Figure 3.10 FIPA Request Interaction Protocol (FIPA, 2003b)

UML Interactive overview diagram focuses on the overview of the flow of control where the nodes are Interactions or InteractionOccurrences fragments. FIPA modelling TC highlights the usefulness of the interaction overview diagrams to express the flow of interactions that are clearer rather than using the frame of an interaction operand. FIPA

also suggests that UML communication diagrams are useful to express Agent-oriented interaction by focusing on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. Furthermore, the UML Timing diagram is useful to show the change in the lifeline state or condition of a classifier instance or classifier role over linear time.

3.2.2 Extending UML Structural Component

UML structural components such as class diagrams are vital UML components that need extending to model Agents and Agent roles. Bauer (2002) emphasises that there is a need to extend UML class diagram to support in relating the definition of Agent interaction protocol and the internal behaviour of an Agent. He expresses that an Agent has three components: communicator, head, and body. The communicator is responsible for doing physical communications by using speech acts; the head deals with goals, states, and beliefs while the body do the pure actions of an Agent.

Figures 3.11, 3.12, and 3.13 depict the Agent class diagrams that the author proposes. Figures 3.11 and 3.13 show the internal structure of the Agent while figure 3.12 shows the extension for UML classes to accommodate Agents. Bauer (2002) describes the Agent roles as in his previous work (Bauer, 2001) to have general form "instance-1.....instance-n / role-1....role-m: Class", which denotes a distinguished set of Agent instances instance-1, ..., instance-m satisfying the Agent roles role-1,.., role-m with n,m >= 0 and class it belongs to. He identifies that sending and receiving of communicative acts are the main interface of an Agent to its environment. Figure 3.14 illustrates incoming and outgoing messages respectively. To trigger the required behaviour, the Agent matches the incoming message against incoming communicative acts of the Agent. He provides an extension to distinguish class for "well formed formula" (wff) for all kinds of logical descriptions of the state, independent of the underlying logic. Bauer (2002) highlights the

need to define two variables of type wff for a pure goal-oriented semantics: permanent-goals and actual goals, while BDI semantics need four variables: beliefs, desires, intentions, and goals.

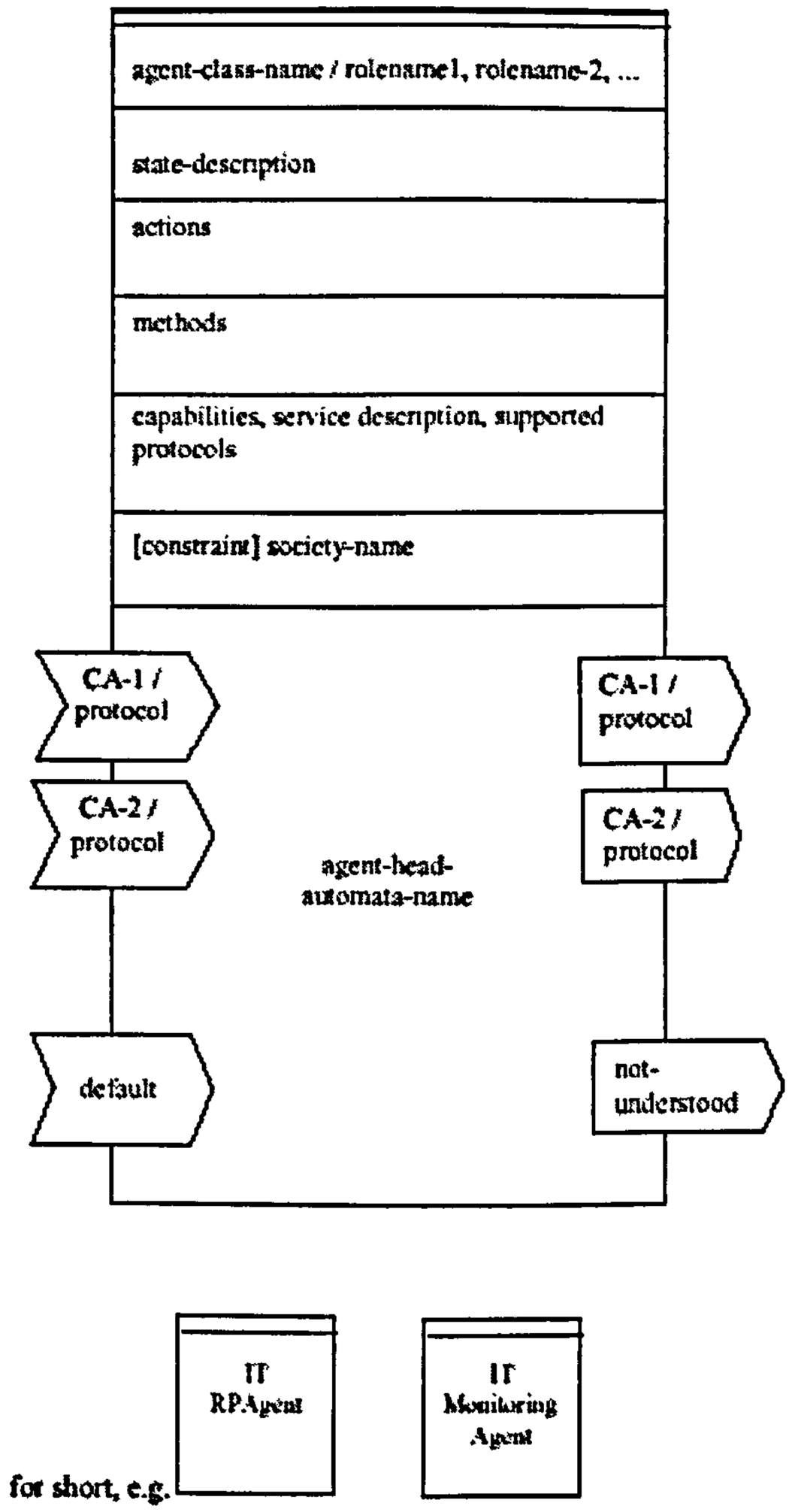


Figure 3.11 Agent class diagram and its abbreviation (Bauer, 2002)

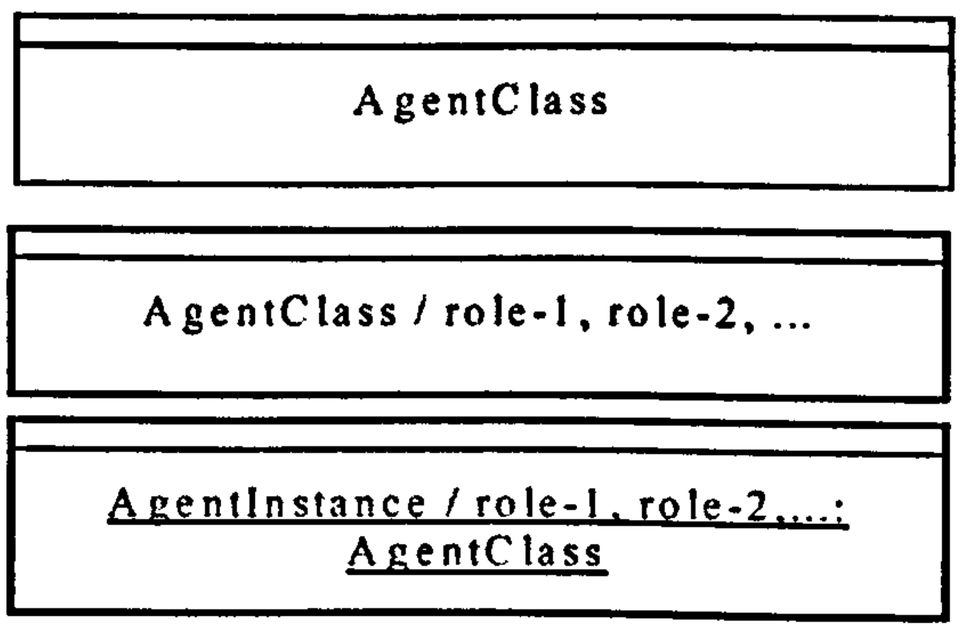


Figure 3.12 Different Kinds of Agent classes (Bauer, 2002)

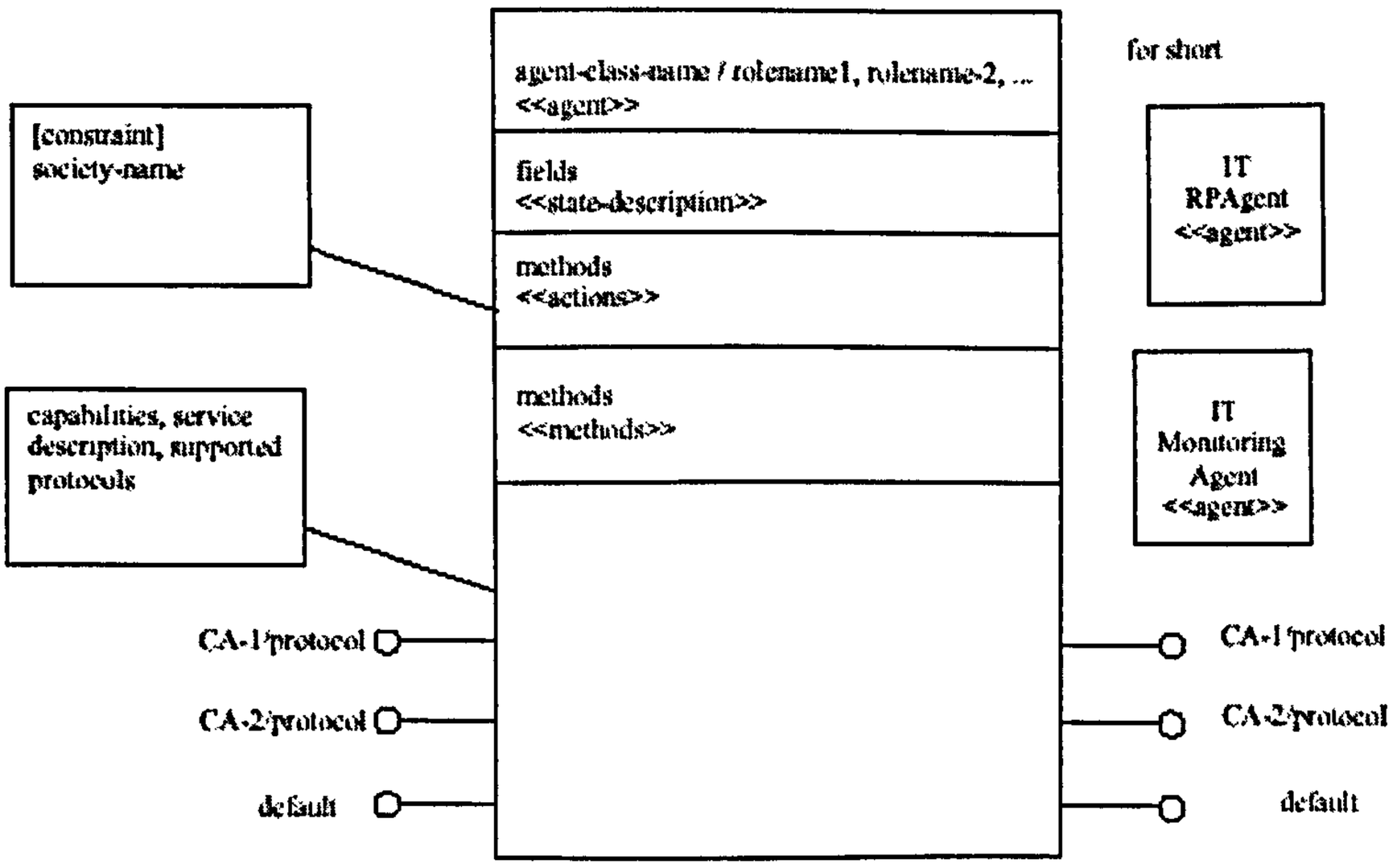


Figure 3.13 using UML class diagrams to specify Agent behaviour and its abbreviations (Bauer, 2002)

Bauer (2002) emphasises that UML sequence and collaborating diagrams are suitable for the definitions of an Agent's head behaviour while state and activity diagrams are more suitable for abstract specification of the behaviour of an Agent's head. He highlights that analysts can define pro-active behaviour through two different ways: using pro-active actions or using the Agent head automata. The Agent itself triggers Pro-active actions by using a timer or by reaching a special state. Incoming messages do not trigger Pro-active behaviour.

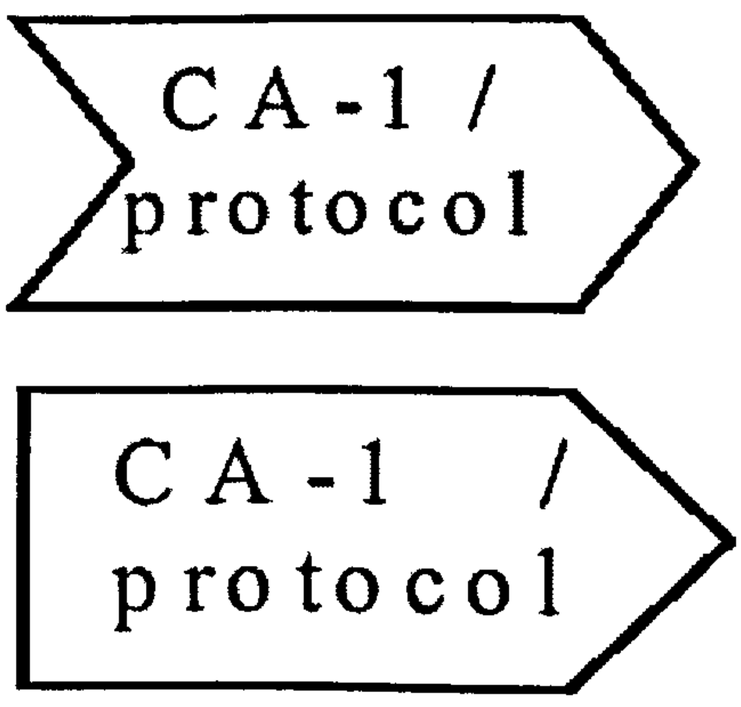


Figure 3.14 incoming and outgoing messages

3.2.2.1 FIPA Agent Class Superstructure Metamodel

FIPA modelling Technical Committee (TC) (FIPA, 2004) extends UML 2.0 class specification to accomplish the relative complexity and differences between Agent-oriented and object-oriented systems. FIPA modelling TC extends UML 2.0 classifier to have Agent Classifier that provides classification of Agent instances (figure 3.15). In addition, Agent Classifier specialises in two ways: Agent Physical Classifier and Agent Role Classifier. Agent Role Classifier describes sets of normative features or roles that Agents may hold or request to play. While Agent Physical Classifier describes sets of core or primitive features for those associated instances of the Agents it classifies. A man can be a father, a manager, or a husband; Agents Role Classifier will describe such roles. Agent Physical classifier describes the feature of the man: height, eyes colour, etc.

FIPA Modelling TC identifies that Agent classifier is the core modelling constructs. Such constructs define the instance of Agents for a system and the classification for those Agents (FIPA, 2004). FIPA Modelling TC defines Agent Class description, as "an Agent is an instance specification specifying the existence of an entity that is classified by an Agent Classifier; which completely or partially describes the entity" (FIPA, 2004). In addition, FIPA Modelling TC identifies groups as a set of Agents that associates together by some common interest or purpose. Groups encourage and support interaction among those Agents that are members of the same Group class extends UML 2.0 structured classifier. Thus, each Group is a UML composite structure. Groups can act as Agents, Agentified group[5], or establish a set of Agents for a purpose, that is non-Agentified group[6].

---

[5] "An Agentified group is a group that is also a subclass of Agent" (FIPA 2004, 14).

[6] "A Non-Agentified group is a group that is not a subclass of Agent" (FIPA 2004, 14).

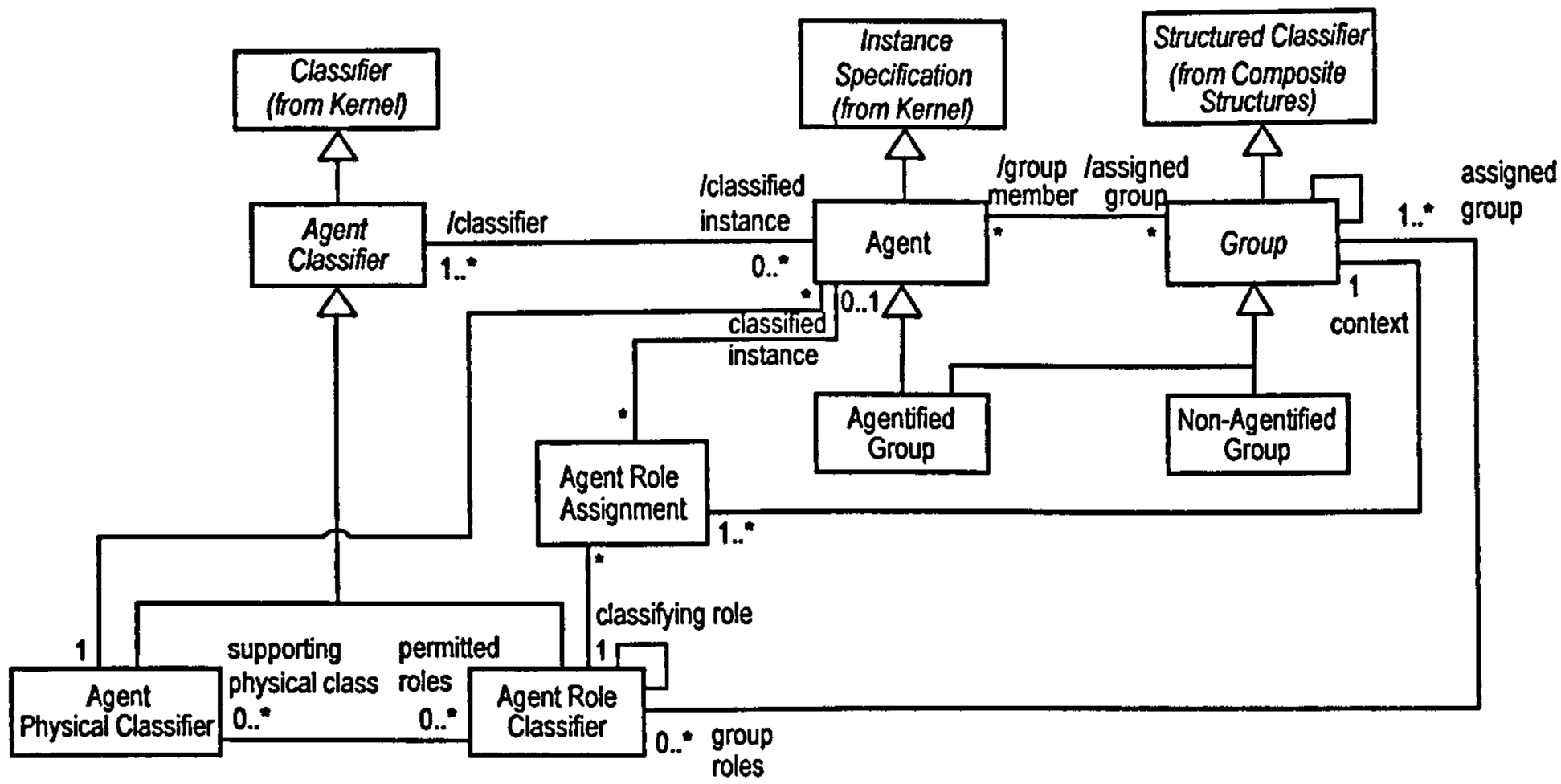


Figure 3.15 FIPA proposed Agent class abstract syntax (FIPA, 2004)

Agent Unified Modelling Language research is the most intensive research area for using UML to model Agent-oriented systems. The research in this direction has influenced the new standards of UML 2.0 to include notations for more complex behaviours than offered in UML 1.x to be able to model some of the Agent-oriented system requirements such as concurrency, parallelism, and branching processes. The importance of the latest attempt by FIPA Modelling TC (June 2004) is currently the only research conducted to use UML 2.0 to model Agent-oriented systems. This research direction does not concentrate on developing a methodology; rather, it uses and extends the basic language of UML 2.0. The AUML research direction focuses on interaction behaviour of Agents with a recent attempt to model Agent Class Superstructure Metamodel.

3.3 Agent Graph Transformation Modelling

Depke et al. (2001) introduce Agent-oriented modelling techniques that use UML version 1.x notation. They divide the modelling process of Agent-based systems like modelling of object-oriented systems: requirements specifications, analysis, and design phases. They express that the informal descriptions of the system functionality identifies the requirements of the system during the requirement specification phase. Through that

phase, Depke et al. (2001) use UML Use case diagrams to describe the functionality and actors of the system. They have introduced an extension to Use Case diagram by adding Agents that are UML actors but with a square head. The Use case diagram shows Agents required for the system and their Use Cases (figure 3.16). Then analysts describe each use case by proposing graph transformation rule that shows modelling a before-after scenario of the use case (figure 3.17). Depke et al. (2001) use the UML sequence diagram to describe communication between actors participating in a Use case.

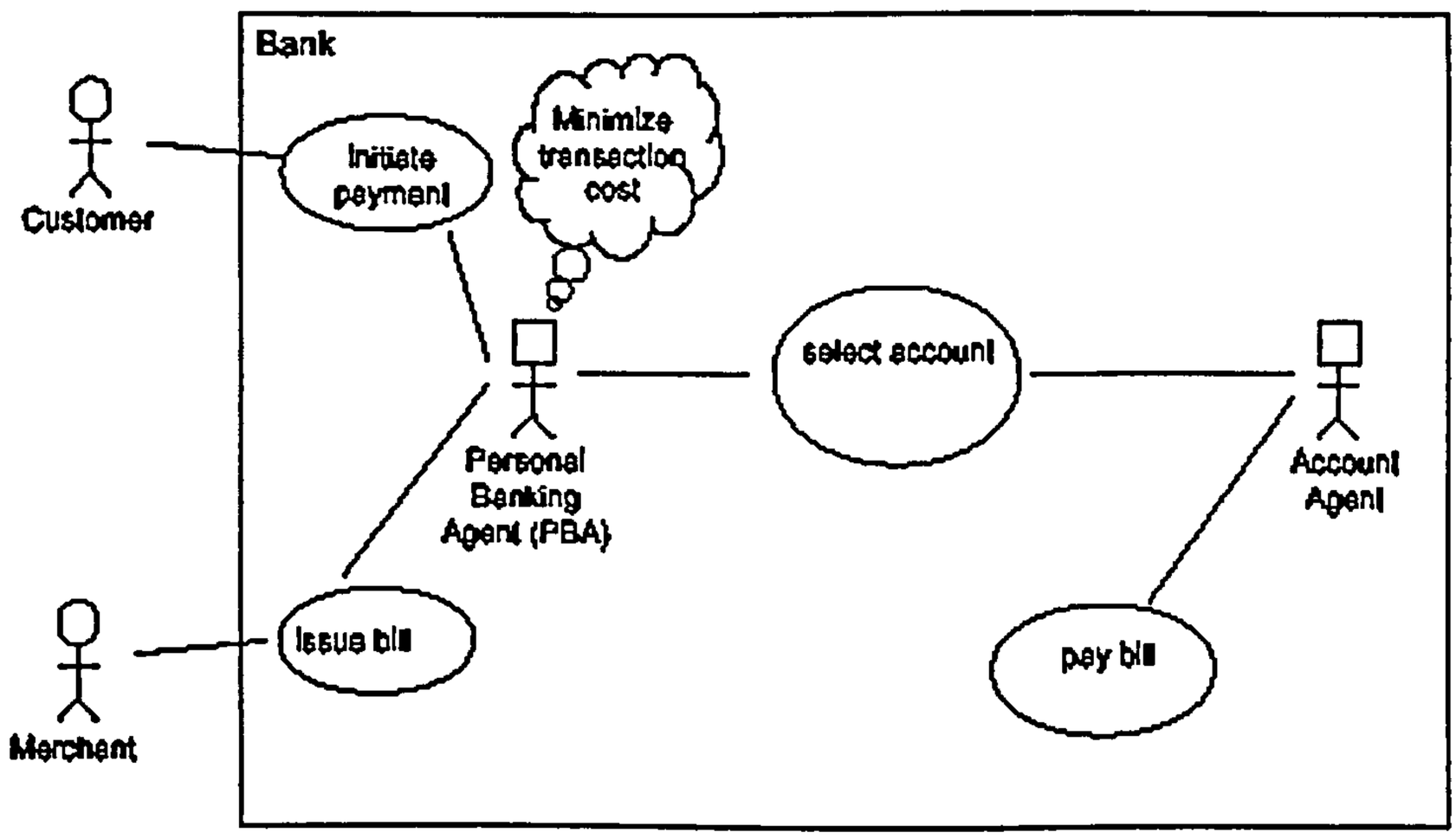


Figure 3.16 Use Case diagram for banking example (Depke et al. 2001, 108)

Similar to object-oriented analysis phase, the analysis phase consists of a structural model, a dynamic model, and a functional model. The structural model -Agent class diagram specifies the types of objects and Agents, their attributes, associations, and messages. Agent classes depict an active class with bold borders that have an extra compartment for messages.

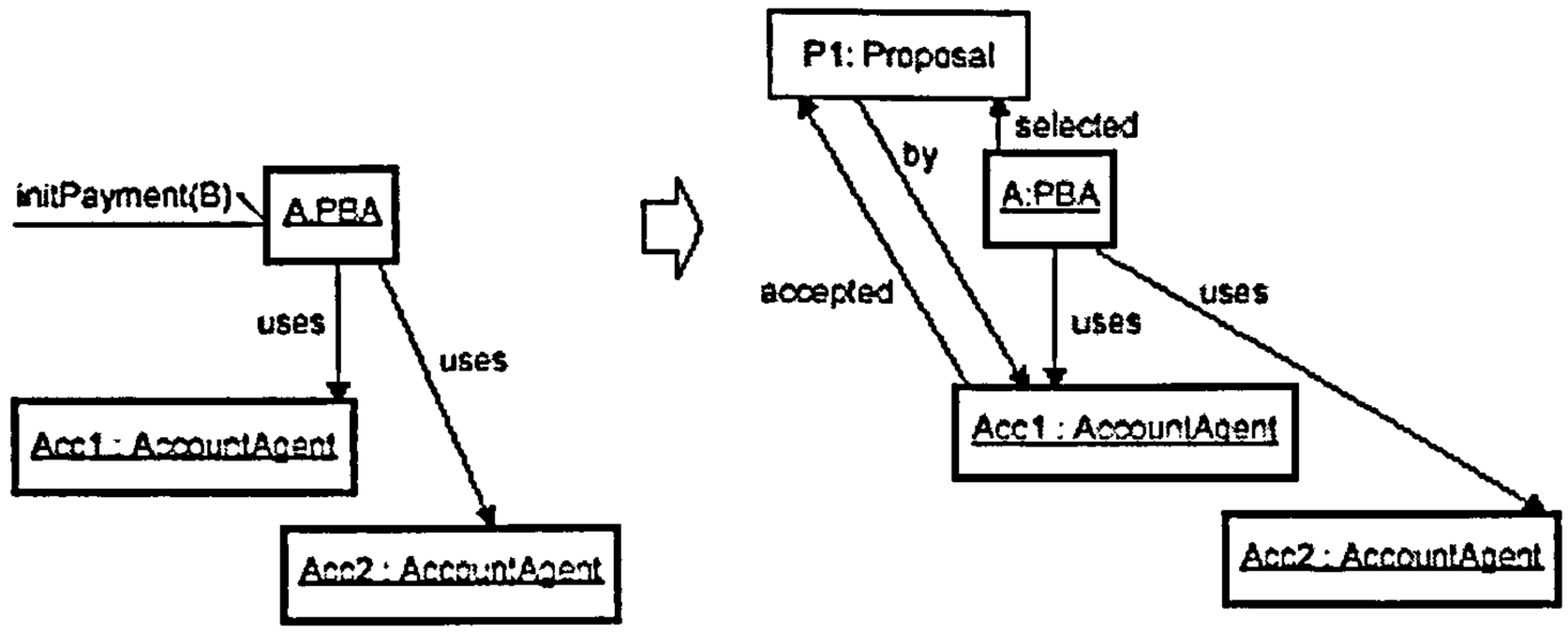


Figure 3.17 Graph transformation rule (Depke et al, 2001)

The functional model specifies the overall effect of a use case on the state of the system. They use a theory of graph transformation act as a mechanism to specify the pairs of graphs that represent before-after scenarios of use cases. Figure 3.18 illustrates three rules specifying the possible effect of the use case "Select account" shown in figure 3.16. The dynamic model focuses on the communication required to execute a certain protocol. Sequence diagram describes communication that occurs in each graph transformation resulting from the functional model. The Design phase concentrates on how the system will conduct the functions that the analysis phase describes. Similar to the analysis phase, the design phase describes three models: a structural model, a dynamic model, and a functional model.

In the structural model, Depke et al. (2001) refine the class diagram of the analysis phase and add the Agent's autonomous operations in an extra compartment. They distinguish between the Agent's message and operation notations, whereas in objects, the notion of method integrates both notions. The dynamic model specifies the ordering of operations an Agent of this class may perform. They use state diagrams to represent the ordering of operations. The functional model shows how operations declared in the design model affects the state of the system. Graph transformation rules specify operations declared in the structural model.

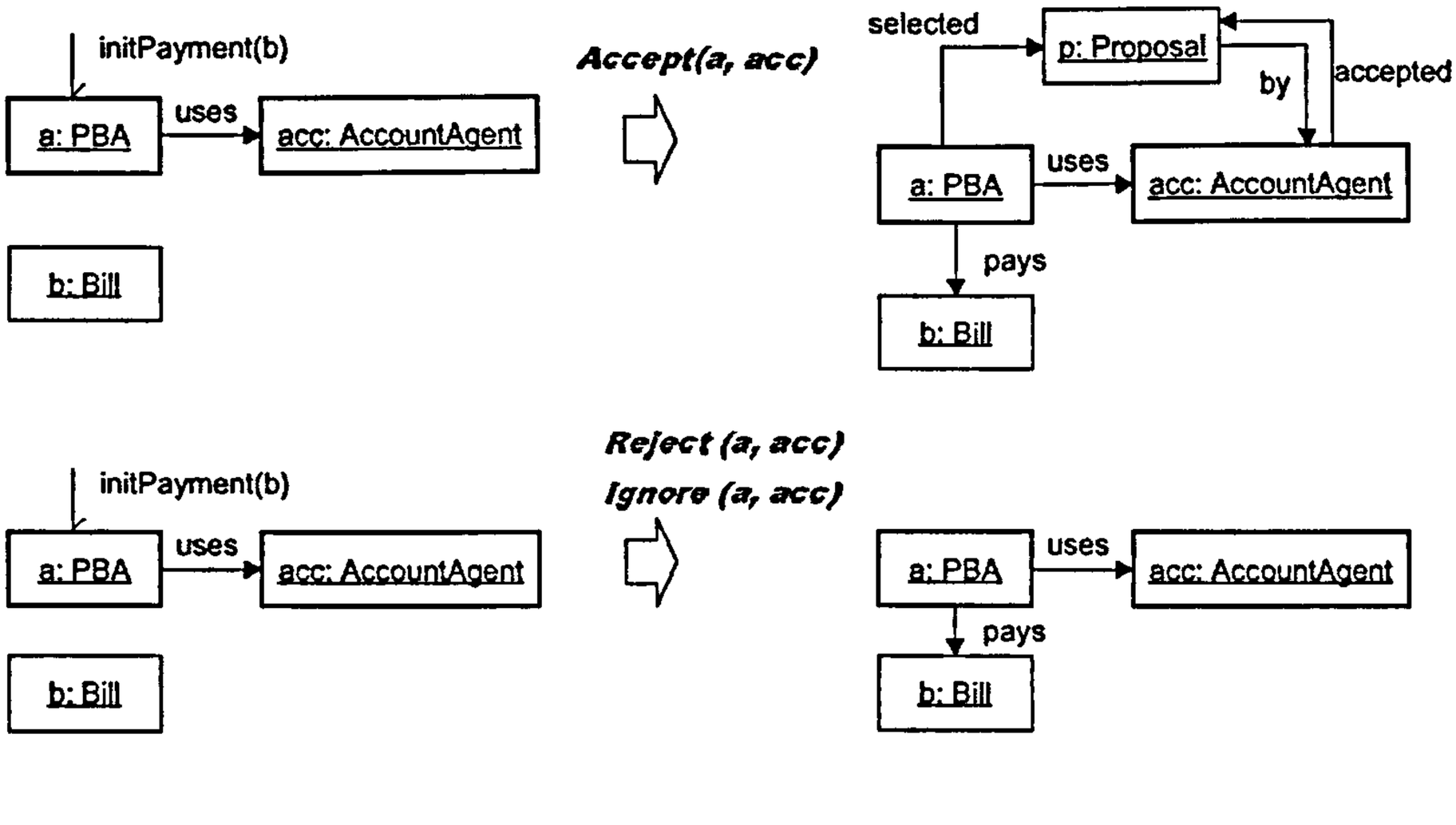


Figure 3.18 Three rules specifying the possible result of each interaction (Depke et al, 2001)

The effort exerted by Depke et al. (2001) focuses on developing a method for processing Agent-oriented systems. Similar to object-oriented methods process, their method process starts with identifying system requirements. Then the analysis phases and ends up with the design phase. The method uses and extend components from UML 1.x, which needs to be revised after the publication of the new UML standard, UML 2.0. Software developers of object-oriented systems will also need to learn more about graph transformation rules in order to be able to use the methods for developing Agent-oriented systems.

3.4 MESSAGE / UML

Caire et al. (2002) claim that they present a methodology for engineering systems of software Agents (MESSAGE) by integrating the best of the two approaches: the first approach applies existing software engineering methodologies to develop Agent-oriented systems, and the second develop a new methodology based on Agent theories. They highlight that MESSAGE has well defined concepts and a notation that uses UML

notations whenever possible. They extend UML by modelling the concepts and diagrams for the Agent knowledge level. The diagrams extend UML class and activity diagrams. Caire et al. (2002) argue that the MESSAGE modelling language shares a common Meta modelling language with UML and a Meta Object Facility (MOF). They extend the UML metamodel with "knowledge level"[7] Agent-oriented concepts. Caire et al. (2002) use UML as starting point and append entities and relationship concepts required for Agent-oriented modelling. MESSAGE uses standard UML as "data level" modelling language but provides additional "knowledge level" concepts that MESSAGE metamodel defines.

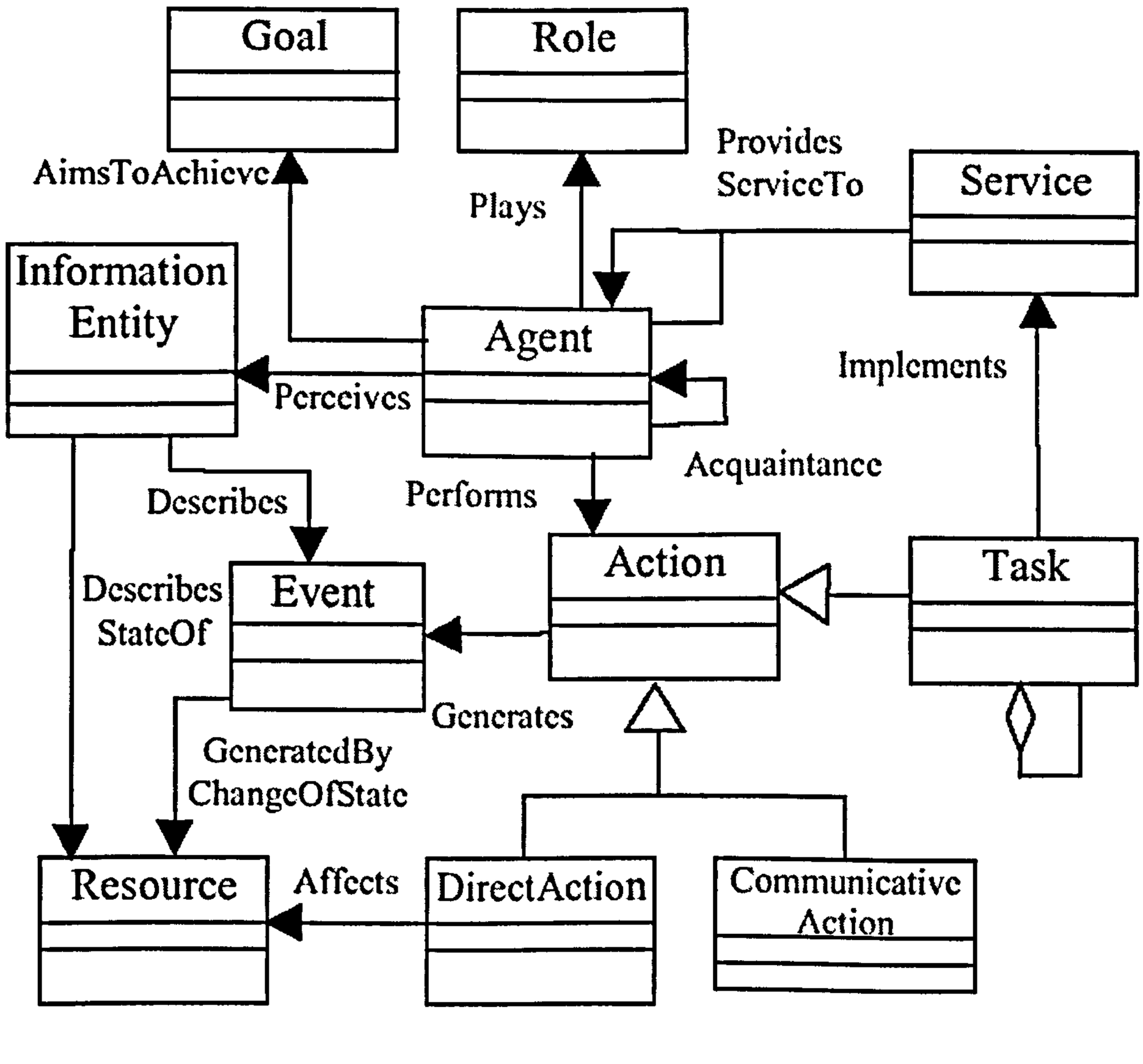


Figure 3.19 Agent centric MESSAGE concepts (Caire et al, 2002)

[7] Knowledge Level System is a system that "rationally brings to bear all its knowledge onto every problem it attempts to solve" (Lemon et al. 2004, .cogarch4/toc_defs/defs_theory/knowlevel.html)

MESSAGE Knowledge level entity concepts have mainly three categories of entities: concrete, activity, and mental-state entities. The main types of the concrete entity category are Agent, organisation, role, and resources. The major types of the activity category are tasks, interactions, and interaction protocols. Goal is one type of mental-state entity category. Figure 3.19 shows the links between these knowledge level concepts.

The analysis model identifies first the top level of decomposition; level "0" with subsequent stages of refinement that result in the creation of models at level "1." Level "0" identifies the system stakeholders and environment. Then the analyst drafts the Organisation and Goal/task views. Caire et al. (2002) highlight that level "0" focuses on the identification of entities and their relationships according to the metamodel. Level "1" concentrates on identifying the structure and behaviour of entities.

Caire et al. (2002) introduce different refinement strategies that analysts can use to refine level "0" models such as organisation-centred approaches, Agent-centred approaches, interaction oriented approaches, or goal / task decomposition approaches. Organisation-centred approaches concentrate on analysing overall system properties such as the system structure and the services offered. Agent-centred approaches concentrate on identifying the required Agents. Interaction-centred approaches concentrate on suggesting progressive refinements of interaction scenarios that describe the internal and external behaviour of organisations and Agents. Goal/task decomposition approach focuses on functional decomposition. Caire et al. (2002) suggest that it is preferable to use a combination of different refinement approaches with frequent loop-backs among them.

Similar to the second direction, Caire et al. use UML 1.x for their MESSAGE methodology. MESSAGE uses some of UML diagrams but also identify new views to model Agent-oriented features. The diagrams extend UML class and activity diagrams. It will not be direct transformation for well-trained object-oriented systems analysts using UML to work with the MESSAGE methodology. MESSAGE needs to review UML 2.0

diagrams to identify similarity. It will be better to adhere, as much as possible, to the core UML diagrams.

3.5 Agent-Object Relation Models

Wagner (2003a, b) proposes an Agent-oriented modelling language for the analysis and design of organizational information system called Agent-Object-Relationship modelling language (AORML or AOR), where an entity is an Agent, an event, an action, a claim, a commitment, or an ordinary object. Special relationships between Agents and events, actions, claims and commitments supplement the fundamental association, generalization, and aggregation relationships of UML class diagrams. Wagner (2003a, 138) emphasises that AORML or AOR is an extension of UML. AOR models have two types: external and internal models. External models adopt the perspective of an external observer who is observing the Agents and their interaction in the problem domain under consideration. The internal model adopts the internal view of a particular Agent under modelling process. External AOR model consists of Agents, communicative and non-communicative action events, non-action events, commitments, / claims between Agents, ordinary objects, various designated relationships, and ordinary associations. External AOR model represents a conceptual analysis view of the problem domain without mentioning any software artefact. External AOR diagrams include one or combination of Agent diagrams: interaction frame diagrams, interaction sequence diagrams, and interaction pattern diagrams (Wagner, 2003a, and 2003b).

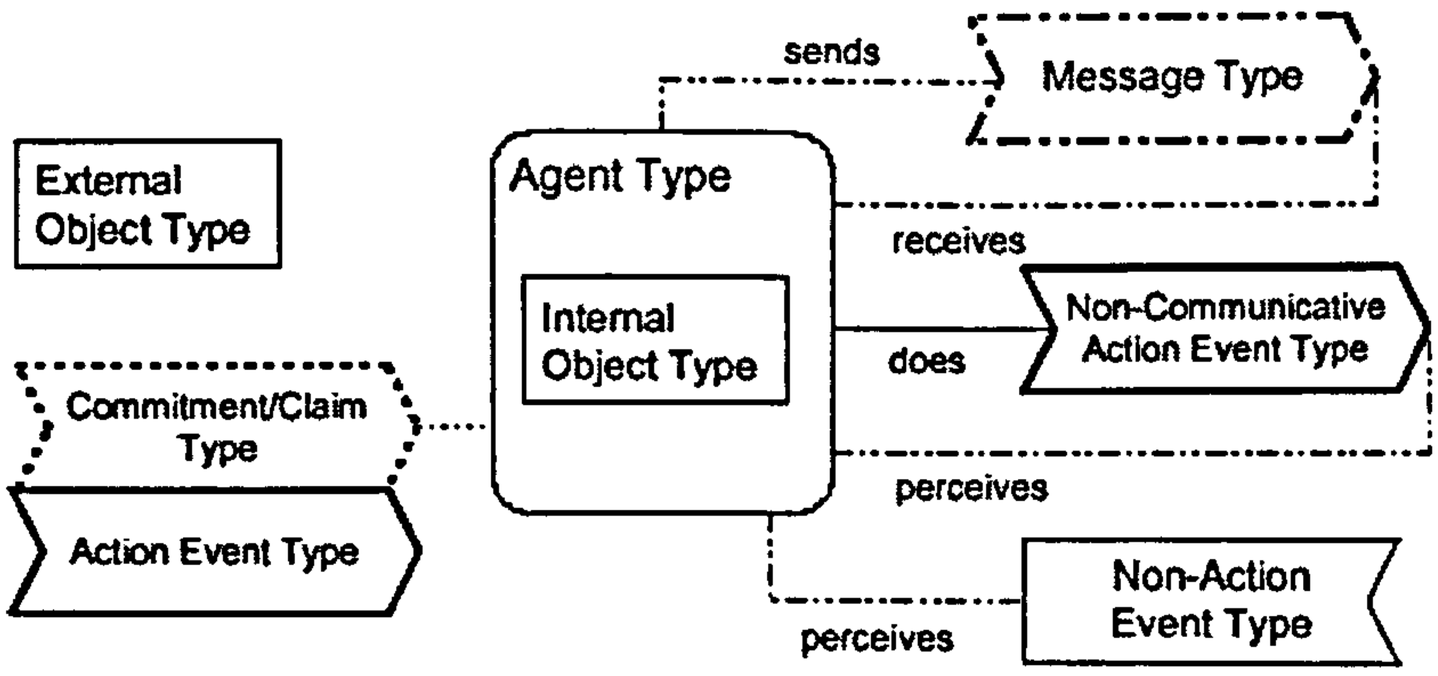


Figure 3.20 The core elements of External AOR modelling (Wagner 2003a, b)

Agent diagrams depict Agent types of the domains, certain relevant object types, and relationships among them. Interaction frame diagrams represent the action event types and commitment/claim types that determine the possible interactions between two Agent types. Interaction sequence diagrams illustrate prototypical instances of interaction processes. Interaction pattern diagrams focus on general interaction patterns expressed by means of a set of reaction rules defining an interaction process type. Figure 3.20 shows the core elements of external AOR modelling (Wagner, 2003a, b).

Internal AOR model consists of the following: other Agents, actions, commitments, events, claims, objects, designated relationships, and associations. Internal AOR models may consist of one or a combination of reaction frame diagrams, reaction sequence diagrams, and reaction pattern diagrams. Reaction frame diagrams depict other Agents, actions, and event types as well as the commitment and claim types that determine the possible interactions with them. Reaction sequence diagrams depict prototypical instances of interaction processes in the internal perspective. Reaction pattern diagrams focus on the reaction patterns of the Agent under consideration expressed by means of reaction rules. Figure 3.21 illustrates a comparison between entity relationship modelling, UML, and AOR (Wagner, 2003a, b).

(Extended) ER	UML	External AORML	Internal AORML
entity type	class active class sent signal received signal	object type agent type message type commitment/claim type	object type agent type outgoing message type incoming message type commitment type claim type
relationship type	association	association sends receives does perceives hasCommitment hasClaim	association isSentTo isReceivedFrom isPerceivedBy isCreatedBy hasCommitment Towards hasClaimAgainst
–	–	reaction rule	reaction rule
ER diagram	class diagram	agent diagram interaction frame diagram	reaction frame diagram
–	sequence diagram	interaction sequence diagram	reaction sequence diagram
–	activity diagram state machine d.	activity diagram	activity diagram state machine d.
–	–	interaction pattern diagram	reaction pattern diagram

Figure 3.21 A comparison of some important concepts of ER, UML, and AORML (Wagner 2003a, b)

Wagner (2003a, b) highlights the strengths of AOR, with respect to UML, lies in the availability of a richer set of basic ontological concepts as compared to UML. This allows capturing more semantics of a domain. AOR allows integrating state and behaviour modelling in one diagram. He identifies that a major weaknesses of the approach is the lack of an entire development path from analysis to implementation and AOR does not allow modelling the proactive behaviour of Agents.

Agent-Object Relation Modelling language (AOR), fourth direction, is one of the few methods that integrate objects and Agent in the same model. AOR uses and extends UML 1.x components, but it will not be a direct transformation for object-oriented developers to use such method. The approach is still lacking important aspects such as the

lacking of defined entire development path from analysis to implementation. AOR needs more work to be able to act as a standard approach for developing Agent-oriented systems. In addition, AOR should review UML 2.0, similar to MESSAGE, to identify similarity, and use the core language as much as possible.

3.6 Conclusion

This chapter discusses four directions to model Agent-oriented systems by using Unified Modelling language. The four directions are Agent Unified Modelling Language, Agent Graph Transformation modelling, Methodology for Engineering Software Agent (MESSAGE), and Agent-Object Relation Model (AOR). Agent Unified Modelling Language research is the most intensive research area for using UML to model Agent-oriented systems. The research in this direction has influenced the new standards of UML 2.0 to include notations for more complex behaviours than UML 1.x to model some of the Agent-oriented systems requirements such as concurrency, parallelism, and branching. The importance of the latest attempt by FIPA Modelling TC is currently the only research conducted to use UML 2.0 to model Agent-oriented systems. This research direction does not concentrate on developing a methodology like the other three directions; rather, it uses and extends the basic language of UML 2.0. The AUML research direction is focusing on the interaction behaviour of Agents where other Agent features need more research.

Depke et al. (2001) efforts focus on developing a method for developing Agent-oriented systems. Their method process, similar to object-oriented method, starts with identifying system requirements, then the analysis phases and ends up with the design phase. The method uses and extend components from UML 1.x, which needs to be revised after the publication of the new UML standard, UML 2.0. In addition, software developers of object-oriented systems will need to learn more about graph transformation rules to be able to use the methods for developing Agent-oriented systems.

Similar to the second direction, MESSAGE has used UML 1.x for their methodology. Not only does the methodology use some of UML diagrams but also identifies new views to model Agent-oriented features. The diagrams extend UML class and activity diagrams. It will not be direct transformation for well-trained object-oriented systems analysts using UML to work with the MESSAGE methodology. MESSAGE needs to review UML 2.0 diagrams to identify similarity. It will be better to stick as much as possible to the core UML diagrams.

Agent-Object Relation Modelling language (AOR), fourth direction, is one of the few methods that integrate objects and Agents in the same model. AOR uses and extends UML 1.x components but it will not be a direct transformation for object-oriented developers to use such methods. The approach still lacks important aspects such as the lack of defined entire development path from analysis to implementation. AOR needs more work to be able to act as a standard approach for developing Agent-oriented systems. In addition, AOR should review UML 2.0, similar to MESSAGE, to identify similarity, and use the core language as much as possible.

The main objective of using UML as a base modelling language for describing Agent-oriented systems is to allow software developers using object-oriented systems to migrate more easily towards the development of Agent-oriented systems. Currently, we cannot declare that any of the available UML Agent-oriented efforts can act as a standard modelling language or a method for Agent-oriented systems as UML for object-oriented systems. AUML tries to use and extend UML to describe interaction among Agents. No research direction of the above has explored how the new UML notations, UML 2.0, can model different aspects of Agent-oriented systems such as the ability to learn, the ability to reach agreement among Agents, and the ability to describe mobility.

Chapter 4: Exploring UML 2.0 to Model Structural Components of Intelligent Agents

This chapter explores the capability of UML 2.0 to model structural features of Intelligent Agents. During the writing of the thesis, the only work done to use UML 2.0 to model structural components of Agents was from the Foundation for Intelligent Physical Agent (FIPA). The research builds upon, explores, and utilises this work and provides further development to model the structural components of learning behaviour of Intelligent Agents. Section 4.2 discusses two scenarios to model Intelligent Agent structural features by using the FIPA Agent Class Superstructure Metamodel. Section 4.3 introduces the required learning compartment attributes that describe the learning features of Intelligent Agent.

4.1 Introduction

As mentioned in Chapter 3, section 3.2.1.1, FIPA modelling Technical Committee (TC) is the only published work (during the writing of the thesis) to extend UML 2.0 to model Agent structural and behaviour features. FIPA modelling TC extends UML 2.0 Classifier to have Agent Classifier that provides classification of Agent instances (figure 4.1). This Agent Classifier is able to describe the differences between Agents and Objects. UML 2.0 standards allow such extension which is not available in UML 1.X. Agent Classifier specialises in two ways: Agent Physical Classifier and Agent Role Classifier (FIPA, 2004). Until June 2004, FIPA Modelling TC has been the only effort that uses UML 2.0 to model Agent-oriented systems.

This has encouraged the author to use FIPA legal UML 2.0 extensions to explore their capability in modelling structural components of the learning dimension of Intelligent Agents. To describe the learning features of Intelligent Agents, a Learning compartment is added to the structural notation of Intelligent Agents. The learning compartment should

contains attributes to identify the Intelligent Agents learning goal, commitment strategy, learning strategy, learning feedback method, learning location, learning schedule and duration, the use of background knowledge during learning, and the input knowledge representations.

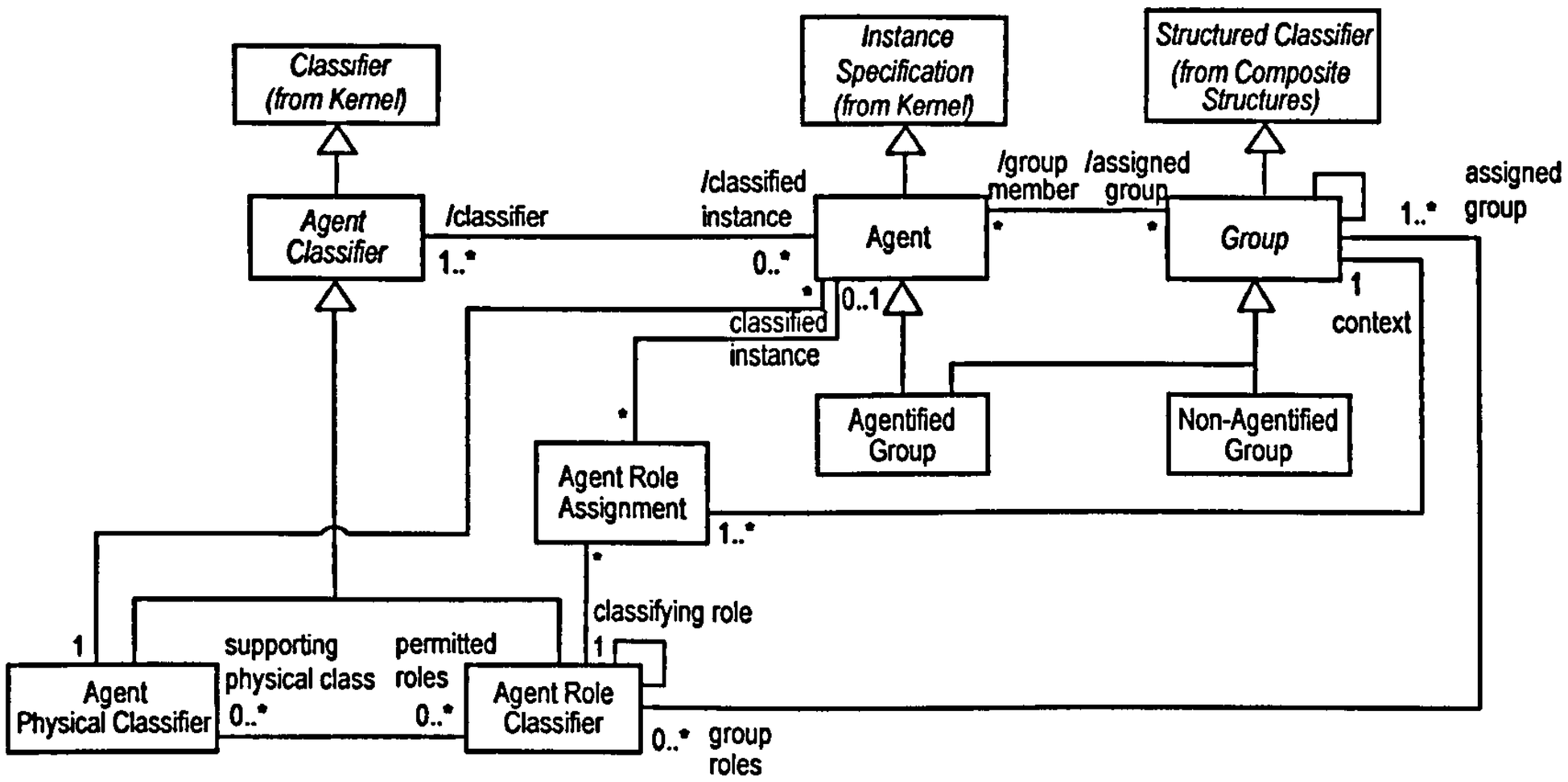


Figure 4.1 FIPA proposed Agent class abstract syntax (FIPA, 2004)

4.2 Modelling Intelligent Agents

The differences between Intelligent Agents and Agents should be clear in order to model structural components of Intelligent Agents by using FIPA Agent class superstructure meta-model, described in chapter 3, section 3.2.1.1. The differences can have two scenarios: the first scenario recognises that the relation between Intelligent Agents and Agents is similar to the relation between an active object and a passive object. An active object has its own thread of control while a passive object executes within the context of other objects. Thus, an active object has a different notation from a passive object. The second scenario identifies the Intelligent Agent as an Agent with extra attributes. The first scenario requires Intelligent Agent to have a distinct notation that is different from Agent notation and a Boolean attribute *"islearning."* In UML 2.0, the

analyst identifies a Class is active, that each of its instances has its own thread of control, if "*isActive*" parameter equals true. The notation of active class is a class box with an additional vertical bar on each side as shown in figure 4.2.

Figure 4.2 Active Class (OMG 2003, 387)

FIPA modelling TC proposes the notation of the Agent to be a solid-outline rectangle, containing underlined concatenation of the instance name (if any), a colon (':') and the classifier name or names. Intelligent Agents can have the same notation but with vertical bar on each side as shown in figure 4.3. In addition, a learning compartment is added to provide learning attributes.

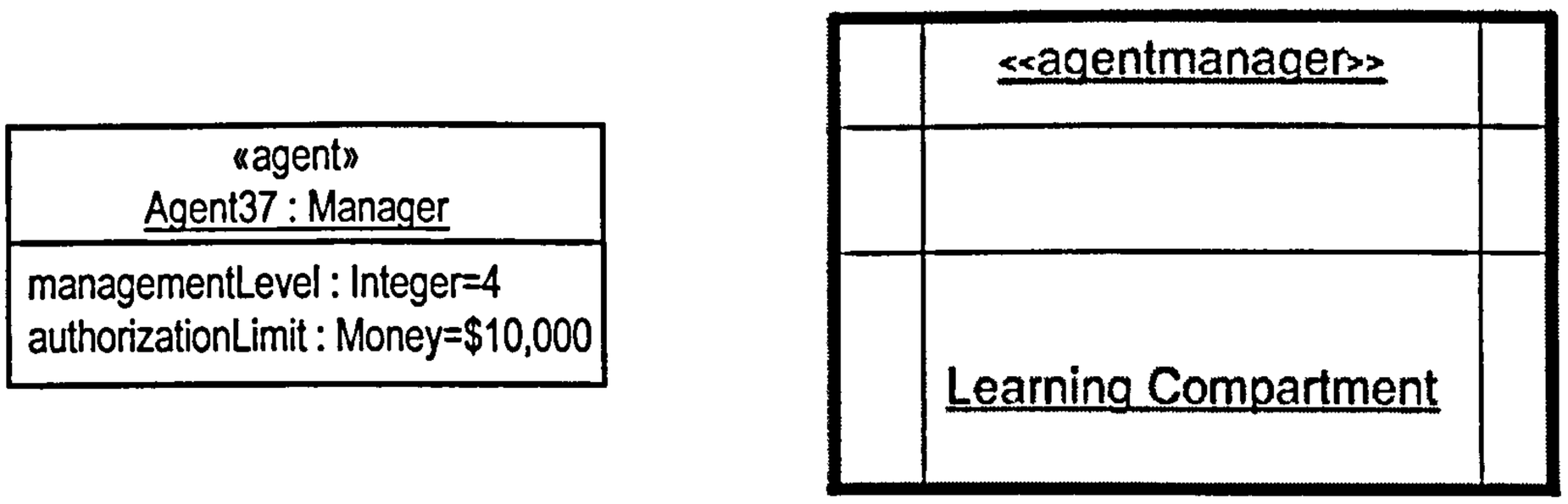


Figure 4.3 : FIPA Agent notation (FIPA, 2004) and proposed Intelligent Agent notation

The second scenario uses Agent Physical classifier that FIPA modelling TC introduces. This scenario deals with Intelligent Agent as an Agent with extra attributes. The Agent Physical classifier will have a compartment to hold learning attributes that describe the learning behaviour of the Intelligent Agent. "An Agent Physical Classifier is an Agent Classifier defines those sets of core, or primitive features for those associated instances are Agent that it classifies" (FIPA, 2004). The notation of Agent is the same

notation that FIPA proposes. Figure 4.4 depicts the modelling of Intelligent Agent and shows the learning compartment in Agent physical classifier that holds learning attributes.

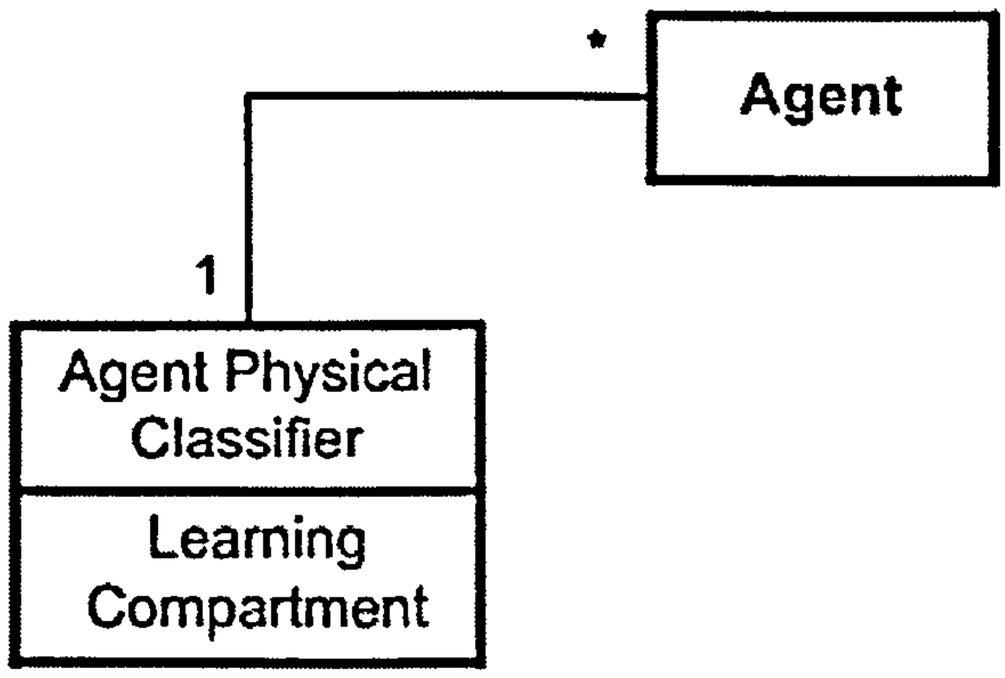


Figure 4.4 Agent class and Agent Physical Classifier

4.3 Learning Compartment Attributes:

Learning compartment provides the designer with attributes that describe the learning features of the Intelligent Agent. These learning attributes guide the designers in identifying the learning characteristics such as learning algorithms, communication requirements, and reasoning capabilities. The following section describes the proposed attributes that the learning compartment includes.

Learning-goal : literalinteger[8] [1]	This attribute identifies the type of the learning goal. "1" – synthetic learning "2" – analytic learning "3" – both
Commitment-strategy: literalinteger[1..*]	This attribute defines the type of commitment strategy the Intelligent Agent uses during learning. The following is the values of the attributes: "0"- Unknown commitment strategy "1"- Blind commitment "2"- Single-minded commitment "3"- Open-minded commitment
Background-knowledge: Boolean [1]	This attribute identifies if the Intelligent Agent uses background knowledge during the learning process. True: uses background knowledge False: does not use background knowledge.

[8] " A literal integer specifies a constant integer value" (OMG 2003, 49)

Learning-strategy: literalstring[9][1..*]	This attribute identifies the learning strategies that the Intelligent Agent uses. The first character identifies the learning strategy and the other two describe its variant. "000"- Unknown "100"- Learning from observation and discovery – unknown features "101"- Learning from observation and discovery – Passive learner "102"- Learning from observation and discovery – Active learner "200"- Learning from examples "210"- Learning from examples – instance to class "211"- Learning from examples – part to whole "221"- Learning from examples – positive examples "222"- Learning from examples – positive and negative examples "231" –Learning from examples – incremental "232" –Learning from examples –non incremental "300"- Learning from analogy "310" -Learning from analogy – transformational "320" -Learning from analogy - derivational "400" -Learning from instructions "400" -Learning from instructions - unidirectional "410" -Learning from instructions - interactive "500"- Rote Learning "510"- Rote learning – by being programmed "520"- Rote learning – by memorising
Learning-feedback: literalstring[1..*]	This attribute identifies the learning feedback method that the Intelligent Agent uses. "100" – unsupervised learning "200" – supervised learning "300" – reinforcement learning "310" – reinforcement learning – Immediate Reward "311" – reinforcement learning – Terminal State "312" – reinforcement learning – End of Learning "320" – reinforcement learning – exploitation "330" – reinforcement learning - exploration
Learning-location: string [1...*]	This attribute identifies the location(s) where Intelligent Agent conducts learning
Learning-schedule: timeinterval[10] [1...*]	This attribute describes the schedule and the duration of learning.
Knowledge-representation: literalintegers[1..*]	This attribute identifies the knowledge representation that the Intelligent Agent uses during the learning process. 0 – parameters in algebraic expression 1 – decision trees 2 – formal grammar 3 – production rules 4 – formal logic based expression 5 – predicate logic 6 – graph and network 7 – frames and schemas 8 – computer program 9 – taxonomies 10 –Bayesian network

[9] "A literal string specifies a constant string value" (OMG, 2003)

[10] "A Time interval defines the range between two Time expressions" (OMG, 2003).

4.3.1 Learning Goal Attribute

Learning goal attribute identifies whether the learning goal type is synthetic, analytic or both. Synthetic learning objective is to acquire new knowledge that is not available by the knowledge possessed while analytic learning transfers the knowledge already possessed into the form that is most desirable and/or effective for achieving the given learning goal. Synthetic learning employs induction as the primary inference while analytic learning employs deduction as the primary inference (Michalski, 1993). Designers should identify the required resources to accomplish the learning goal. In so far as synthetic learning is concerned, designers should estimate the required space to hold the new acquired data, the acquisition rate, and the access privileges required to access and collect the new data. Designers also identify the capability needed by programming language to accomplish the learning goal type. Synthetic learning needs programming language that is able to provide an induction mechanism, that is tracing backward, while analytic learning needs programming language able to perform a deduction mechanism, that is tracing forward. For analytic learning, analysts should identify the method of refining the current hypothesis and whether the Intelligent Agent should keep a copy of current and new hypotheses.

4.3.2 Commitment Strategy Attribute

As mentioned in Chapter 2, Section 2.3, commitment strategy has an impact on the behaviour of Intelligent Agents. The strategy can be a blind, single-minded, or an open-minded strategy (Wooldridge, 2002). Analysts can use two scenarios during the analysis of Intelligent Agent to design the commitment strategy: the first scenario analysts design the commitment strategy of Intelligent Agent according to the type of environment while in the second, they design the Intelligent Agent according to the required commitment strategy.

For the first scenario, analysts will identify whether the Intelligent Agent will work in a dynamic or a static environment. Intelligent Agents working in static environment can have blind commitment strategy, whereas Intelligent Agents working in dynamic environment can use single-minded or open-minded strategy. For the second scenario, analysts should identify how Intelligent Agents conclude learning. Intelligent Agents using blind commitment strategy and working in dynamic environment will consume their efforts without realising that the learning goal is not achievable. Analysts can identify constraints that affect the continuation of learning. In an open-minded commitment strategy, analysts should identify when the Intelligent Agent should believe that the learning goal is no longer valid to stop achieving the learning goal, such as the expiration of the learning goal after a specific duration.

4.3.3 Background Knowledge Attribute

The *background knowledge* attribute identifies to what level Intelligent Agents use their background knowledge during the learning process. The analysts would provide notes about the constraint and specification for the use of Intelligent Agents background knowledge. If the learning strategy attribute identifies that the Intelligent Agents use learning from examples, that is incremental learning, analysts should provide a note about which type of incremental learning Intelligent Agents are using: no-memory, partial memory, or complete memory. Each type of incremental learning will need a special design. For example, for complete memory type, designers should develop efficient mechanism to provide a fast search in current and past hypotheses. During the learning process, Intelligent Agents will need more space for partial or complete memory rather than no memory type. This might be critical for a multi-Agent environment where Multi-Intelligent Agents are using a partial or a complete memory leaning type. Mobile Intelligent Agents that also use a partial or a complete memory learning type need more

time to travel between servers than mobile Intelligent Agents using no-memory learning type. Intelligent Agents size increases during the learning process; designers should take special consideration during the development of such mobile Intelligent Agents to cater for such increase.

4.3.4 Learning Strategy Attribute

Learning strategy attribute identifies the type of learning strategy that Intelligent Agents use. They can have single learning strategy or multi-learning strategies. According to the learning strategy of Intelligent Agent, the designer designs the learning algorithm, and identifies the required resources needed to perform the algorithm. *Learning strategy* attribute identifies whether the learning Agent is utilising learning from observation and discovery, learning from examples, learning by analogy, learning from instructions, or rote learning strategies. The attribute also can describe the variants of the utilised learning strategy.

4.3.4.1 Learning from Observation and Discovery

Learning strategy attribute equals to "100" indicates that the Intelligent Agent is capable of learning from observation and discovery. The Intelligent Agent should be able to monitor the actions and reactions of the environment or the users to learn. Designers should identify the type of learning from observation whether it is clustering[11] or discovery. Clustering learning would guide the designer to use learning algorithm to conduct statistical, conceptual, or kohonen net clustering techniques. For discovery learning, the designer might use theory-driven or data-driven algorithms (Smith 1996,

---

[11] Clustering algorithms organises unclassified objects into a hierarchy of classes by measuring the similarities between objects and gathering maximally similar objects into the same group or cluster" (Smith 2004, ../staffpages/serengul/Clustering.htm).

../ML/unsupervised.html#4.2). *Learning strategy* attribute of "101" indicates that the Intelligent Agent is a passive learner while *learning strategy* attribute "102" indicates the Intelligent Agent is an Active learner. Intelligent Agent with passive-learner features should be able to monitor actions of the environment or users. As far as an active learner is concerned, designers should identify triggers at which Intelligent Agent starts exploring the reactions of users or the environment, such as time triggers or action triggers. It should also be able to trigger the environment or users to monitor their feedback; designers should provide the Intelligent Agent with required resources, as access rights, to be able to perform exploration. Furthermore, designers should be able to identify the learning algorithm that Intelligent Agent will use during exploration. Other attributes from the learning compartment, such as *background knowledge* attribute, will guide the designer in identifying the algorithm.

4.3.4.2 Learning from Examples

Intelligent Agents that use learning-from-examples strategy should have access to the source of knowledge and tutor to receive training examples needed for learning. Failure to have such access will terminate the whole process of learning. According to the type of learning from examples, designers should design the reasoning behaviour, learning algorithm, according to the classification method that Intelligent Agent will use. For *learning-strategy* attribute equals "210," instance-to-class type, the Intelligent Agent should use a learning algorithm capable to learn how to classify independent entities into a common class. *Learning-strategy* attribute equals "211,"part-to-whole type, requires the Intelligent Agent to use learning algorithm capable to link different parts of a whole object. In addition, Intelligent Agent with *learning-strategy* attribute equals "221" or "222" stands for learning from positive or positive and negative examples respectively. The types of examples guide the designers to select learning algorithm that can use the type of available examples in learning. For the first case, designers should identify conditions to limit

overgeneralisations, as there are no negative examples to perform this task. For the second case, learning from positive and negative examples would require the designers to use learning algorithms that employ positive examples to generalise concepts, and use negative example to prevent overgeneralisations such as candidate-elimination algorithm[12].

For *learning strategy* attribute of "231,"(Learning from examples – incremental) designers should identify a learning algorithm capable of updating the hypothesis after providing each example, such as candidate-elimination, ID4, ID5, or ID5R algorithms. As for *learning strategy* attribute "232," non-incremental learning, designers should identify whether the Intelligent Agent requests to start and stop learning or the teacher has the control. In addition, designers should identify the proper conditions for the learning process, where the absence of the Intelligent Agent will not trigger hazard situations. Nevertheless, designers should identify non-incremental learning algorithm such as ID3 algorithm.

4.3.4.3 Learning by Analogy

Intelligent Agent with *learning strategy* attribute equals to "300" is achieving learning by analogy which is a complex strategy that requires the Intelligent Agent to be able to conduct inductive and deductive inferences. Intelligent Agent using learning from analogy can mimic the human-like features of learning. As learning by analogy is a knowledge-intensive-learning mechanism and depends on background knowledge and experiences, Intelligent Agent should be capable of storing, indexing, and retrieving its experience to perform effective learning. *Leaning strategy* attribute has value of "310" and exhibits transformational analogy capabilities. Designers should be able to identify appropriate learning algorithm for the intelligent Agent, such as case-based reasoning.

---

[12] Candidate-Elimination algorithm "finds all describable hypotheses that are consistent with the observed training examples" (Mitchell 1997,29)

Intelligent Agent with *learning strategy* attribute equals to "320" exhibits capability to learn by the derivational analogy method. Designers can identify the learning algorithm used by the Intelligent Agent that may use case-based learning that depends on its own experience. This entitles the Intelligent Agent to be able to memorise, index, and restore its own experience to be able to conduct learning.

4.3.4.4 Learning from Instructions

Intelligent Agent with a *learning strategy* attribute equals to "400" exhibit learning by instruction method. Designers should provide the needed resources to accomplish communication between the Intelligent Agent and Instructor, such as friendly-user interface or authorisation. They should also identify the learning algorithm that will transform instructions received to executable procedures. *Learning strategy* attribute of "410" indicates that the type of learning from instruction is uni-directional where Instructor provides instructions to the Intelligent Agent on a sequential basis. Designers should identify when these sequential instructions are available and should provide the Intelligent Agent with the proper conditions to receive the instructions and transform them into operational procedures. *Learning strategy* attribute equals to "420" indicates that Intelligent Agent learns through interactive learning from instruction. Designers should identify conditions where the instructor would provide his/her instructions; when the Intelligent Agent identifies that, it should ask for instruction or advice from instructor. Designers should also provide all the needed resources for conducting such communication, as the Intelligent Agent might be in a hazardous situations and might request the instruction or advice from the instructors.

4.3.4.5 Rote Learning

Intelligent Agent with *learning strategy* attribute equals "500" indicates that it has capabilities to learn by rote-learning strategy. Designers should identify the required

resources to store, index, and retrieve the provided learning knowledge. The *learning strategy* attribute of "510" indicates that Intelligent Agent conducts rote learning by being programmed type. Designers should identify when the Intelligent Agent will receive the program from the instructor, the method for the Intelligent Agent to receive the input data and transform it to operational actions. Intelligent Agent with *learning strategy* attribute of "520" identifies that the rote-learning type is accomplished through memorising. Designers should estimate the amount of input knowledge and provide the Intelligent Agent with the required capacity to store the input knowledge and efficient methods for indexing and restoring the stored knowledge.

The learning strategy attribute guides the designer to the type of the learning strategy the Intelligent Agent uses. Each learning strategy has different variants that require different algorithms to perform learning. In addition, some of the learning strategies are knowledge intensive processes that require the designer to provide efficient mechanisms for acquiring, indexing, and restoring background and newly acquired knowledge. Designers should also identify the method of communication between Intelligent Agent and the source of knowledge such as learning from example strategy to allow the success of the learning process. Failure to understand and provide the required resources for the learning strategy jeopardises the Intelligent Agent capability to learn and adapt to new situations.

4.3.5 Learning Feedback Attribute

Learning feedback attribute identifies the feedback method that the Intelligent Agent uses during learning. The learning feedback can be supervised learning, unsupervised learning, or reinforcement learning. There is a direct relation between the *learning strategy* attribute and the *learning feedback* attribute. When the *learning strategy* attribute is "1XX"- that is learning from observation and discovery, the *learning feedback*

attribute is "100"- that is unsupervised learning. When the *learning strategy* attribute is "2XX"- that is learning from examples, the learning feedback attribute is "200" – that is the supervised learning method. According to the *learning feedback* method, designers provide the Intelligent Agent with the required communication skills to perform the learning process. For supervised learning, Intelligent Agent should be capable to communicate with the source of knowledge, which is the tutor. Therefore, the designers should identify the method of communication established between the Intelligent Agent and the tutor. The tutor can be a user or another Intelligent Agent. If the tutor is a user, Intelligent Agent should have a user-friendly interface while if the tutor is another Agent, the Agent communication language can be implemented. The Intelligent Agent that has multi-type tutors should be able to communicate with any type. In that way the Intelligent Agent will be more complex. For unsupervised learning, the Intelligent Agent should be capable of monitoring environmental actions and reactions or users during the learning process.

Reinforcement learning method requires the Intelligent Agent to be able to communicate with users, other Intelligent Agents, or objects to receive the reward. Designers need to identify the main elements of reinforcement learning, such as policy, a reward function, value functions, and model of the environment. According to the *learning feedback* attribute, designers should identify how the Intelligent Agent will receive its rewards for its actions. *Learning feedback* attribute equals to "310" indicates that the Intelligent Agent should receive its reward immediately after each action. This method has considerable communication between the Intelligent Agent and its tutor. Designers should provide resources to accomplish such intensive communication method. *Learning feedback* attribute of "311" highlights that designers should identify the terminal state where the Intelligent Agent receives its reward. It also points out that required communication resources should be available at this state to allow communication between the Intelligent Agent and the tutor. Intelligent Agent with *learning feedback* attribute

"312" will receive its reward at the end of the learning process. This type is a less communicative load than the other previous feedback methods. Intelligent Agent can exploit its existence hypothesis, "320," or explore new opportunities, "330." For each state, the designer should guarantee the resources required for conducting the learning process.

Learning feedback attribute has a direct impact on the performance of the Intelligent Agent. Each type of feedback method has different requirements of the communication capabilities and learning algorithms. Designers should clearly understand the behaviour of Intelligent Agent, and identify the right learning algorithm. Designers will identify the supervised and unsupervised learning algorithms during the identification of the learning strategy attribute whether the type of the learning strategy is learning from observation and discovery or learning from examples. Reinforcement learning requires the designer to identify the main elements needed to conduct learning. This attribute can be renamed to reinforcement learning attribute, as the supervised and unsupervised feedback method can be identified from the learning strategy attribute; however, for classical classification of learning, it is better to keep the attribute with the same name.

4.3.6 Learning Location Attribute

Learning location attribute specifies the location(s) where the Intelligent Agent performs learning. Analysts specify one location for static Intelligent Agent and different locations for mobile Intelligent Agent. Designers should take into consideration the requirements and constraints at the learning location to allow the Intelligent Agent to conduct learning. Intelligent Agents that are using reinforcement learning and would like to perform an exploitation or exploration processes should be able to trigger the environment at the learning location. It will need to have access, authorisation and privileges to monitor the environment or user reactions... etc.

4.3.7 Learning Schedule Attribute

Learning schedule attribute provides information about when the Intelligent Agent conducts learning and the duration of the learning process. During the identified learning duration, designers should make sure that the Intelligent Agent would have all the required resources to accomplish learning. For multi-Agent environments, designers should construct a mechanism to reduce the effects of competition between Intelligent Agents to use available resources during learning, such as memory usage, network bandwidth… etc. Analysts should provide notes to the Agent class diagram about constraints for conducting learning during the identified learning schedule.

4.3.8 Knowledge Representation Attribute

Knowledge representation attribute provides the designer with the types of inputs representations. According to the representation type, designers develop the algorithm for manipulating such representation. Intelligent Agent can acquire inputs from single or multiple representations. Intelligent Agents that acquire input from multiple representations are more complex to develop. Analysts should provide a Note to describe the data input specification such as time and duration for the data acquiring process. There are variants for each type of representation. Analysts can add more codes to such attributes, indicating its type and requirements.

4.4 Conclusion

This chapter has introduced two scenarios to use FIPA modelling TC Agent class superstructure meta-model to model Intelligent Agents: the first scenario extends Agent class and the second one uses the available Agent Physical Classifier and has introduced a learning compartment to model Intelligent Agents. The second scenario is more appropriate as there will be no need to develop new Intelligent Agent structures. The

second scenario also allows for global usage of UML to model Intelligent-Agent oriented systems.

The learning compartment provides description about the learning behaviour of Intelligent Agents. This description will guide the designers and the developers during both the design and the development phases of the system. The description consists of attributes and notes. The attributes describe the main learning parameters, and the notes provide behaviour description or constraints for the attached attribute. The learning compartment contains the following attributes: *learning-goal, commitment-strategy, learning-strategy, learning-feedback method, knowledge-representation, background-knowledge, knowledge-representation, and learning-schedule.*

During the writing of the thesis, the only work done to use UML 2.0 to model structural components of Agents was from the Foundation for Intelligent Physical Agent (FIPA). The research builds upon, explores, and utilises this work and provides further development to model the structural components of learning behaviour of Intelligent Agents. Using the FIPA UML Agent Class superstructure meta-model would allow the development of a single modelling language to model Objects, Agents, and Intelligent Agents.

Chapter 5: Exploring UML 2.0 to Model the Learning Behaviour of Intelligent Agents

This chapter explores the capability of the behaviour components of the Unified Modelling Language (UML) version 2.0 to model the learning behaviour of Intelligent Agents. Section 5.1 introduces the work done in this chapter. Section 5.2 explores the capability of UML 2.0 activity diagrams and sequence diagrams to model learning strategies of an Intelligent Agent, namely learning from observation and discovery and learning from examples. The section proposes different scenarios of learning strategies with examples from real life applications. Section 5.3 evaluates if UML 2.0 state machine diagrams can model specific reinforcement learning algorithms, namely dynamic programming, Monte Carlo, and temporal difference algorithms. Section 5.4 provides conclusions of the result of the work done in this chapter.

5.1 Introduction

Unified Modelling Language (UML) version 2.0 is an upgraded release from UML 1.X especially behaviour diagrams: activity diagrams, use case diagrams, state machine diagrams, and interaction diagrams. Research conducted during the past five years for extending UML 1.X to model Agent-oriented systems encourages the Object Management Group (OMG) to include features in the new UML standard to be able to model dynamic behaviours such as concurrency, branching, and paralleling processes. This research uses UML activity diagrams and sequence diagrams to model learning behaviour. Activity diagrams show the workflow of the learning process while sequence diagrams highlight collaboration between Intelligent Agent and other entities during the learning process.

The research explores two types of learning strategies: learning from observation and discovery and learning from examples. Learning from Observation and discovery is

an appropriate learning strategy for Intelligent Agents. Intelligent Agents will mostly work in dynamic environments where examples for all environment states cannot be determined. However, Intelligent Agents can use a learning from examples strategy to enhance their skills if the rate of change in the environment is slower than the learning process, or if there is a capability of cloning Intelligent Agents, and replacing them after learning. The research provides different scenarios for each learning strategy case and examples of real-life applications of the discussed scenarios to visualise the use of UML 2.0.

The research also evaluates whether UML 2.0 behaviour diagrams, namely state machine diagrams , can model the learning feedback methods, namely learning reinforcement. It explores the capability of state machine diagrams to model specific types of dynamic programming algorithms, namely, iterative policy evaluation, policy evaluation, policy improvement, and value iteration algorithms. It also investigates if state machine diagrams are able to model Monte Carlo algorithms, namely policy evaluation and control algorithms. Moreover the research uses this type of diagram to model temporal difference (TD) algorithms, namely TD prediction, Sarsa, Q-learning, Sarsa (λ), and Q(λ) algorithms.

Reinforcement learning and unsupervised learning are the appropriate learning feedback methods for Intelligent Agents working in a dynamic environment. Learning from observation and discovery strategy uses the unsupervised learning feedback method while learning from examples uses the supervised learning feedback method.

5.2 Modelling Learning Strategies of Intelligent Agents

This section explores UML version 2.0 capabilities in modelling Intelligent Agents' learning strategies. The research concentrates on two learning strategies; namely, learning from observation and discovery and learning from examples. The research provides different scenarios for each learning strategy. UML 2.0 behaviour diagrams model the

scenario by illustrating UML 2.0 activity and sequence diagrams for each scenario. The research also highlights examples from real life internet applications for the proposed scenarios.

5.2.1 Learning from Observation and Discovery Strategy

Learning from observation and discovery is an appropriate learning strategy that Intelligent Agents can use especially in dynamic environments. For Intelligent Agents working in dynamic environments, it is very difficult to program the required behaviour for every status of the environment or action of users. Intelligent Agents should have the capability to update the current hypothesis by observing the actions and reactions of either the environment or users. Intelligent Agents should have the ability to trigger the environment or the users to discover their reactions, to confirm or negate the current hypothesis. Below are different modelling scenarios of learning from observation and discovery strategy.

5.2.1.1 Scenario 1.1:"Learning From Observation and Discovery, Passive Observation, Event action"

This scenario describes the Intelligent Agent that monitors the actions of the environment or users, and learns by passive observation strategy. The Intelligent Agent starts the learning process as soon as the environment sensors detect an action. Figure 5.1 shows the UML activity diagram of this scenario with the parameters of the learning strategy at the figure title. The following steps describe the UML activity diagram of this scenario:

1. The activity diagram contains two swim lanes: environment sensors and Intelligent Agent (iAgent). Each swim lane contains actions and activities of the owner of the mentioned swim lanes.

2. The Intelligent Agent records the received actions of the environment sensors with which it observes the environment status.

    a. If the Agent decides to continue monitoring the actions of the environment, the control of flow moves to the *event-arrive* action (Italic format represents UML activity titles) in the environment sensors' swim lane.

    b. If the Agent decides to stop observation, flow of control moves towards the *cluster-actions* activity.

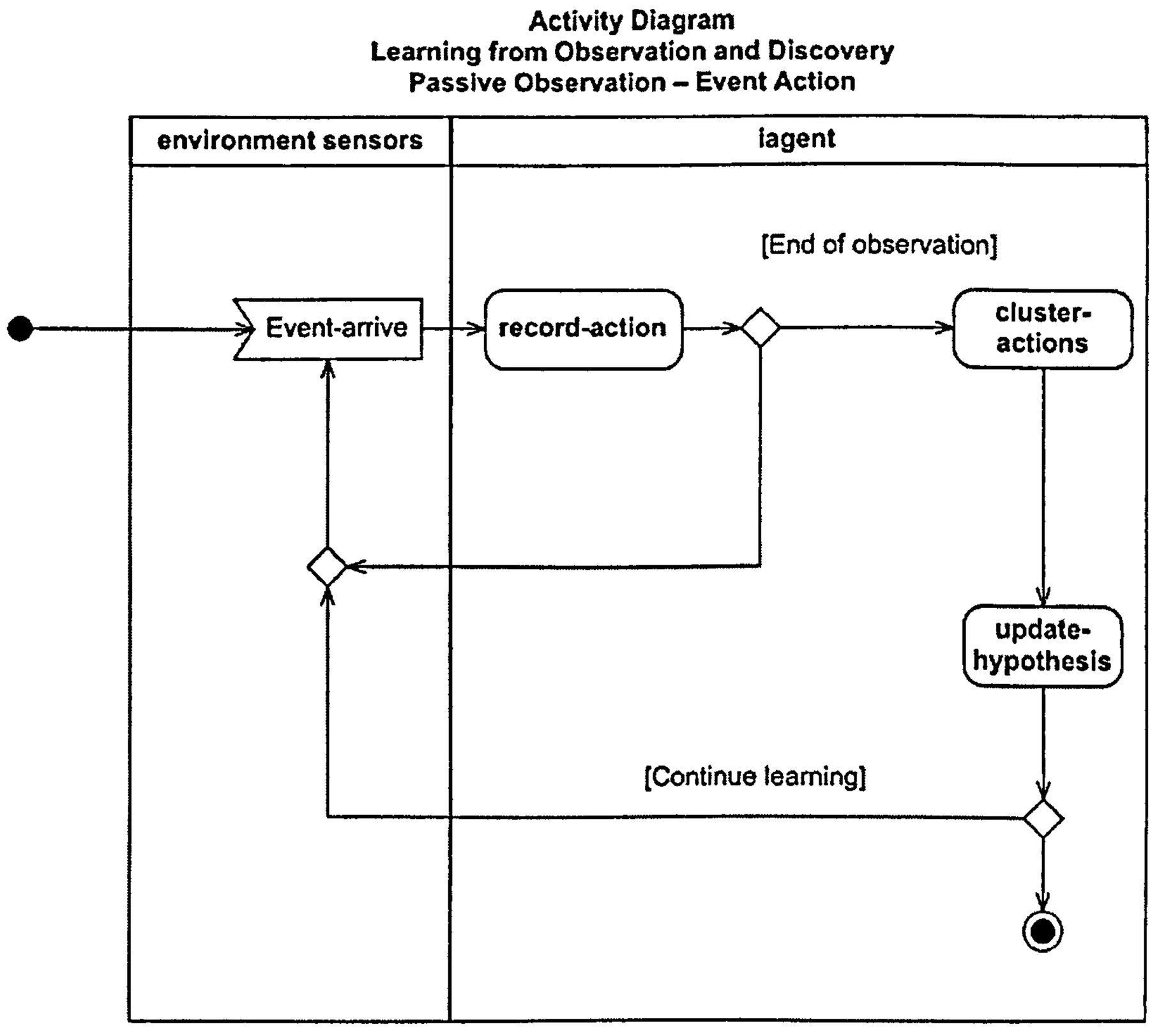


Figure 5.1 Activity diagram of scenario 1.1

3. After classifying the recorded actions, the Intelligent Agent updates the current hypothesis.

a. If the Intelligent Agent wishes to continue learning, the flow of control will point to the *event-arrive* action state.

b. If the Intelligent Agent decides to finalise learning, the flow of control will point to the end of flow action state.

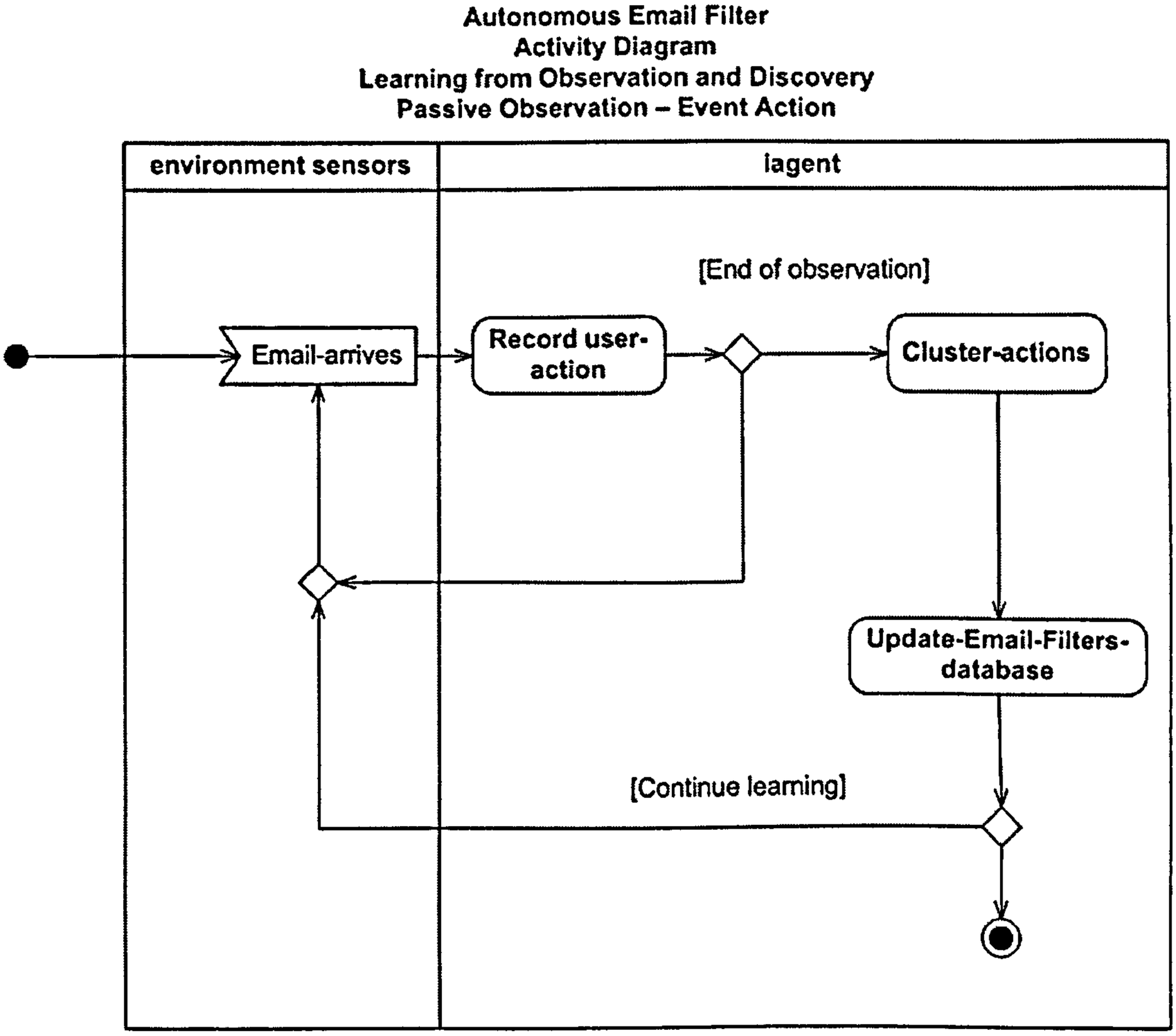


Figure 5.2 Activity diagram of email filter Intelligent Agent

An example of this scenario is an Intelligent Agent that filters users' email messages. The Intelligent Agent assists the user to sort the received email messages and move each email message to its relevant folder. Through the observation process, the Intelligent Agent starts to mimic the same actions the user conducts after a specific threshold count of similar actions repeated a number of times. For example, the Intelligent Agent observes that the user deletes without reading an email message. If the user deleted an email five times without reading from the same sender, the Intelligent Agent

automatically updates the current hypothesis to allow the deletion of any new email message received from that email address. Such a technique is very useful in combating email spamming and managing junk emails. Figure 5.2 shows an activity diagram for an Intelligent Agent that is acting as an email filter.

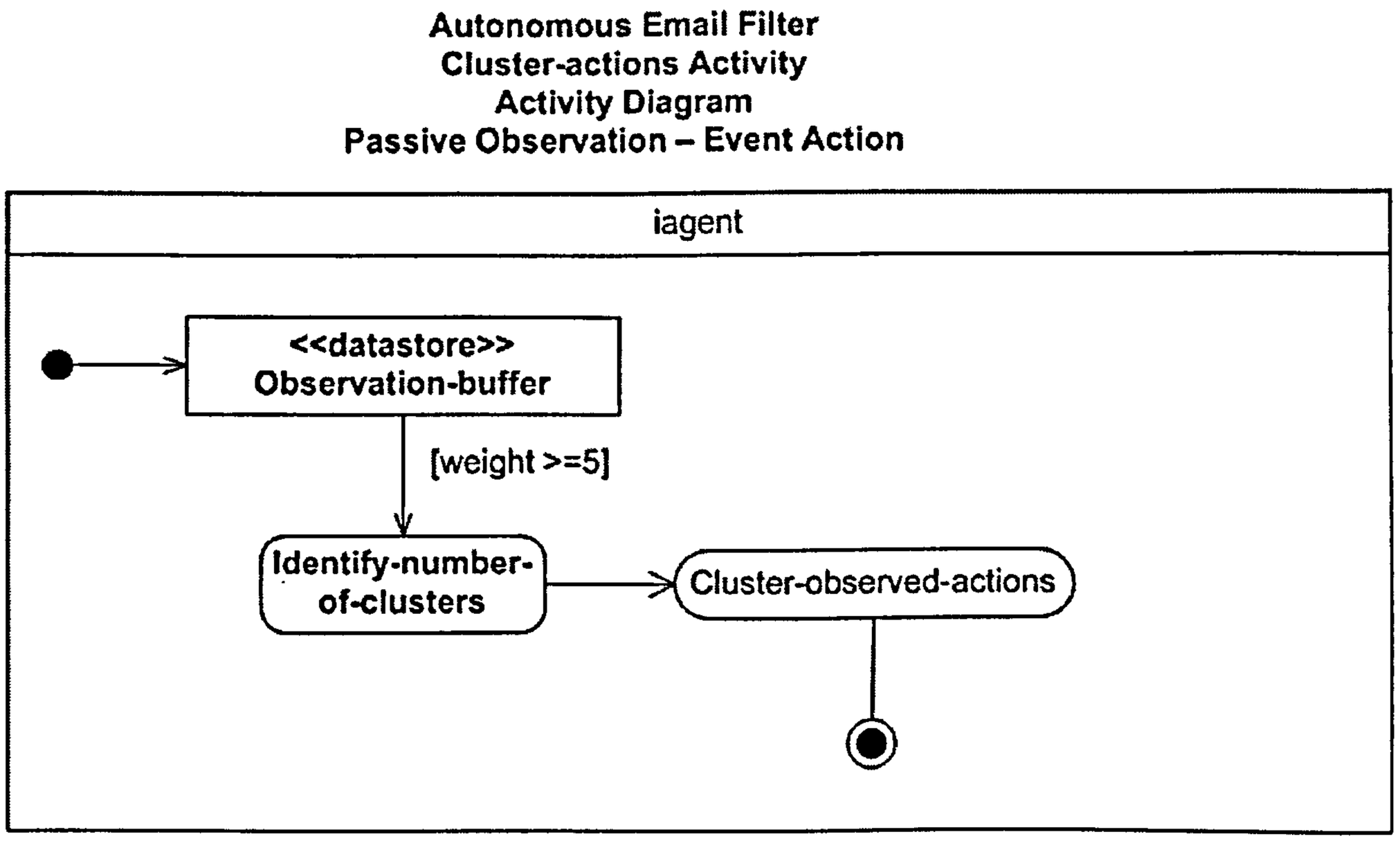


Figure 5.3 cluster-actions activity diagram

UML 2.0 activity diagrams show actions or activities of a process. Activity is a combined action abstract to allow diagrammatic visualisation. Figure 5.3 shows the activity diagram of the decomposition for the *cluster-actions* activity. The following are the steps taken by the Intelligent Agent to cluster the recorded actions:

1. Retrieve the actions that the user has repeated five times or more from the same email address from the *observation_ buffer* data-store node, where the Agent has recorded observations.
2. Identify the number of clusters according to the number of actions.

3. Assign each email address according to the email taken by the user to its relevant cluster. Different conceptual clustering algorithms can be used for this activity
4. The flow of control points to the end of flow action state.

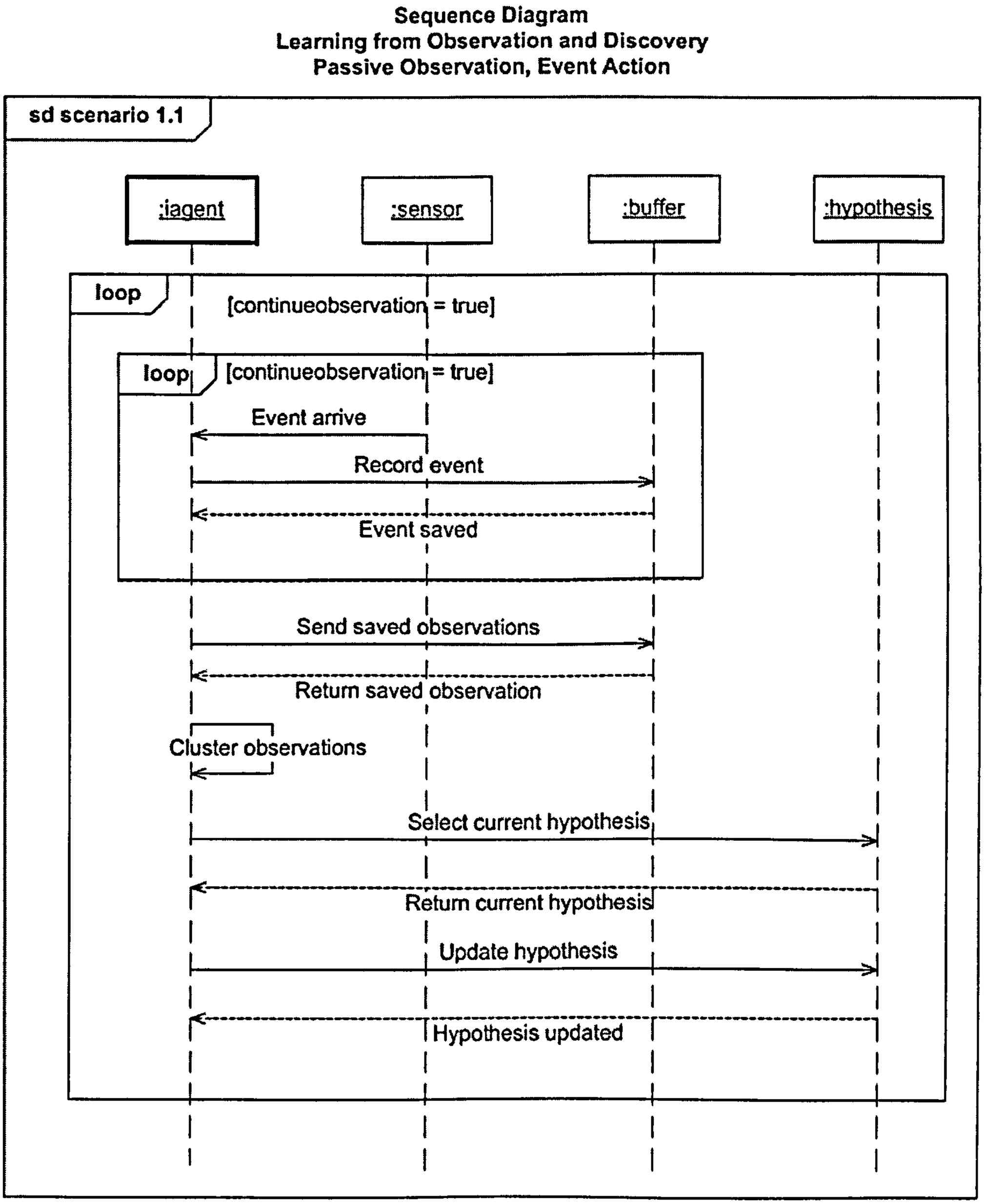


Figure 5.4 Sequence diagram of scenario 1.1

The UML sequence diagram identifies four entities for scenario 1.1: Intelligent Agent, environment sensor, buffer, and hypothesis. The notation of the Intelligent Agent is as proposed in chapter 4. The interaction fragment loop indicates that the Intelligent Agent conducts learning, until it decides to stop learning. This shows the autonomous behaviour of the Intelligent Agent. Figure 5.4 shows the sequence of messages between the Intelligent Agent and the other entities during the learning process.

5.2.1.2 Scenario 1.2: "Learning from Observation and Discovery, Passive Observation, Time Event"

This scenario describes an Intelligent Agent that uses learning from observation learning strategy. The Intelligent Agent is a passive learner as in the previous scenario. The Intelligent Agent starts to learn actions of the environment or users according to specific time or duration rather than action received, as in the previous scenario. The following are the steps that describe the UML activity diagram of this scenario:

1. The activity diagram contains two swim lanes: environment sensors and Intelligent Agent. Each swim lane contains actions and activities of the owner of the above mentioned swim lanes
2. The Intelligent Agent starts to record the actions of the environment or users when a time signal arrives. The Intelligent Agent observes the environment status through environment sensors.
    a. If the Intelligent Agent decides to continue monitoring events, the control of flow moves to the *time-event* action in the environment sensor's swim lane.
    b. If the Intelligent Agent decides to stop monitoring events, object flow moves towards the *cluster-observations* activity. Recorded actions will be delivered to the *cluster-observations* activity
3. After clustering actions, the object flow transfers the clustered actions to the *update-hypothesis* activity. The Intelligent Agent updates the current hypothesis.

a. If the Intelligent Agent wants to resume learning, the flow of control will point to the *time-event* action state.

b. If the Intelligent Agent decides to finalise learning, the flow of control will point to the end of flow action state.

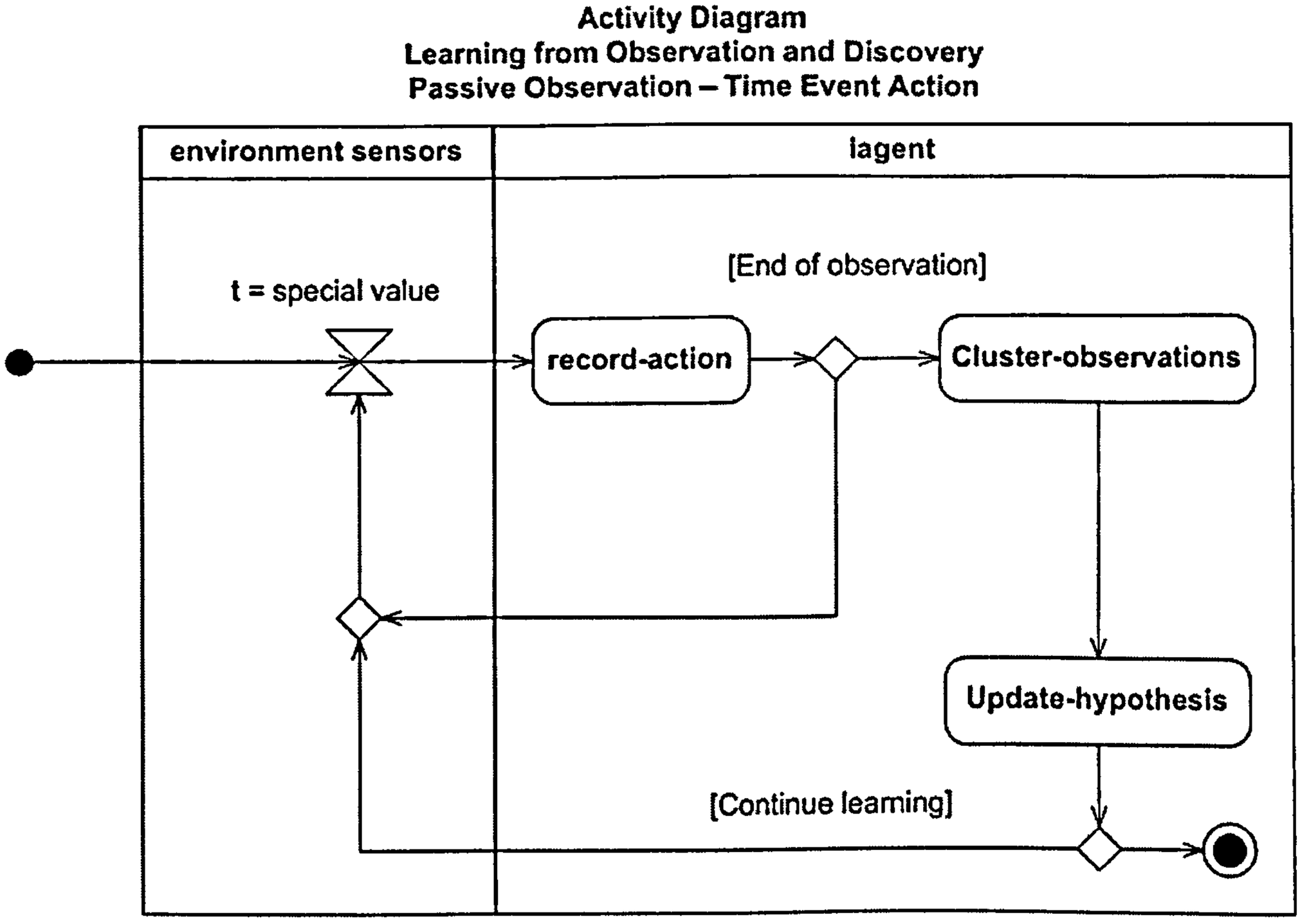


Figure 5.5 Activity diagram of scenario 1.2

An example of such a scenario is an Intelligent Agent that learns the internet browsing behaviour of enterprise employees. The Intelligent Agent starts to identify a schedule for the browsing habits of the employees. The Intelligent Agent monitors if there are some websites visited regularly during a specific duration, and classifies them according to their access time. The Intelligent Agent browses the identified websites and updates the cache of the enterprise internet server before the identified time. Such process assists in the efficient use of the enterprise internet bandwidth. The Intelligent Agent learns the enterprise employees browsing habits from observing the visited websites; any change in any previous learned schedule will require the Intelligent Agent to update the

hypothesis with a new browsing schedule. Figure 5.6 shows the activity diagram of the enterprise Internet server.

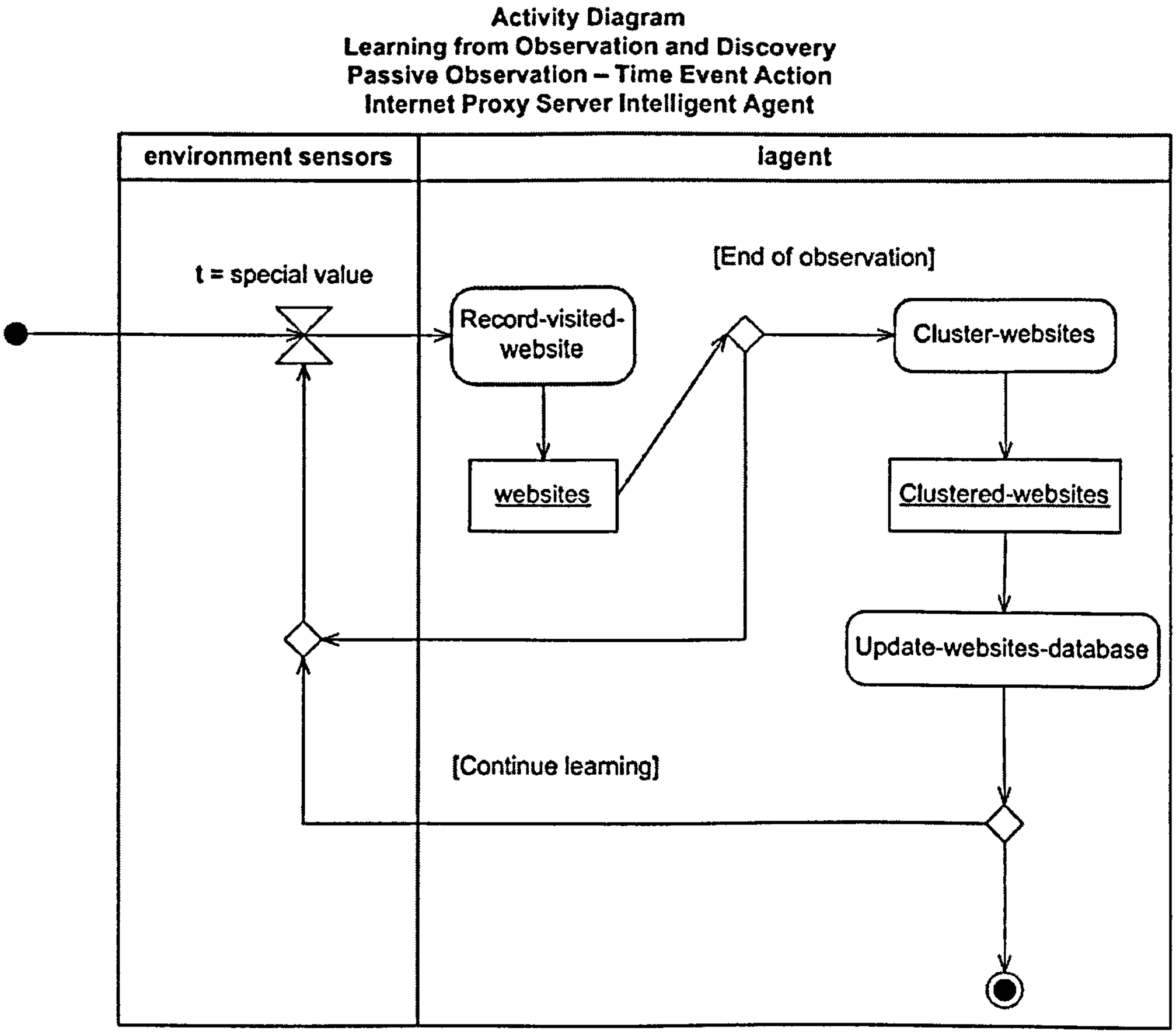


Figure 5.6 Activity diagram of internet proxy server Intelligent Agent

UML Sequence diagram of scenario 1.2 (figure 5.7) shows the time constraint T where the Intelligent Agent starts to record action when the time constraint parameter is equal to T. This sequence diagram shows the sequence of messages between the Intelligent Agent and other entities in the system. The Intelligent Agent starts learning when the time variable t equals T.

Sequence Diagram
Learning from Observation and Discovery
Passive Observation, Time Event Action

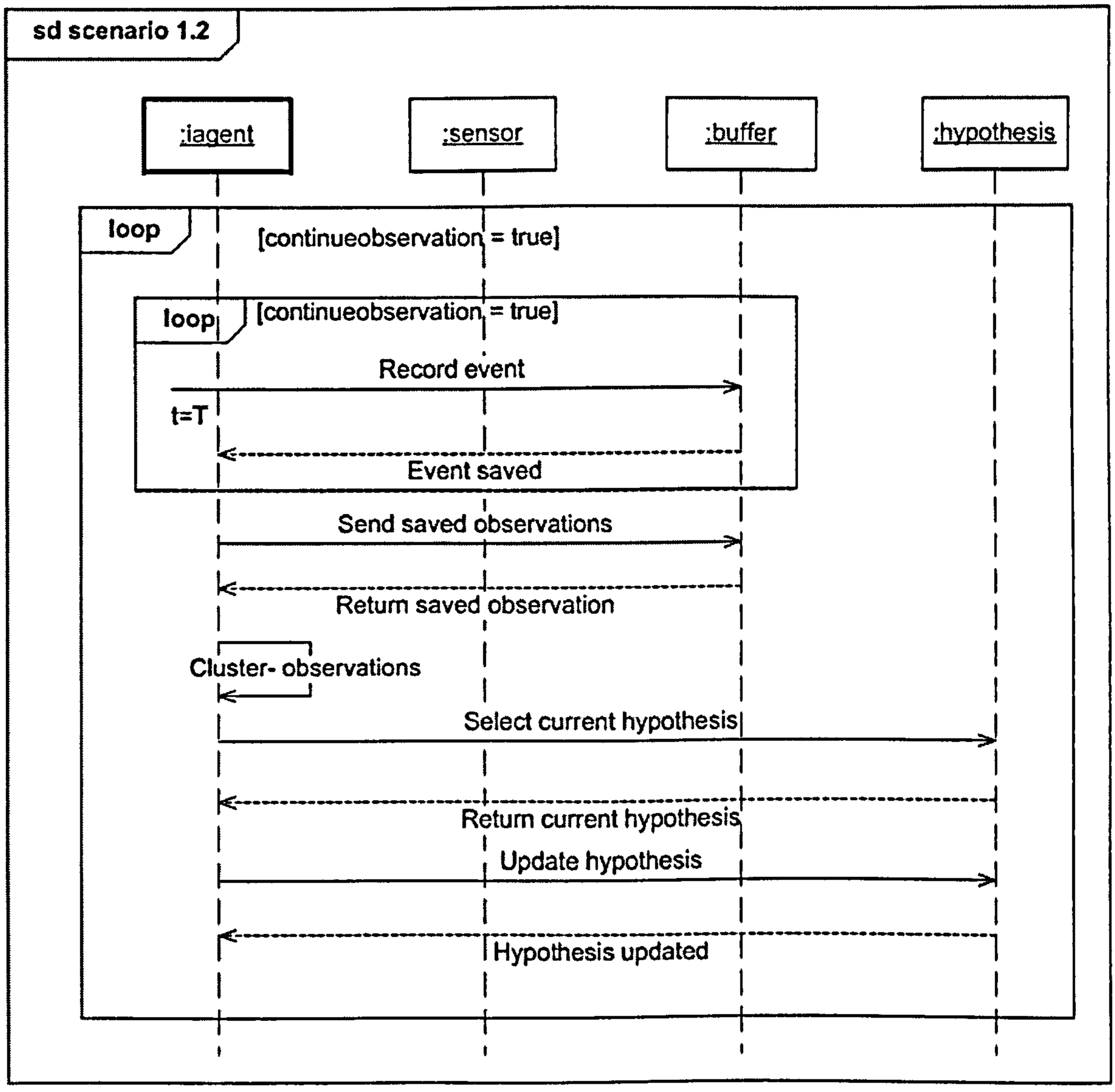


Figure 5.7 UML Sequence diagram for scenario 1.2

5.2.1.3 Scenario 1.3: "Learning from Observation and Discovery, Passive Observation, Time or Action event"

This scenario integrates the above two scenarios (figure 5.8). The Intelligent Agent starts to learn actions of the environment or user at a specific time or action. The Intelligent Agent is also a passive learner during this scenario. This scenario integrates event action and time event nodes to allow the Intelligent Agent to start learning as soon as one of the nodes is active. The following steps describe the UML activity diagram:

1. The activity diagram contains two swim lanes: environment sensors and Intelligent Agent. Each swim lane contains the actions and activities of its owner.
2. The Intelligent Agent monitors and records time events and actions received by the environment sensors.
    a. If the Intelligent Agent decides to continue monitoring events, the control of flow is directed towards the monitoring actions: time events and action events.
    b. If the Intelligent Agent decides to stop monitoring events, object flow points towards the *cluster-observations* activity. Recorded actions will be delivered to the *cluster-observations* activity
2. After clustering the recorded actions, object flow delivers clustered actions to the *update-hypothesis* activity.
3. The Intelligent Agent updates the current hypothesis.
    a. If the Intelligent Agent wants to resume learning, the flow of control will point to the monitoring actions
    b. If the Intelligent Agent decides to finalise learning, the flow of control will point to the end of flow action state

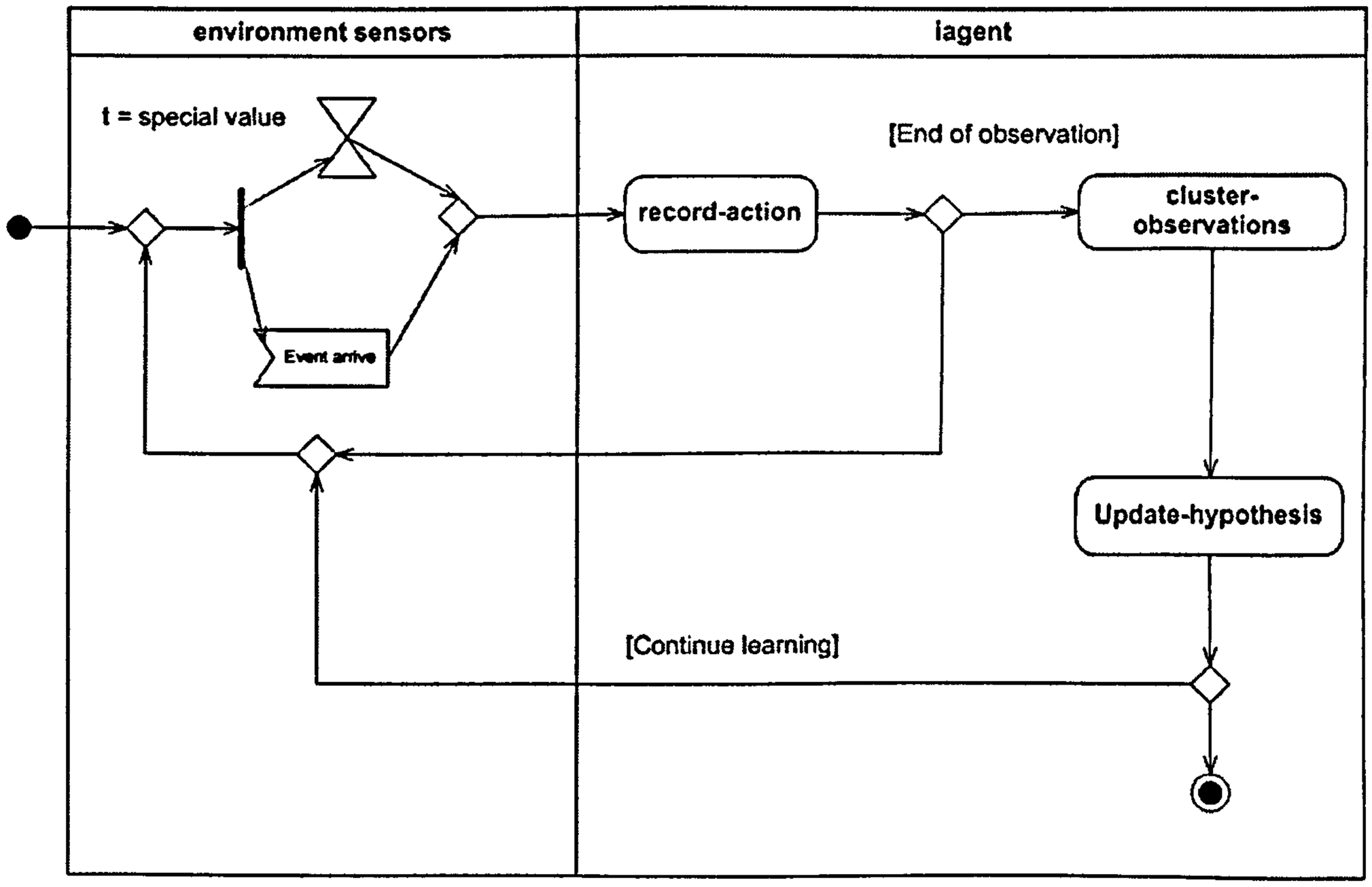


Figure 5.8 UML activity diagram for Scenario 1.3

An example of such a scenario is the previous Intelligent Agent example of scenario 1.2 where the Intelligent Agent not only monitors time and clusters the visited website according to the time users who have visited this website, but also monitors the updating schedule of the selected websites. Thus, the Intelligent Agent updates the enterprise internet cache with new versions of the selected websites according to their updating rate. This allows the Intelligent Agent to access the internet according to the updating schedule and not to congest the enterprise internet bandwidth with browsing the selected websites regularly. Figure 5.9 illustrates the activity diagram of internet proxy Intelligent Agent during learning.

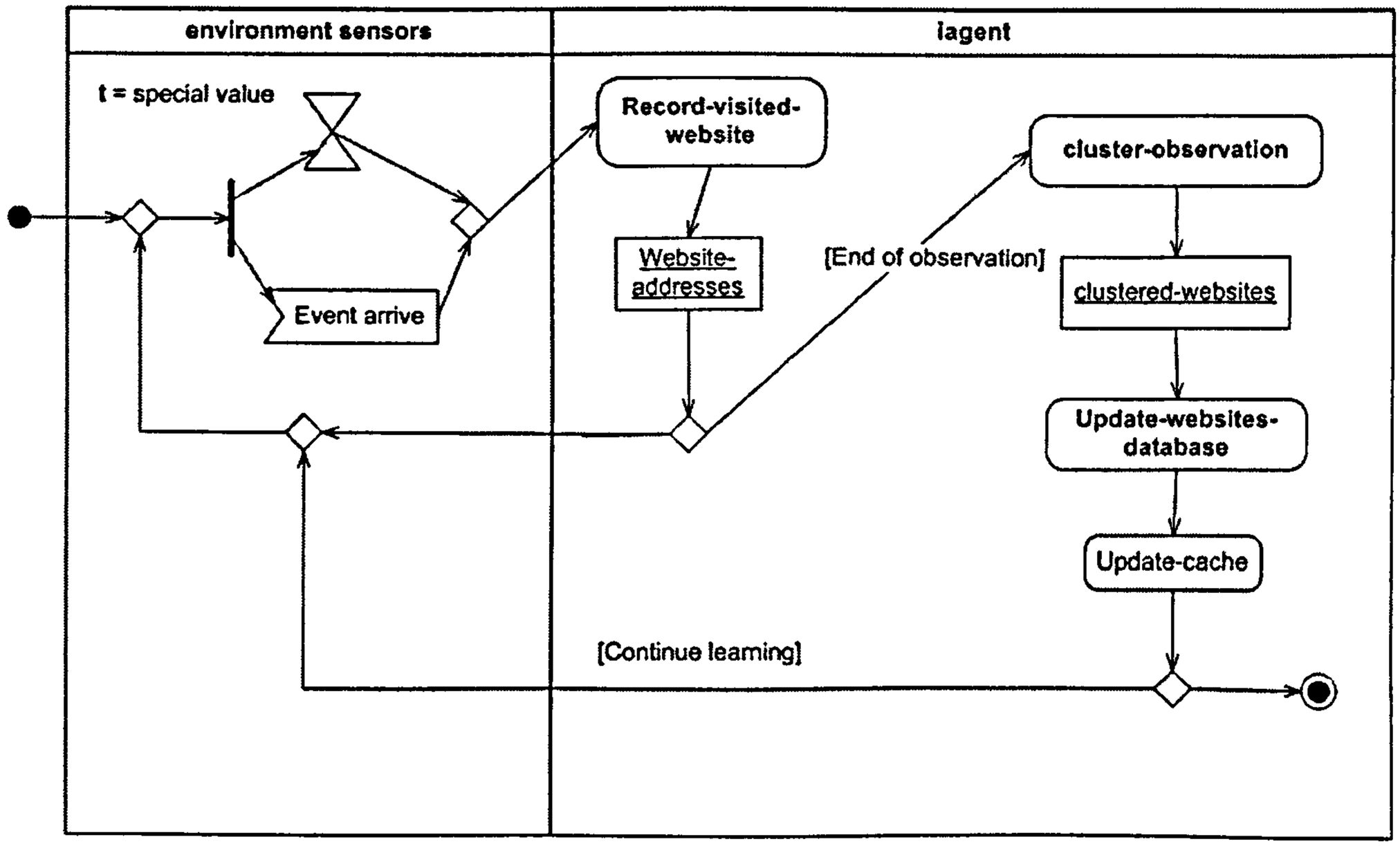


Figure 5.9 UML activity diagram for Internet Proxy Server

The UML sequence diagram in Figure 5.10 shows the combination of looping and parallel notation to describe the behaviour of an Intelligent Agent. The sequence diagram depicts how that Intelligent Agent communicates with three entities: environment sensor, buffer, and hypothesis. The sequence diagram demonstrates the sequence of message and requests between the Intelligent Agent and other entities during the learning process. Two loops and one parallel interaction fragment control the learning process. This sequence diagram is more complex than the previous two diagrams.

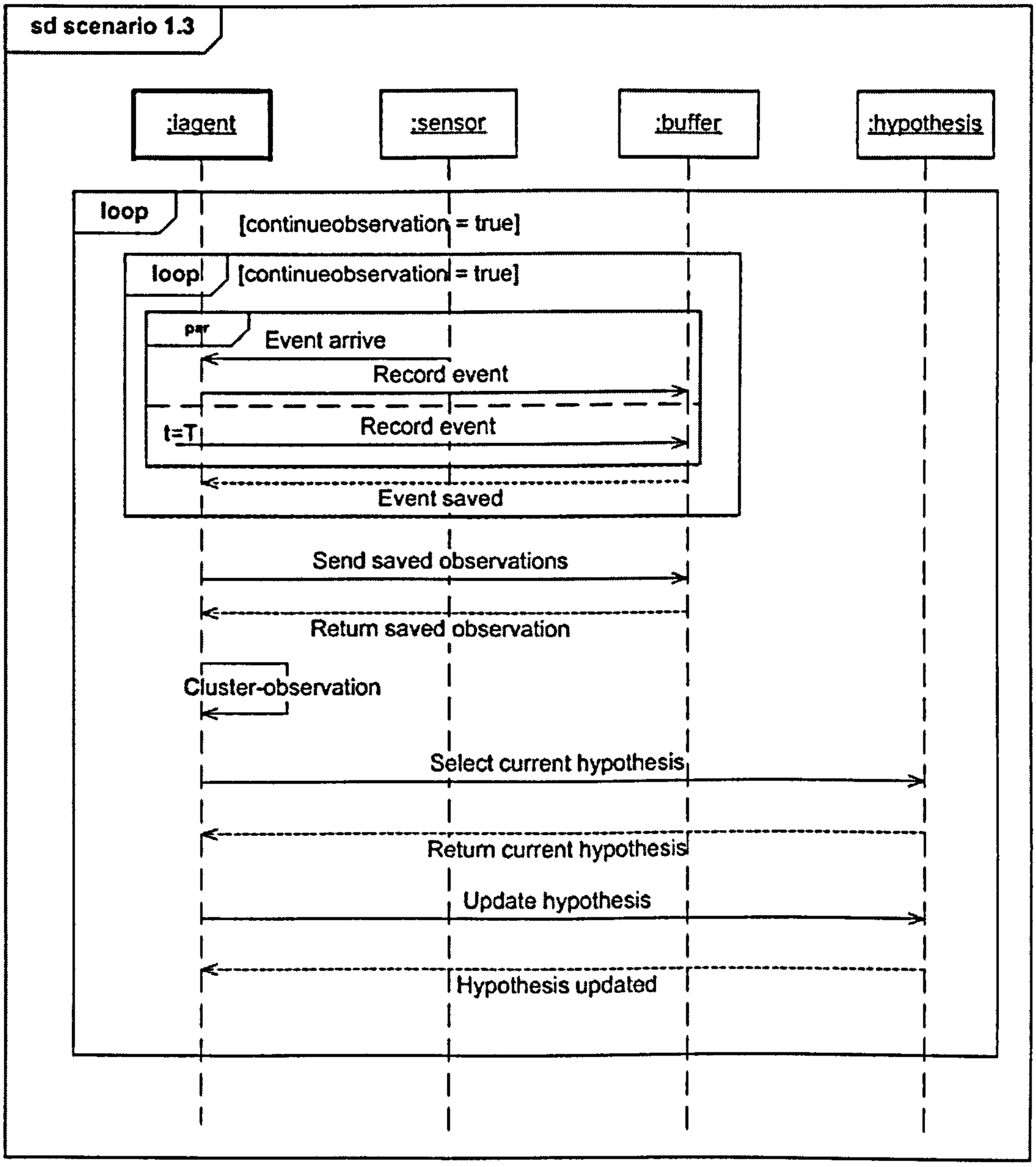


Figure 5.10 UML sequence diagram for scenario 1.3

5.2.1.4 Scenario 1.4: "Learning From Observation and Discovery, Active Experimentation, Event Action"

Unlike the previous scenarios, this scenario describes Intelligent Agents that act as an active learner. The Intelligent Agent learns the environment or user reactions after issuing a triggering action. According to the observed reaction , the Intelligent Agent updates its hypothesis by confirming or negating an entry in the hypothesis.

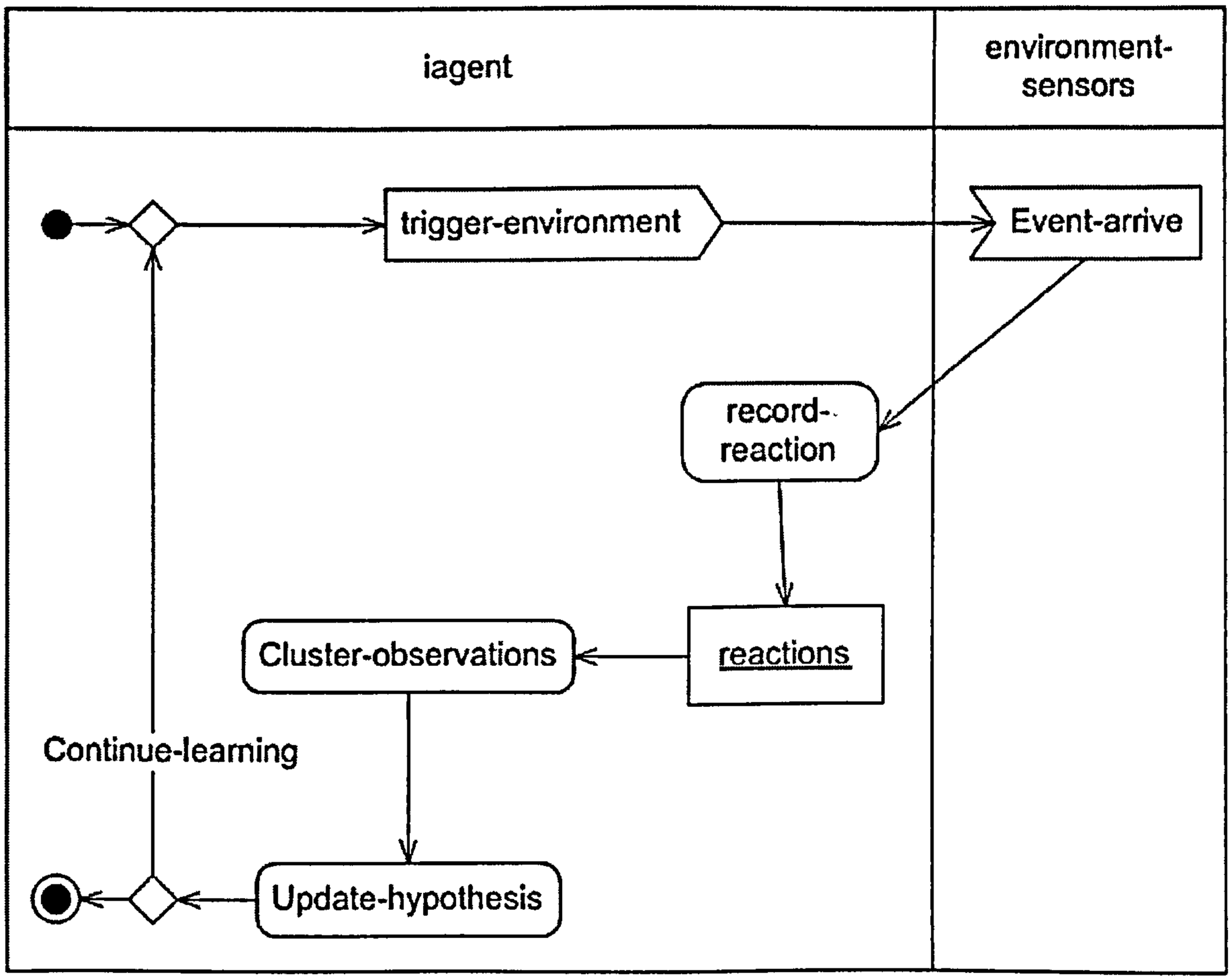


Figure 5.11 UML activity diagram for Scenario 1.4

Figure 5.11 shows a UML activity diagram for scenario 1.4. This activity diagram introduces the node by sending a signal action to trigger the environment or a user. The following steps describe the UML activity diagram:

1. The activity diagram contains two swim lanes: environment sensors and Intelligent Agent. Each swim lane contains actions and activities of the owner of the said swim lanes.
2. The Intelligent Agent issues a triggering action to the environment.

3. The Intelligent Agent monitors the reaction of the environment and as soon as the Intelligent Agent observes reaction, it records the reaction and passes the reaction to the *cluster-observations* activity.
4. The Intelligent Agent classifies the reaction and passes the classified reaction to the *update-hypothesis* activity.
5. The Intelligent Agent updates the current hypothesis by confirming or negating an entry in the Intelligent Agent's hypothesis.
    a. If the Intelligent Agent decides to continue learning, the flow of control points to the merge node to provide another trigger action.
    b. If the Intelligent Agent decides to finalise learning, the flow of control points to the end of flow action state.

Figure 5.12 shows UML sequence diagram for scenario 1.4. The diagram illustrates how the Intelligent Agent sends a message to trigger the environment through a gate to an entity outside the sequence diagram. The Intelligent Agent receives the reaction to the triggered message through the environment sensor.

An example of such a scenario is 'enterprise Intelligent Agent search assistance' When an employee searches through the enterprise search engine, the search engine might return the results in multi-pages due to the large number of returned links. Users browse the upper entries in the returned list, and do not give much attention to results at the bottom of the list. The search engine sorts the entries in the returned list according to the rank of the link. The rank of links depends on the number of times employees have browsed the link with the same or similar queries.

The Intelligent Agent would like to discover whether the returned links at the bottom of the returned list might have higher rank or not. The Intelligent Agent selects some of the lowest ranking returned links and lists them at the top section of the returned list when an employee executes a similar query. The Intelligent Agent observes the

reactions of the employees and learns if they browse the link or not. If the number of employees browsing the link exceeds a specific threshold, the rank of the link is increased; otherwise, the link keeps its low rank. The Intelligent Agent search assistance would assist the employees to share their experience of searching and browsing with each other and would allow better search results as each enterprise or group has common interests.

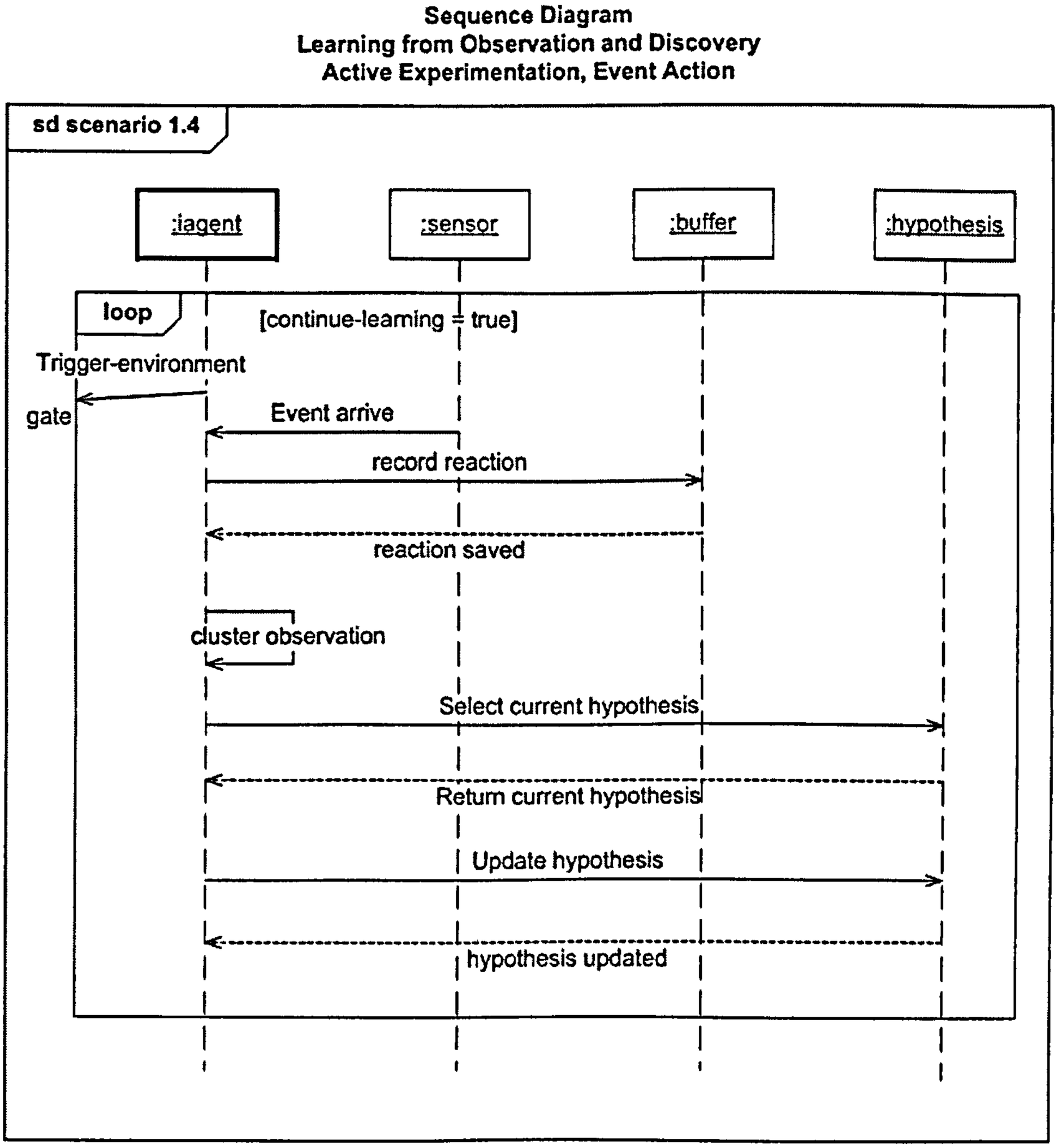


Figure 5.12 UML Sequence diagram for scenario 1.4

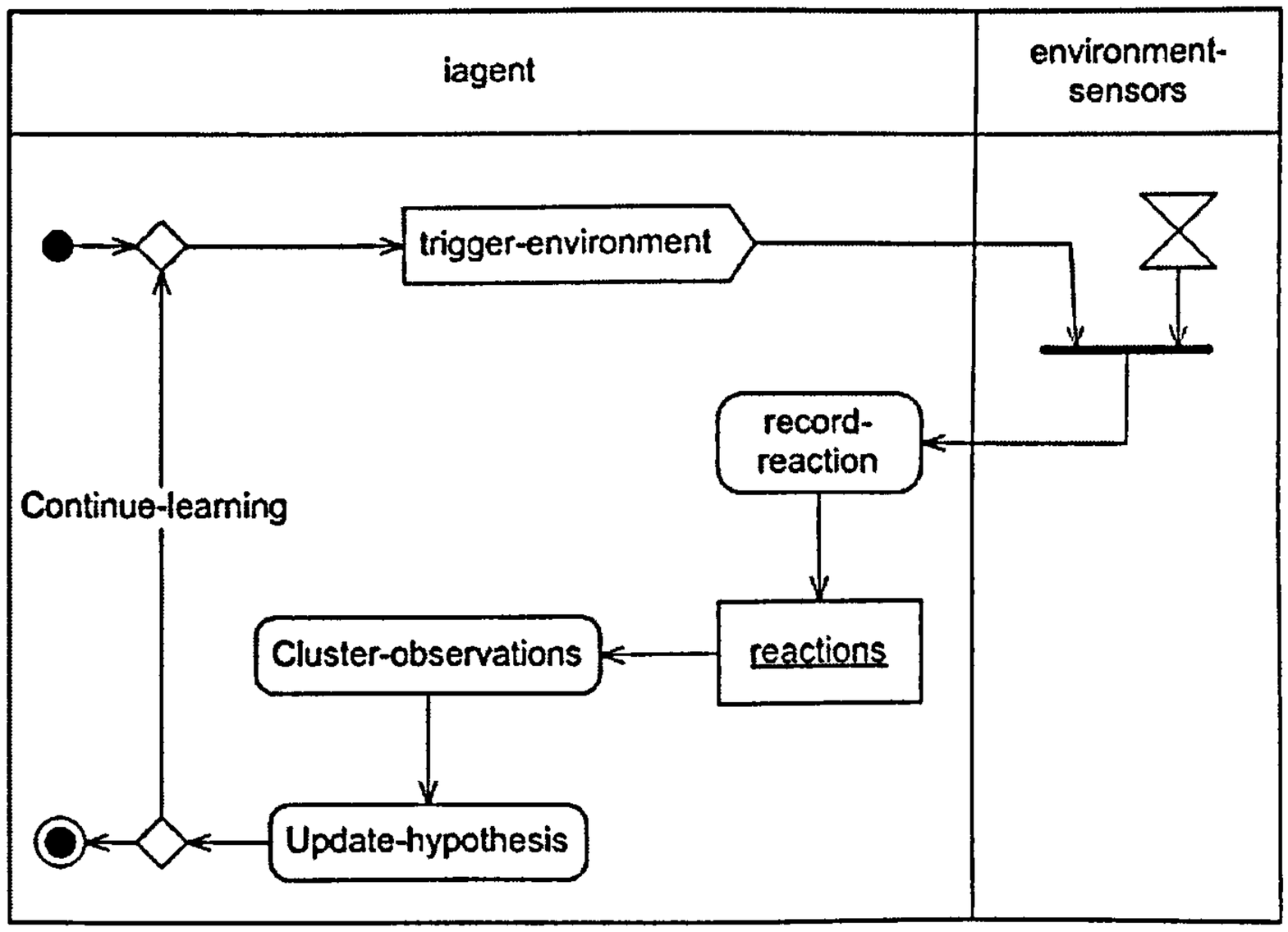


Figure 5.13 UML activity diagram for Scenario 1.5

5.2.1.5 Scenario 1.5: "Learning from Observation and Discovery, Active Experimentation, Time reaction"

This scenario describes an Intelligent Agent that issues a trigger action to the environment or a user, and monitors the reaction related to time. According to the observed action, the Intelligent Agent updates the current hypothesis by confirming or negating an entry in the hypothesis. The UML activity diagram is similar to the previous scenario but only differs in having a node of the type 'Accept time event action' rather than the 'Accept event action' node. The following steps describe the UML activity diagram:

1. The activity diagram contains two swim lanes: environment sensors and Intelligent Agent. Each swim lane contains actions and activities of the owner of the abovementioned swim lanes.
2. The Intelligent Agent issues a triggering action to the environment or the user.

3. The Intelligent Agent monitors the reaction of the environment over a specific duration, and records the received reaction.
4. The Intelligent Agent clusters the observed reaction, and passes the observed clustered reactions to the *update-hypothesis* activity.
5. The Intelligent Agent updates its hypothesis by confirming or negating an entry.
    a. If the Intelligent Agent wants to continue learning, the flow of control will point to the merge node for providing another triggering action.
    b. If the Intelligent Agent decides to finalise learning , the flow of control will point to the end of flow action state

The Enterprise Internet Search Intelligent Agent is an example of such a scenario where the Intelligent Agent tests the response time for searching through specific web servers. The Intelligent Agent sends a query for different web servers and records the response time. According to the response time, the Intelligent Agent updates its mechanism by sending a query to slow-response servers before fast-response servers. This allows the Intelligent Agent to acquire the results of the query from all servers over an acceptable duration. Such a learning mechanism will enhance the performance of the Intelligent Agent in terms of time and quality.

Figure 5.14 shows UML sequence diagram for scenario 1.5. The diagram shows two-loop interaction fragment frames. The first one continues to execute until the Intelligent Agent decides to stop observation and learning. The second loop stops when the t operand is equal to T, where T represents a specific time. The rest of the sequence diagram is similar to the sequence diagram of the previous scenario.

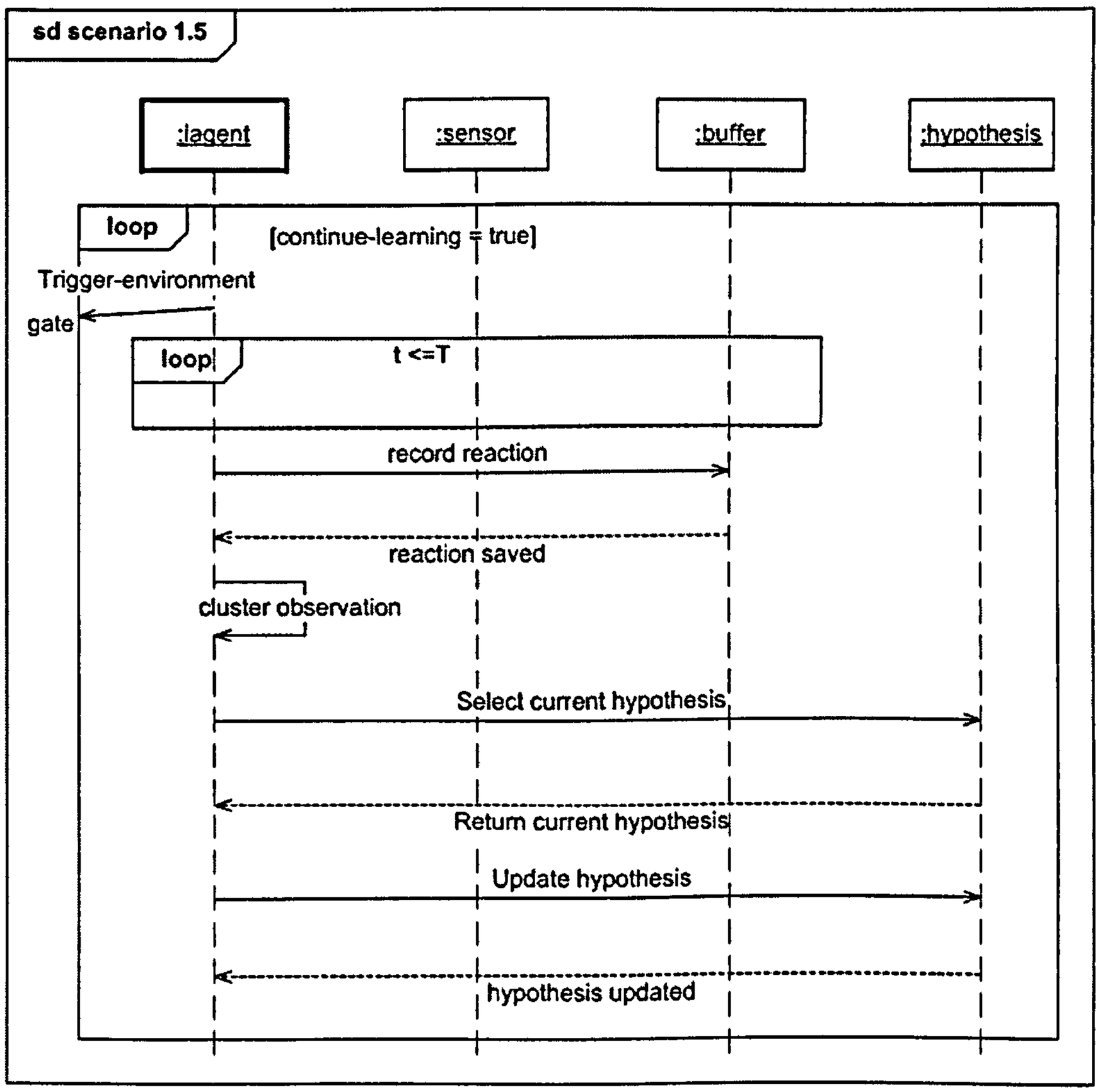


Figure 5.14 UML sequence diagram for scenario 1.5

5.2.2 Learning from Examples Strategy

Learning from example strategy is not adequate for Intelligent Agents that are working in dynamic environments. It is very difficult to provide examples for all situations that Intelligent Agents will deal with. Intelligent Agents can use the learning from examples strategy, if the rate of change in the environment is slow, or if there is a process of cloning the Intelligent Agent and replacing it after learning. The objective of using learning from examples is to enhance an Intelligent Agents' skills. Below are some scenarios for using UML version 2.0 behaviour diagrams to model learning from example strategy.

5.2.2.1 Scenario 2.1: "Learning from Examples, Non-Incremental Learning, Use Background Knowledge, Retrieve all Examples, Classify"

This scenario describes an Intelligent Agent that uses learning from examples as a learning strategy. The Intelligent Agent learns through non-incremental learning, and uses available background knowledge in classifying the available examples. After receiving all the available examples, the Intelligent Agent starts the classification process. The Intelligent Agent updates the current hypothesis with the new classified examples. The UML 2.0 activity diagram for scenario 2.1 has two swim lanes, one for the Intelligent Agent and the other for the tutor. The tutor can be a user, another Intelligent Agent, or any source of knowledge that can provide examples. Figure 5.15 shows the activity diagram for scenario 2.1. The following steps describe the scenario:

1. The Intelligent Agent requests the tutor to send an example.
    a) If there is an available example, the tutor sends the requested example, and the control of flow points towards 'receive example activity' (Step 3)
    b) If there is no available example, the tutor returns end of example flag and the control of flow point toward the join node (step 4).
2. The Intelligent Agent conducts parallel actions shown by the fork node:
    a) The Intelligent Agent stores the received example in a buffer;
    b) The Intelligent Agent requests another example from the tutor.
3. When the end of an example flag is true, the Join node transfers the stored examples in the buffer with a relevant hypothesis to the *classify_examples* activity.
4. After classifying the received examples, the flow of control points towards *update_hypothesis* activity.
5. After updating the hypothesis, the control of flow terminates at End of flow node.

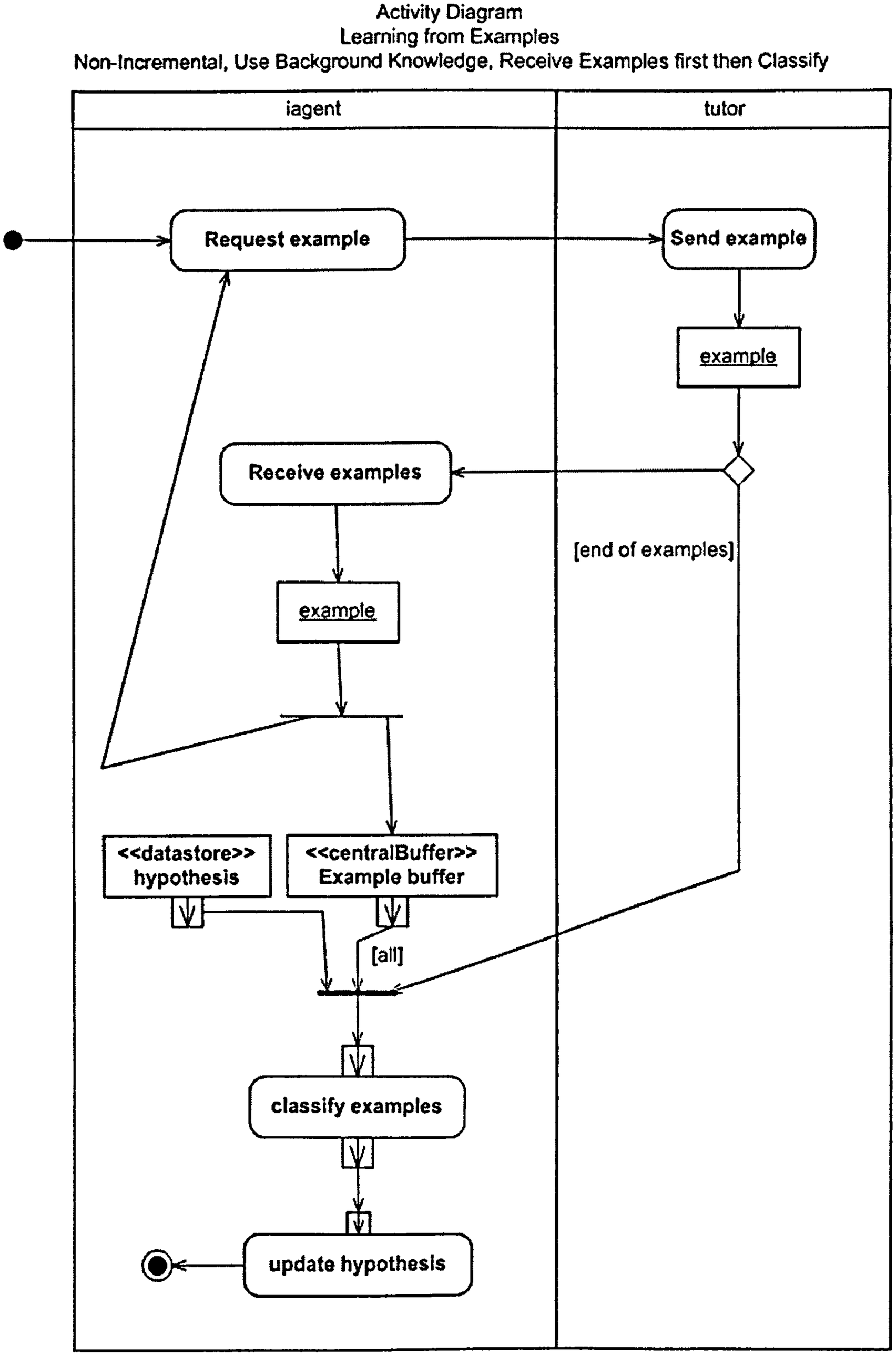


Figure 5.15 UML activity diagram for scenario 2.1

Figure 5.16 illustrates UML sequence diagram for this scenario. The Intelligent Agent communicates with three entities to conduct learning from examples: tutor, buffer, and hypothesis.

UML 2.0 introduces the central-buffer[13] and data-store[14] object nodes; buffer and hypothesis are central-buffer and data-store object nodes respectively. The loop fragment shows that the Intelligent Agent conducts learning from examples until there are no more available examples.

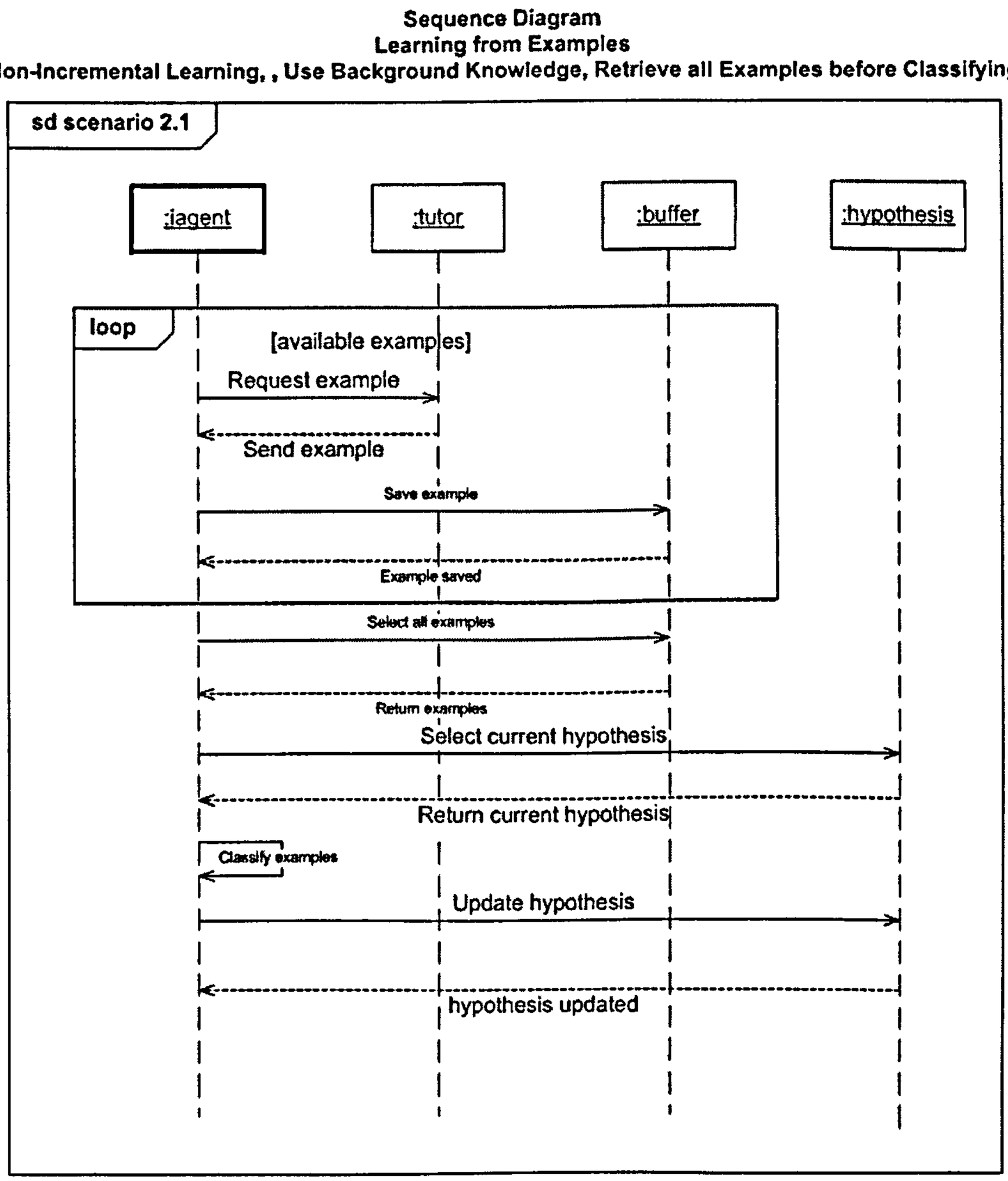


Figure 5.16 UML sequence diagram for scenario 2.1

---

[13] Central buffer node is an "object node for managing flows from multiple sources and destination" (OMG, 2003)

[14] Data store node is "a central buffer node for non-transit information" (OMG, 2003).

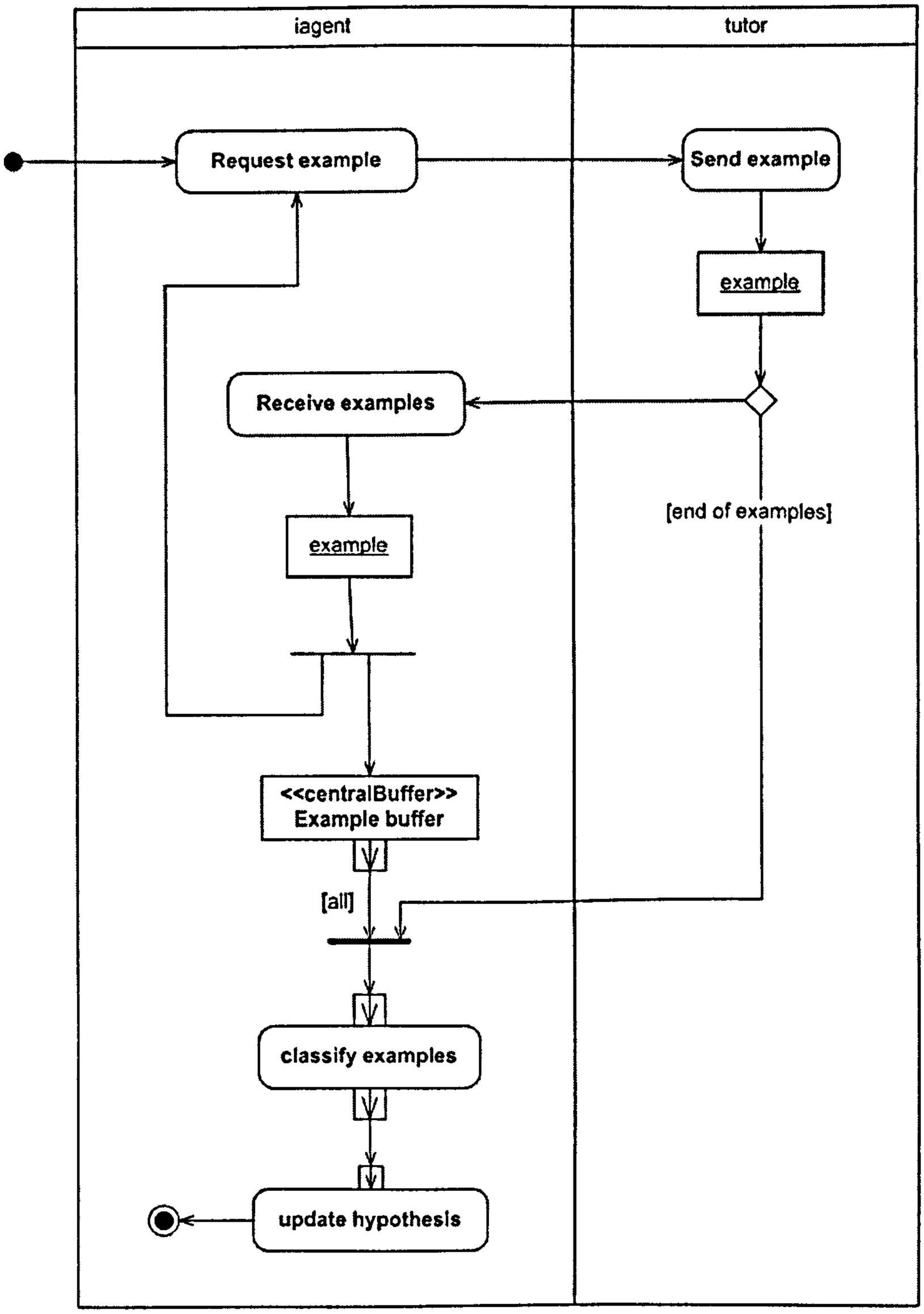


Figure 5.17 UML activity diagram of scenario 2.2

5.2.2.2 Scenario 2.2: "Learning from Examples, Non-Incremental, Does not Use Background Knowledge, Receive Examples, Classify"

The difference between this scenario and the previous one lies in the use of background knowledge in classifying examples. In this scenario, the Intelligent Agent

does not use background knowledge in the classifying examples. Figure 5.17 shows that examples stored in the buffer are the only input to the "classifying examples activity". As in the previous scenario, the activity diagram has two swim lanes: one for the Intelligent Agent and another for the tutor. The tutor can be any source of knowledge that can supply examples, such as user, another Agent...etc.

The activity diagram steps for this scenario are the same as for the previous one but they are only different at Step 3. The inputs to the classify-examples activity are the examples in the buffer. The Intelligent Agent does not use the available hypothesis during the classification process. Figure 5.17 illustrates the activity diagram for this scenario.

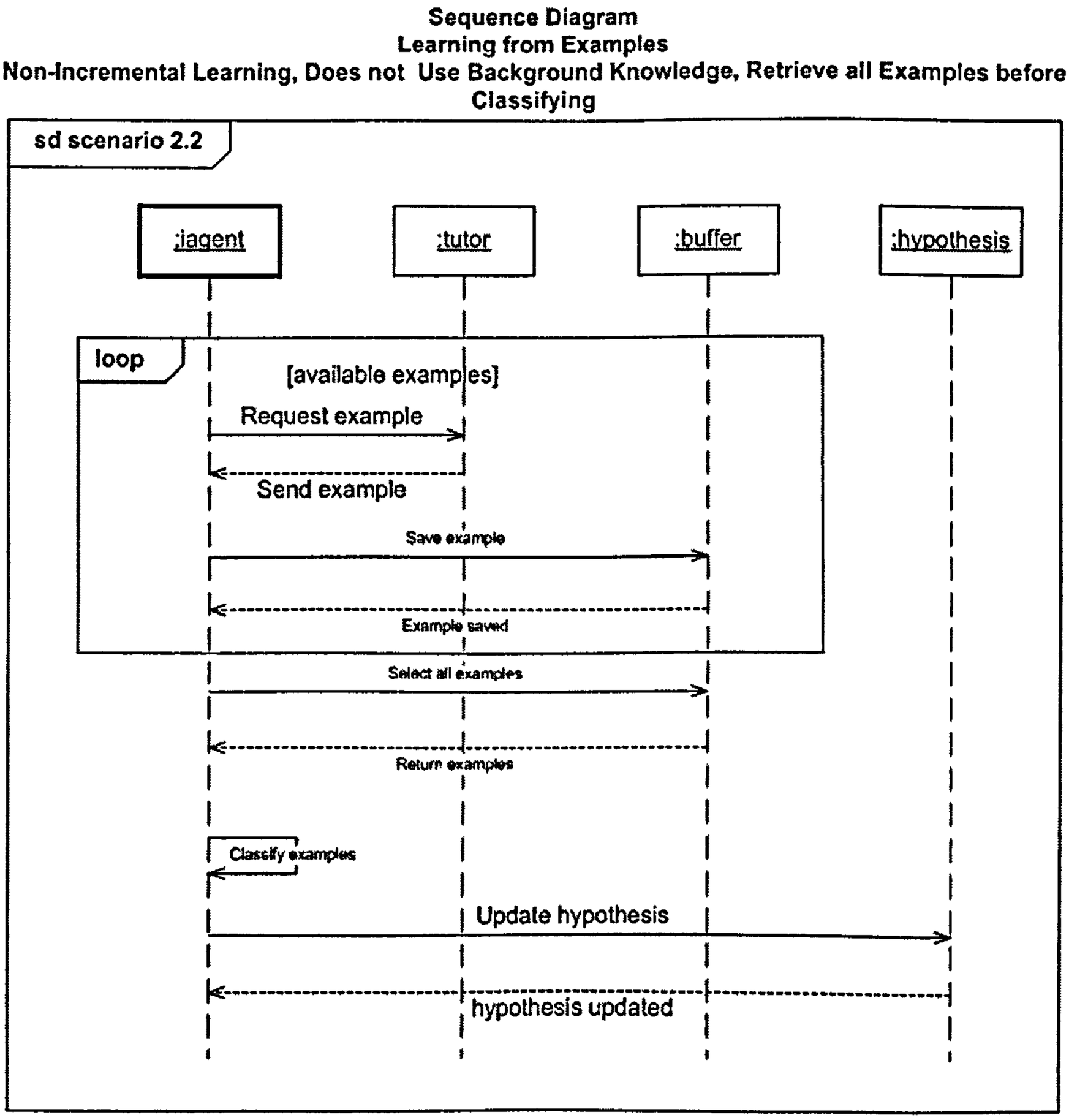


Figure 5.18 UML sequence diagram for scenario 2.2

The UML sequence diagram for this scenario shows that the Intelligent Agent communicates with the available hypothesis only to update its entry after classifying the received examples. Figure 5.18 demonstrates the sequence of communication between the Intelligent Agent and other entities within the sequence diagram.

5.2.2.3 Scenario 2.3: "Learning from Examples, Non-incremental, Use Background knowledge, Receive example, classify, update hypothesis, and then request another example"

This scenario describes the Intelligent Agent that uses the learning from example strategy. The learning process is a non-incremental one, and the Intelligent Agent uses background knowledge during the classification of the received example. The Intelligent Agent classifies the example immediately before requesting another example. The learning process ends when *end of examples* flag is true. Figure 5.19 illustrates an activity diagram for this scenario. The Intelligent Agent requests an example from the tutor, classifies the example, and updates the hypothesis before requesting a new example. The following steps describe the scenario:

1. The activity diagram contains two swim lanes, the Intelligent Agent and the tutor swim lanes.
2. The Intelligent Agent requests an example from the tutor
    a. If end of examples flag is false, tutor sends example.
    b. If end of examples flag is true, flow of control points towards the end of flow node.
3. The Intelligent Agent receives the example and integrates the example with the current hypothesis.
4. The Intelligent Agent classifies the received example.
5. The Intelligent Agent updates the hypothesis with the new classification.
6. The Intelligent Agent requests new examples (Step 1).

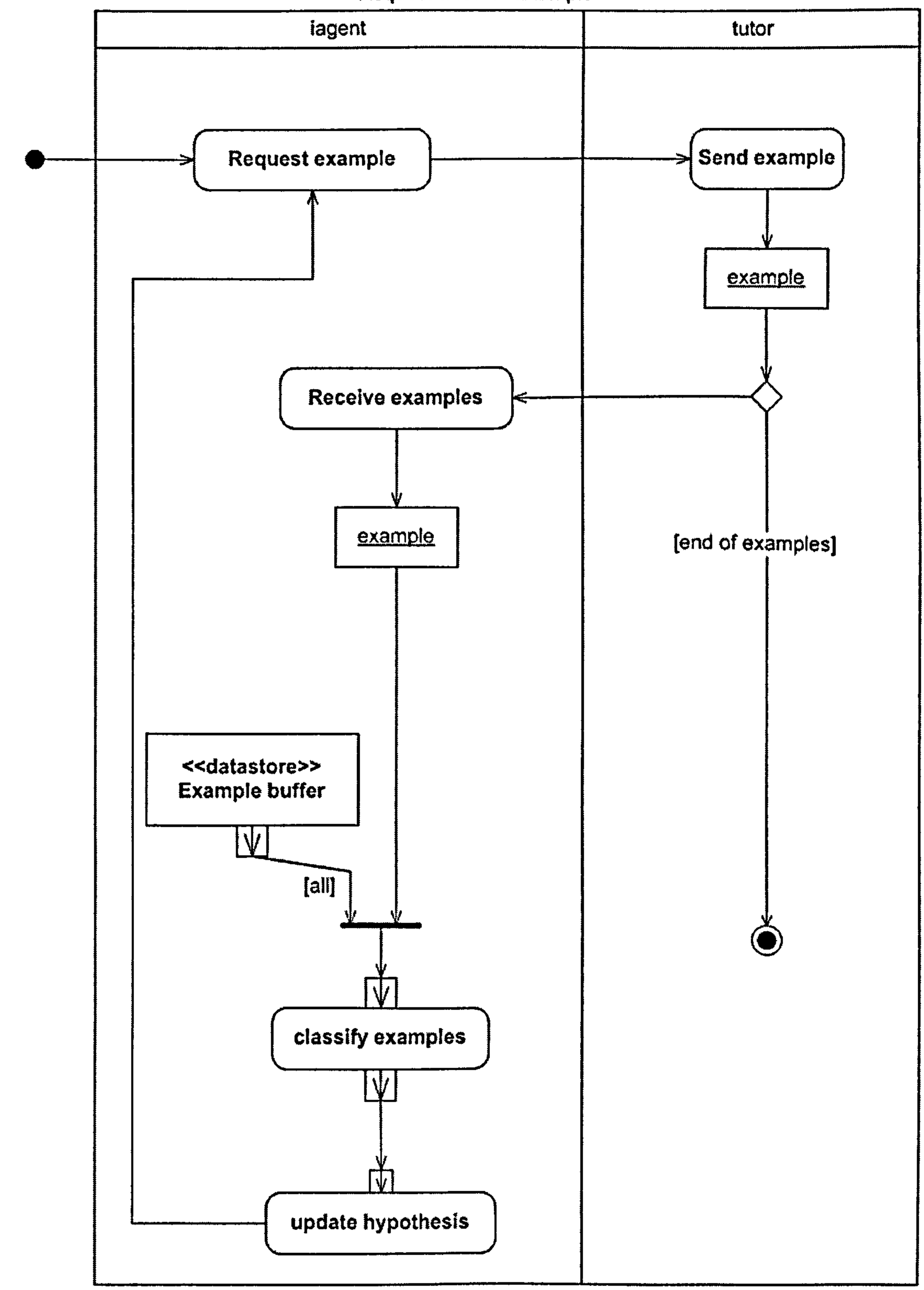


Figure 5.19 UML activity diagram for scenario 2.3

Unlike the previous scenario, the UML sequence diagram (figure 5.20) for this scenario shows that the Loop Interaction Fragment holds all communications between the Intelligent Agent and other entities in the sequence diagram. The diagram has three lifelines: Intelligent Agent, tutor, and hypothesis. The Intelligent Agent terminates learning when the available-example flag is false. This scenario is recommended when all entities are available and on-line.

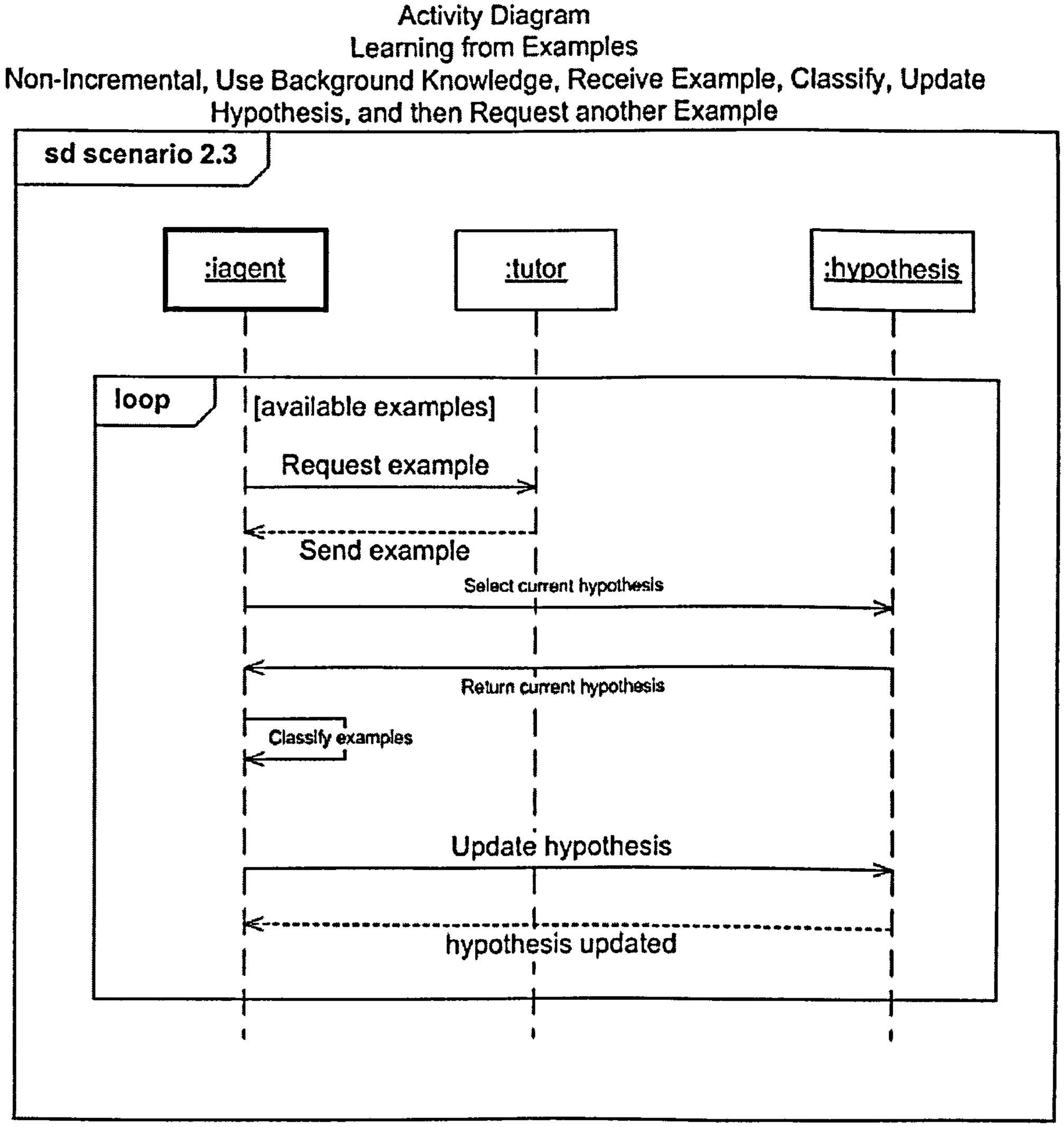


Figure 5.20 UML sequence diagram for scenario 2.3

5.3 Modelling Learning Feedback Methods of Intelligent Agent

There are three types of learning feedback methods: supervised learning, unsupervised learning, and reinforcement learning. Learning from examples is a supervised learning feedback method whereas learning from observation and discovery is an unsupervised learning feedback method. The most appropriate learning feedback methods for Intelligent Agents that deploy in dynamic environments are reinforcement learning and unsupervised learning. In dynamic environments, it is very difficult to assign a tutor that provides the correct answer for every action Intelligent Agents carry out. Nevertheless, Intelligent Agents require some guidance for the achieved performance that is lacking in unsupervised learning feedback method. Unlike the two previous sections in modelling scenarios, the following section evaluates the capability of UML state machine diagrams to model specific reinforcement learning methods and algorithms, namely dynamic programming, Monte Carlo, and temporal difference methods and their different algorithms.

5.3.1 Reinforcement Learning Feedback Methods

Intelligent Agents that use the reinforcement-learning method learn what to do, and how to map situations to actions in order to maximise a numerical reward signal (Sutton 1998). Unlike supervised learning, the tutor does not inform the Intelligent Agents which actions to take; rather, Intelligent Agents discover which actions yield the most reward by trying them. Reinforcement learning is adequate for learning from interaction and when the environment is dynamic. The Intelligent Agent receives some evaluation of its actions but the tutor does not tell it the correct action (Russell and Norvig, 1995). The tutor in reinforcement learning can be a user, an environment, or another Agent. The following sections evaluate if UML 2.0 state machine diagrams can model dynamic programming, Monte Carlo, and temporal difference algorithms.

5.3.1.1 *Dynamic Programming – Policy Iteration Algorithm*

The Intelligent Agent uses Policy iteration algorithm to find an optimal policy that it can use. Figure 5.21 shows the policy iteration algorithm that contains policy evaluation and policy improvements algorithms. Figure 5.22 depicts a UML state machine diagram for policy iteration algorithm. The diagram shows five states; *initial, initialisation, policy_evaluation, policy_improvement*, and *final* states. *The policy_evaluation* and *policy_improvement* are composite states that contain sub-states. The initialisation state initialises value states and policy used by the intelligent agent.

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Figure 5.21 Policy iteration algorithm (Sutton and Barto, 1998)

State Machine Diagram

Reinforcement Learning – Dynamic Programming - Policy Iteration Algorithm

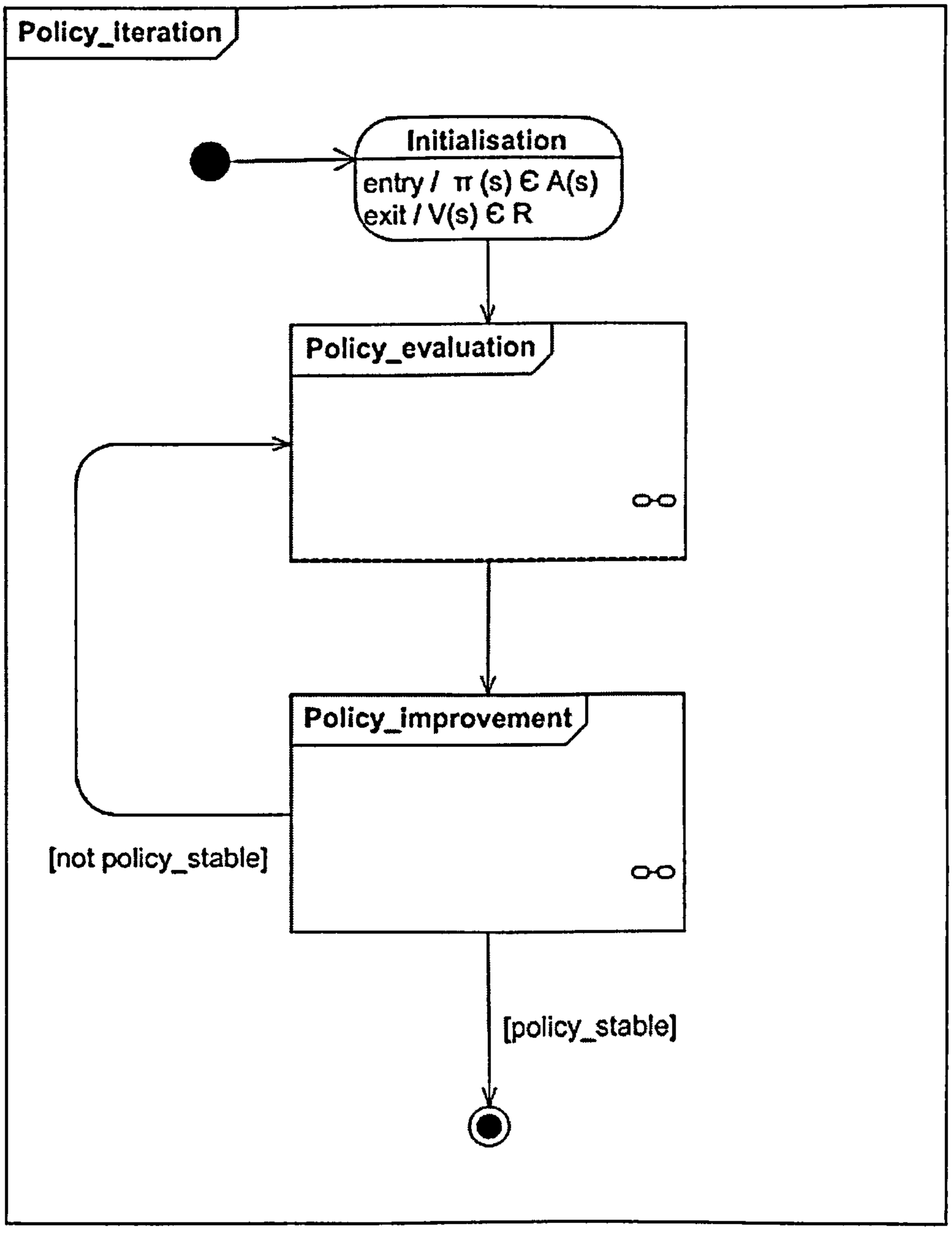


Figure 5.22 UML State machine diagram for policy iteration algorithm

The following sections describe the *policy_evaluation* and the *policy_improvement* sub-states. The Intelligent Agent stops learning when the policy_ stable parameter is true, i.e. stable policy. If the policy_stable parameter is false the Intelligent Agent transfers to the *policy_evaluation* state to continue the exploration of stable policy.

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

Figure 5.23 Iterative policy evaluation algorithm (Sutton and Barto, 1998)

5.3.1.2 Dynamic Programming - Policy Evaluation Algorithms

The policy evaluation algorithm attempts to change the current value function to resemble the true value function of the current policy. Figure 5.23 depicts the policy evaluation algorithm as described by Sutton and Barto (1988) and Figure 5.24 shows UML state machine diagram that models the algorithm. The diagram has three states: *start, initialise,* and *Policy_evaluation* states. Each state contains activities that the Intelligent Agent performs during the state. *Policy_evaluation* state is a composite state that contains sub-states for the Intelligent Agent. The following are the descriptions of the components of the state machine diagram:

1. *Initialise* state: While the Intelligent Agent is in this state, it acquires the policy through *Read* π activity and initialises the Value function, V(s) to 0 for all states.
2. *Policy-evaluation* state: This a composite state that consists of multiple states as follows:
    a. *Initialise_s* state: During this state the Intelligent initialises counter s and Δ to zero
    b. *Learn_valuefn*: The Intelligent Agent conducts three activities during this state; the first one *setvalue* (v, V(s)) which set v to the current V(s), where s is the counter for the current state. The second *setvalue* activity sets V(s) to

the results of $\Sigma_a \pi (s,a) \Sigma s` \rho^a_{ss'} [R^a_{ss'} + \partial V(s')]$. The Third setvalue activity sets Δ to highest value of current Δ and the |v-V(S)|.

c. If the value of Δ is less than a specific, small, positive number

   i. The agent would go to the *StopLearning* state. During the *StopLearning* state the agent sets V^{π} to V

   ii. Go to final state

d. Otherwise, the Intelligent Agent will continue to the decision "pseudostate", if this is the last state s in S.

   i. Set s to 0 and then transfers flow to state b, *Learn_valuefn* sub-state.

   ii. Else set s to s' and then go to state b, *Learn_valuefn* sub-state.

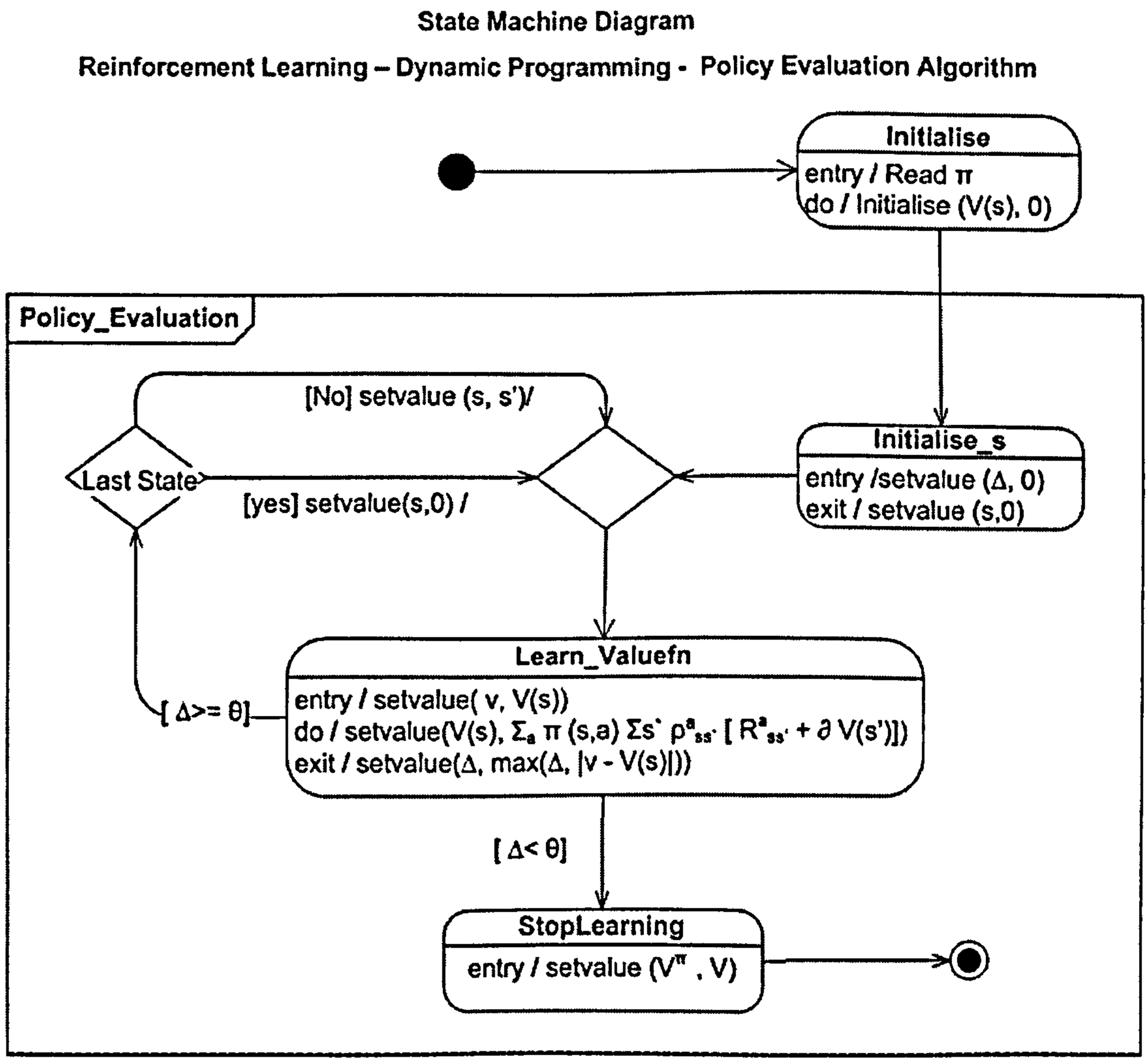


Figure 5.24 State Machine Diagram for policy evaluation algorithm

5.3.1.3 Dynamic Programming – Policy Improvement Algorithm

The policy improvement algorithm tries to change the current policy to make it better. Figure 5.25 depicts the UML state machine diagram for the policy improvement algorithm as stated in Figure 5.21. The diagram shows five states: *initial, initialise_policy, set_temp, improve_policy,* and *decision* states. The following describe the main sub-states of the policy improvement state:

1. *Initialise_Policy* state: During this state, the Intelligent Agent sets the policy-stable parameter to true and s parameter to zero, first state.
2. *Set_temp* state: During this state, the Intelligent Agent sets the b parameter to π (s), current policy value for state s.

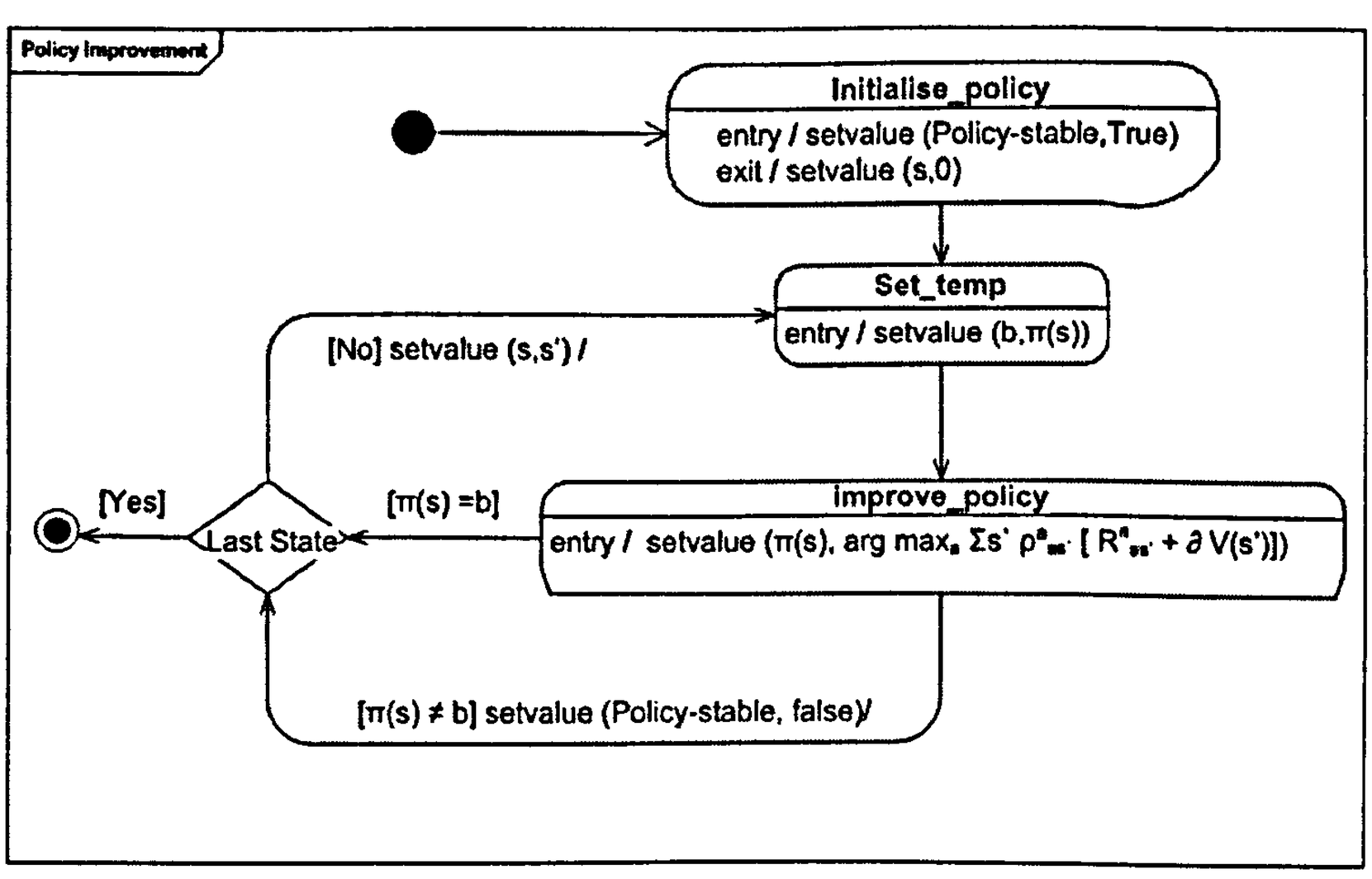


Figure 5.25 UML State machine diagram for policy improvement algorithm

3. *Improve_policy* state: during this state, the Intelligent Agent sets the current policy value for state s to "arg maxa Σs` $\rho^{a}$ss` [Rass' + ∂ V(s')]".

a. If $\pi(s)$ is not equal to b, then the intelligent Agent sets policy_state parameter to false and transits to the *decision* state

b. Else, go to decision state.

4. *Decision* state:

a. If the current state is not the last state, then Intelligent Agent increments s to s' and goes back to set_temp state to continue learning

b. Else, the Intelligent Agent moves to final state to exit from the policy improvement composite state.

5.3.1.4 Dynamic Programming - Value Iteration Algorithm

Figure 5.26 depicts the value iteration algorithm as identified by Sutton and Brato (1998). The UML state machine diagram of this algorithm has three states; *initial, initialize,* and *value_iteration* states (figure 5.27). *The value_iteration* state is a composite state that includes six sub-states; *set_s, set_valuefn, learn_valuefn, decision, stoplearning,* and *final node* sub-states.

Initialize V arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

Figure 5.26 Value iteration algorithm (Sutton and Barto, 1998)

During each state or sub-state of the UML state machine diagram of the value-iteration algorithm, the Intelligent Agent performs the following activities:

1. *Initialise* state: The Intelligent Agent acquires the policy under evaluation and initialises V to 0 for all available states.
2. *Set_s* sub-state. The Intelligent Agent set Δ and s to 0.
3. *Set_valuefn* sub-state: During this state, the Intelligent Agent sets the parameter v to the current value function, V(s).
4. *Learn_valuefn* sub-state: The Intelligent Agent conducts two activities during this state: *entry* and *exit* activities. At *entry* activity, it calculates current Value function V(s) by the getmax procedure. Before exiting from this state, it sets Δ to the highest of Δ and $|v - V(s)|$.
5. If $\Delta < \theta$ (a small positive number)
    i. *StopLearning* sub-state: The Intelligent Agent sets policy π (s) to arg maxa $\Sigma s` \rho^a ss` [Rass' + \partial V(s')]$
    ii. Go to *Final* state.
6. Else; transfer flow to *Decision* sub-state
    i. If this is the last state then the Intelligent Agent sets s to 0
    ii. else, the Intelligent Agent sets s to s'
7. The Intelligent Agent returns to *set_Valuefn* sub-state to continue learning.

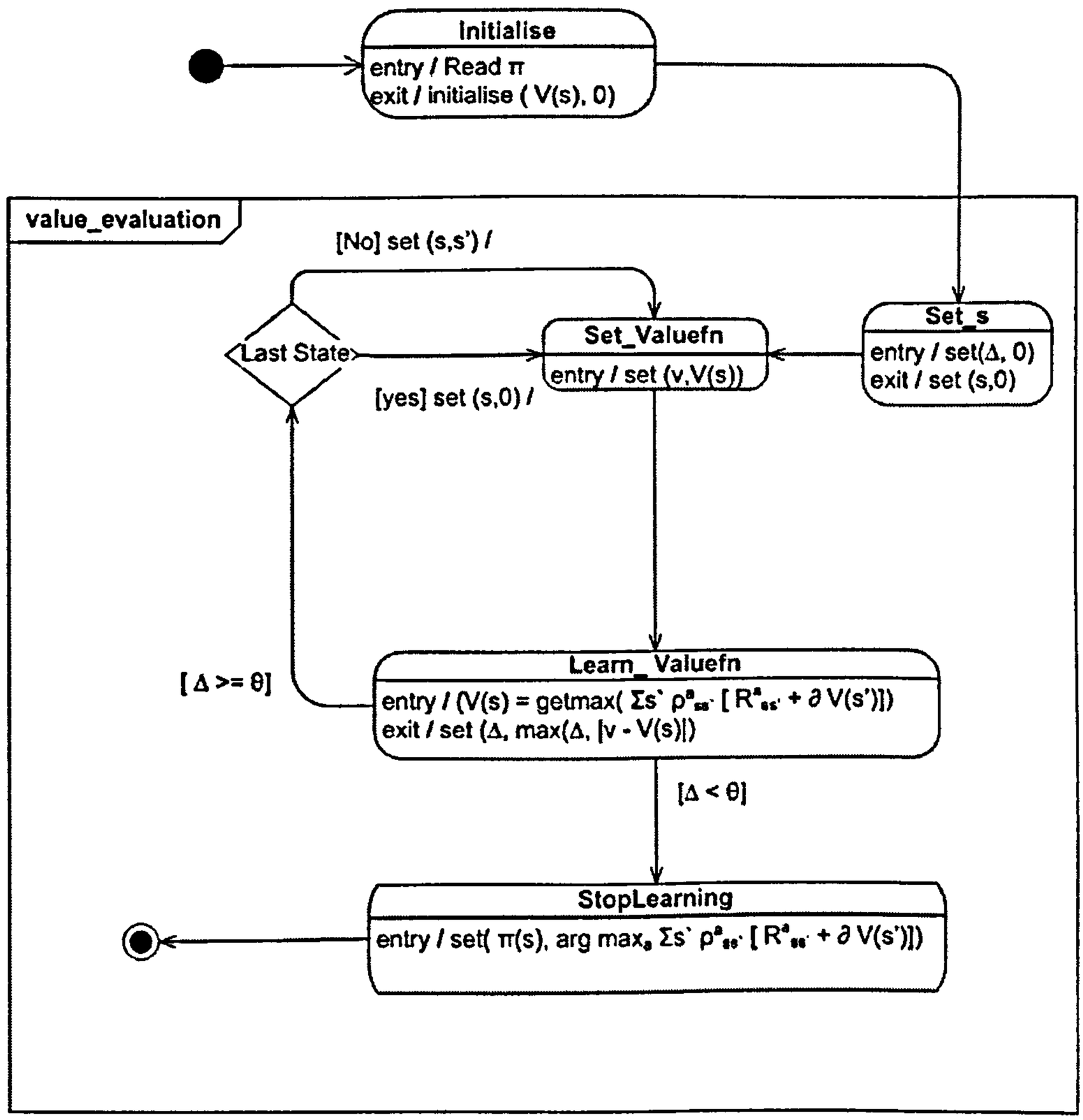


Figure 5.27 UML state machine diagram for value evaluation algorithm

5.3.1.5 Monte Carlo Policy Evaluation Algorithm

Figure 5.29 depicts the UML state machine diagram for first-visit Monte Carlo policy evaluation algorithm that is shown in Figure 5.28. The diagram has four states; *initial, initialise get_episode,* and *receive_return* states. During each state, the Intelligent Agent conducts the following activities:

1. *Initialise* state: The Intelligent Agent initialises policy, sets V to an arbitrary state-value function, and initialises R(s) to an empty list.
2. *Get_episiode* state: The Intelligent Agent requests to get an episode through the *generate_episode* activity and set counter s to -1.

Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    (a) Generate an episode using π
    (b) For each state s appearing in the episode:
        $R \leftarrow$ return following the first occurrence of s
        Append R to $Returns(s)$
        $V(s) \leftarrow$ average($Returns(s)$)

Figure 5.28 First-visit Monte Carlo method for V estimating V^{π} (Sutton and Barto, 1998)

3. *Receive_return* state: During this state the Intelligent Agent increment counters at entry, receives reward R through *get_reward* activity, appends R to R(s) through *append_R* activity, and calculates V(s) through averaging R(S). The *get_reward* activity provides the return following the first occurrence of s.
4. If this is the last state, the flow returns to *get_episode* state to generate a new episode.
5. Else, the Intelligent Agent loops back to *receive_return* state.

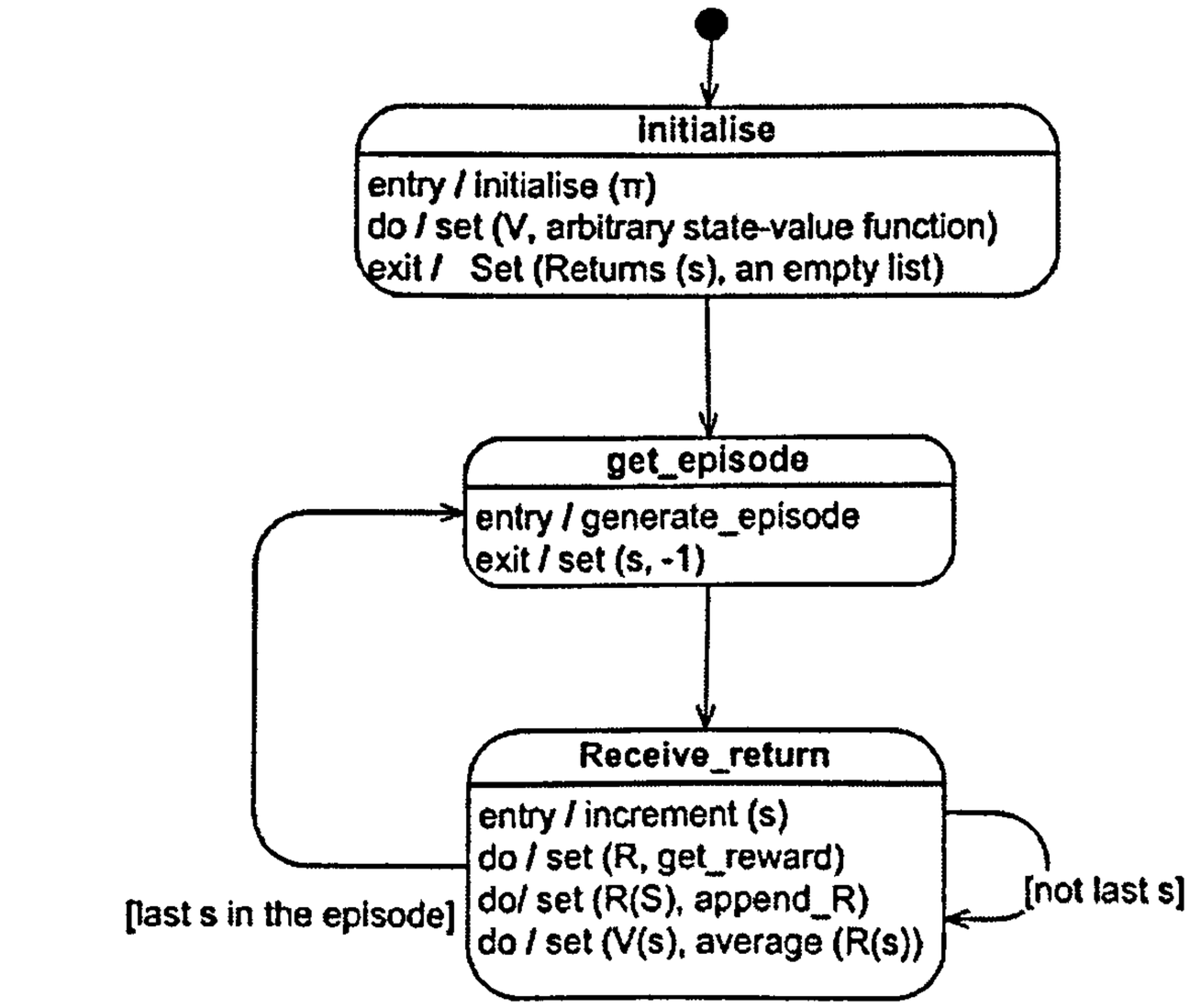


Figure 5.29 UML state machine diagram for first-visit Monte Carlo policy evaluation algorithm

This algorithm does not have a final node as the Intelligent Agent always continues to learn the value function for the current episode or generate a new episode to enhance its performance. The *get_reward* activity can be adjusted to provide a return at each visit of s, thus the UML state machine diagram will be the same for Every-visit Monte Carlo policy evaluation algorithm.

Sutton and Barto (1998) have provided the Black Jack card game as an example of the Monte Carlo policy evaluation algorithm. Figure 5.30 depicts the UML state machine diagram for the Intelligent Agent which tries to learn the best policy to play the card game. The diagram has five main states; *initialise, receive_cards, decision, dealer_turn,* and *receive return* states. The diagram also includes *start, final,* and *merge* states. The states include activities that the Intelligent Agent conducts during the active state as follows:

1. *Initialise* state: This state has three activities; the first activity, *initialise* π, initialises the policy that the intelligent agent uses during the game. This policy is to hit whenever the sum of the cards is less than 20. The second activity, *set V*, initialises V to an arbitrary state-value function, while the third activity, Return(S), sets the Return(s) to an empty list.
2. *Receive-Cards* state: During this state, the Intelligent Agent receives cards from the Dealer.
3. *Current_Sum* decision state: This decision state routes the flow according to the current sum of cards available with the Intelligent Agent as follows:
    a. For Current Sum < 20: The Intelligent Agent requests cards and the flow is transferred to the *initialise* state (step 2).
    b. For 20 >= Current Sum < 21: The Intelligent Agent sticks and the flow of transfer points to the *dealer_turn* state.
    c. For Current Sum > 21: The flow transfer to the *Receive_return* state.
4. *Dealer_turn* state: During this state, the Intelligent Agent waits until the dealer continues playing

5. Receive_return state: The Intelligent Agent conducts three activities during this state, in the first it gets the reward by *get_reward* activity. The second one is to append the return to the return list. Third, it calculates the value function, V(s), by averaging Return(s).
    a. If the Intelligent agent decides to stop playing, the flow transfers to final node.
    b. Else, the flow transfers returns to *receive_cards* status and the Intelligent Agent requests the start of a new game.

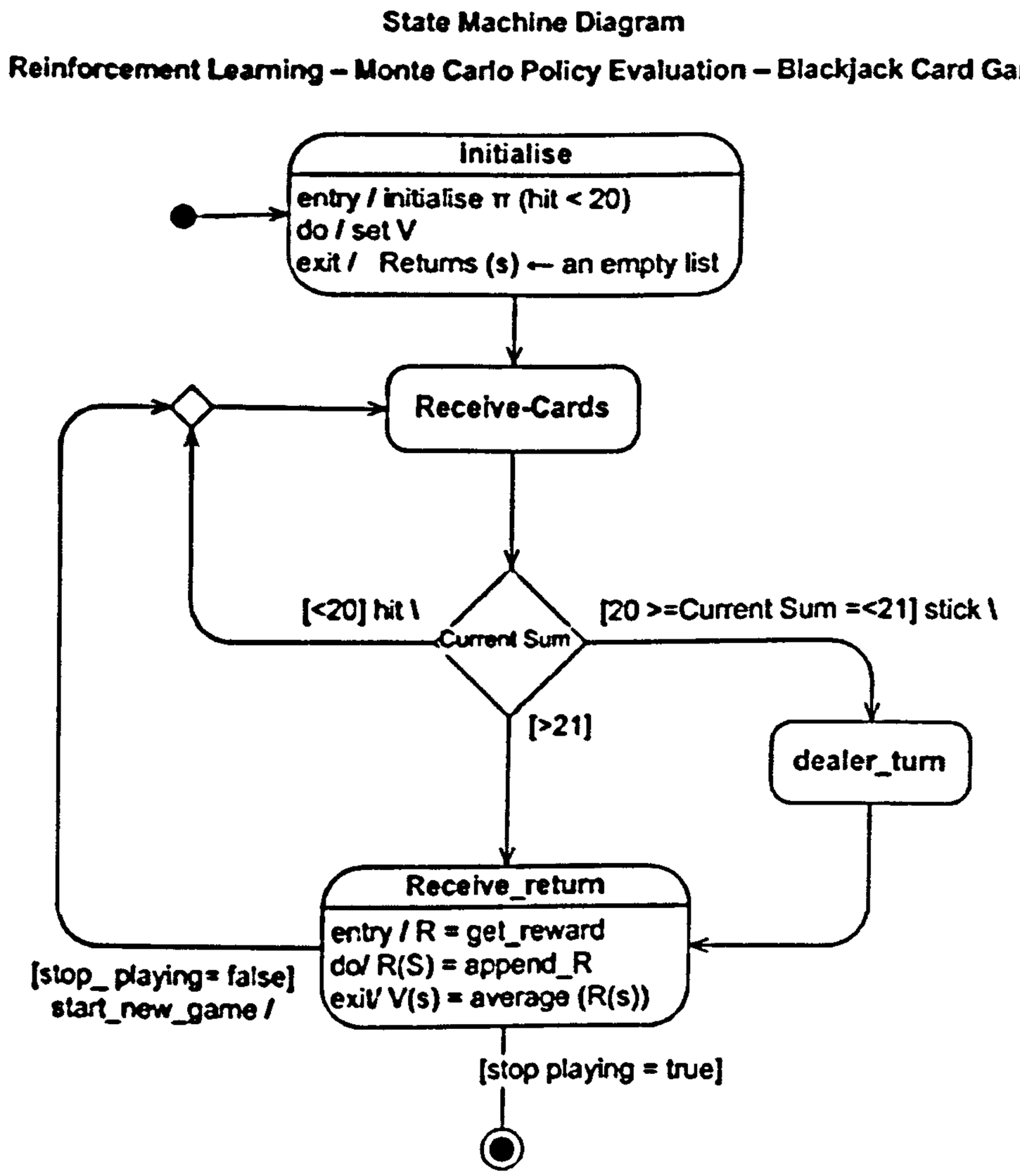


Figure 5.30 UML state machine diagram for Black Jack game

5.3.1.6 Monte Carlo Control Algorithm

Figure 5.31 shows the Monte Carlo control algorithm assuming exploring starts (Sutton and Barto, 1998). The UML state machine diagram for this algorithm consists of six states; *initial*, *initialise*, *get_episode*, *receive_returns*, *improve_policy*, and *decision* states. The algorithm has infinity loop, which the diagram describes by the absence of the final node. The following is the description of the states and the activities that the Intelligent Agent conducts:

1. *Initialise* state: During this state the Intelligent Agent conducts three activities; at entering this state it initialises the policy π(s), then sets action-value, Q(s,a), to an arbitrary values, and at exit it sets Returns(s,a) to an empty list.

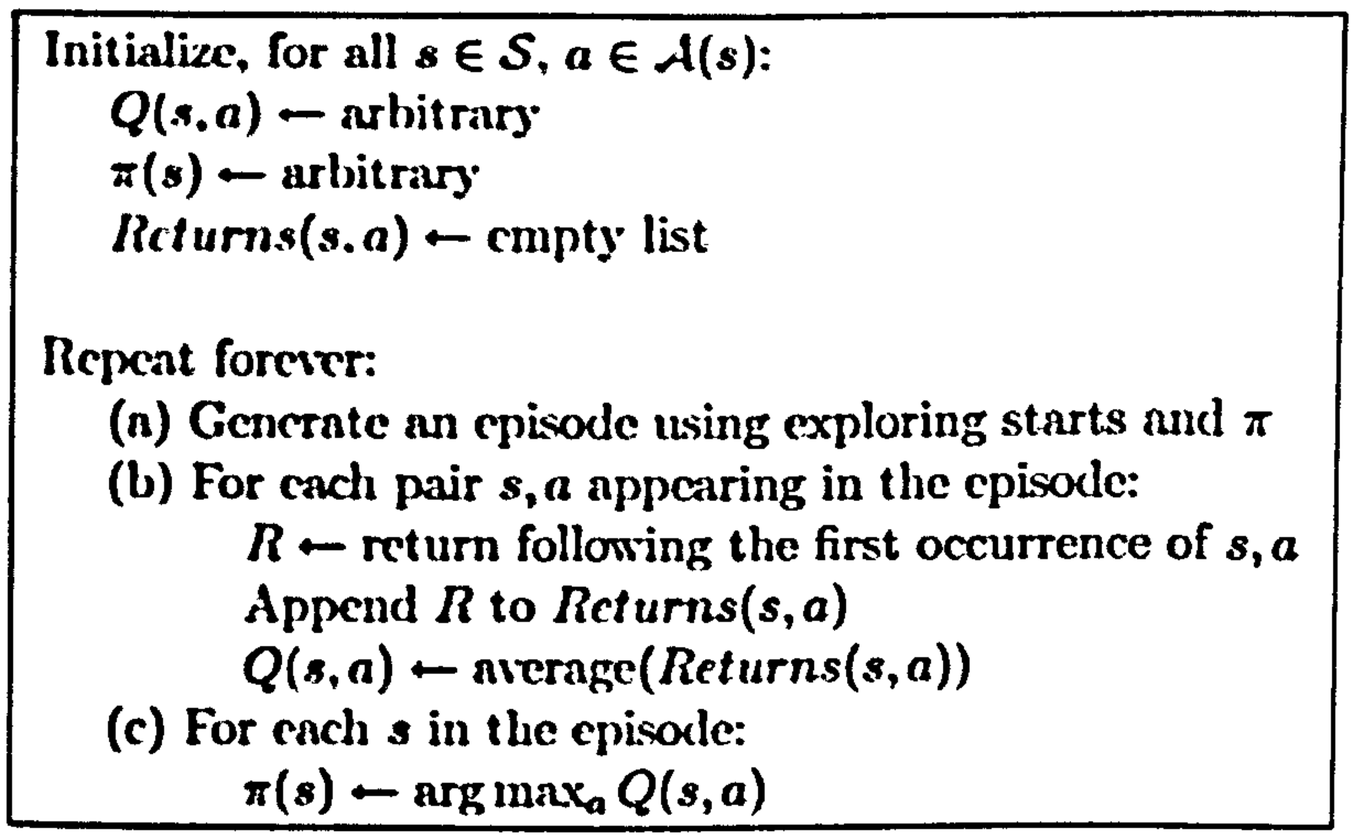


Figure 5.31 Monte Carlo control algorithm assuming exploring starts (Sutton and Barto, 1998)

2. *Get_episode* state: The Intelligent Agent generates episode and sets parameters "a" and "s" to zero.
3. *Receive_returns* state: During this state, the Intelligent Agent receives rewards, R(s,a) by the *get_reward* activity, appends R (s,a) to the Return (s,a) list by the *append_R* activity, and calculates the action-value function, Q(s,a) by averaging Returns(s,a) list.

a. If this is the last episode state, s, the flow transfers to *improve_policy* state.

b. Else, the flow transfers to the decision state.

4. Decision state:

   a. If this is the last action for this state, counter "s" is incremented and flow returns to *receive_returns* state.

   b. Else, counter "a" is incremented and flow returns to *receive_returns* state.

5. *Improve_policy* state: During this state, The Intelligent Agent updates the current policy by setting policy $\pi(s)$ to $\max_a Q(s,a)$.The flow transfers to *get_episode* state to generate a new episode.

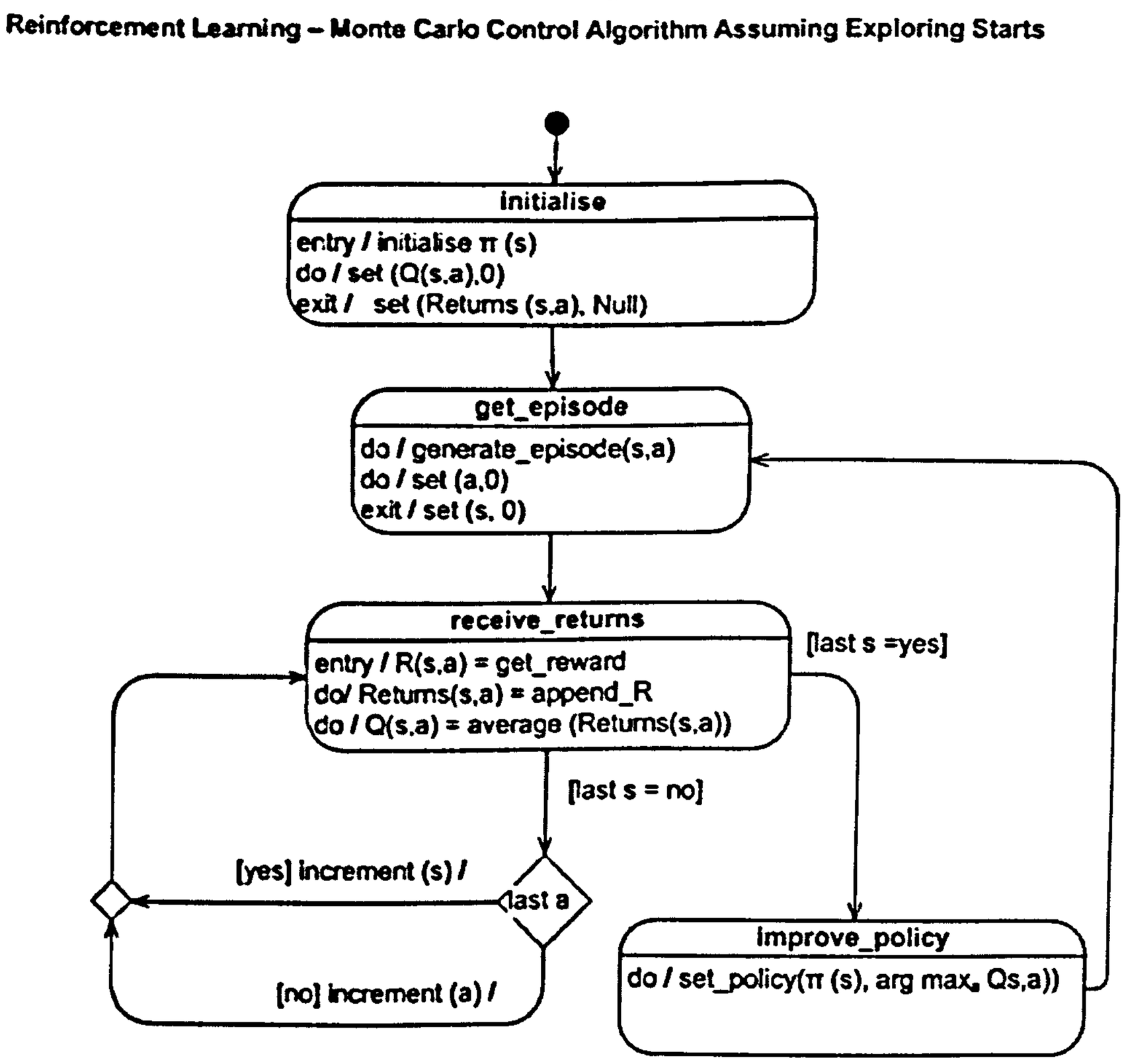


Figure 5.32 UML state machine diagram for the Monte Carlo control algorithm assuming exploring starts

5.3.1.7 Temporal Difference Predication Algorithm

Figure 5.33 shows the algorithm of tabular Temporal Difference (TD) prediction algorithm (Sutton and Barto, 1998). The UML state machine diagram for this algorithm (Figure 5.34) has four states; *initial, initialise get_episode,* and *update_v* states. The Intelligent Agent conducts the following activity within the following states:

1. *Initialise* state: The Intelligent Agent initialises V(s) such as to zero and read the policy, $\pi(s)$ that needs evaluation.
2. *Get_episode* state: The Intelligent Agent generates a episode by the *generate_episode* activity, and sets the value of α, ∂, and s. if there are no new episodes, flow transfers to Final flow state
3. Update_v state: During this state the Intelligent Agent gets the value of V(s) and V(s') by using *get_value* activity, receives rewards through the *get_reward* activity, and calculates the expected V(s). Before exiting from this state, it sets s to s'.
   a. If the Intelligent Agent reaches a terminal state, then the flow transfers to *get_episode* state to generate new episode.
   b. Else, the flow loops back to the *update_v* state.

Initialize $V'(s)$ arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a; observe reward, r, and next state, s'
    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
    $s \leftarrow s'$
  until s is terminal

Figure 5.33 Tabular TD(0) for estimating V^{π} (Sutton and Barto, 1998)

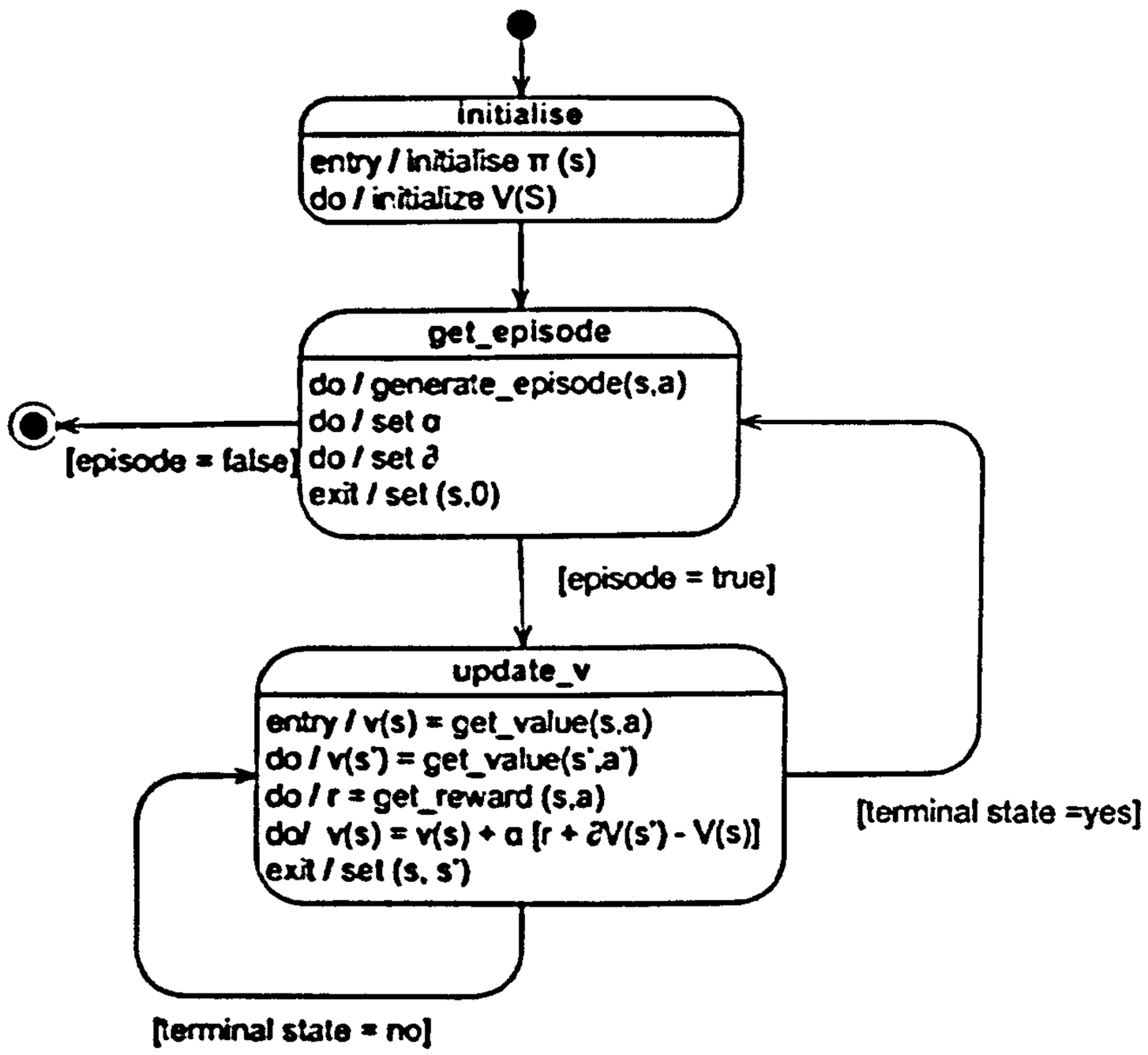


Figure 5.34 UML state machine diagram for TD algorithm of figure 5.33

5.3.1.8 Temporal Difference – Sarsa Algorithm

Figure 5.35 depicts the temporal difference - Sarsa algorithm. The UML state diagram for this algorithm (figure 5.36) has four states; *initial, initialise get_episode,* and *update_Q states.* The Intelligent Agent conducts activities during each state on entering, during or exiting from this state. The following are the activities that the Intelligent Agent conducts during each state:

1. *Initialise* state: The Intelligent Agent initialises both the policy that it will use and the action-value, Q(s,a), that it will calculate to an arbitrary value.
2. Get_episode state: The Intelligent Agent initialises counter "s" on entering this state; "s" represents state number in the episode. It also generates episode to perform learning. The Intelligent Agent also sets α and ∂ to

specific values to use them for computing the action-value function. If there are no new episodes, flow transfers to *final* flow state.

3. Update_Q state: The Intelligent Agent updates the action-value function during this state by conducting the flowing activities: The Intelligent Agent gets the Q value for s and s' in the generated episode by *get_Qvalue* activity. It also receives the reward through the *get_reward* activity. Then it computes the Q(s,a) according to the shown equation. Finally, it increments s to s' and a to a' before exiting from that state, where s' and a' are the next state and action respectively.

    a. If s is at a terminal state of this episode, then flow transfers to the *get_episode* state

    b. Else, the flow loops backs to continue in updating the action-value function.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Figure 5.35 Sarsa: An on-policy TD control algorithm (Sutton and Barto, 1998)

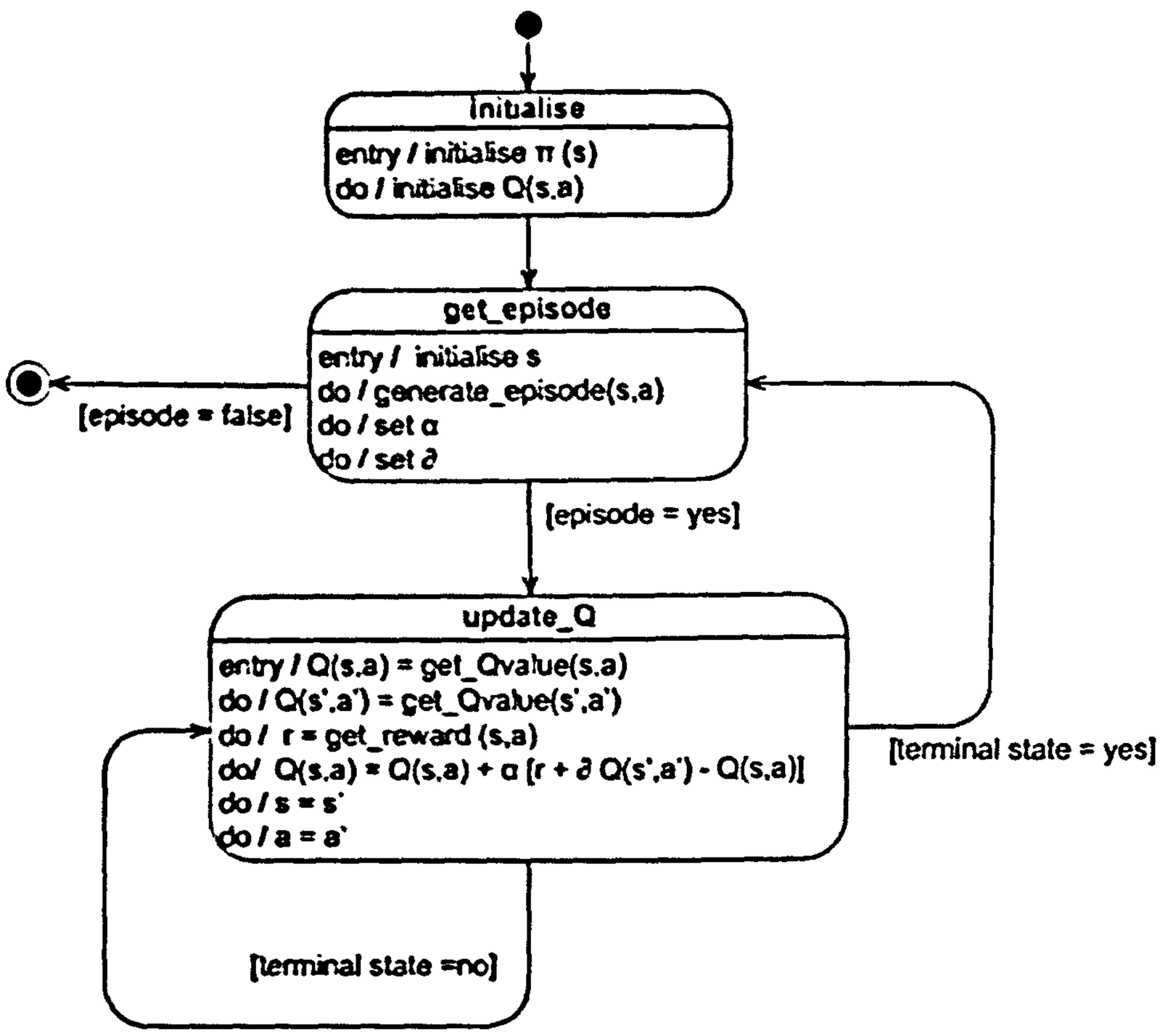


Figure 5.36 UML state machine diagram for sarsa algorithm

5.3.1.9 Temporal Difference Q-learning Algorithm

Figure 5.37 describes the Q-learning algorithm that Sutton and Barto (1998) have described. The UML state machine diagram of the Q-learning Algorithm (Figure 5.38) contains four states; *initial, initialise get_episode,* and *update_Q* states. The following are the activities that Intelligent Agents conducts during each state:

1. *Initialise* state: During this state the Intelligent Agent initialises the Q(s,a) to an arbitrary value such as 0 and the policy, π (s) that it will use during learning.
2. Get_episode state: The Intelligent Agent initialises counter "s" that represents number of state in the episode, conducts the activity *generate_episode(s,a),* and sets learning rate α and the discount factor ∂.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  until s is terminal

Figure 5.37 Q-learning Algorithm (Sutton and Barto, 1998)

3. *Update_Q* state: During this state the Intelligent Agent updates the value of Q(s,a). It calculates the values of Q(s,a) and Q(s',a'). *Get_maxQvalue* activity selects the maximum action-state for the next episodic state, Q(s',a'). The Intelligent agent receives the reward, r and updates Q(s,a) according to the shown equation. The Intelligent Agent sets s and a to s' and a' respectively. If s is a terminal state then flow transfers to the *get_episode* state. Else, the flow loops back to *update_Q* state.

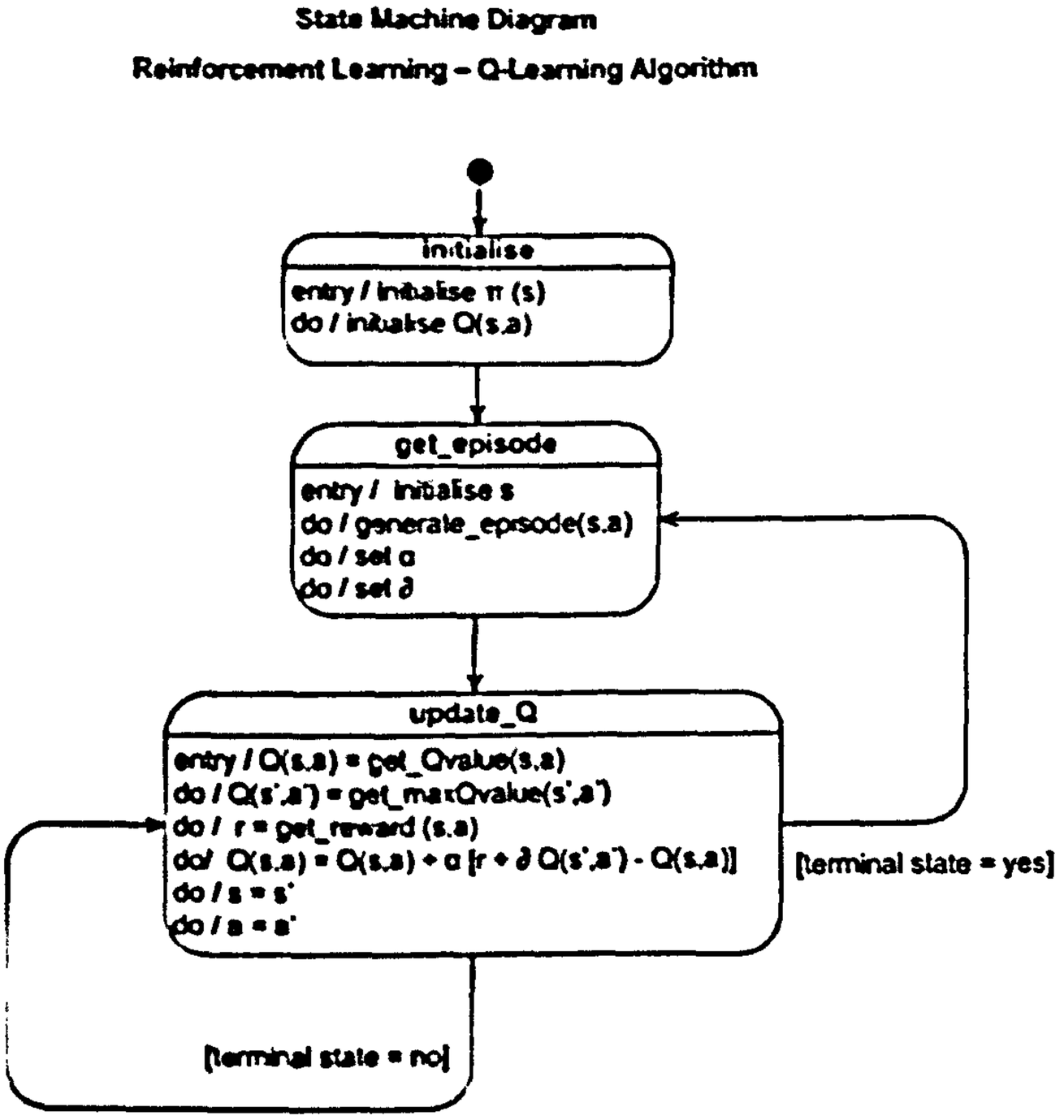


Figure 5.38 UML state machine diagram for q-learning algorithm

5.3.1.10 Temporal Difference – Sarsa (λ) Algorithm

Figures 5.39 and 5.40 show the tabular Sarsa (λ) algorithm and its UML state machine diagram respectively. The state machine diagram has three main states as in the previous Sarsa algorithm described in section 5.3.1.8, but each state has different activities conducted by the Intelligent Agent as follows:

1. *Initialise* state: During this state, the Intelligent Agent initialises the policy that it will use during learning and action-values function Q(s,a) that it will learn for all s and a such as 0. It also sets e(s,a) to 0 for all s and a.
2. *Get_episode*: The Intelligent Agent initialises s and a when entering this state. Then it generates episodes by the *generate_episode (s,a)* activity. The Intelligent Agent before leaving this state, sets α and ∂ to the required values. If there are no more episodes the flow transfers to *final* state, else it continues to the next state.

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s,a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Figure 5.39 Tabular Sarsa (λ) Algorithm (Sutton and Barto, 1998)

3. Update_Q state: During this state, the Intelligent Agent conducts the main activities that it needs to conduct learning. First, the Intelligent Agent conducts the *takeaction* activity to get the next state, s'. Then it receives the reward by the *get_reward* activity. Following these two activities, the

Intelligent Agent chooses a' from s' through the *policy.selectaction* activity. Then it gets the values for current and next action-state function, Q(s,a) and Q(s',a'), by conducting *get_Qvalues* activity. The next activity is to compute δ according to the given equation. For all the available s and a in the episode, the Intelligent Agent conducts *update_Qvalues* activity to update Q and e according to the following equations: Qs,a)= Q(s,a) + α δ e(s,a) and e(s,a)= γ λ e(s,a)

Finally, s and a are incremented to s' and a' respectively. If s is a terminal state, then flow transfers to *get_episode* state or else it loops back to *update_Q* state.

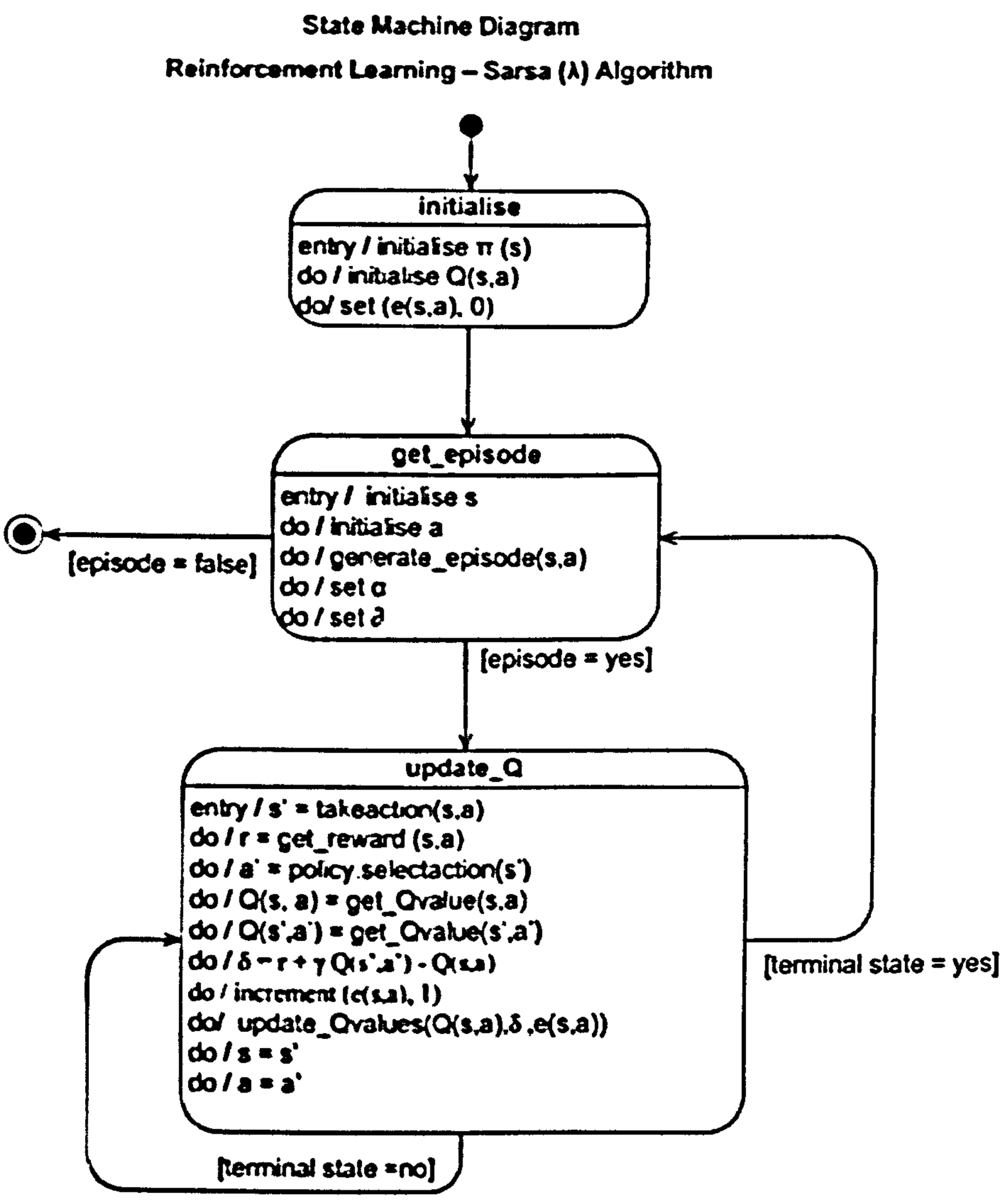


Figure 5.40 UML state machine diagram for tabular Sarsa (λ) algorithm

5.3.1.11 Temporal Difference – Q(λ) Algorithm

Figure 5.41 shows the tabular version of Watkins's Q(λ) algorithm as described by Sutton and Barto (1998) Watkins's algorithm combines Q-learning and eligibility traces. The UML state machine diagram for this algorithm (figure 5.42) has four states; *initial, initialise, get_episode,* and *update_Q* states. The difference between this state machine diagram and the previous Q-learning state machine diagram is in the activities conducted within each state. The activities the Intelligent Agent conducts are as follows:

1. *Initialise* state: during this state the Intelligent Agent initialise the policy and action-value function, Q(s,a). It also sets e(s,a) to 0.
2. *Get_episode*: The Intelligent Agent initialises variable s and a. It also generates an episode to start the learning process through the *generate_episode(s,a)* activity. Finally, it sets α and ∂ to the required values.

```
Initialize Q(s, a) arbitrarily and e(s, a) = 0, for all s, a
Repeat (for each episode):
    Initialize s, a
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ε-greedy)
        a* ← arg max_b Q(s', b) (if a' ties for the max, then a* ← a')
        δ ← r + γQ(s', a*) − Q(s, a)
        e(s, a) ← e(s, a) + 1
        For all s, a:
            Q(s, a) ← Q(s, a) + αδe(s, a)
            If a' = a*, then e(s, a) ← γλe(s, a)
                        else e(s, a) ← 0
        s ← s'; a ← a'
    until s is terminal
```

Figure 5.41 Tabular version of Watkins's Q(λ) algorithm (Sutton and Barto, 1998)

3. *Update_Q* state: This is the main activity of the state machine diagram through which the Intelligent Agent updates the action-value function. The Intelligent Agent conducts the following activities during this state:
    a. At entry the Intelligent Agent conducts an action through the *takeaction(s,a)* activity and identifies the next state, s'.

b. The Intelligent agent receives the reward, r, through the activity *get_reward(s,a)*. The *get_reward (s,a)* activity observes the *takeaction(s,a)* activity and sets r to the received reward.

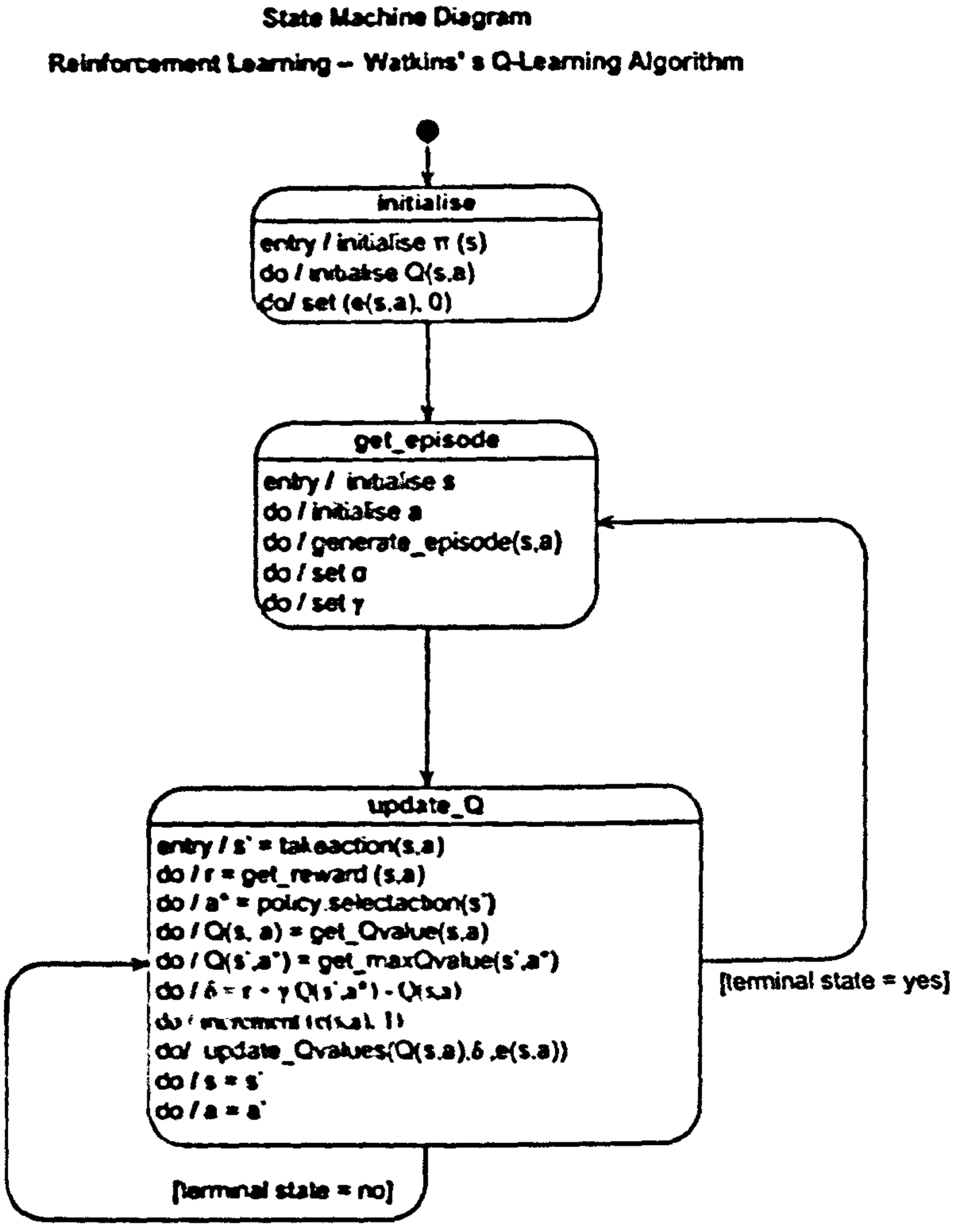


Figure 5.42 UML state machine diagram for Watkins's Q(λ) algorithm

c. Through the *policy.selectacion(s')* activity, the Intelligent Agent identifies a*.

d. Then the Intelligent Agent computes Q(s,a) and Q(s',a*) through the *get_Qvalues(s,a)* and *get_maxQvalues(s,a*)*.

e. The Intelligent Agent computes δ through the given equation and increment eligibility trace, e(s,a).

f. For all s and a, the Intelligent Agent updates Q(s,a) and e(s,a) by the *update_Qvalues(Q(s,a)* activity, δ, e(s,a)).

g. Lastly, the Intelligent Agent sets s to s' and a to a'

h. If s is a terminal state then flow transfers to the *get_episode* state, else flow goes back to *update_Q* state.

5.4 Conclusion

This chapter shows the ability of UML version 2.0 behaviour diagrams, namely activity diagrams and sequence diagrams, to model the learning behaviour of Intelligent Agents that use learning from observation and discovery as well as learning from examples strategies. It also illustrates the capability of state machine diagrams of UML 2.0 to model specific reinforcement learning algorithms, namely dynamic programming, Monte Carlo, and temporal differences algorithms.

UML 2.0 behaviour diagrams provide more advanced modelling techniques than previous behaviour diagrams in UML version 1.x. The introduction of techniques to model parallelism, alternative execution, and looping behaviour in sequence diagrams allow modelling complex behaviour that are exhibited by Intelligent Agents such as learning behaviour. UML 2.0 activity diagram also uses Petri-net-like notations that allow a powerful tool for modelling the workflow. The activity diagrams show the workflow of the learning process while the sequence diagrams show the sequence of interaction between Intelligent Agent and other entities required to perform the learning process. State machine diagrams provide capable tools to describe the behaviour of a single Intelligent Agent across different states. During the writing of the thesis, available tools to model applications by using UML version 2.0 were limited. The research writer has used Microsoft Visio program with UML stencil for version 2.0 templates to model the above diagrams.

This chapter provides different scenarios to explore whether the activity and sequence diagrams can model learning from observation and discovery. Intelligent Agents deploy mainly in dynamic environments requiring learning without gaining advice from a source of knowledge. The source of knowledge is not able to provide advice about the correct actions for all the different states of the dynamic environment. Furthermore, activity and sequence diagrams are capable of modelling scenarios for Intelligent Agents that use a learning from examples strategy. Intelligent Agents that work in dynamic environments can use a learning from examples strategy in one of the following conditions:

1) The rate of examples is higher than the rate of changes in the environment or

2) Cloning the Intelligent Agent and then replacing it after learning.

UML 2.0 state machine diagrams have proven resourceful for modelling specific reinforcement learning methods and their different algorithms. This chapter illustrates the capability of modelling different types of dynamic programming algorithms namely, iterative policy evaluation, policy evaluation, policy improvement, and value iteration algorithms. It also shows the capability of state machine diagrams to model Monte Carlo algorithms, namely policy evaluation and control algorithms. Moreover These type of diagrams are also successful in modelling temporal difference (TD) algorithms, namely TD prediction, Sarsa, Q-learning, Sarsa (λ), and Q(λ) algorithms. Appendices A,B, and C provide user guides for the main components of activity, sequence, and state machine diagrams to allow researchers in agent-oriented systems use the UML 2.0 diagrams in modelling the learning components of Intelligent Agents.

Future information systems will contain objects, Agents, and Intelligent Agent as the main software entities. However, it will be impractical to imagine for the time being that Agent-oriented systems will replace object-oriented ones. The integration of the capabilities for both systems will pave the way for implementing reliable and capable software systems especially for dynamic environments.

Chapter 6: Conclusions and Future Research Directions

6.1 Thesis Conclusions

This thesis has shown the ability of structural and behavioural components of the Unified Modelling Language version 2.0 to model the learning behaviour of Intelligent Agents. The research focuses on the learning of a single Agent, as it is the core of multi-agent systems. The research shows the ability of UML version 2.0 behaviour diagrams, namely activity diagrams and sequence diagrams, to model the learning behaviour of Intelligent Agents that use learning from observation and discovery as well as learning from examples strategies. The research also illustrates the capability of UML 2.0 state machine diagrams to model specific reinforcement learning algorithms, namely dynamic programming, Monte Carlo, and temporal difference algorithms.

UML 2.0 behaviour diagrams provide more advanced modelling techniques than previous behaviour diagrams in UML version 1.x. The introduction of techniques to model parallelism, alternative execution, and looping behaviour in sequence diagrams allow the modelling of complex behaviour that is exhibited by Intelligent Agents such as learning behaviour. UML 2.0 activity diagram also uses Petri-net-like notations that allow a powerful tool for modelling the workflow of the learning process while the sequence diagrams show the sequence of interaction between the Intelligent Agent and other entities required to perform the learning process. UML 2.0 state machine diagrams provide tools capable of describing the behaviour of a single Intelligent Agent across different states. Appendices A,B, and C provide user guides for the main components of activity, sequence, and state machine diagrams to allow researchers in agent-oriented systems to use the UML 2.0 diagrams in modelling the learning components of Intelligent Agents.

The research highlights that learning is a crucial feature for Intelligent Agents. Intelligent Agents are Agents that can learn and reach agreement with other Agents or

users. It recognises different learning components required to model the learning behaviour of Intelligent Agents such as learning goals, learning strategies, and learning feedback methods.

The research has highlighted the need to extend the structural components of UML 2.0. During the writing of the thesis, the Foundation for Intelligent Physical Agent (FIPA) proposed an extension to UML 2.0 to model structural components of Agents. This research utilises FIPA's UML 2.0 Agents' structural components extension and provides further development to model the structural components of not only Agents but also Intelligent Agents. It has introduced two scenarios for using FIPA modelling TC Agent class superstructure meta-model to model Intelligent Agents: the first scenario extends Agent class while the second one uses the available Agent Physical Classifier to model Intelligent Agents. The second scenario is more appropriate as there will be no need to develop a new Intelligent Agent structures. It introduces a learning compartment to the Agent Physical Classifier that provides a description of the learning behaviour of the Intelligent Agent. This description will guide both designers and developers during both the design and the development phases of the system. The description consists of attributes and notes. The attributes describe the main learning parameters, and the notes provide behaviour description or constraints for the attached attribute. The learning compartment contains the following attributes: *learning-goal, commitment-strategy, learning-strategy, learning-feedback method, background-knowledge, knowledge-representation, and learning-schedule.*

The main objective of using UML 2.0 as a base modelling language for describing Agent-oriented systems is to allow software developers using object-oriented systems to shift more easily towards the development of Agent-oriented systems. Currently, no one can declare that any of the available UML Agent-oriented efforts can act as a standard modelling language or method for Agent-oriented systems as UML for object-oriented systems. No research direction that the author has tackled in this research has explored

how the new UML notations, UML 2.0, can model the intelligence aspects of Agent-oriented systems such as the ability to learn and the ability to reach agreement among Agents.

The research has shown the capability of activity and sequence diagrams to model a learning from observation and discovery strategy. Intelligent Agents deploy mainly in dynamic environments which require learning without gaining advice from a source of knowledge. The source of knowledge is not able to provide advice about the correct actions for all the different states of the dynamic environment. Moreover, activity and sequence diagrams are able to model scenarios for Intelligent Agents that use a learning from examples strategy. Intelligent Agents that work in dynamic environments can use a learning from examples strategy in one of the following conditions:

1) The rate of examples is higher than the rate of changes in the environment or

2) Cloning the Intelligent Agent and then replacing it after learning.

UML 2.0 state machine diagrams have proven resourceful for modelling specific reinforcement learning methods and algorithms. The research shows the capability of these diagrams to model dynamic programming algorithms namely, iterative policy evaluation, policy evaluation, policy improvement, and value iteration algorithms. It also uses the state machine diagrams to model Monte Carlo algorithms, namely policy evaluation and control algorithms. UML 2.0 state machine diagrams are also successful in modelling temporal difference (TD) algorithms, namely TD prediction, Sarsa, Q-learning, Sarsa (λ), and Q(λ) algorithms.

Future information systems will contain objects, Agents, and Intelligent Agent as the main software entities. However, it will be impractical to imagine for the time being that Agent-oriented systems will replace object-oriented ones. The integration of the capabilities for both systems will pave the way for implementing reliable and capable software systems especially in dynamic environments.

6.2 Future Research Directions

This Thesis will stimulate new directions for future research into Agent-oriented systems. The first research direction is expected to concentrate on the Intelligence dimension of Intelligent Agents. The second one focuses on customizing UML 2.0 for the use of modelling Agents and Intelligent Agents. The third direction aims to develop a standard methodology for developing software systems that integrates the capabilities of objects, Agents, and Intelligent Agents.

The first future research direction inspired by this thesis would explore the capability of UML 2.0 in modelling the ability to reach agreement, being a major component of the intelligence dimension. This ability includes negotiation and argumentation skills. This research is expected to shed more light on the ability of UML behaviour components to model the three main types of negotiation: one-to-one, one-to-many, and many-to-many (Wooldridge, M., 2002). It would also sustain the ability of UML to model the different types of argumentation skills: logical, emotional, visceral, and kisceral modes (Wooldridge, M., 2002). There are efforts to model Agent interaction protocols by using UML 2.0, as in FIPA, 2003b, but there is still a need to model the negotiation and argumentation skills exhibited by Intelligent Agents.

The second future research direction would focus on customising UML 2.0 to model Agents and Intelligent Agents components. The proposed research direction would continue to explore the capability of other UML 2.0 components such as state machine diagram, use case diagram, and interaction view diagram to model Agents and Intelligent Agents. This research direction would also investigate the capability of modelling communication patterns among objects, Agents, and Intelligent Agents. The research can customise UML 2.0 to present Agents and Intelligent Agents components profiles.

The third future research direction would focus on producing a software development methodology that is able to model objects, Agents, and Intelligent agents as the main software entities. The methodology would study the requirements of objects,

Agents, and Intelligent Agents during the different phases, such as analysis, design, implementation, and testing phases. The use of UML 2.0 as the modelling language for this methodology would allow the use of one modelling language to model the above mentioned diversified software entities. The development of future software systems, that include objects, Agents, and Intelligent Agents, requires a new standard methodology that is able to cover all the requirements of the software development life cycle.

This thesis has stressed the eligibility of Intelligent Agents as the most appropriate software entities to work in dynamic environments, and provide adaptive behaviour to meet their designated tasks. Thus, the thesis paves the way for an increase in attention to the intelligence dimension of Intelligent Agents. Future research directions will not only concentrate on the agency dimension but will also concentrate on the dimension of intelligence itself.

References

Ambler, S., 2005a. UML 2 Object Diagrams. Available from: http://www.agilemodeling.com/artifacts/objectDiagram.htm [Accessed 2 May 2005]

Ambler, S., 2005b. *UML Class Diagrams*. Available from: http://www.agilemodeling.com/artifacts/classDiagram.htm [Accessed 2 May 2005]

Bauer, B., 2002. UML Class Diagrams Revisited in the context of Agent-Based Systems. *In:* M.J. Wooldridge, G. Weib, and P. Ciancarini, eds. *Agent-Oriented Software Engineering II: Second International Workshop, AOSE 2001, Montreal, Canada May 2001, Revised Paper and Invited Contribution, Lecture Notes in Computer Science 2222.* Berlin: Springer-Verlag, 101-118.

Bauer, B., Muller, J., and Odell, J., 2001. Agent UML: A Formalism for Specifying MultiAgent Interaction Diagrams. *In:* P. Ciancarini and M.J. Wooldridge, eds. *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000, June 2000 Limerick Ireland, Revised Papers, Lecture Notes in Computer Science 1957.* Berlin: Springer-Verlag, 91-103

Bell, D., 2004. *UML Basics: The Class Diagram. An Introduction to Structure Diagrams in UML 2.* Available from: http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/ [Accessed 29 April 2005]

Bigus, J., and Bigus, J., 2001. *Constructing Intelligent Agents Using Java.* 2nd ed. New York: John Wiley & Sons, Inc.

Booch, G., 1994. *Object Oriented Design with Applications.* 2nd ed. Redwood city, CA: The Benjamin/Cummings Publishing Company, Inc.

Brenner, W., Zarnekow, R., and Wittig, H., 1998. *Intelligent Software Agents: Foundations and Applications.* Berlin: Springer-Verlag.

Burmeister, B., 1996. Models and Methodology for Agent-oriented Analysis and Design. *In:* K. Fischer, ed. *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, 17-19 September 1996 Dresden.* Kaiserslautern: DFKI Document, 96-106.

Caglayan, A. And Harrison, C., 1997. *Agent Source Book.* New York: John Wiley & Sons, Inc.

Caire G., Coulier W., Garijo F., Gomez J., Pavon J., Leal F., Chainho P., Kearney P., Stark J., Evans R., and Massonet P., 2002. Agent Oriented Analysis using MESSAGE/UML. *In:* M.J. Wooldridge, G. Weib, and P. Ciancarini, eds. *Agent-Oriented Software Engineering II: Second International Workshop, AOSE 2001, Montreal, Canada May 2001, Revised Paper and Invited Contribution, Lecture Notes in Computer Science 2222.* Berlin: Springer-Verlag, 119-135.

Carbonell, J., 1986. Derivational Analogy: A theory of Reconstructive Problem Solving and Expertise Acquisition. *In:* R. Michalski, J.G. Carbonell, and T. Mitchel, eds. *Machine Learning Volume II: An Artificial Intelligence Approach.* Los Altos, CA: M. Kaufmann Publishers, 371-392.

Carbonell, J., Michalski, R., and Mitchell, T., 1983. *An Overview of Machine Learning.* In: R. Michalski, J.G. Carbonell, and T.M. Mitchell, eds. *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: M. Kaufmann Publishers, 3 – 23.

Coetzee, F., 2005. *UML 2.0.* Xpdian. Available from: http://www.xpdian.com/UML2.0.html [Accessed 5 May 2005]

Depke R., Heckel R., and Malte Kuster J., 2001, Agent-Oriented Modeling with Graph Transformation. *In:* P. Ciancarini and M.J. Wooldridge, eds. *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000, Limerick, Ireland June 2000, Revised Paper, Lecture Notes in Computer Science 1957.* Berlin: Springer-Verlag, 105 – 119.

FIPA, 2003a. *FIPA Modeling: Agent Class Superstructure Metamodel, initial draft for review at London FIPA.* Available from: http://www.auml.org/auml/documents/CD2-03-10-31.doc [Accessed 13 Sept 2004].

FIPA, 2003b. *FIPA modelling: Interaction Diagrams, Version 2003-07-02.* Available from: http://www.auml.org/auml/documents/ID-03-07-02.doc [Accessed 13 September 2004].

FIPA, 2004. FIPA Modeling: Agent Class Superstructure Metamodel, The result of London FIPA meeting and interim work 21 January 2004 version. Available from: http://www.auml.org/auml/documents/CD2-04-01-21.doc [Accessed 13 September 2004].

Flores-Mendez, R., 1999. Towards a Standardization of Multi-Agent System Frameworks *Cross Roads*, 5 (4).

Fowler, M., 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd ed. Boston: Addison-Wesley.

Greneserth, M. and Ketchpel, S., 1994. Software Agents. Communication of the American Computing Machinery, 37 (7), 48- 53.

Holzmann G., 1997. The Model Checker Spin. IEEE Transactions on Software Engineering, 23 (5), 279-295.

Huget M., 2003. Extending Agent UML Protocol Diagram. *In:* F. Giunchiglia, J. Odell, and G. Weid, eds. *Agent-Oriented Software Engineering III: Third International Workshop, ASOE 2002, Bologna, Italy, July 2002, Revised Papers and Invited Contributions, Lecture Notes in Computer Science 2585.* Berlin: Springer-Verlag, 150- 161.

Koning J. and Romero-Hernandez, I., 2003. Generating Machine Processable Representations of Textual Representations of AUML. *In:* F. Giunchiglia, J. Odell, and G. Weid, eds. *Agent-Oriented Software Engineering III: Third International Workshop, ASOE 2002, Bologna, Italy, July 2002, Revised Papers and Invited*

Contributions, Lecture Notes in Computer Science 2585. Berlin: Springer-Verlag, 126 – 137.

Lemon, B., Pynadath, D., Taylor, G., and Wray, T., 2004. *Cognitive Architectures,* Available from: http://ai.eecs.umich.edu/cogarch4/index.html [Accessed April 2004]

Lind J., 2002. Specifying Agent Interaction Protocols with standard UML. *In:* M.J. Wooldridge, G. Weib, and P. Ciancarini, eds. *Agent-Oriented Software Engineering II: Second International Workshop, AOSE 2001, Montreal, Canada May 2001, Revised Paper and Invited Contribution, Lecture Notes in Computer Science 2222.* Berlin: Springer-Verlag, 136 -147.

Luger, G. and Stubblefield, W., 1993. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* 2nd ed. Palo Alto, CA: The Benjamin/Cummings Publishing Company, Inc.

Michalski, R., 1986. Understanding the Nature of Learning: Issues and Research Directions. *In:* R. Michalski, J.G. Carbonell, T. Mitchel, eds. *Machine Learning Volume II: An Artificial Intelligence Approach.* Los Altos, CA: M. Kaufmann Publishers, 3-25

Michalski, R., 1993. Basic concepts of Inferential Theory of Learning and Their Use for Classifying the Learning Process. *In:* A. Meyrowitz and S. Chipman, eds. *Foundations of Knowledge Acquisition Machine Learning.* Boston: Kluwer Academic Publishers, 1- 41.

Michalski. R., 1994. Inferential Theory of Learning: Developing Foundations for Multistrategy Learning. *In:* R. Michalski and G. Tecuci, eds. *Machine Learning: A Multistrategy Approach Volume IV.* San Francisco: Morgan Kaufmann Publishers, 3-61.

Michalski, R., Carbonell, J., and Mitchell, T., 1983. *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: M. Kaufmann Publishers.

Michalski, R., Carbonell, J., and Mitchell, T., 1986. *Machine Learning: An Artificial Intelligence Approach, Volume II.* Los Altos, CA: M. Kaufmann Publishers.

Mitchell, *T., 1997. Machine Learning. Boston: WCB/McGraw-Hill.*

Object Management Group (OMG), 2004. *Unified Modelling Language: Superstructure version 2.0, Revised Final Specification (ptc/04-10-02).* Available from http://www.omg.org/cgi-bin/doc?ptc/2003-08-02 [19 [Accessed 27 April 2005]

Object Management Group (OMG), 2005. *Introduction to OMG's Unified Modeling Language™ (UML®). Available from: http://www.uml.org [Accessed 27 April 2005]*

Odell, J., 2002. *Objects and Agents Compared. Journal of Object Technology,* 1(1), 41-53. Available from: http://www.jot.fm/issues/issue_2002_05/column4 [Accessed 13 September 2004].

Odell, J., Parunak, H., and Bauer, B., 2000. Extending UML for Agents. *In:* G. Wagner, Y. Lesperance, and E. Yu, eds. *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, July 30 – August 3 2000, Austin, TX, AOIS Workshop at AAAI 2000.* Menlo Park, California: AAAI, 3-17.

Odell, J., Parunk, H., and Bauer, B., 2001. Representing Agent Interaction Protocols in UML. . *In:* P. Ciancarini and M.J. Wooldridge, eds. *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000, Limerick, Ireland June 2000, Revised Paper, Lecture Notes in Computer Science 1957.* Berlin: Springer-Verlag, 121-140

Petrie, C. 1996. Agent-based Engineering, the web, and intelligence. *IEEE Expert,* 11 (6), 24-29.

Russell, S. and Norvig, P., 1995. *Artificial Intelligence: A Modern Approach.* New Jersey: Prentice Hall International.

Schank, R. and Kass, A., 2000. *Explanations, Machine Learning, and Creativity. In:* Y. Kodtatoff and R. Michalski, eds. *Machine Learning: An Artificial Intelligence Approach Volume III.* Palo Alto, CA: Morgan Kufmann, 31 – 48.

Sein, S. and Weiss, G., 1999. Learning in MultiAgent Systems. *In:* G. Weiss, ed. *MultiAgent Systems: A Modern Approach to Distributed Artificial Intelligence.* Cambridge, MA: MIT Press, 259 – 298.

Smith, S., 1996. *Machine Learning Review* [online]. Middlesex University, School of Computing Science. Available from: http://www.cs.mdx.ac.uk/staffpages/serengul/ML/table.html [Accessed 13 September 2004].

Sparx Systems, 2005a. UML 2 Activity Diagram. Available from: http://sparxsystems.com.au/resources/uml2_tutorial/uml2_activitydiagram.html [Accessed 15 July 2005].

Sparx Systems, 2005b. UML 2 Sequence Diagram. Available from: http://sparxsystems.com.au/resources/uml2_tutorial/uml2_statediagram.html [Accessed 15 July 2005].

Sparx Systems, 2005c. UML 2 State Machine Diagram. Available from: http://sparxsystems.com.au/resources/uml2_tutorial/uml2_statediagram.html [Accessed 15 July 2005].

Sutton, R. and Barto, A., 1998. *Reinforcement Learning: An introduction.* Cambridge, MA: MIT Press.

Wagner G., 2003a. *A UML Profile for External AOR Models. . In:* F. Giunchiglia, J. Odell, and G. Weid, eds. *Agent-Oriented Software Engineering III: Third International Workshop, ASOE 2002, Bologna, Italy, July 2002, Revised Papers and Invited Contributions, Lecture Notes in Computer Science 2585.* Berlin: Springer-Verlag, 138-149.

Wagner G., 2003b. The Agent-object-Relationship MetaModel: Towards a Unified View of State and Behavior. *Information Systems*, 28 (5).

Wooldridge, M., 2000. *Reasoning About Rational Agents.* Cambridge, MA:MIT Press.

Wooldridge, M., 2002. *An Introduction to MultiAgent Systems.* Chichester, England: John Wiley & Sons Ltd

Wooldridge, M., and Cinancaini, P., 2001. *Agent-Oriented Software Engineering: The State of the Art. In:* P. Ciancarini and M.J. Wooldridge, eds. *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000, June 2000 Limerick Ireland, Revised Papers, Lecture Notes in Computer Science 1957.* Berlin: Springer-Verlag, 1–28.

Wooldridge, M., Jennings, N., and Kinny, D., 2000. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3 (3), Kluwer Academic Publisher, 285-312.

Further Reading

Adida, B., 1997. Weaving the Web. *IEEE Internet Computing*, 1 (4), 91-93.

Allam, H., Denham, M., and Denham, S., 2001. Framework for Assessing Intelligent Agent-based Information Systems Methodologies. *In*: *Proceeding of the Ninth International Conference on Artificial Intelligence Applications, Feb 12-15 2001, Egypt.* Cairo: AUC, 135-142.

Anderoli, J., Pacull, F. and Pareschi, R., 1997. XPECT: A framework for electronic commerce. *IEEE Internet Computing,* 1 (4), 40-48.

Avison, D., and Fitzgerald, G., 1998. *Information Systems Development: Methodologies, Techniques, and Tools.* 2nd ed. Maidenhead: McGraw-Hill Book Company Europe.

Bergenti, F., and Poggi, A., 2000. Exploiting UML in the Design of Multi-Agent Systems, *In:* A. Omicini, R. Tolksdorf and F. Zambonelli, eds. *Engineering Societies in the Agent World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000, Revised Papers*, Lecture Notes in Computer Science, 1972. Berlin: Springer, 96-103.

Booch, G., 1994. *Object Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, CA: The Benjamin/Cummings Publishing company, Inc.

Booch, G., Rumbaugh, J., and Jackobson, I., 1999. *The Unified Modeling Language User Guide*. Reading, MA:Addison-Wesley.

Chavez, A., Moukas, A., and Maes, P., 1997, Challenger: A Multi-Agent System for Distributed Resources Allocation. *In: Agents 97: Proceedings of the first International Conference on Autonomous Agents, Feb. 5-8, 1997, Marina del Rey, CA*. Association for computing machinery (ACM), 323 – 330.

Choi, Y. and Yoo, S., 1997. Multi-Agent Learning Approach to WWW Information Retrieval using Neural Network. *In: Agents 97: Proceedings of the first International Conference on Autonomous Agents, Feb. 5-8, 1997, Marina del Rey, CA*. Association for Computing Machinery (ACM), 23- 30.

Coad, P., and Yourdon, E., 1991. *Object-Oriented Analysis*. 2nd ed. NJ: Prentice-Hall,Inc.

Cooling, J., 1997. *Real-time Software Systems: An introduction to structured and object-oriented design.* London: International Thomson Computer Press.

Dean, T., Allan, J., and Aloimonos, Y., 1995. *Artificial Intelligence: Theory and Practice.* New York: The Benjamin/Cummings Publishing Company, Inc.

Doyle, J,. and Dean, T., 1996. Strategic Directions in Artificial Intelligence. *ACM Computing Survey*, 28 (4), 653-670.

Ernest, E., Candy, L., Jones, R., and Soufi, B. 1994. Support for Collaborative Design: Agents and Emergence. *Communication of ACM* (7), 41-47.

Etzioni O. and Weld D. 1994. A Softbot-Based Interface to the Internet. *Communication of ACM*, 37 (7), 72-76.

Ferber, J., 1999. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence.* Harlow, England: Addison-Wesley.

Filman, R. and Pena-Mora, F. 1997. The Arachnoid tourist. *IEEE Internet Computing*, 1 (4), 31-32.

Finin, T., Fritzson, R., Mckay, D., and McEntire, R., 1994. KQML as an Agent Communication Language. *In: CIKM 94.Proceedings of the Third International Conference on Information and Knowledge Management, November 29 – December 2 Maryland, USA.* New York: ACM, 456-463.

FIPA Technical Committee C, 1999. Extending UML for the Specification of Agent Interaction Protocols, Response to the OMG Analysis and Design Task Force UML RTF 2.0 Request for Information. Available from: ftp://ftp.omg.org/pub/docs/ad/99-12-03.pdf [Accessed 13 September 2004].

Flake, S., Geiger, C., and Kuster, J., 2001. Towards UML-based Analysis and Design of Multi-Agent Systems. *In: International Symposium on Information Science Innovations in Engineering of Natural and Artificial Systems (ENAIS 2001), March 2000, Dubai, UAE.*

Foner, L., 1997. Yenta: A Multi-Agent, Referral-based Matchmaking System. *In: AGENTS '97. Proceedings of the first international conference on Autonomous Agents*, February 5-8, 1997, Marina del Rey, CA. NY: Association for Computing Machinery (ACM), 301 – 307.

Forsyth, R., 1989. *Machine Learning: Principles and Techniques.* London: Chapman and Hall Ltd.

Gather, L.1997. Internet security: Is it in the cards. *Computer* (5), 18-20.

Goldman, C., and Rosenschein, J., 1999. Partitioned MultiAgent Systems in Information Oriented Domains. *In: Agents '99, Proceedings of the third annual conference on Autonomous Agents, May 1-5, 1999, Seattle, WA.* NY: Association for Computing Machinery (ACM), 32-39.

Graham, I., 1994. *Object Oriented Methods.* 2nd ed. Wokingham, England: Addison-Wesley Publishing Company.

Hawryszkiewycz, I.T., 1991. *Introduction to System Analysis and Design.* 2nd ed. Sydney: Prentice Hall.

Huber, M., and Hardley, T., 1997. Multiple Roles, Multiple Teams, Dynamic Environment: Autonomous Netrek Agents. *In*: *AGENTS '97. Proceedings of the first international conference on Autonomous Agents,* February 5-8, 1997, Marina del Rey, CA. NY: Association for Computing Machinery (ACM), 332-339.

Iglasis, C., Garji. M., Gonzalez, J., and Velasco, J., 1996. A methodological Proposal for MultiAgent Systems Deveeding extending CommonKADS. *In: Proceeding of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshops, Banff, Canada, 25-1/17.* Available from: http://ksi.cpsc.ucalgary.ca/KAW/KAW96/iglesias/Iglesias.html [Accessed 19 September 2004]

Janca, P., and Gilbert, D., 1998. Practical Design of Intelligent Agent Systems. *In:* N. Jennings and M. Wooldridge, ed. *Agent Technology: Foundations, Applications, and Markets*, Berlin: Springer-Verlag, 73-89.

Jennings, N., and Wooldridge, M., 2000. Agent-Oriented Software Engineering. *In:* J. Bradshaw, ed. *Handbook of Agent Technology*. MA: AAAI/MIT press.

Karjoth, G., Lange, D., and Oshima, M. 1997. A security model for AGLETS. *IEEE Internet Computing*, 1 (4), 68-77.

Kautz H., Selman, B., and Coen M., 1994. Bottom-up design of software Agents. *Communication of ACM*, 37 (7), 143-146.

Khoshafian, S. and Abnous, A., 1990. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. New York: Wiley.

Kiniry, J. and Zimmerman, D., 1997. Special Feature: A hands-on Look at Java Mobile Agents. *IEEE Internet Computing*, 1 (4), 21-30.

Kinny, D., Georgeff, M., and Rao, A., 1996. A Methodology and Modelling Technique for systems of BDI-Agents. *In*: W. Van de Velde and J. W. Perram, ed. *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a MultiAgent World*, LNAI Volume 1038, Berlin, Germany: Springer-Verlag, 56--71.

Kotz, D., Gray, R., Nog, S., Rus, D., Chawila, S., and Cybenko, G. 1997. Agent TCL: Targeting the needs of mobile computers. *IEEE Internet computing*, 1 (4), 58-67.

Krulwich, B. 1997. Automating the Internet: Agents as user surrogates. *IEEE Internet Computing*, 1 (4), 34-37.

Labrou, Y. and Finin, T., 1994. A semantics approach for KQML--a general-purpose communication language for software Agents. *In: CIKM 94: Proceedings of the Third International Conference on Information and Knowledge Management*, New Your: ACM SIGMOD Anthology, 447-455.

Lucarella, D. and Zanzi, A., 1996. A Visual Retrieval Environment for Hypermedia Information Systems. *ACM transaction on Information Systems,* (1), 3-29.

Maes, P. 1994. Agents that reduce work and information overload. *Communication of the ACM,* 37 (7), 31-40.

Maes, P. 1997. Patteis Maes on software Agents: Humanizing the Global Computer. *IEEE Internet Computing,* 1 (4), 10-19.

Mahalingam, K. and Huhns, N. 1997. A Tool for Organizing Web Information. *IEEE Computer*, 30 (6), 80-83.

Malherio, B. and Oliveria, E., 1997. Environmental Decision Support: a Multi-Agent Approach. *In: AGENTS '97. Proceedings of the first international conference on Autonomous Agents, February 5-8, 1997 Marina del Rey, CA*. NY: Association for Computing Machinery (ACM), 540 - 541.

Marsella, S., Adibi, J., Al-Onaizan, Y., Kaminka, G., Muslea, I., Tambe, M., 1999. On being a teammate: Experiences acquired in the design of RoboCup teams. *In: Agents '99, Proceedings of the third annual conference on Autonomous Agents, May 1-5, 1999, Seattle, WA, USA.* NY: Association for Computing Machinery (ACM), 221-227.

Marshall, A.D., 2001, AI2 Courseware [Online], Computer Science Department, Cardiff University. Available from http://www.cs.cf.ac.uk/Dave/AI2/AI_notes.html [Accessed 14 September 2004].

Martin, J. and Odell, J., 1992. *Object-oriented Analysis and Design.* NJ, USA: Prentice Hall.

Mason, C. and Matwin S. 1995. Environmental Applications of AI. *IEEE Expert*, 10 (6), 12-13.

Meyrowitz, A. and Chipman, S., 1993. *Foundations of Knowledge Acquisition Machine Learning.* Boston: Kluwer Academic Publishers.

Michalski, R., 1986. Understanding the Nature of Learning: Issues ad Research Directions. *In:* Y. Kodtatoff and R. Michalski, eds. *Machine Learning: An Artificial Intelligence Approach Volume III.* Palo Alto, CA: Morgan Kufmann,P 3 –25.

Milewski, A. and Lewis, S. 1997. Delegating to software Agents. *International Journal for Human-Computer Studies*, 46 (4), 485-500.

Muller, Jorg P., 1996. The Design of Intelligent Agents: a Layered Approach. *Lecture Notes in Artificial Intelligence,* 1177. Springer-verlag,

Norman, D. 1994. How might people interact with Agents. *Communication of the ACM,* 37 (7), 68-71.

Object Management Group, 2001. *OMG Unified Modeling Language Specification Version 1.5* [Online]. OMG. Available From http://www.omg.org/cgi-bin/doc?formal/03-03-01 [Accessed 19 September 2004]

Object Management Group, 2003. *UML 2.0 Superstructure Specification* [Online]. OMG. Available from http://www.omg.org/cgi-bin/doc?ptc/2003-08-02 [19 September 2004]

Ohsuga, A., Nagai, Y., Irie, Y., Hattori, M., and Honiden, S., 1997. Plangent: An approach to making mobile Agents intelligent. *IEEE Internet Computing,* 1 (4), 50-57.

Plu, M., 1998. Software Technologies for building Agent Based Systems in Telecommunications. *In:* N. Jennings and M. Wooldridge, ed. *Agent Technology: Foundations, Applications, and Markets*, Berlin: Springer-Verlag, 241-266.

Quinlan, J., 1986, The Effect of Noise on concept learning. *In:* R. Michalski, J.G. Carbonell, T. Mitchel, eds. *Machine Learning Volume II: An Artificial Intelligence Approach.* Los Altos, CA: M. Kaufmann Publishers 149 – 166.

Ram, P. and Abarbanel, R., 1997. Enterprise computing: The Java factor. *IEEE Computer,* 30 (6), 115-117.

Rao, S., and Georgeff, P., 1991. Modeling Rational Agents within A BDI- Architecture. *In:* J. Allen, R. Fikes, and E. Sandewall, ed. *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91),* San Mato, CA: Morgan Kaufmann, 473--484.

Rao, S., and Georgeff, P., 1996, BDI-Agents: From Theory to Practice. *In*: V. Lesser and L. Gasser, *eds. Proceedings of the First International Conference on Multi-Agent-System (ICMAS '95), 12 – 14 June 1995 San Francisco*, CA: AAAI, 312-319.

Riecken, D. 1994. Intelligent Agents. *Communication of ACM*, 37 (7), 18-21.

Rowe, A. and Davis, S., 1996. *Intelligent Information Systems: Meeting the Challenge of the Knowledge Era.* West Port: Quorum Books.

Rumbaugh, J., Jacobson, I., and Booch, G., 1999. *The Unified Modeling Language Reference Manual.* Reading: Addison-Wesley.

Rus, D. and Subramanina, D. 1997. Customizing Information Captures and Access. *ACM Transaction on Information Systems*, 15 (1), 67-101.

Russell, S., 1996. Chapter 4: Machine Learning. *In:* M. A. Boden Ed. *Handbook of Perception and Cognition volume 14.* Artificial Intelligence, Academic Press.

Sankaranarayanan, A. and Mataric, M., 1997. The Multi-Agent-Based Calculator (MASC) System. *In: Agents '98, Proceedings of the second annual conference on Autonomous Agents, May 9-13, 1998, Minneapolis, MN, USA.* New York: Association for Computing Machinery (ACM), 465-466.

Singh, M., and Huhns, M., 1996. Internet-Based Agents: Applications and Infrastructure. *IEEE Internet Computing,* 1 (4), 8-9.

Subrahmanian, V., Chen, S., Hendler, J., Hull, R., and Tannen, V. 1996. Smart mediators and Intelligent Agents (panel). *In: CIKM 96. Proceeding of the Fifth International Conference on Information and Knowledge Management, Rockville, Maryland, USA, November 12-16, 1996.* New York: ACM, 343.

Taylor, D., 1998. *Object Technology: A Manager's Guide*. 2nd ed. Reading, MA: Addison-Wesley.

Thomas, G. and Fischer, G. 1997. Using Agents to Personalize the web. *In: IUI 97. Proceedings of the 1997 International Conference on Intelligent User Interface, 6th – 9th Jan 1997, Orlando, Florida*. New York: ACM, 53-60.

Thrun, S. and Pratt, L., 1998. *Learning to Learn*. Boston: Kluwer Academic Publishers.

Toomey, C. and Mark, W., 1995. Satellite Image Dissemination via software Agents. *IEEE Expert*, 10 (5), 44 -51.

Wasserman, A.I., Pircher, P.A. and Muller, R.J., 1990. The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer*, 23 (3), 50-62.

Weiß, G., 1996. Adaptation and Learning in Multi-Agent Systems: Some Remarks and Bibliography. *In:* G. Weiß & S. Sein eds. *Adaption and Learning in Multi-agent Systems. Lecture Notes in Artificial Intelligence, Vol. 1042.* Berlin: Springer-Verlag, 1-21.

Whitehead, S., Karlsson, J., and Teneberg, J., 1993. Learning Multiple Goal Behaviour Via Task Decomposition and Dynamic Policy Merging. *In:* Connell and Mahadevan, ed. *Robot Learning,* Norwell, MA: Kluwer Academic Publishers, 45-78.

Winston, H., 1992. *Artificial Intelligence*. 3rd ed. Boston: Addison-Wesley Publishing Company.

Wong, S. 1994. Preference-based decision making for cooperative knowledge-based systems. *ACM Transaction on Information Systems,* 12 (4), 407-435.

Wooldridge, M., 1999. Intelligent Agent. *In:* G. Weiss, ed. *Multi-Agent Systems: A Modern Approach to Distributed Artificial Intelligence*. MA: MIT Press, 1 – 77.

Appendix A: UML 2.0 Activity Diagram User Guide

Activity diagrams are useful in modelling the workflow of a process conducted interactively between the Intelligent Agent and other players such as users, agents, or environment. This user guide has three sections: the first section is a simple guide that introduces the major components of the Activity diagram to the researcher of Agent learning field. The second section provides a step-by-step guide to sketch a simple activity diagram. The third section shows a table of all the major components of UML 2.0 activity diagram. The following steps guide the researcher in agent learning to use UML 2.0 activity diagrams to model the learning behaviour of Intelligent Agents:

Section A.1: Activity Diagram Components Guide

This section is extracted from the Sparx Systems, 2005a. In UML an activity diagram is used to display the sequence of activities. Activity Diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity Diagrams are useful for Business Modelling where they are used for detailing the processes involved in business activities.

The below diagram shows an example of an Activity Diagram.

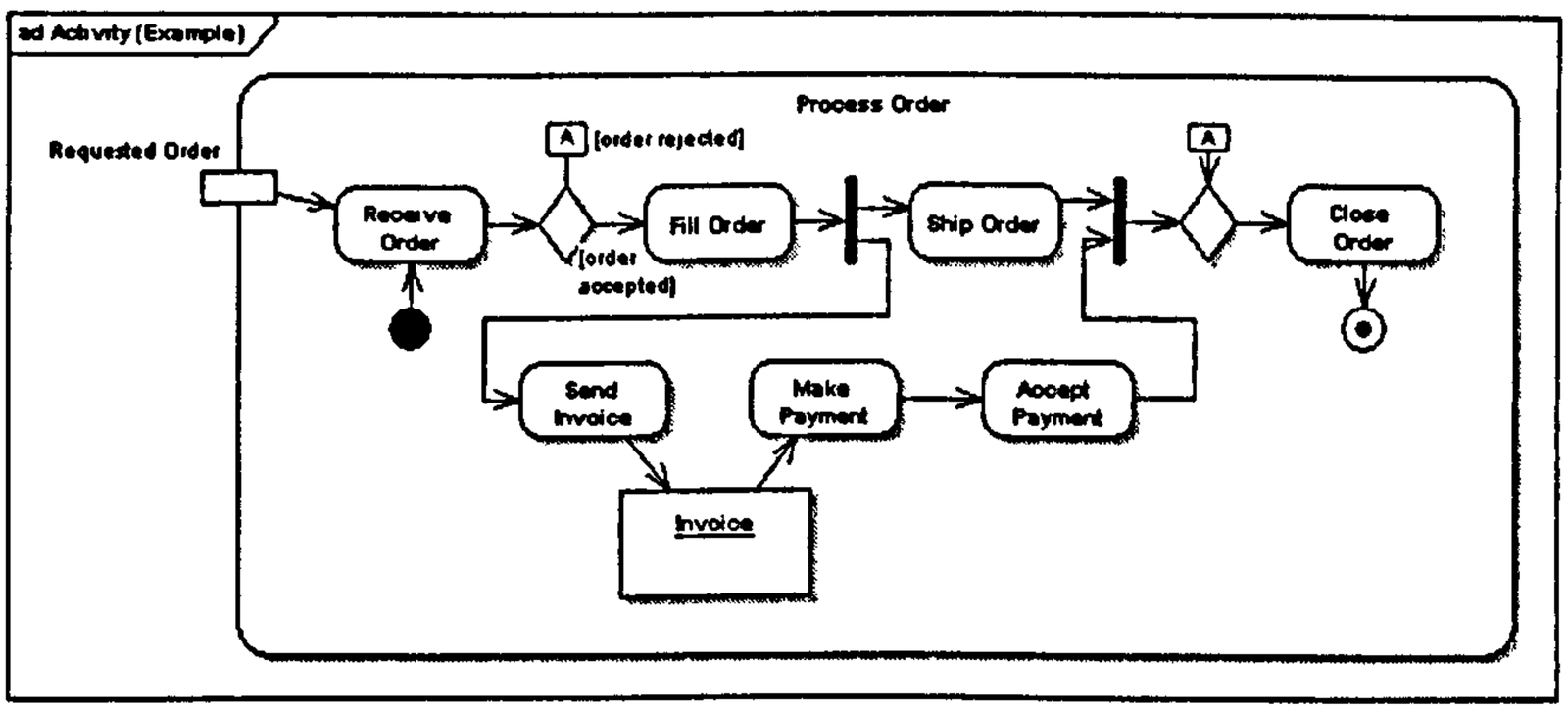


Figure A. 0.1 Example of activity diagram

The following sections describe the elements that constitute an Activity diagram.

Activities

An activity is the specification of a parameterized sequence of behaviour. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity.

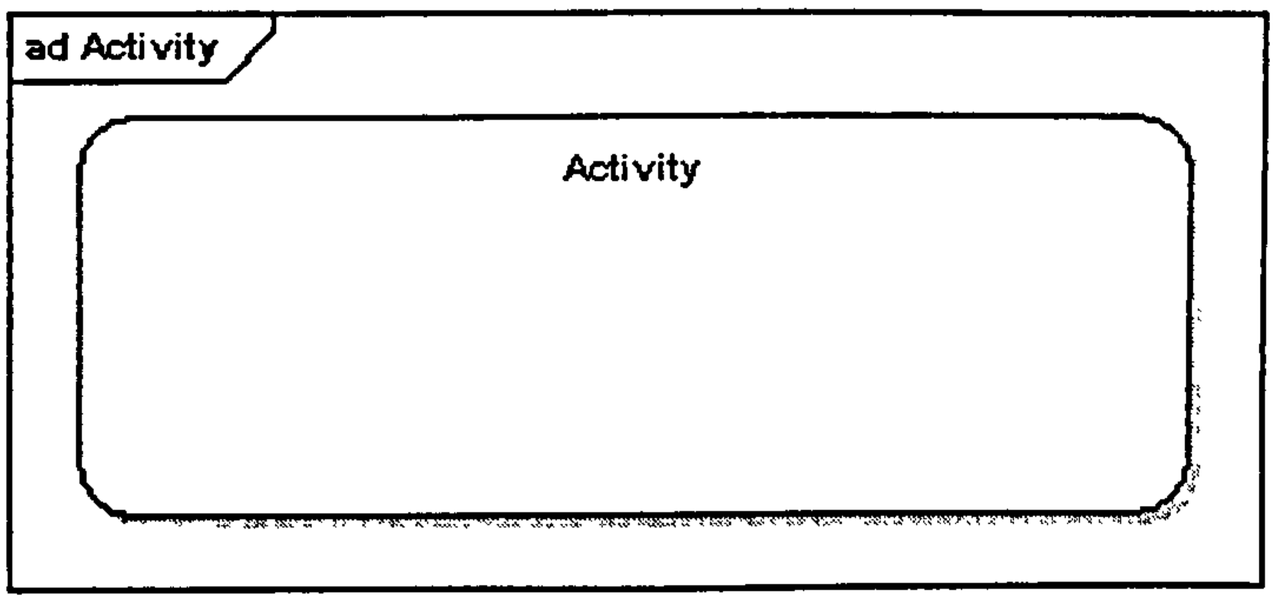


Figure A.0.2 illustration of activity

Actions

An action represents a single step within an activity. Actions are denoted by round-cornered rectangles.

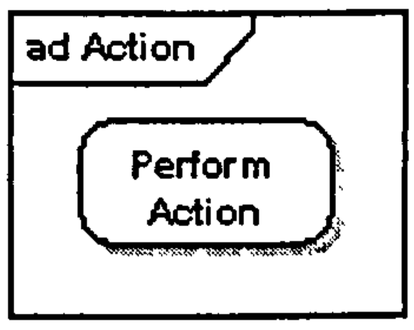


Figure A.0.3 Illustration of Action

Action Constraints

Constraints can be attached to an action. The following diagram shows an action with local pre- and post-conditions.

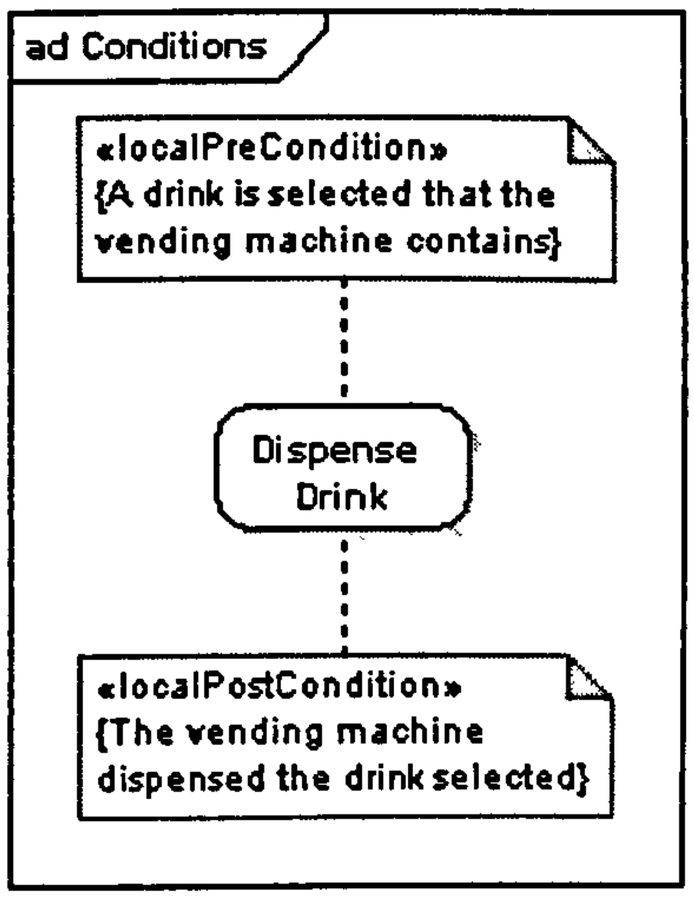


Figure A.0.4 Example of action constraints

Control Flow

A control flow shows the flow of control from one action to the next. Its notation is a line with an arrowhead.

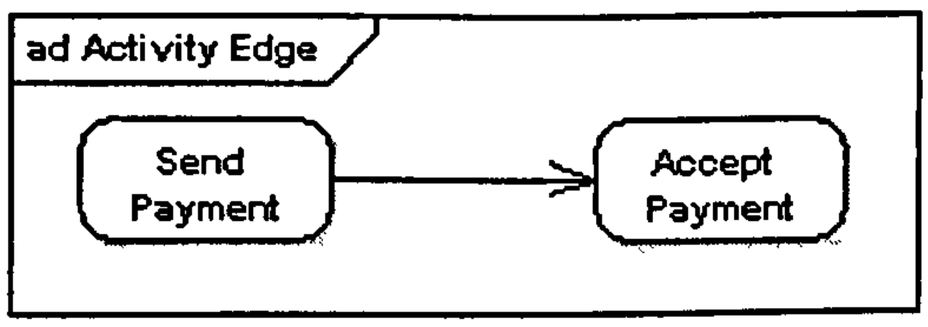


Figure A.0.5 Illustration of control flow

Initial Node

An initial or start node is depicted by a large black spot, as depicted below.

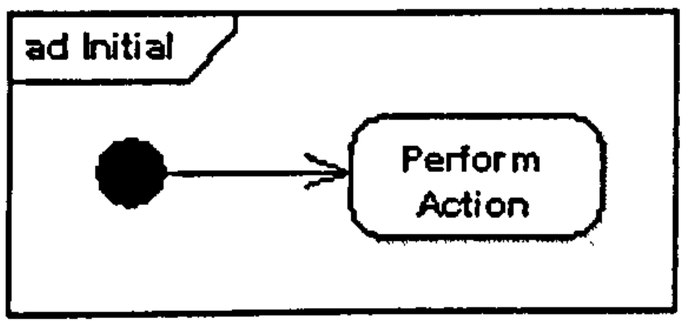


Figure A.0.6 Illustration of initial node

Final Node

There are two types of final node: activity and flow final nodes. The activity final node is depicted as a circle with a dot inside.

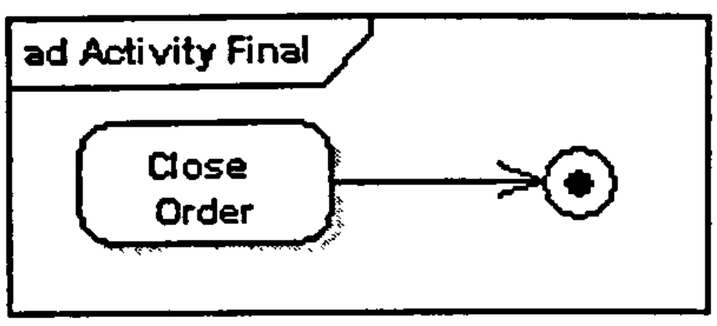


Figure A.0.7 Illustration of final node

The flow final node is depicted as a circle with a cross inside.

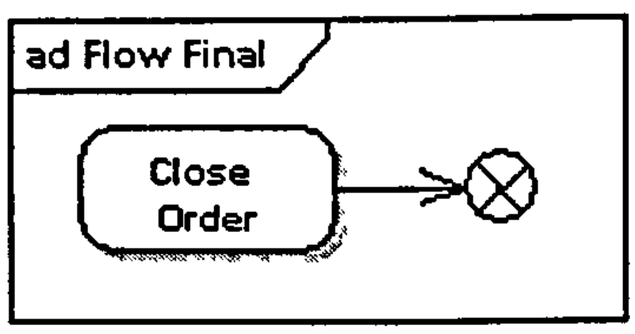


Figure A.0.8 Illustration of flow final node

The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.

Objects and Object Flows

An object flow is a path along which objects or data can pass. An object is shown as a rectangle.

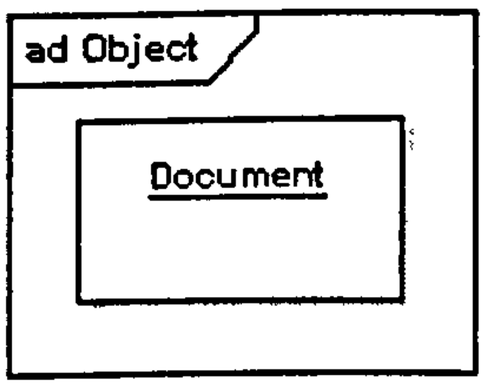


Figure A.0.9 Example of an object

An object flow is shown as a connector with an arrowhead denoting the direction in which the object is being passed.

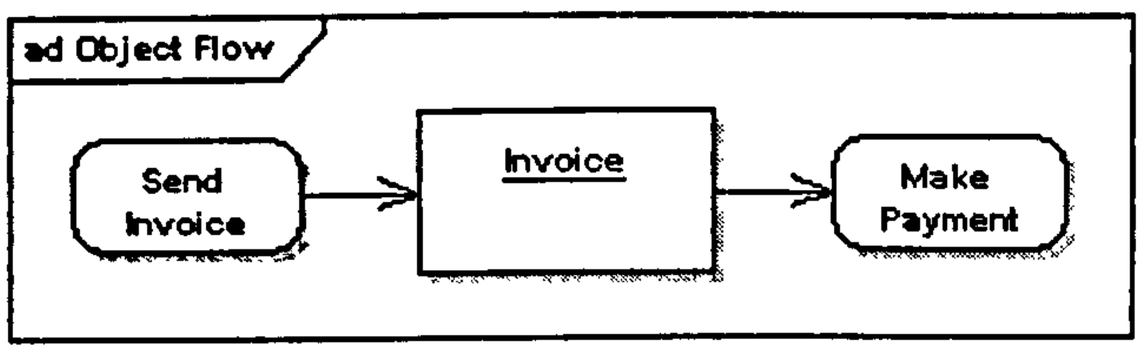


Figure A.0.10 Object flow

An object flow must have an object on at least one of its ends. A shorthand notation for the above diagram would be to use input and output pins.

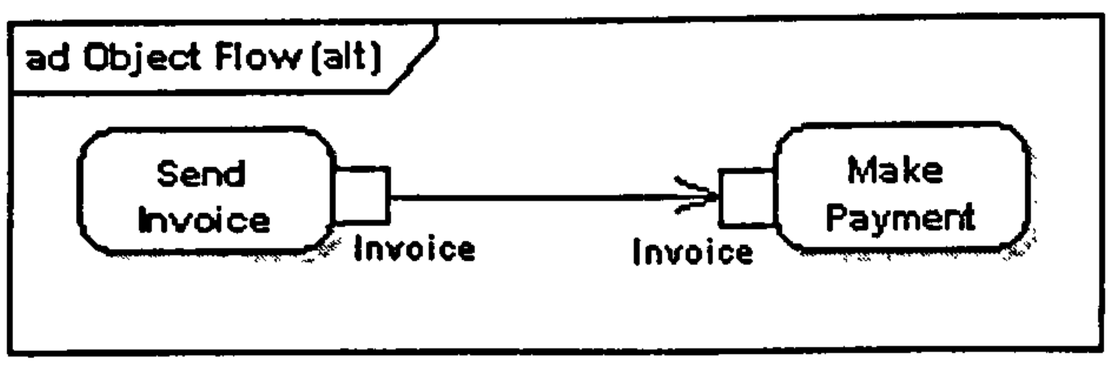


Figure A.0.11 Another illustration of object flow

A data store is shown as an object with the «datastore» keyword.

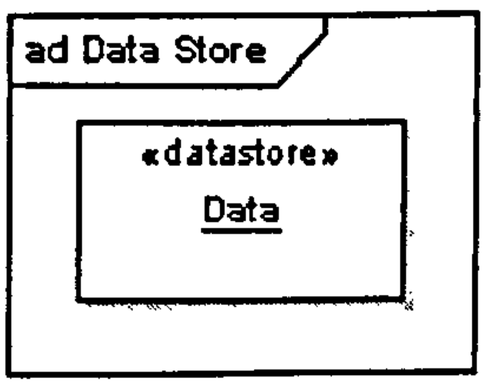


Figure A.0.12 Data store example

Decision and Merge Nodes

Decision nodes and merge nodes have the same notation: a diamond shape. They can both be named. The control flows coming away from a decision node will have guard conditions which will allow control to flow if the guard condition is met. The following diagram shows use of a decision node and a merge node.

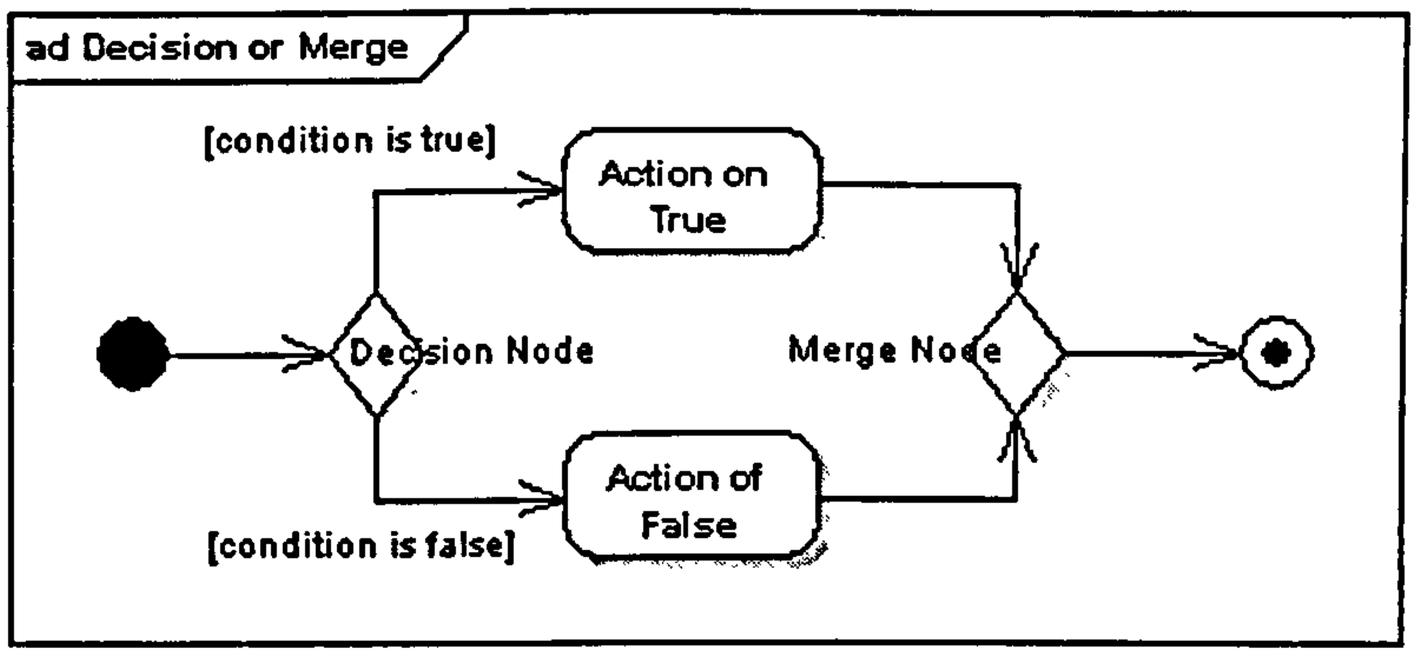


Figure A.0.13 Example of decision and merge nodes

Fork and Join Nodes

Forks and joins have the same notation: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. The following diagram shows an example of their use.

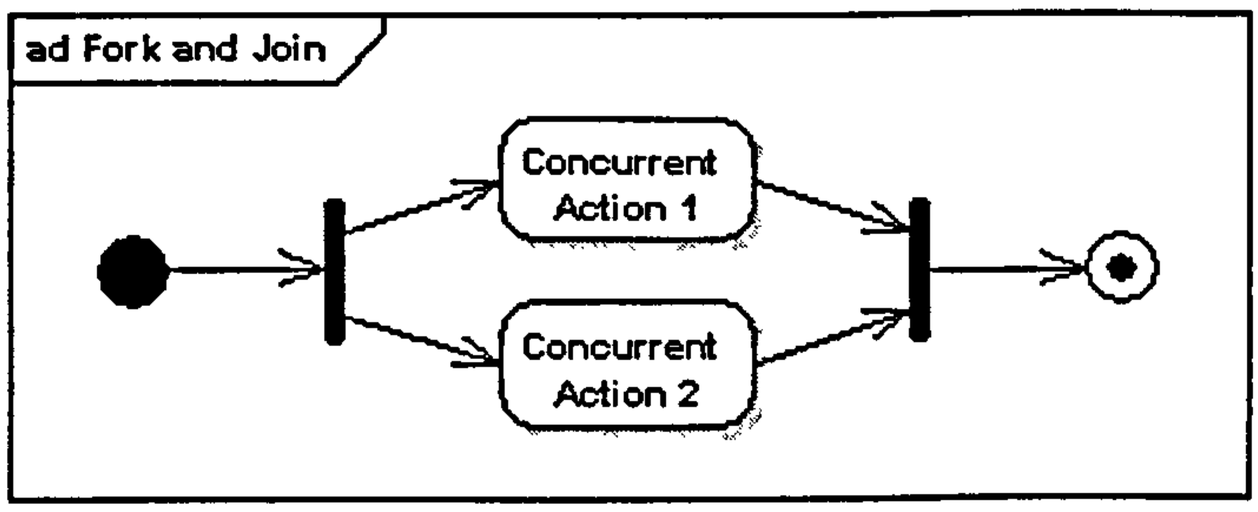


Figure A.0.14 Example of Fork and Join nodes

A join is different from a merge in that the join synchronises two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received. A merge passes any control flows straight through it. If two or more inflows are received by a merge symbol, the action pointed to by its outflow is executed two or more times.

Expansion Region

An expansion region is a structured activity region that executes multiple times. Input and output expansion nodes are drawn as a group of three boxes representing a multiple selection of items. The keyword iterative, parallel or stream is shown in the top left corner of the region.

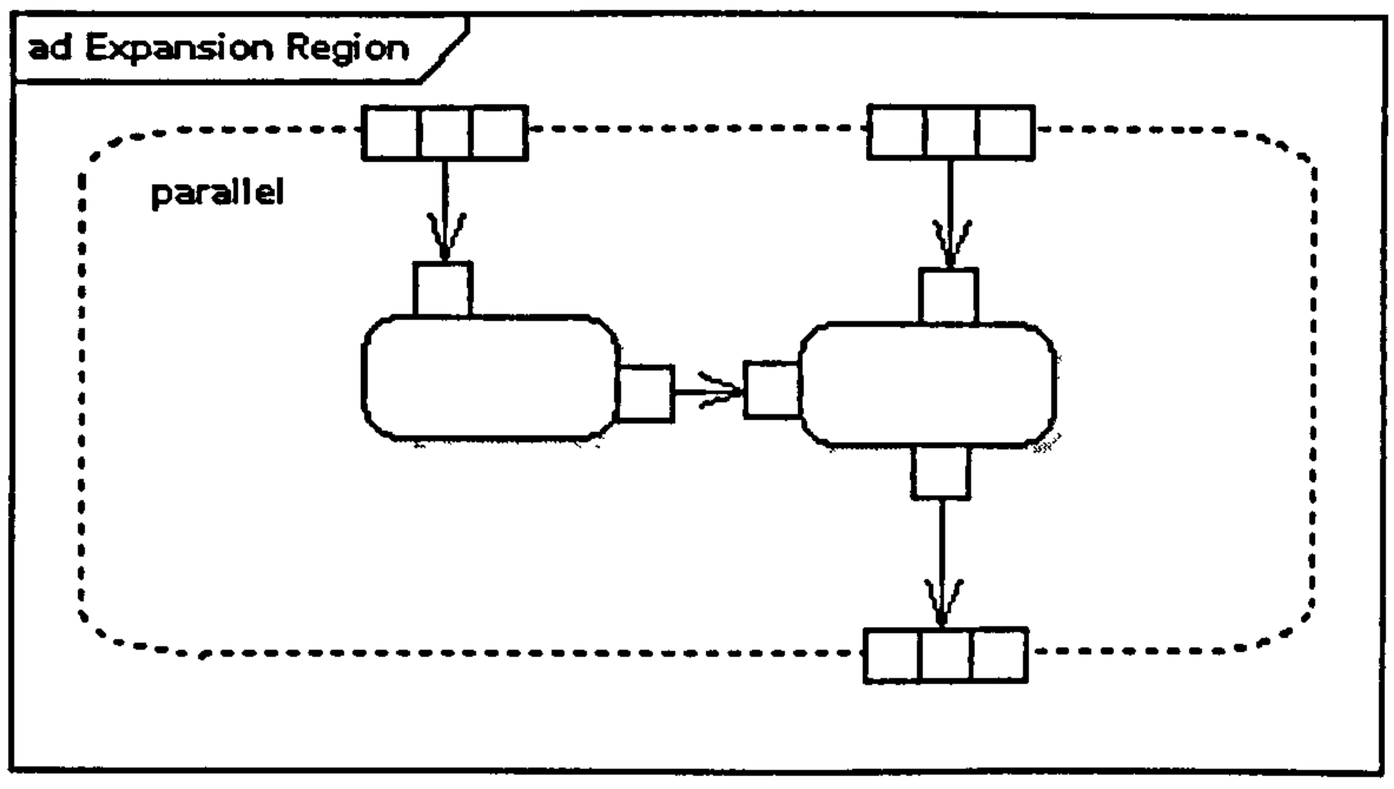


Figure A.0.15 Example of expansion region

Exception Handlers

Exception Handlers can be modelled on activity diagrams as in the example below.

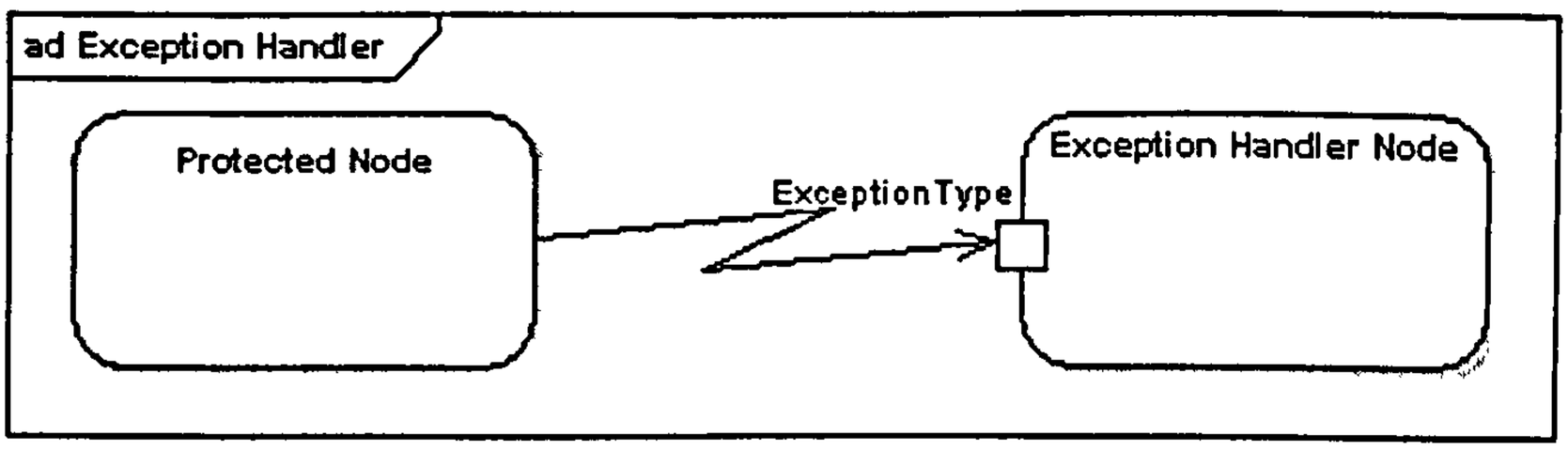


Figure A.0.16 Illustration of exception

Interruptible Activity Region

An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, the Process Order action will execute until completion, when it will pass control to the Close Order action, unless a Cancel Request interrupt is received which will pass control to the Cancel Order action.

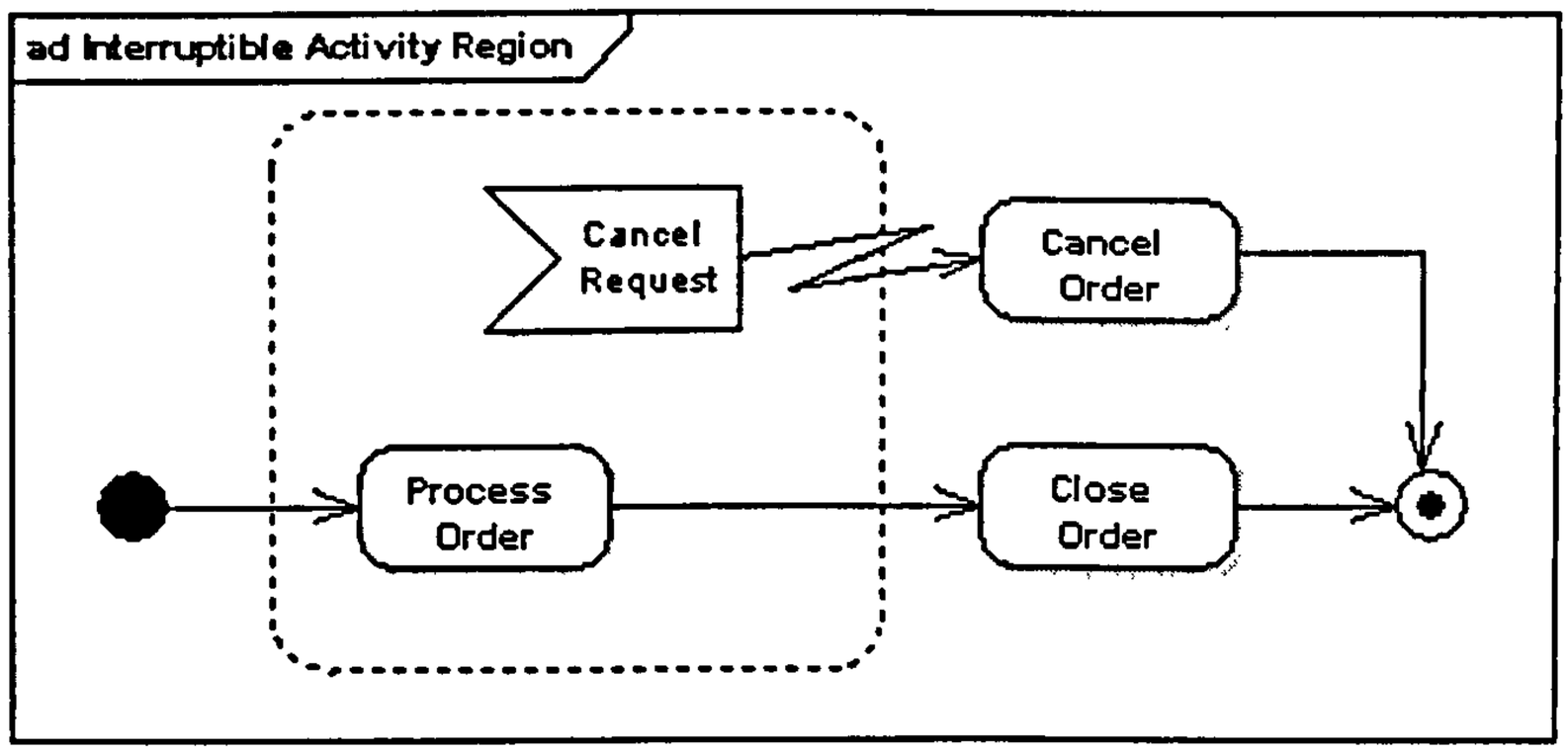


Figure A.0.17 Example of interruptible activity region

Partition

An activity partition is shown as either horizontal or vertical swimlanes. In the following diagram, the partitions are used to separate actions within an activity into those performed by the accounting department and those performed by the customer.

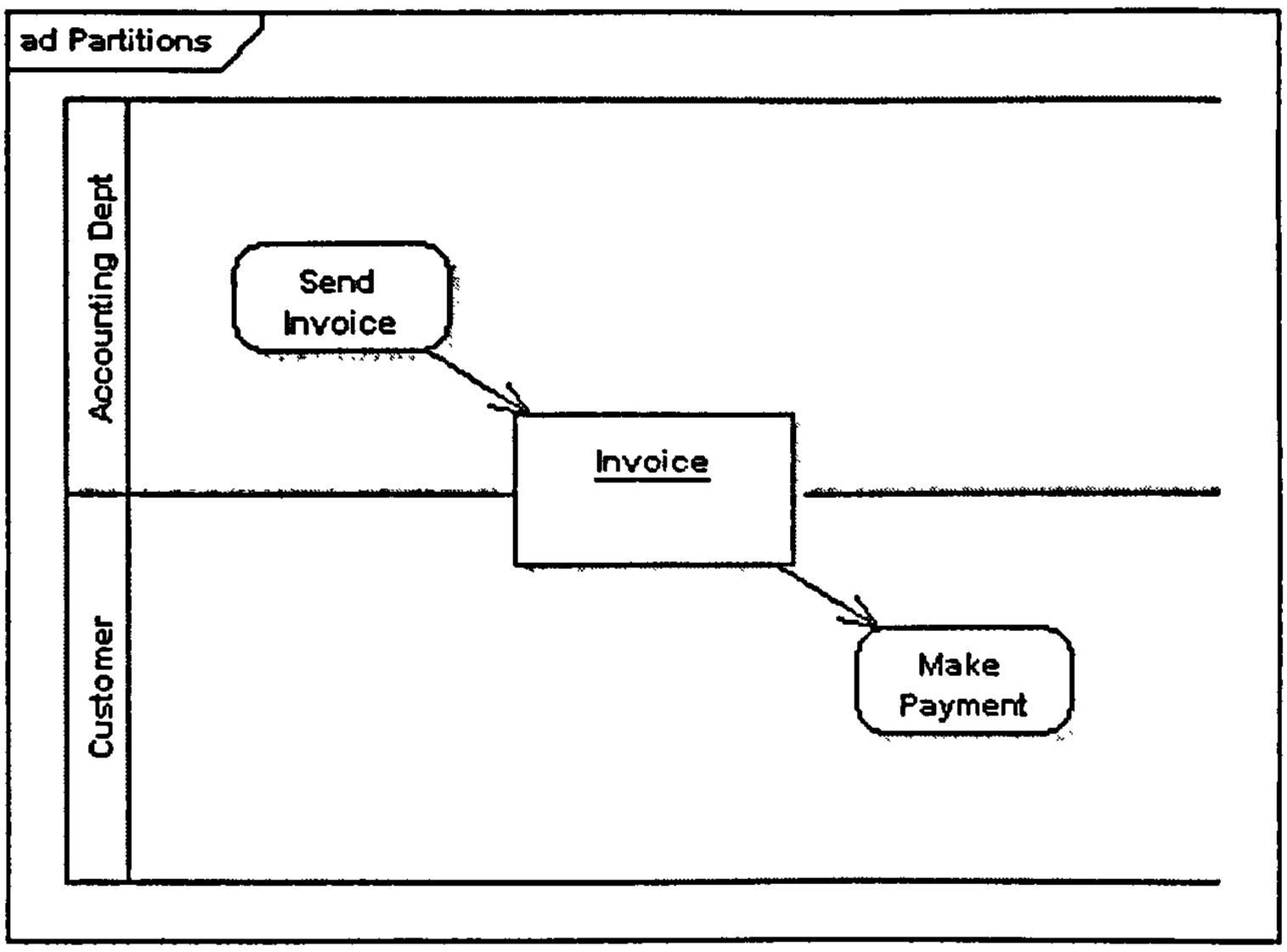


Figure A.0.18 Example of activity diagram with partition

Section A.2: Step by Step UML 2 Activity Diagram Drawing Guide

To understand the following steps, users must read the previous section to be familiar with the UML 2.0 activity diagram's terminologies.

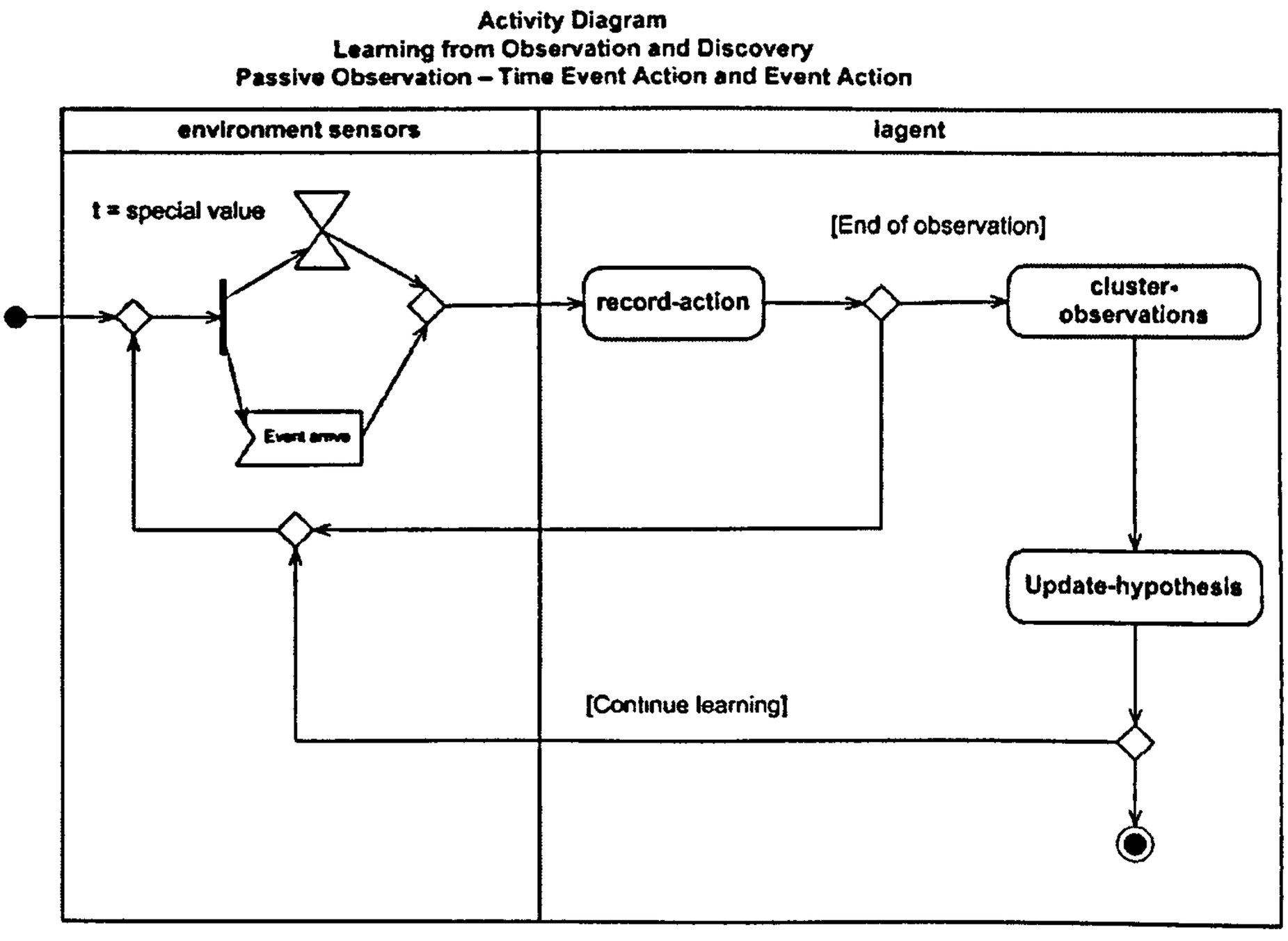


Figure A.0.19 Example of activity diagram

- Step 1: Identify the users, object, and agents that are involved in this activity diagram with the Intelligent Agent
    - For each user, object, agent, or Intelligent Agent , draws a partition. Each partition will include the activities and actions of the partition's owner.
    - Figure A.0.19 shows an activity diagram with two partitions ; one for the Intelligent Agent and the other for the environment sensor agent.
- Step 2: Identify the sensors that the Intelligent Agents uses
    - If the trigger is a time trigger such as a lapse of time of a specific duration or the performance of an action at a specific time, draw the

time signal at the relevant partition . Write the time condition beside the time signal notation

- If the trigger is the occurrence of an action, then draw the accept signal notation at the relevant partition  . Write the triggering action beside the accept signal notation.

- Step 4: Identify the activities and actions that the Intelligent Agent performs
  - Draw the activities and actions that the Intelligent Agent conducts in the relevant partition. Each activity or action must have at least one title.
- Step 5: Identify the activities and actions that other users or agents perform
  - Draw the activities and actions that are conducted by other users in the relevant partition.
- Step 6: Identify Decision Nodes
  - Draw the required decision nodes in the activity diagram, its condition, and the relevant action for each condition as shown in the diagram below.

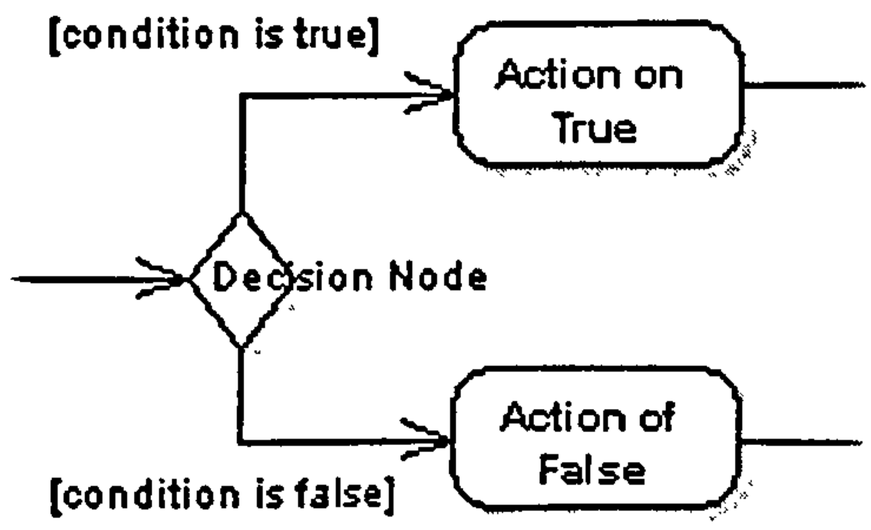


Figure A.0.20 Example of decision node

- Step 7: Identify Fork nodes
  - Draw the needed fork nodes that represent concurrent flow

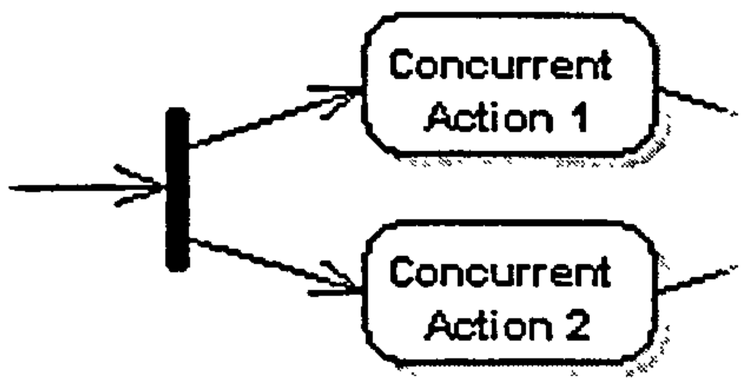


Figure A. 0.21 Fork nodes

- Step 8: identify Join and Merge Nodes
    - Draw the necessary join and merge nodes of control flows. The join node output does not execute until all inflows have been received

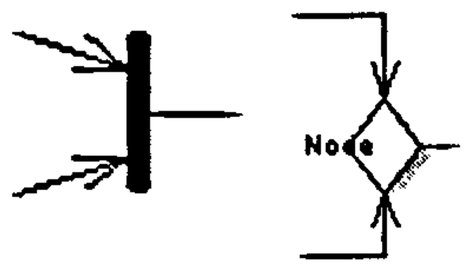
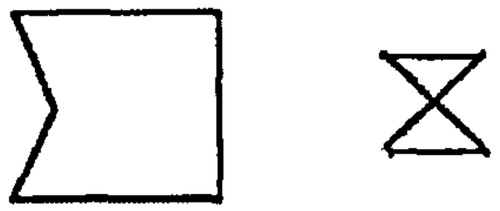
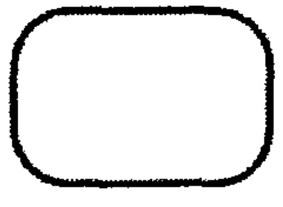


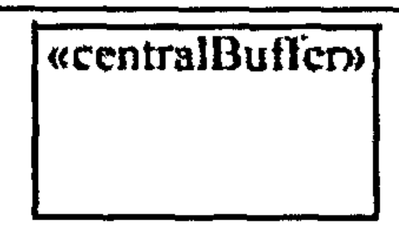
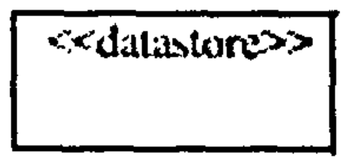
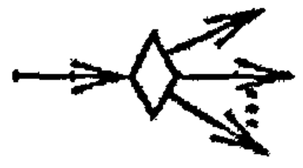
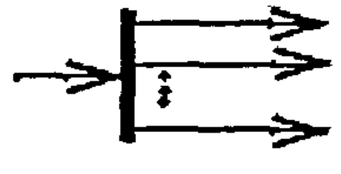
Figure A. 0.22 Notation of join and merge Nodes

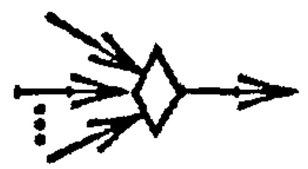
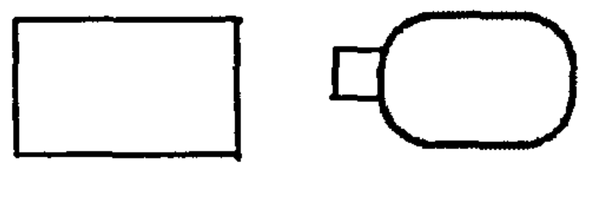
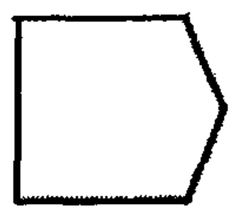
- Step 9: Complete all missing flows
    - Draw all the missing control and object flows between the activity and actions in the activity diagram
- Step 10: Revisit the diagram
    - Refine the diagram from redundant flows or actions.
    - Add exception handling and action interruption notations
    - Add Expansion region notation if needed
- Step 11: Draw initial and final nodes

Section A.3: Table of Activity Diagram Components

The following table illustrates the main components of UML 2.0 activity diagrams as mentioned in *Unified Modelling Language: Superstructure version 2.0, (OMG, 2004).*

Component	Notation	
AcceptEventAction		AcceptEventAction is an action that waits for the occurrence of an event meeting specified conditions.
Action		An action is a named element that is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing in the modelled system, be it a computer system or otherwise.
ActivityFinal		An activity final node is a final node that stops all flows in an activity.
ActivityNode	 *Action node*  *Object node*	An activity node is an abstract class for points in the flow of an activity connected by edges.

CentralBufferNode		A central buffer node accepts tokens from upstream objects nodes and passes them along to downstream object nodes. They act as a buffer for multiple in flows and out flows from other object nodes. They do not connect directly to actions.
ControlNode		A control node is an activity node used to coordinate the flows between other nodes. It covers initial node, final node and its children, fork node, join node, decision node, and merge node.
DataStore		A data store node is a central buffer node for non-transient information.
DecisionNode		A decision node has one incoming edge and multiple outgoing activity edges.
FinalNode		A final node is an abstract control node at which a flow in an activity stops.
FlowFinal		A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity.
ForkNode		A fork node is a control node that splits a flow into multiple concurrent flows.

InitialNode		An initial node is a control node at which flow starts when the activity is invoked.
JoinNode		A join node is a control node that synchronizes multiple flows. It has multiple incoming edges and one outgoing edge.
MergeNode		A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows. A merge node has multiple incoming edges and a single outgoing edge.
ObjectNode	 	An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to, as described in the semantics section.
SendSignalAction		SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity. The argument values are available to the execution of associated behaviours. The requestor continues execution immediately. Any reply message is ignored and is not transmitted to the requestor. If the input is already a signal instance, use SendObjectAction.

Appendix B: UML 2.0 Sequence Diagram User Guide

Sequence diagrams are good for representing the communication behaviour of an Intelligent Agent with other users, objects, agents, or Intelligent Agents . They provide a clear view of the sequence of communication and interaction among the entities of the sequence diagram. This user guide has three sections: the first presents a simple guide to the main components of a sequence diagram, the second section provides a step-by-step guide to allow researchers in the Agent learning field to sketch a simple sequence diagram. The third section shows a table of the main components of a sequence diagram.

Section B1: Sequence Diagram Components Guide

This section is an extraction from Sparx Systems, 2005b. A sequence diagram is a form of interaction diagram, which shows objects as lifelines running down the page and with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good for showing which objects communicate with which other objects and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic..

Lifelines

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is self then that indicates that the lifeline represents the classifier which owns the sequence diagram.

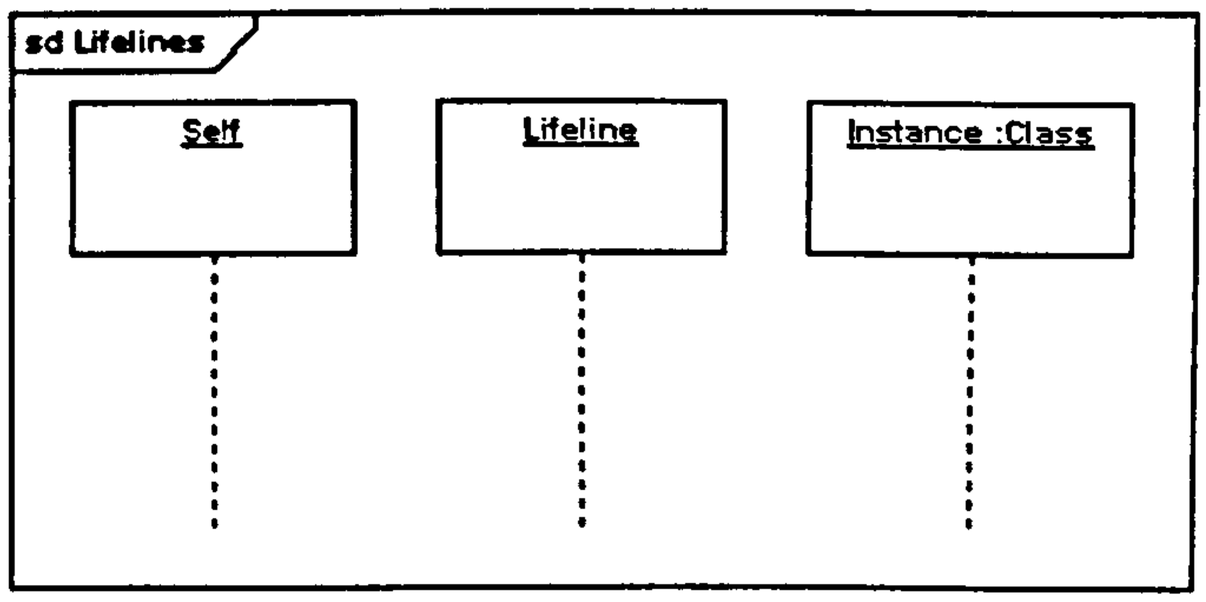


Figure B.0.1 Example of lifelines

Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.

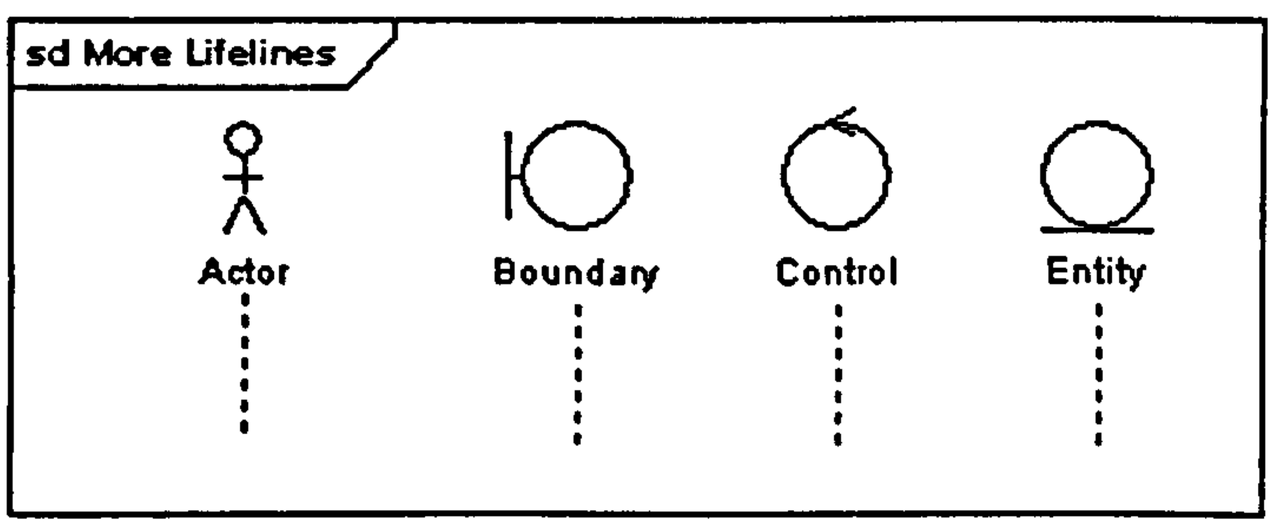


Figure 0.2 Example of actor lifeline

Messages

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by a line arrowhead) and the third is the asynchronous return message (denoted by the dashed line).

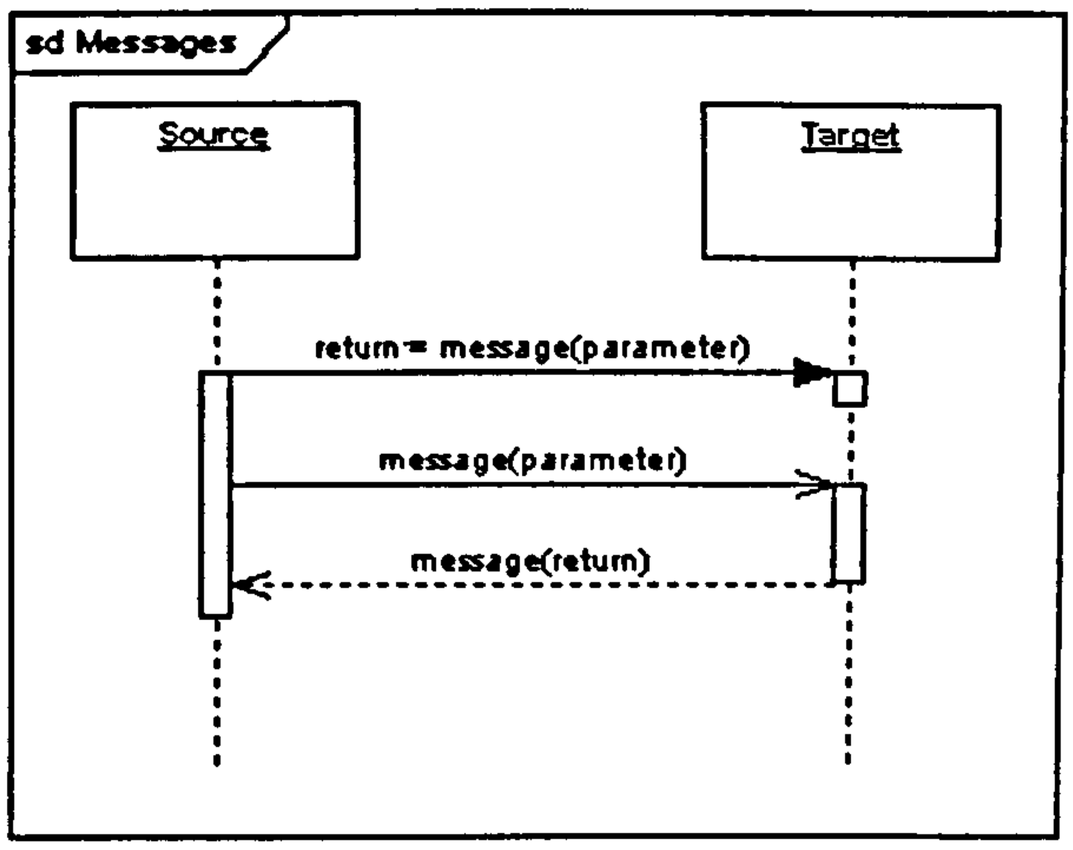


Figure B.0.3 Examples of message exchange

Execution Occurrence

A thin rectangle running down the lifeline denotes the execution occurrence or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning a reply; and the third is the target object receiving an asynchronous message and returning a reply.

Self Message

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline's execution occurrence.

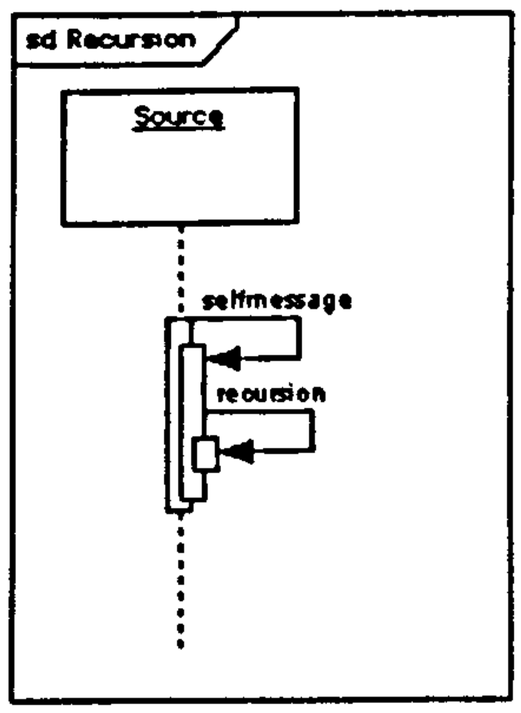


Figure B.0.4 Example of self message

Lost and Found Messages

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.

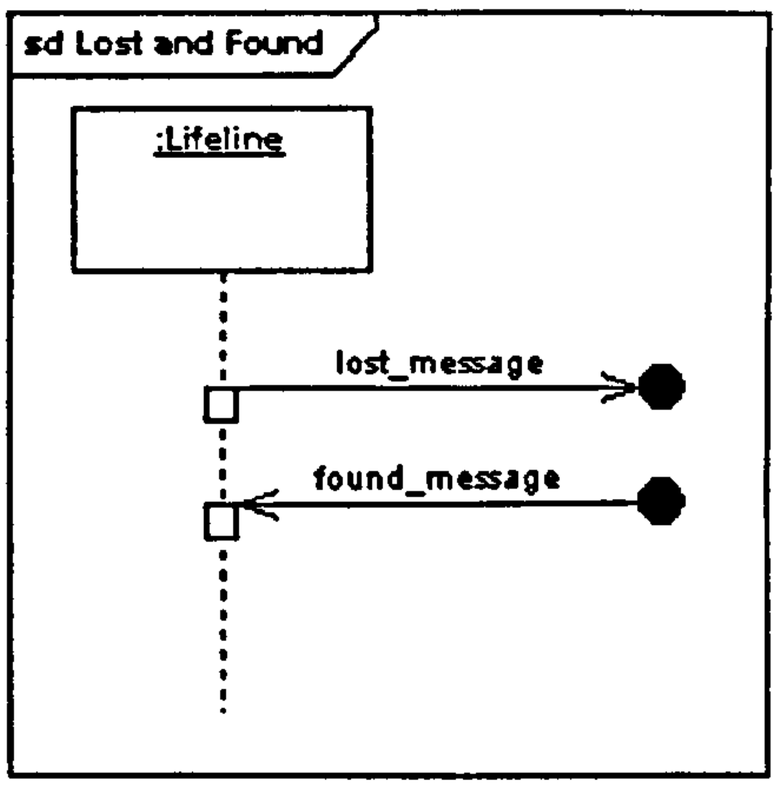


Figure B.0.5 Example of lost and found messages

Lifeline Start and End

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.

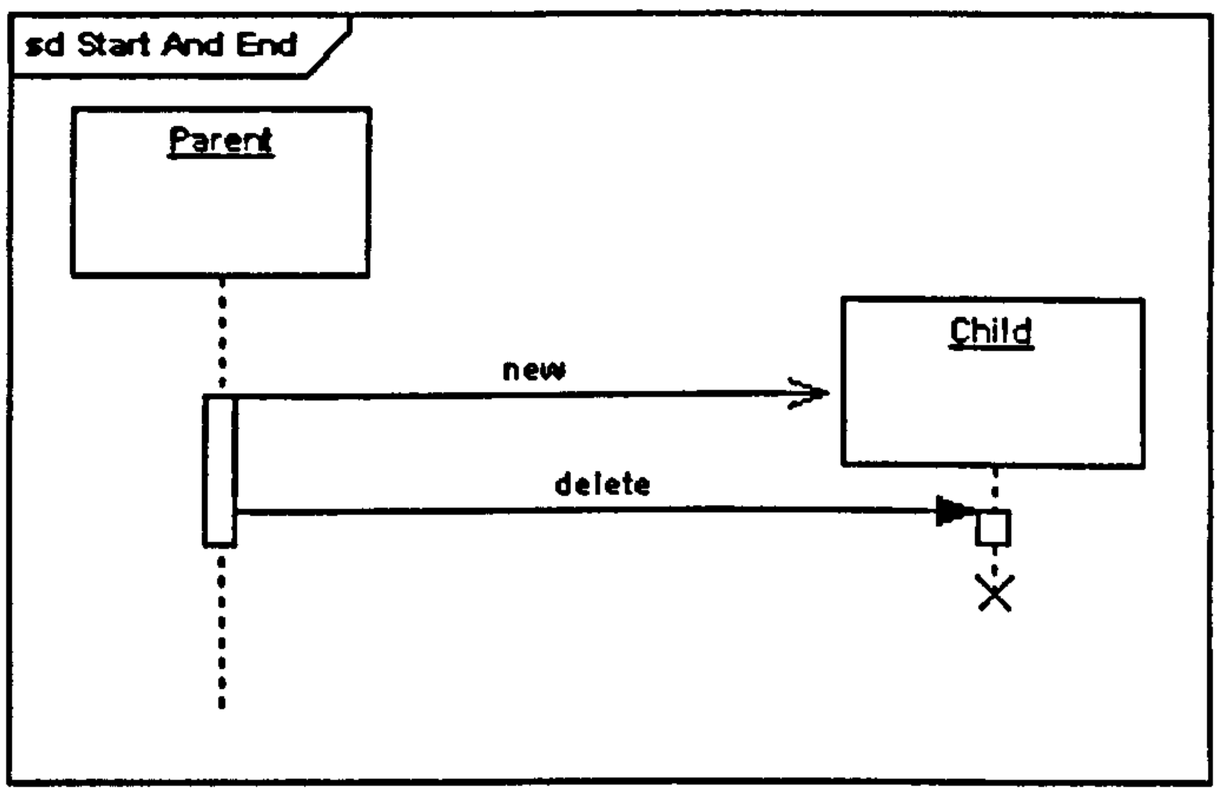


Figure B.0.6 Example of creating and destroying an object

Duration and Time Constraints

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modelling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.

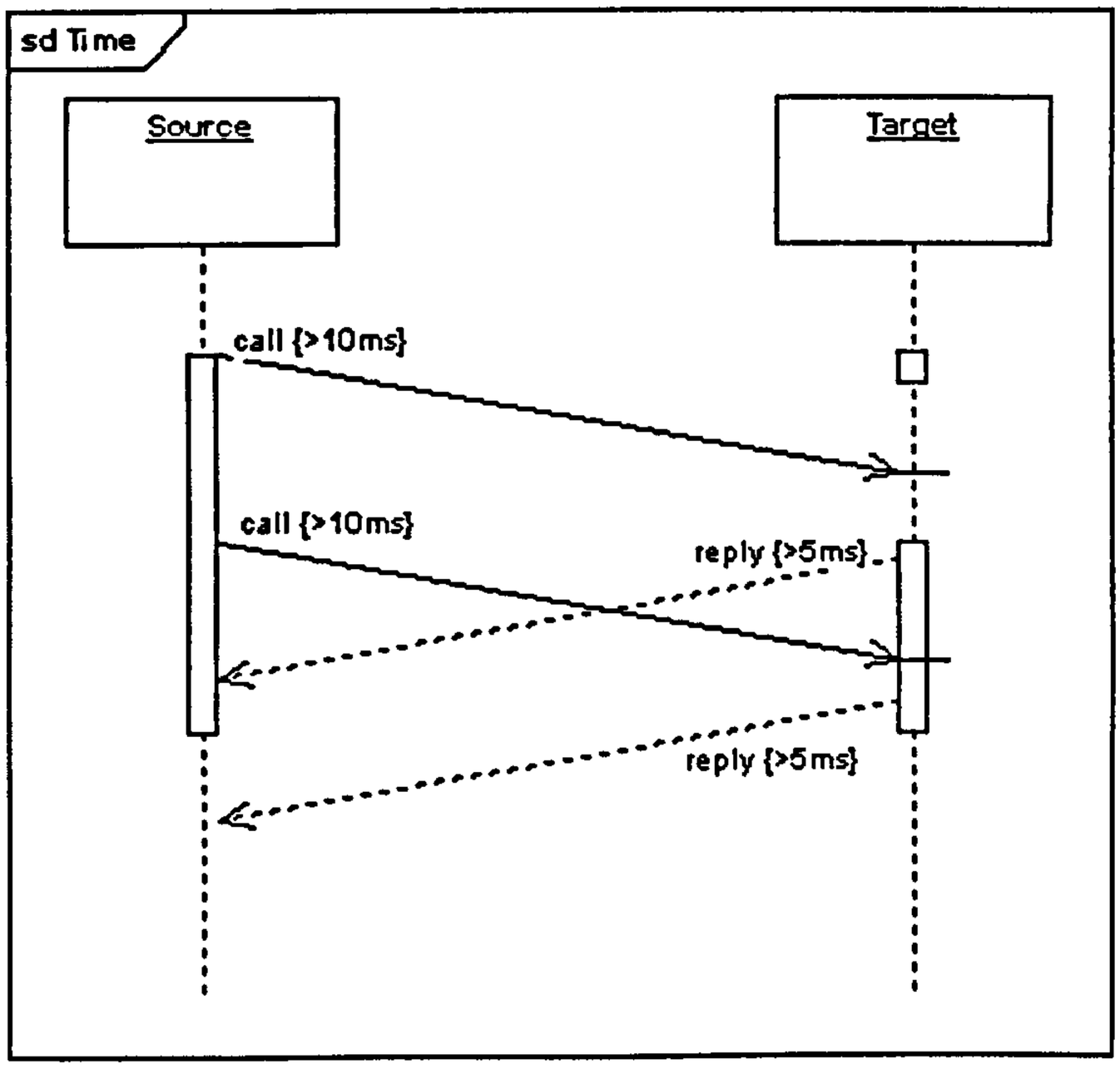


Figure B.0.7 Example of duration and time constraints notation

Combined Fragments

It was stated earlier that Sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:

- Alternative fragment (denoted "alt") models if...then...else constructs.
- Option fragment (denoted "opt") models switch constructs.

- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
- Parallel fragment (denoted "par") models concurrent processing.
- Weak sequencing fragment (denoted "seq") encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don't share a lifeline.
- Strict sequencing fragment (denoted "strict") encloses a series of messages which must be processed in the given order.
- Negative fragment (denoted "neg") encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
- Assertion fragment (denoted "assert") denotes that any sequence not shown as an operand of the assertion is invalid.
- Loop fragment encloses a series of messages which are repeated.

The following diagram shows a loop fragment.

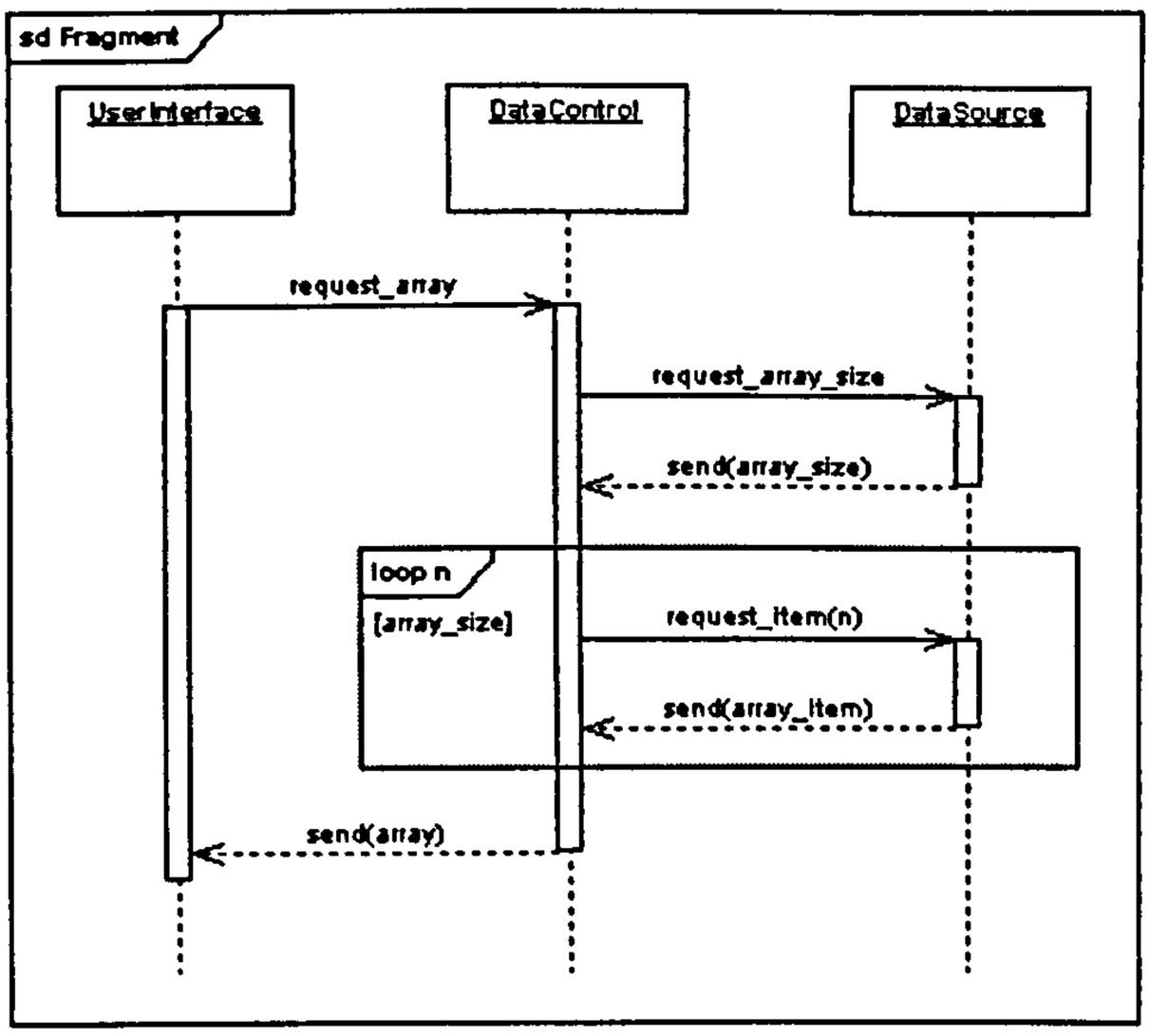


Figure B.0.8 Example of loop fragment

There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

Gate

A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame.

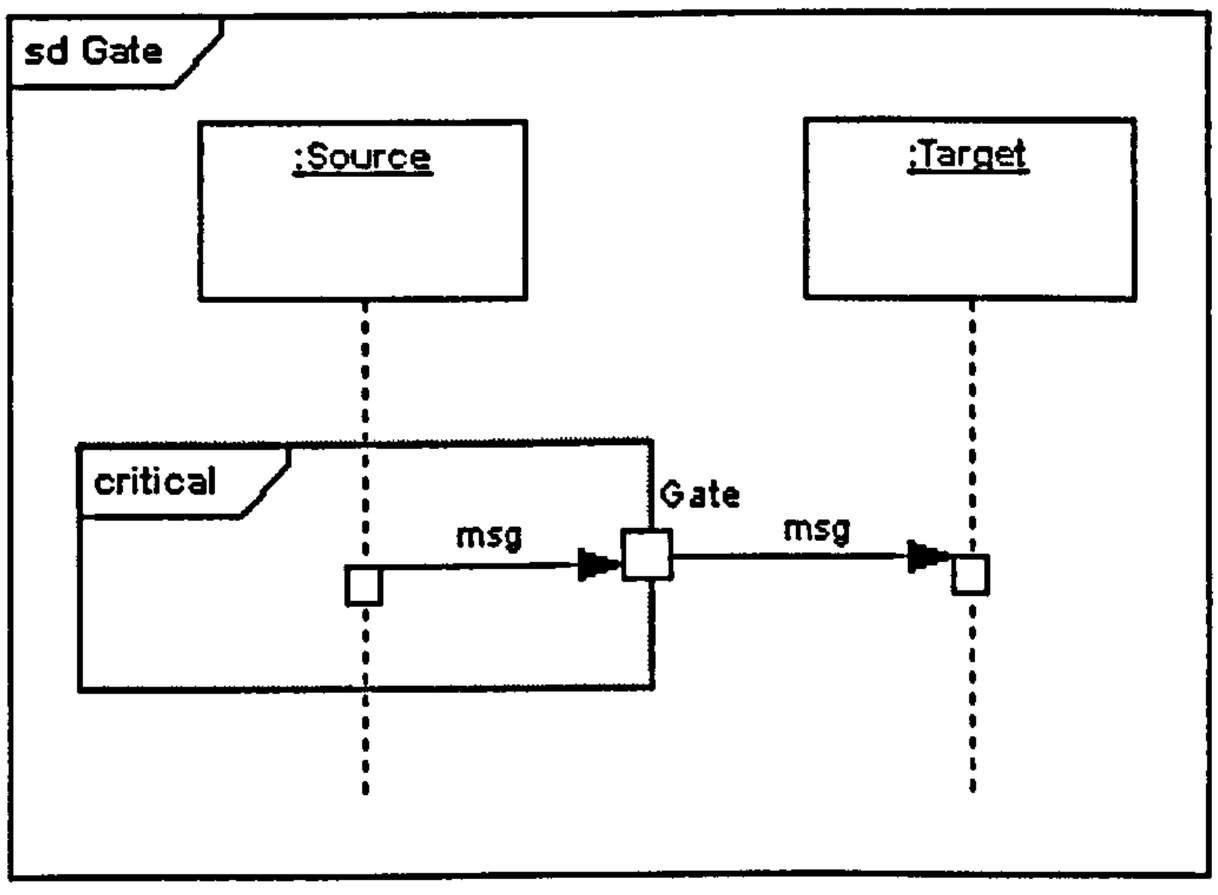


Figure B.0.9 Example of a gate

Part Decomposition

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.

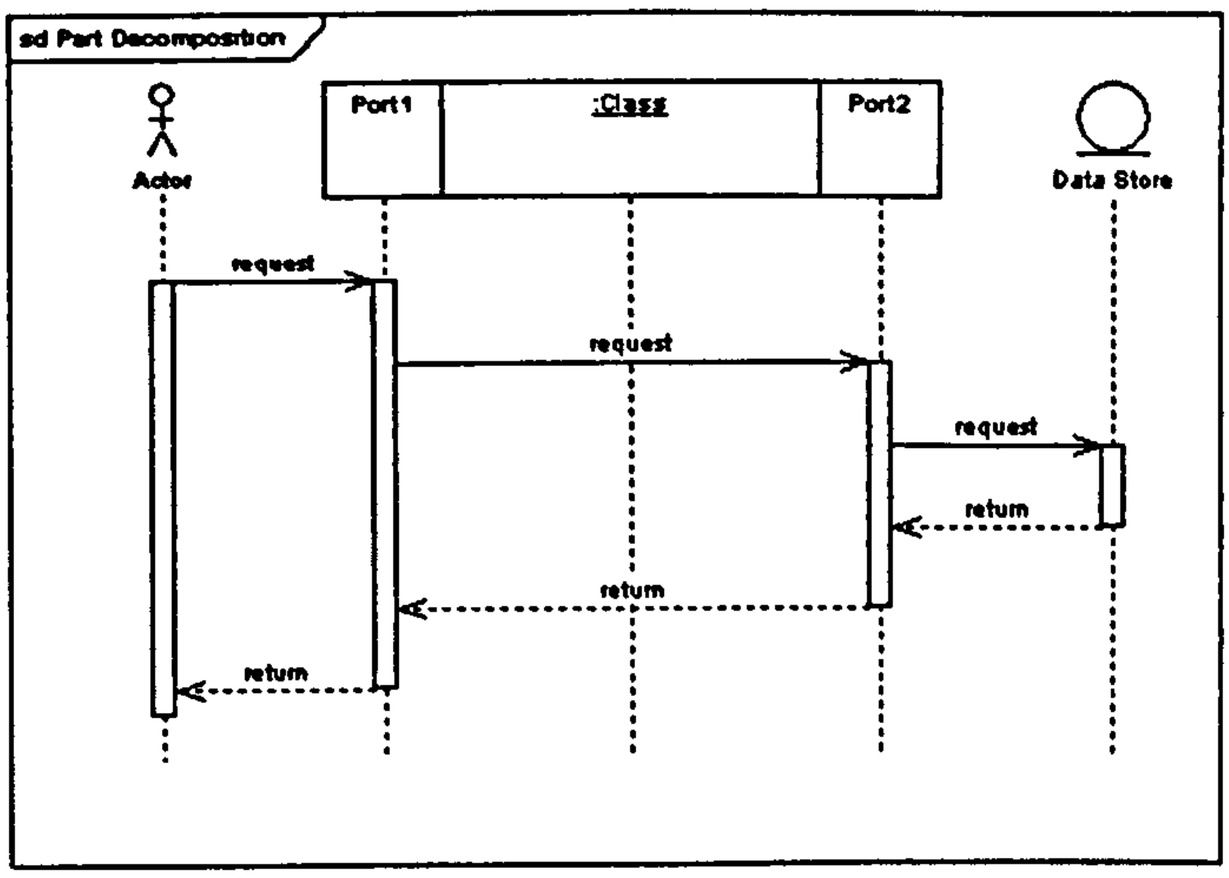


Figure B.0.10 Example of part decomposition

State Invariant / Continuations

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.

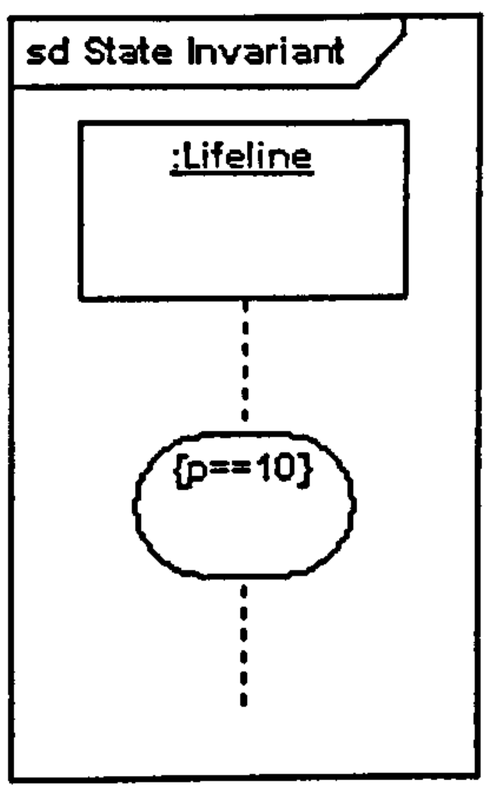


Figure B.0.11 Example of state invariant

A Continuation has the same notation as a state invariant but is used in combined fragments and can stretch across more than one lifeline.

Section B.2: Step by Step UML 2 Sequence Drawing Guide:

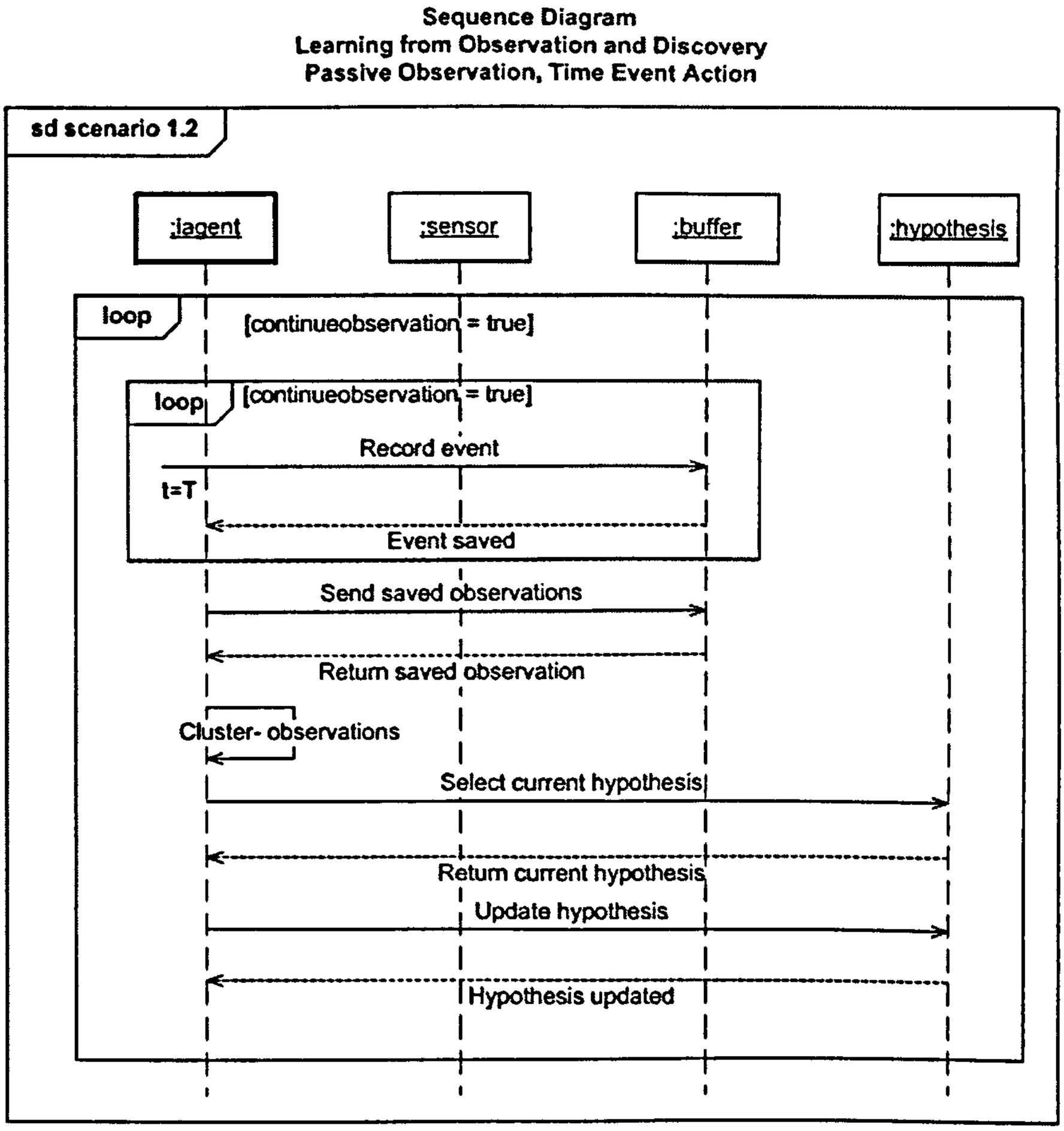


Figure B.0.12 Example of sequence diagram

To understand the following steps, users must first read the previous section to be familiar with the UML 2.0 sequence diagram's terminologies.

- Step 1: Identify and draw lifelines
    - Identify the main entities of the sequence diagram such as Intelligent Agent, users (actors), agents, or objects.
    - Draw the lifeline of each of the identified entities as shown in the example below :

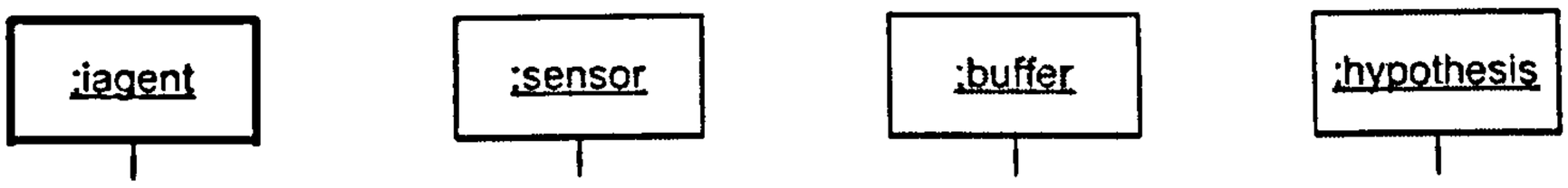


Figure B.0.13 Example of lifelines

- Step 2: Draw messages and Interactions among the sequence diagram entities
    - Draw the arrows that represent the messages and interactions between the Intelligent Agent and the other lifelines of the sequence diagram
    - Draw duration and time constraints if needed
- Step 3: Draw combined frames if needed
    - Identify any procedural logic that needs to be represented by a frame such as parallel or optional interaction
    - For each procedural logic draw a frame that covers the area where this logic applies. Figure B.0.12 shows a sequence diagram with two loop frames.
- Step 4: Identify any needed Gates
    - If there is any message that needs to be transferred outside the existing frame, draw a gate notation to allow this transfer as shown in the below figure.

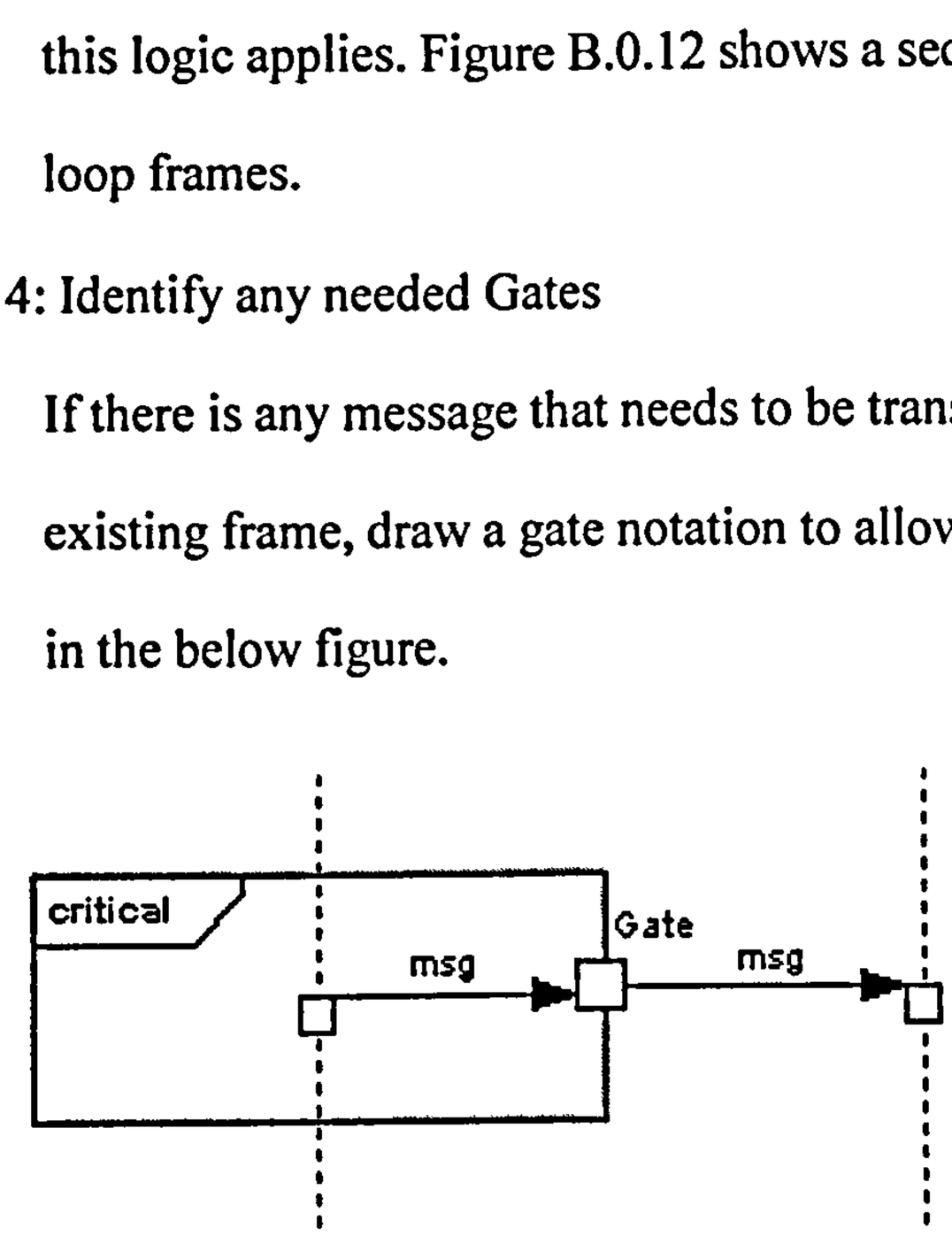


Figure B.0.14 Example of transferring message through a gate

- Step 5: Identify Start and End of lifelines
    - Identify any entity that will be created or destroyed during the sequence diagram
    - Draw the notation of a start and end of the needed lifeline as shown in the example below:

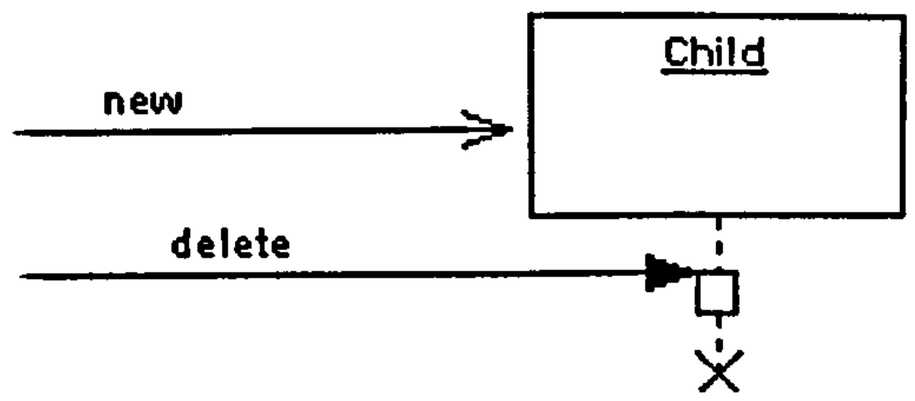
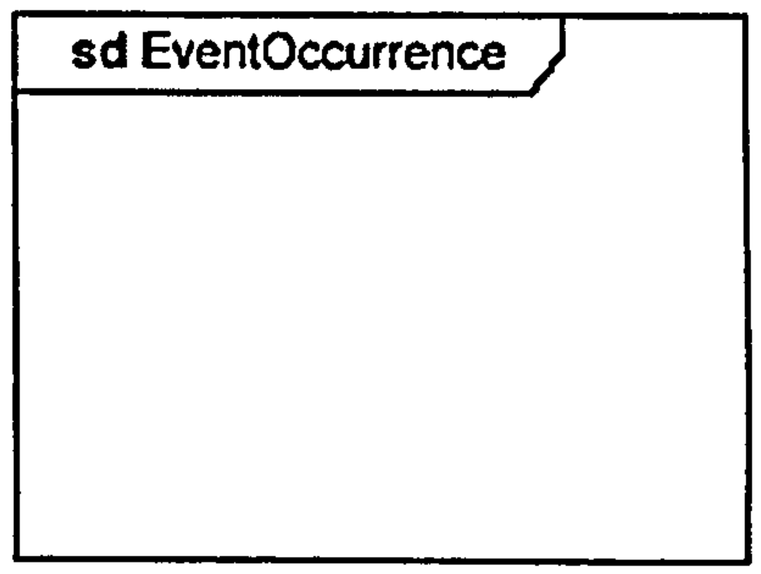
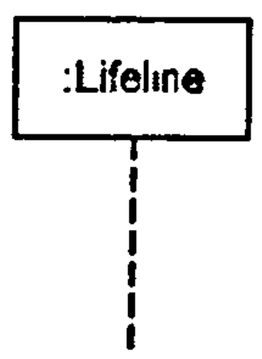
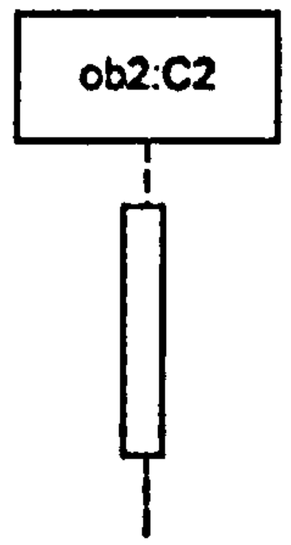
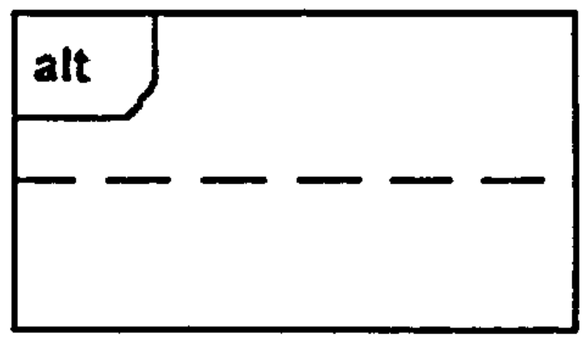
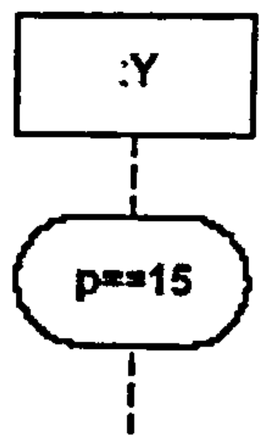


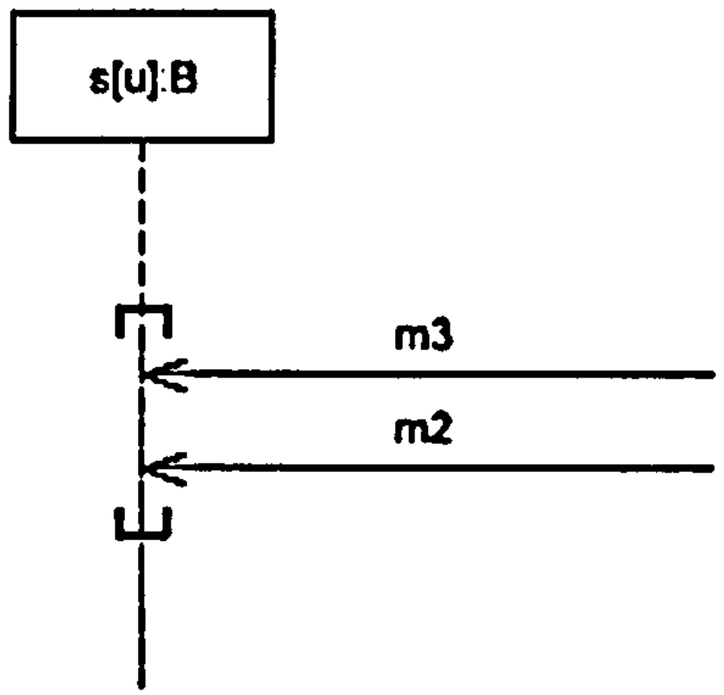
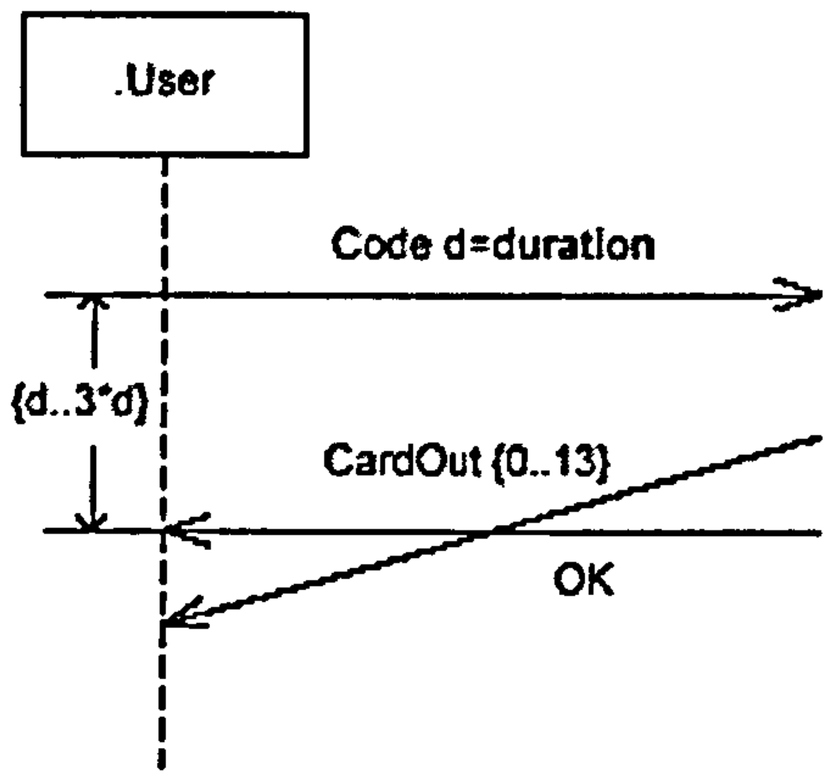
Figure B.0.15 Example of start and end of lifline

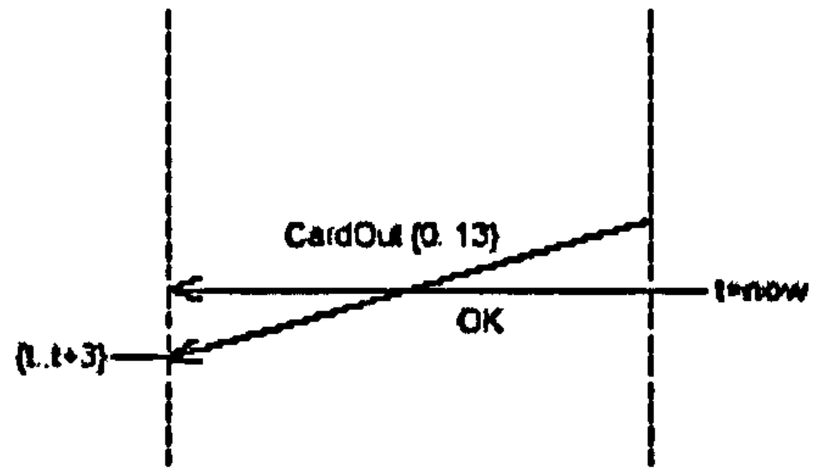
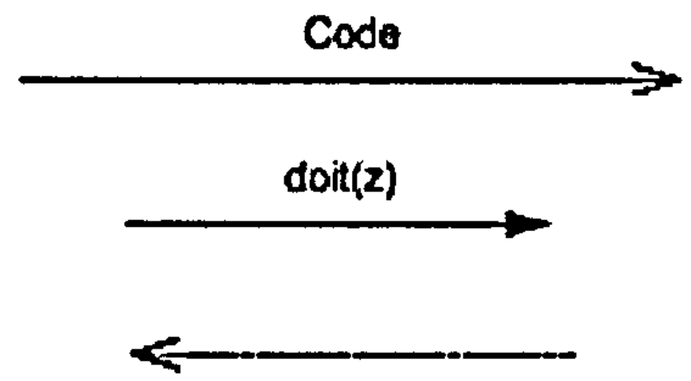
Section B.3: Table of Sequence Diagram Components

The following table illustrates the main components of UML 2.0 sequence diagrams as mentioned in *Unified Modelling Language: Superstructure version 2.0, (OMG, 2004).*

Component	Notation	Reference
Frame		The notation for an Interaction in a Sequence Diagram is a solid-outline rectangle. The keyword sd followed by the Interaction name and parameters is in a pentagon in the upper lefthand corner of the rectangle.
Lifeline		A lifeline represents an individual participant in the Interaction. While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity.

ExecutionSpecification		An ExecutionSpecification is a specification of the execution of a unit of behaviour or action within the Lifeline. The duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications, the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification.
InteractionUse		An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones. It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification.
CombinedFragment		A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner. InteractionOperator Specifies the operation, which defines the semantics of this combination of InteractionFragments.
StateInvariant / Continuations		A StateInvariant is a runtime constraint on the participants of the interaction. It may be used to specify a variety of different kinds of constraints, such as values of attributes or variables, internal or external states, and so on. A StateInvariant is an InteractionFragment and it is placed on a Lifeline. The Constraint is assumed to be evaluated during runtime. The Constraint is evaluated immediately prior to the execution of the next OccurrenceSpecification such that all actions that are not explicitly modeled have been executed. If the Constraint is true the trace is a valid trace; if the Constraint is false the trace is an invalid trace. In other words all traces that have a StateInvariant with a false Constraint are considered invalid.

Coregion		A notational shorthand for parallel combined fragments are available for the common situation where the order of event occurrences (or other nested fragments) on one Lifeline is insignificant. This means that in a given "coregion" area of a Lifeline all the directly contained fragments are considered separate operands of a parallel combined fragment.
Stop		This indicates the termination of the Lifeline of an object.
DurationConstraint DurationObservation		A DurationConstraint defines a Constraint that refers to a DurationInterval. It is shown as a graphical association between a DurationInterval and the constructs that it constrains. The notation is specific to the diagram type. A DurationObservationAction defines an action that observes duration in time and writes this value to a structural feature. It is an action that, when executed, measures an identified duration in the context in which it is executing and writes the obtained value to the given structural feature.

TimeConstraint TimeObservation		A TimeConstraint defines a Constraint that refers to a TimeInterval. The semantics of a TimeConstraint are inherited from Constraints. All traces where the constraints are violated are negative traces i.e. if they occur in practice the system has failed
Message		A Message is a NamedElement that defines one specific kind of communication in an Interaction. A communication can be e.g. raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication given by the dispatching ExecutionSpecification, but also the sender and the receiver. A Message normally associates two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification. Messages come in different variants depending on what kind of Message they convey. The notations shown are for an asynchronous message, a call and a reply.
Lost Message		Lost messages are messages from a known sender, but the reception of the message fails.
Found Message		Found messages are messages from a known receiver, but the sending of the message is not described within the specification.
GeneralOrdering		A GeneralOrdering represents a binary relation between two OccurrenceSpecifications, to show that one OccurrenceSpecification must occur before the other in a valid trace. This mechanism provides the ability to define partial orders of OccurrenceSpecifications that may otherwise not have a specified order. A GeneralOrdering may appear anywhere in an Interaction, but only between OccurrenceSpecifications.

Appendix C: UML 2.0 State Machine Diagram User Guide

State machine diagrams are good for describing the behaviour of an Intelligent Agent during different states of its life. State machine diagrams are not efficient at describing interactions among Intelligent Agents and other entities such as other agents, users, or objects. This user guide has three sections: the first presents a simple guide to the main components of state machine diagrams, the second section provides a step-by-step guide to allow researchers in the Agent learning field to sketch a simple state machine diagram. The third section shows a table of the main components of a state machine diagram.

Section C.1 State Machine Diagram Components Guide

This section is an extraction from Sparx Systems, 2005c. A State Machine Diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

As an example, the following State Machine Diagram shows the states that a door goes through during its lifetime.

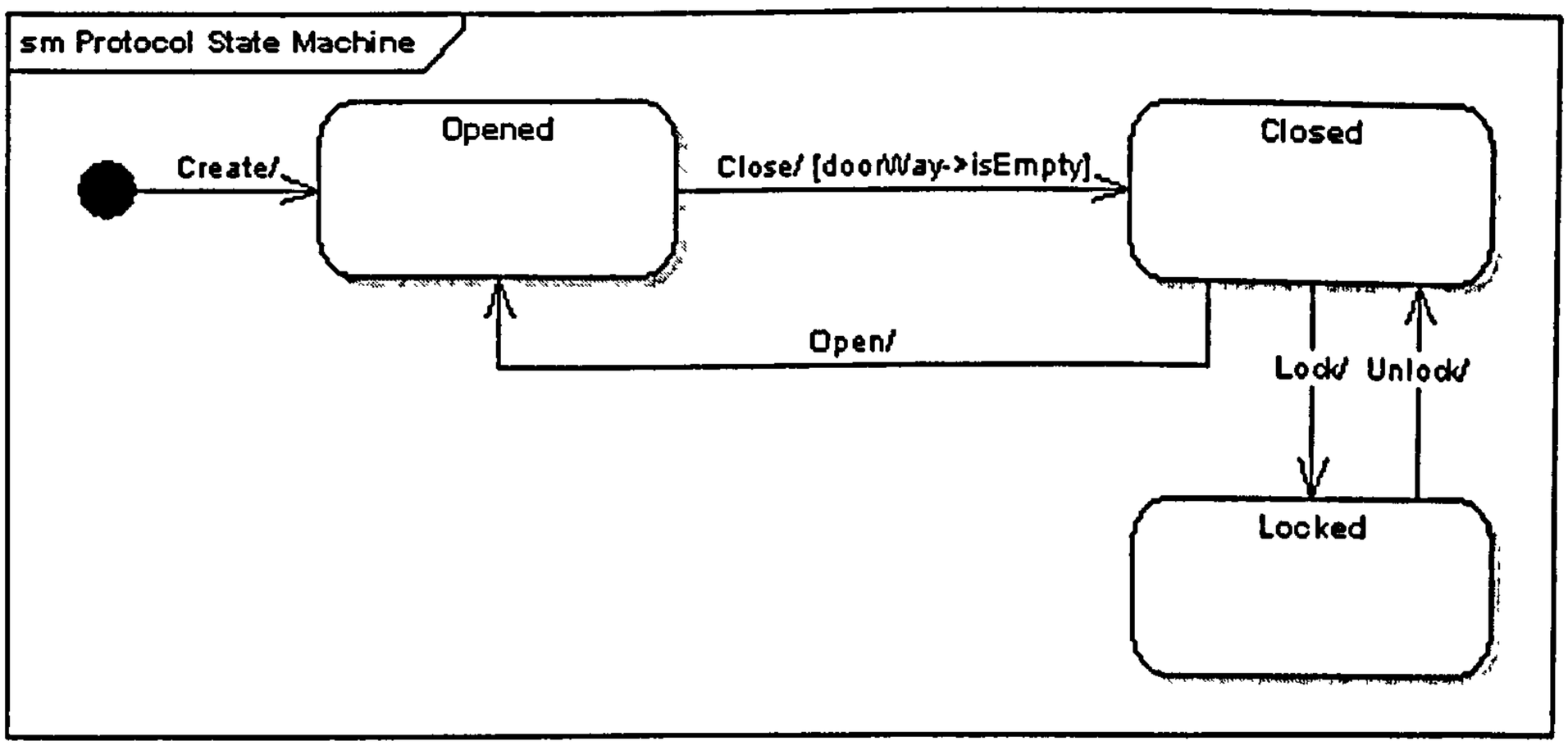


Figure C.0.1 Example of state machine diagram for a door

The door can be in one of three states: Opened, Closed or Locked. It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states: for example, if a door is Opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled. The syntax and conventions used in State Machine Diagrams will be discussed in full in the following sections.

States

A State is denoted by a round-cornered rectangle with the name of the state written inside it.

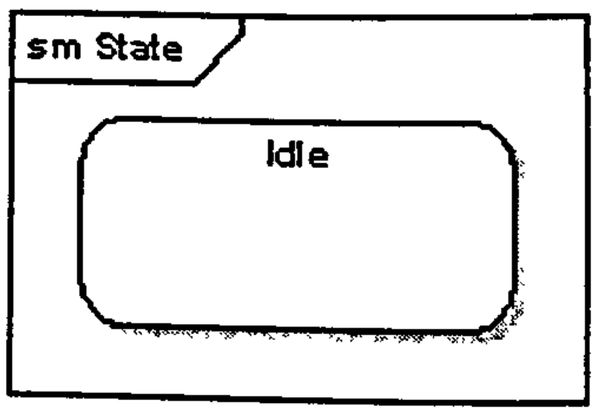


Figure B.0.2 Notation of a state

Initial and Final States

The Initial State is denoted by a solid black circle and may be labelled with a name. The Final State is denoted by a circle with a dot inside and may also be labelled with a name.

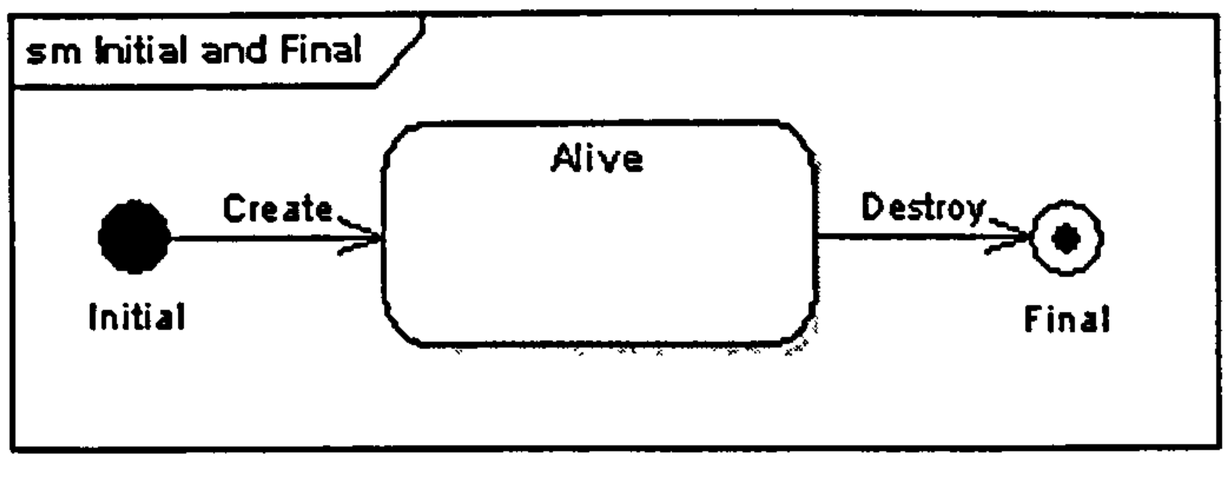


Figure C.0.3 Example of initial and final states

Transitions

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.

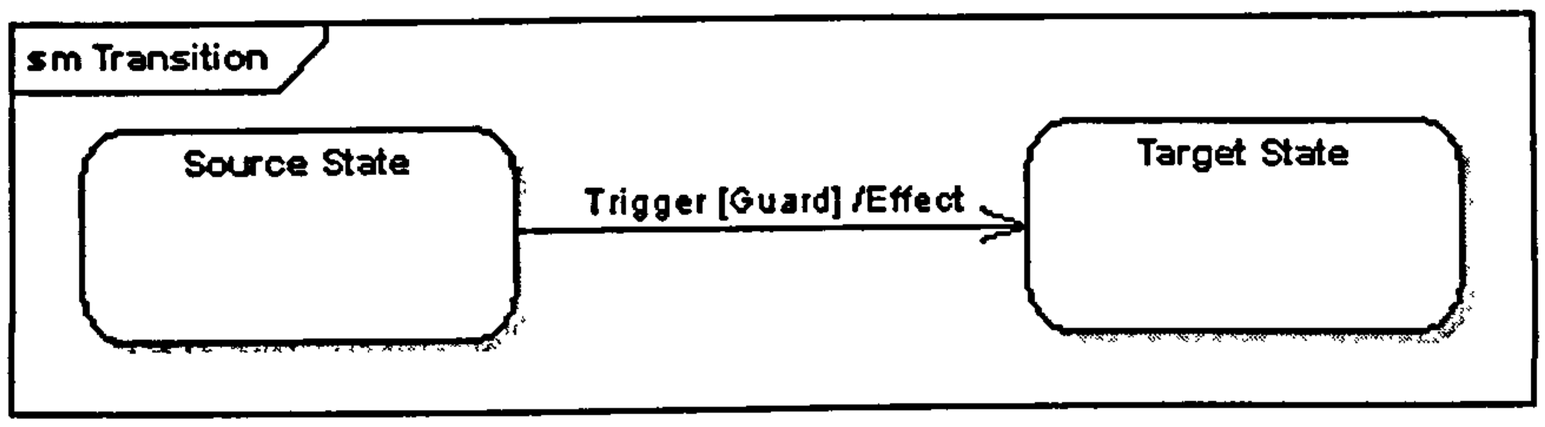


Figure C.0.4 Notation of a transition

"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions

In the transition example above, an Effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.

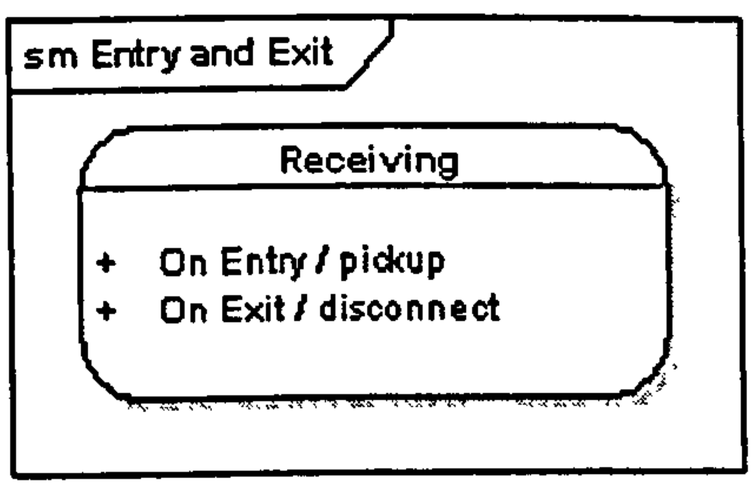


Figure C.0.5 Notation of state actions

It is also possible to define actions that occur on occasion , or actions that always occur. It is possible to define any number of actions of each type.

Self-Transitions

A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.

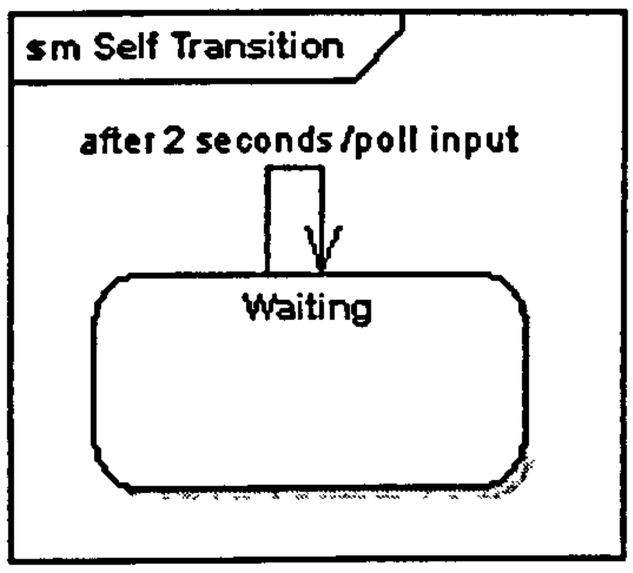


Figure C.0.6 Example of self -transition

Compound States

A state machine diagram may include sub-machine diagrams, as in the example below.

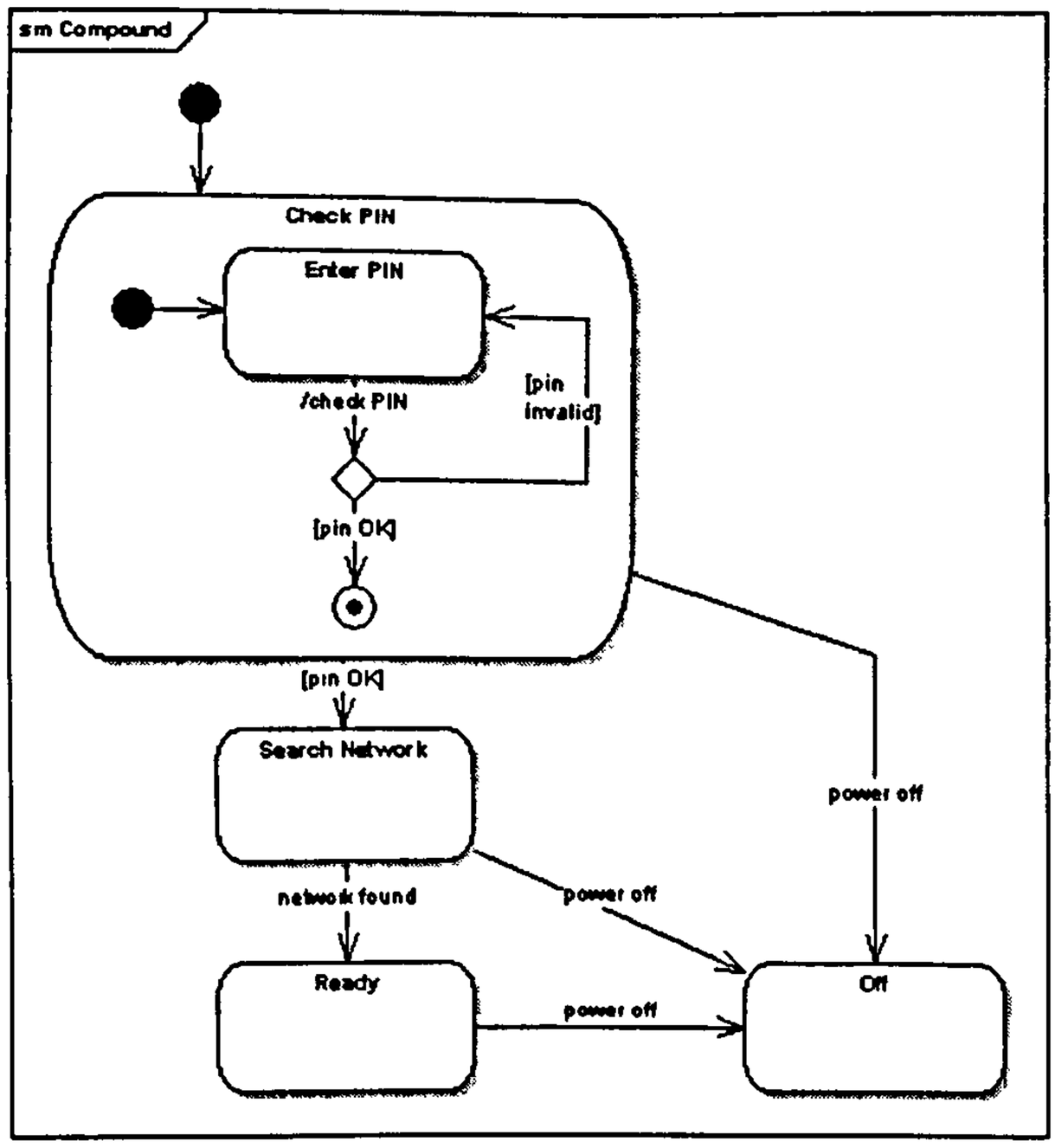


Figure C.0.7 Example of state machine with compound state

An alternative way to show the same information is as follows :

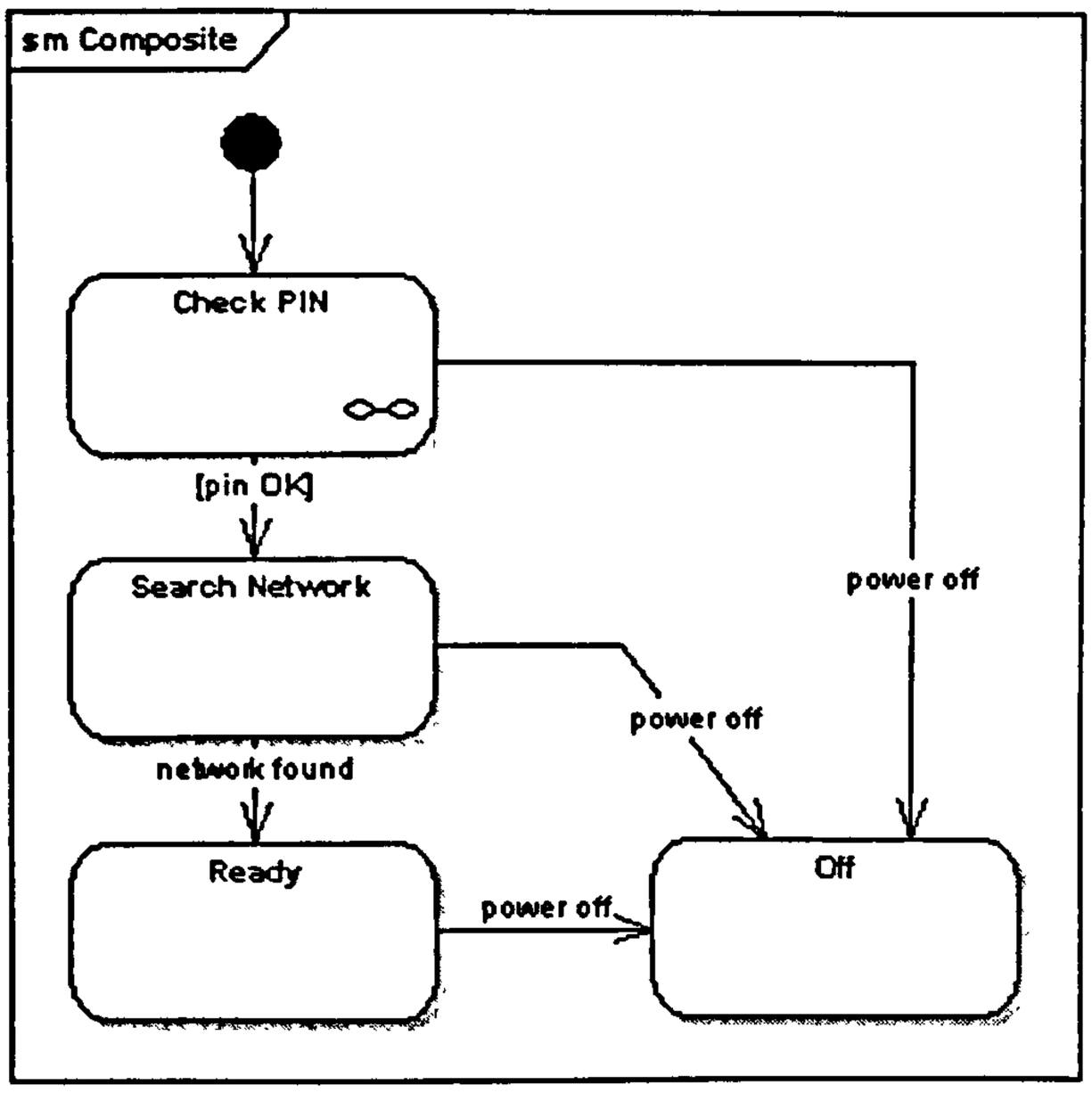


Figure C.0.8 Example of state machine diagram with composite state

The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

Entry Point

Sometimes you won't want to enter a sub-machine at the normal Initial State. For example, in the following sub-machine it would be normal to begin in the Initializing state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the Ready state by transitioning to the named Entry Point.

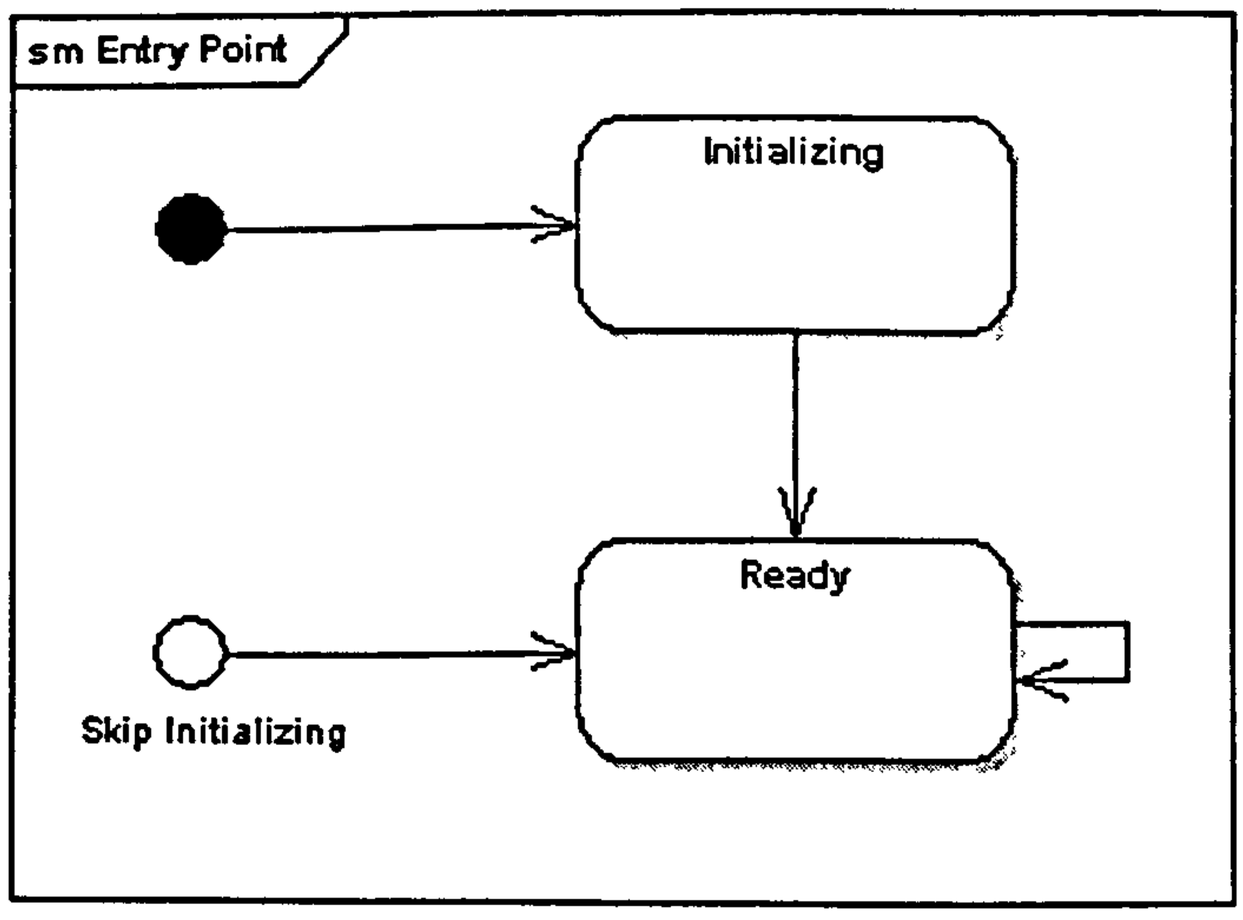


Figure C.0.9 Example of entry points

The following diagram shows the state machine one level up:

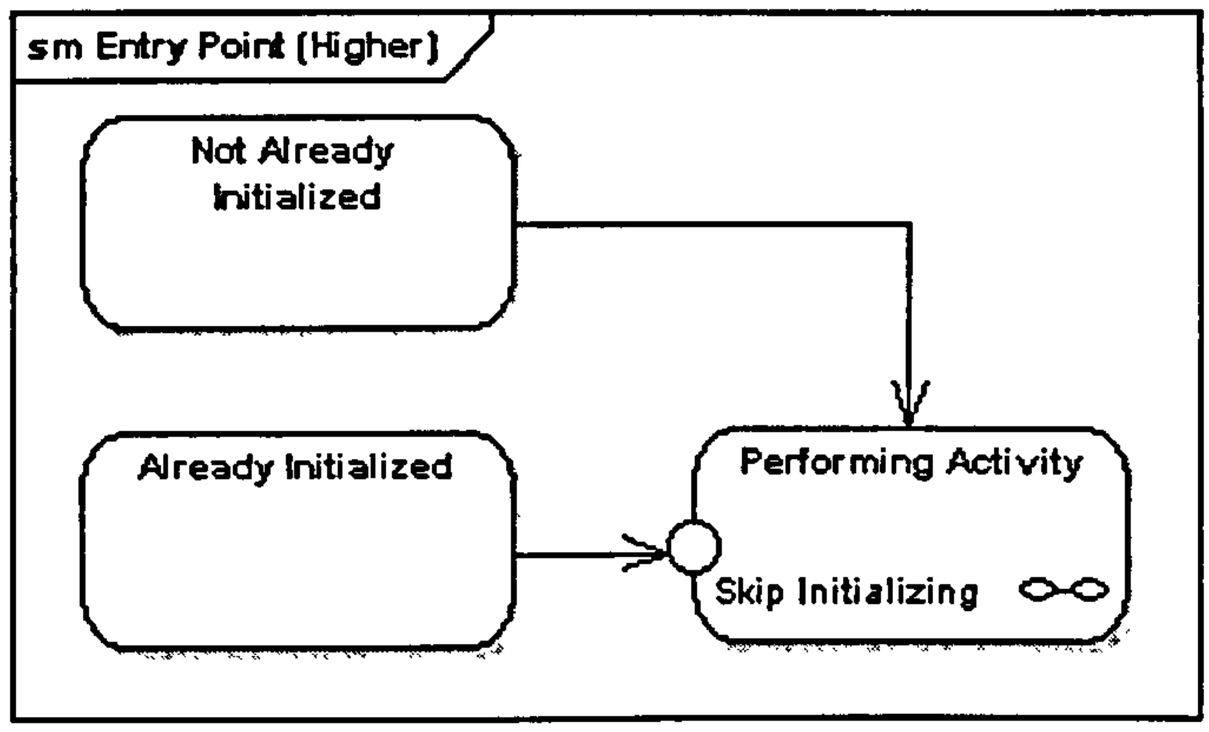


Figure C.0.10 Example of entry point

Exit Point

In a similar manner to Entry Points, it is possible to have named alternative Exit Points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

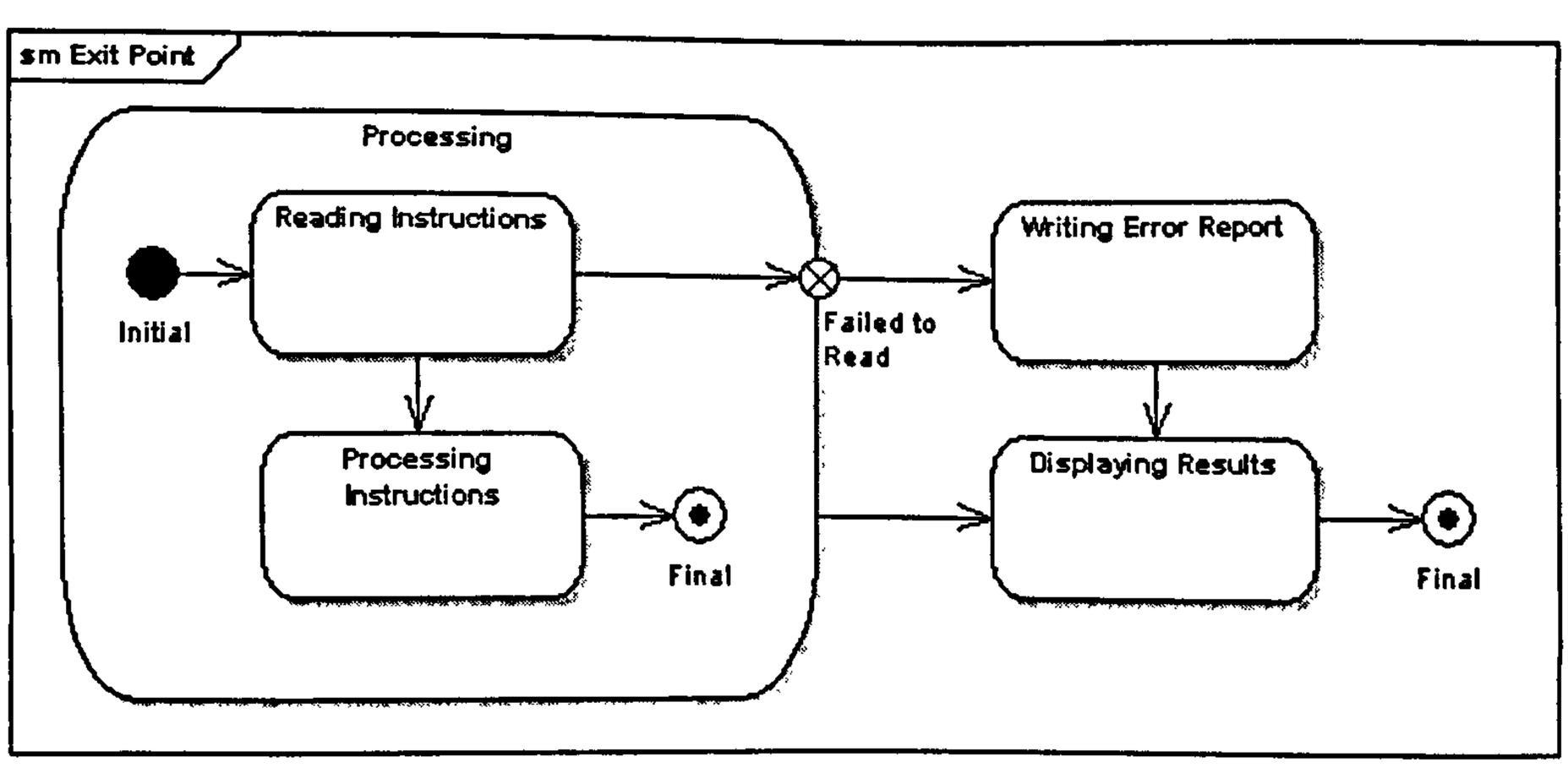


Figure C.0.11 Example of exit point

Choice Pseudo-State

A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at

after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.

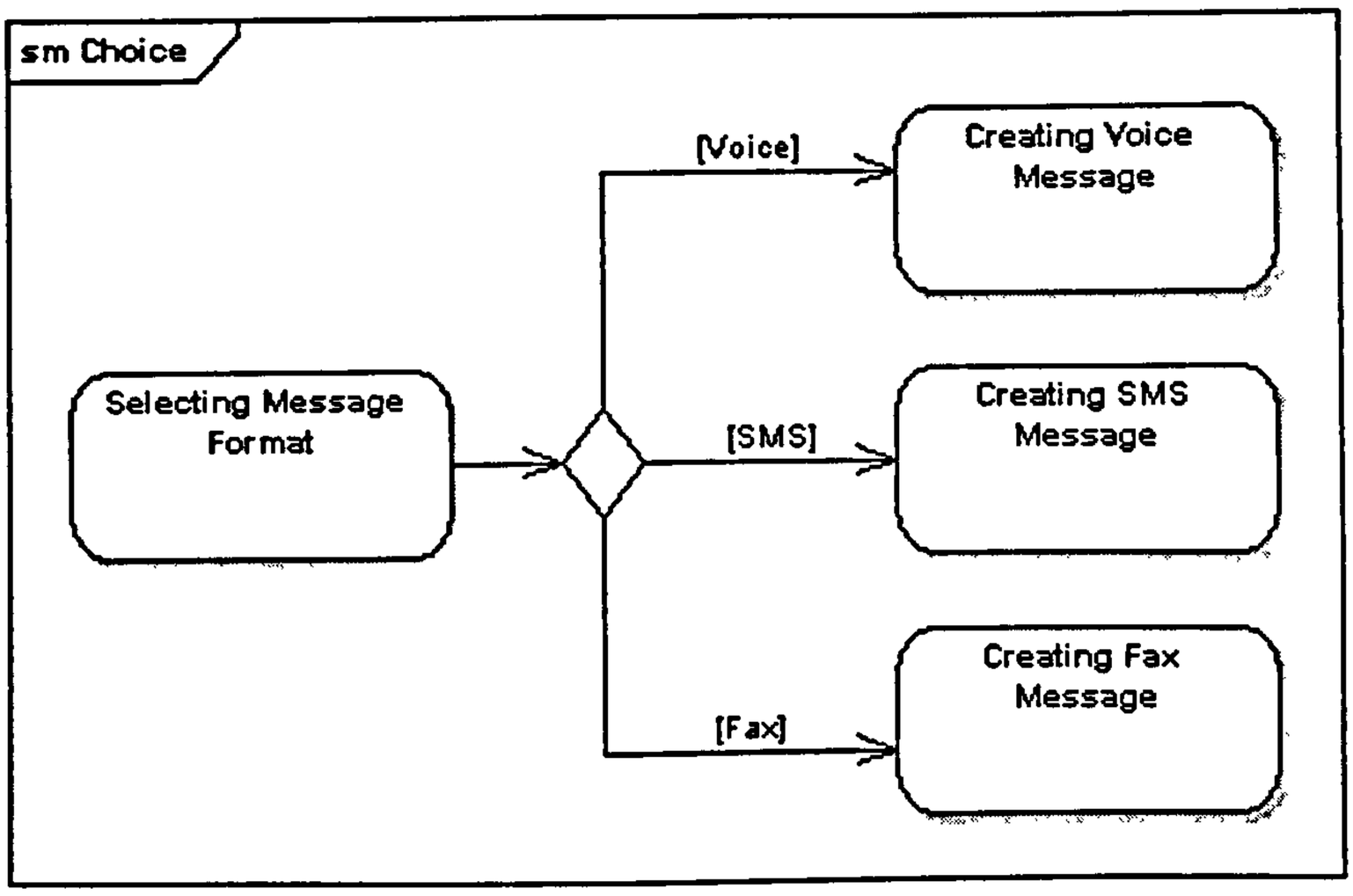


Figure C.0.12 Example of choice pseudo-state

Junction Pseudo-State

Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming and one or more outgoing transitions and a guard can be applied to each transition. Junctions are semantic-free; a junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch as opposed to a choice pseudo-state which realizes a dynamic conditional branch.

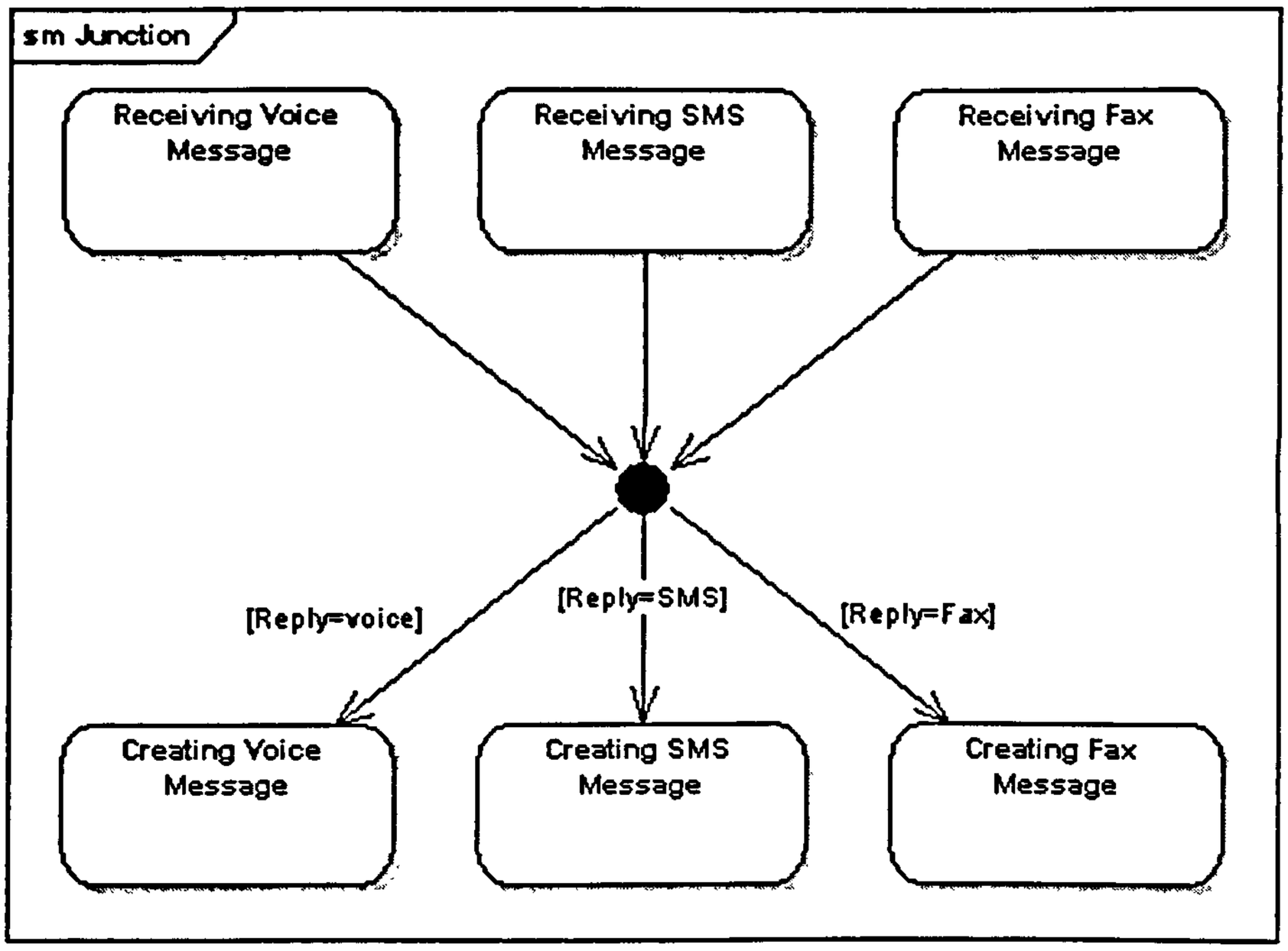


Figure C.0.13 Example of Junction Pesudo-state

Terminate Pseudo-State

Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.

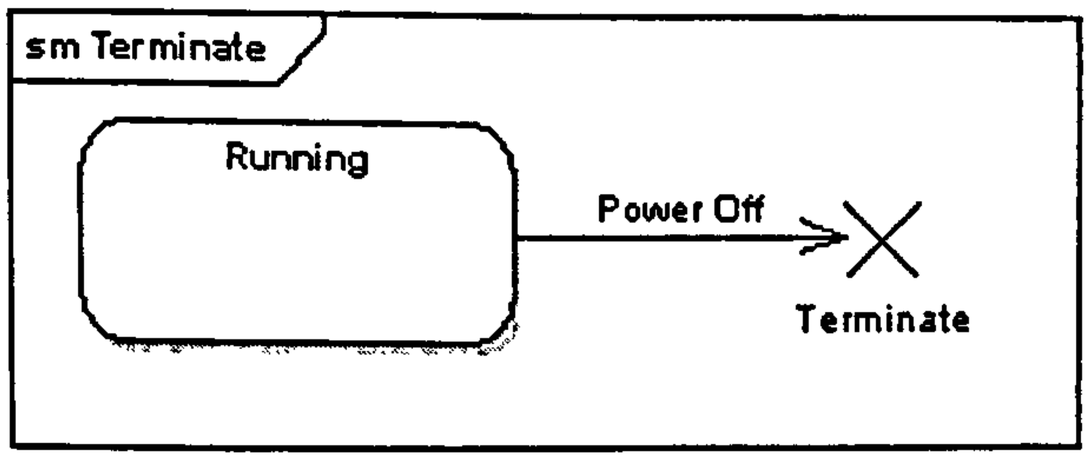


Figure C.0.14 Example of terminate pseudo-state

History States

A History State is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

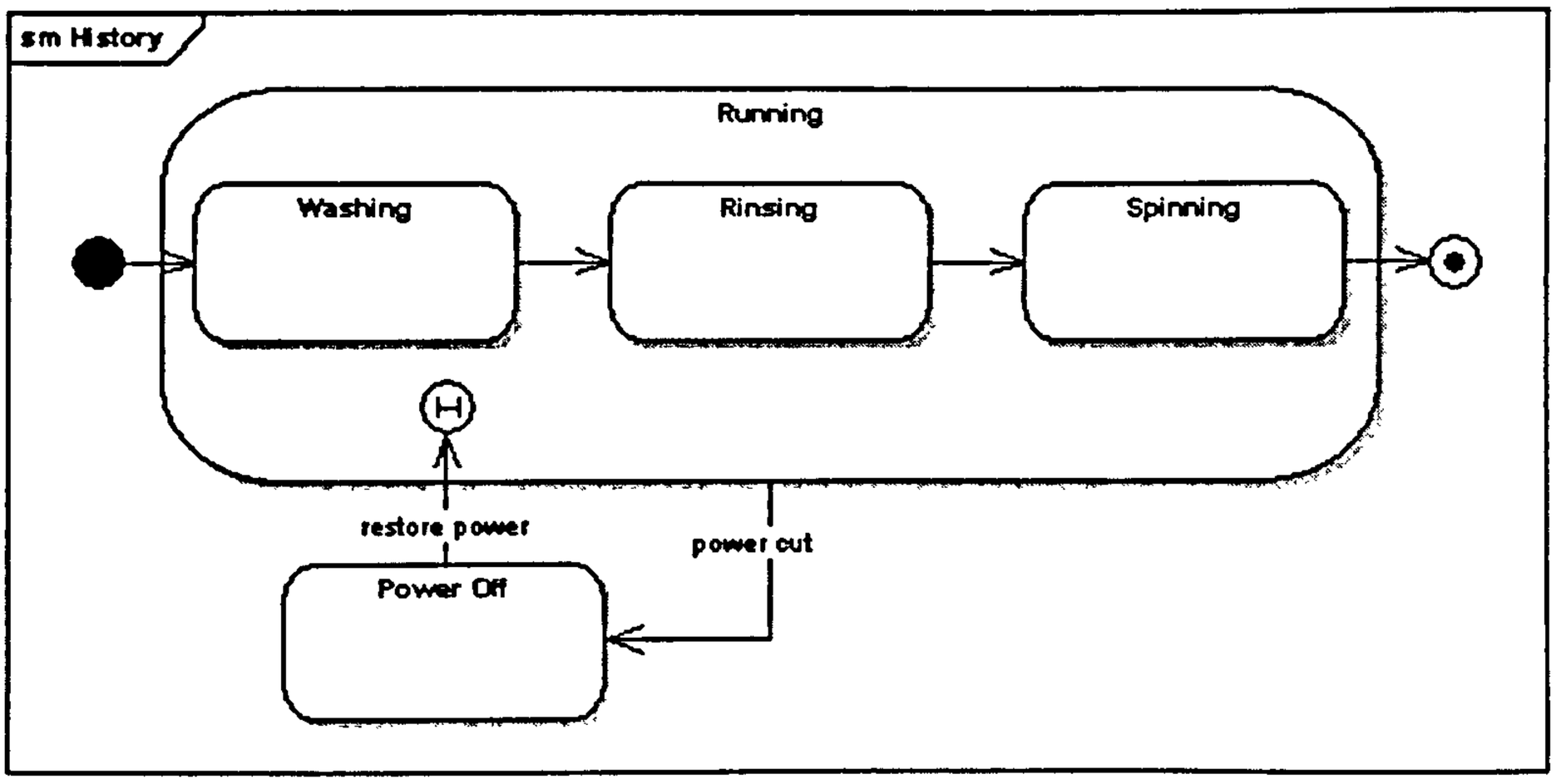


Figure C.0.15 Example of history state

In this state machine, when a washing machine is running it will progress from Washing through Rinsing to Spinning. If there is a power cut, the washing machine will stop running and will go to the Power Off state. Then when the power is restored, the Running state is entered at the History State symbol meaning that it should resume where it last left off.

Concurrent Regions

A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

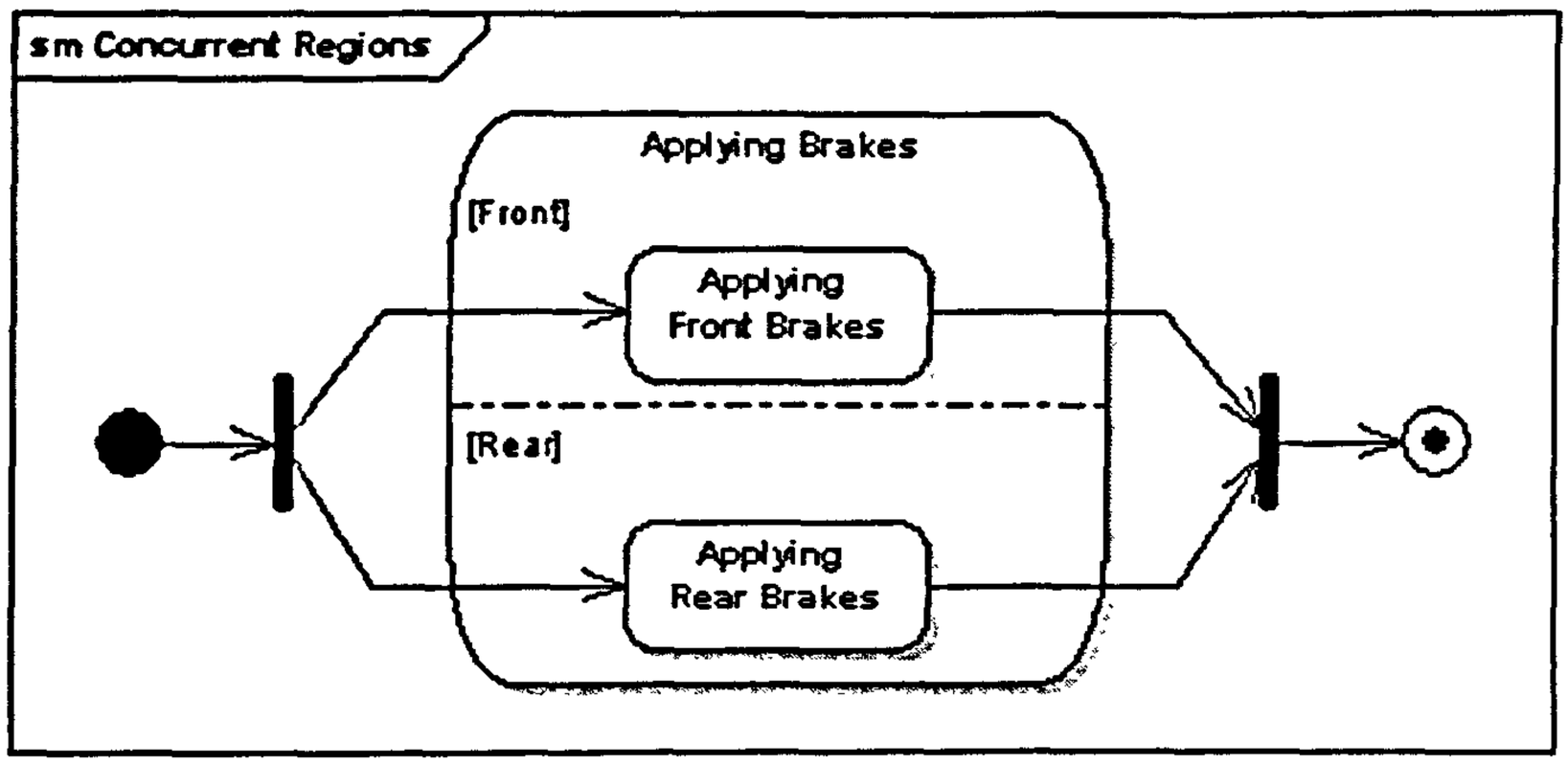


Figure C.0.16 Example of concurrent regions

Section C.2: Step-by-Step State Machine Diagram Drawing Guide

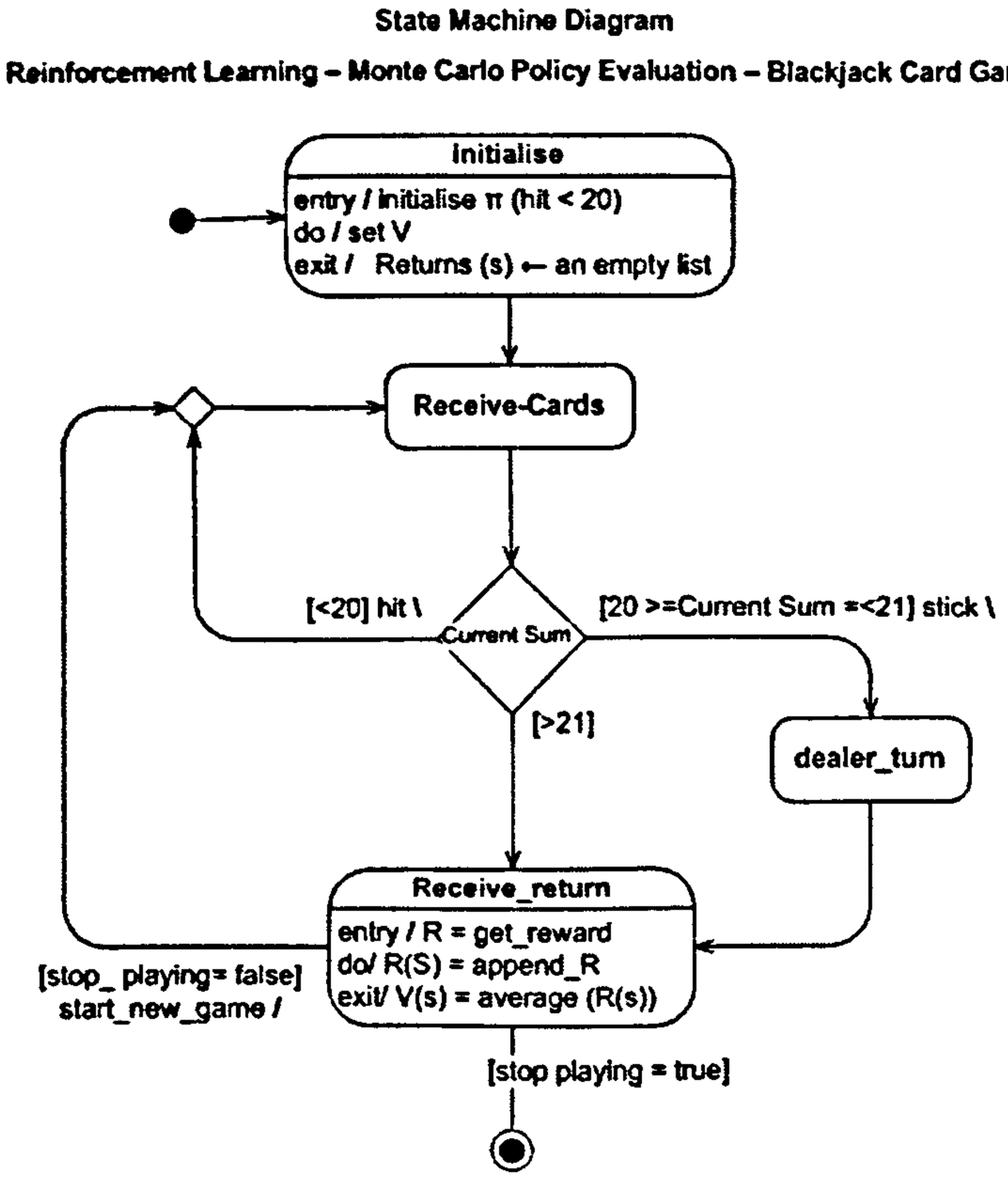


Figure C.0.17 UML state machine diagram for Black Jack game

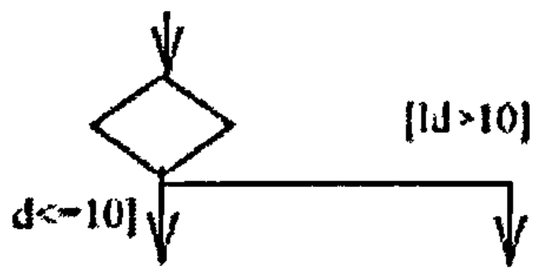
- Step 1: Identify the Intelligent Agent states
    - Identify and draw the states that the Intelligent Agent exhibits during this state machine diagram. The above diagram shows four states: initialise, receive-cards, receive-returns, and dealer-turn states
- Step 2: Identify any choice or junction pseudo states
    - Draw choice or junction pseudo states. The black jack diagram shows choice state titled "current-sum."
- Step 3: Identify if there is any composite or concurrent state
    - Draw the symbol of the composite diagram on the relevant state
- Step 4: Draw the transitions between the states identified above .
    - Draw the transitions between the states as needed
    - Write the trigger / guard on each relevant transition arrow.

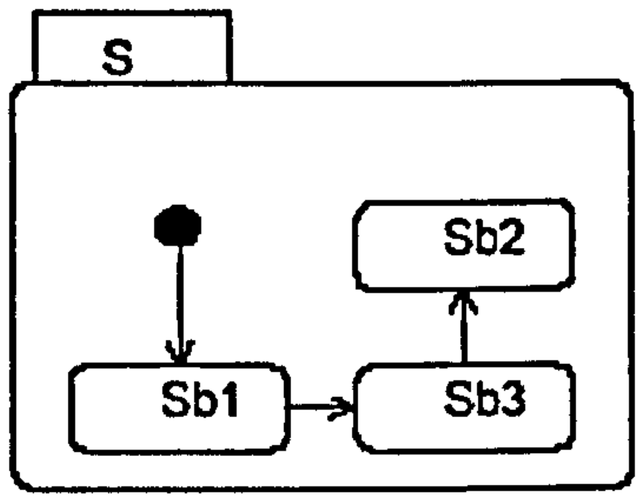
- Step 5: For each state identify the entry, exit, and do actions
    - For each state, write the entry, do, and exit actions if needed.

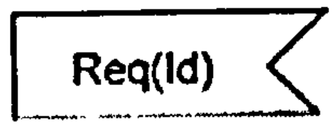
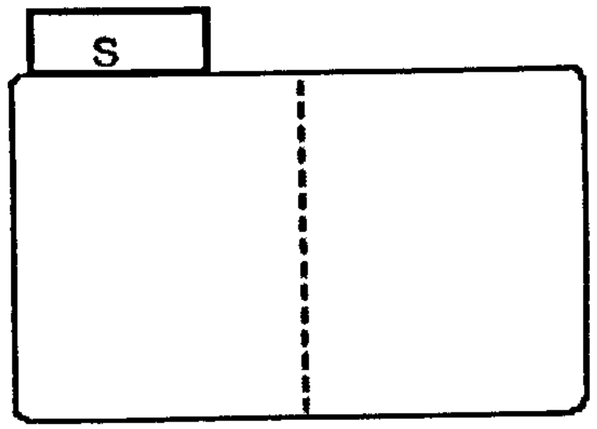
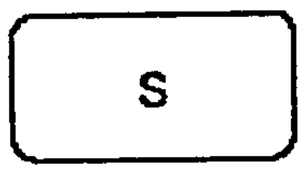
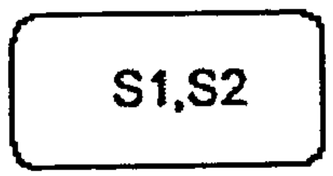
      Receive-return state has three actions : one on entering this state, the second is executed during the state, and the third on exiting the state.
- Step 6: Identify concurrent region
    - Draw any concurrent region needed to identify concurrent behaviour.
- Step 7: Identify initial and terminate states
    - Draw the initial and terminate states

Section C.3: Table of State Machine Diagram Components

The following table illustrates the main components of UML 2.0 state machine diagrams as mentioned in *Unified Modelling Language: Superstructure version 2.0, (OMG, 2004).*

Component	Notation	Reference
Action		An action symbol is a rectangle that contains a textual representation of the action represented by this symbol. Alternatively, the graphical notation defined for this action, if any, may be used instead. The action symbol must follow the signal receipt symbol if one exists for that transition. The action sequence symbol is mapped to an opaque action, or to a sequence node containing instances of actions, depending on whether it represents one or more actions, respectively.
Choice Pseudo state		Choice vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to completion step.

Composite state		A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint sub-vertices and a set of transitions. A given state may only be decomposed in one of these two ways. Any state enclosed within a region of a composite state is called a sub-state of that composite state. It is called a direct sub-state when it is not contained by any other state; otherwise, it is referred to as an indirect sub-state.
Entry point		An entry point pseudostate is an entry point of a state machine or composite state. In each region of the state machine or composite state it has a single transition to a vertex within the same region.
Exit point		An exit point pseudostate is an exit point of a state machine or composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state.
Final state		When the final state is entered, its containing region is completed, which means that it satisfies the completion condition.
History, deep pseudo state		Deep History represents the most recent active configuration of the composite state that directly contains this pseudostate
History, shallow pseudo state		Shallow History represents the most recent active sub-state of its containing state (but not the sub-states of that sub-state).
Initial pseudo state		An initial pseudo state represents a default vertex that is the source for a single transition to the default state of a composite state.

Junction pseudo state		Junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states.
Receive signal action		The trigger symbol is shown as a five-pointed polygon that looks like a rectangle with a triangular notch in one of its sides (either one). It represents the trigger of the transition.
Send signal action		Signal sending is a common action that has a special notation described in SendSignalAction. The actual parameters of the signal are shown within the symbol. On a given path, the signal-sending symbol must follow the trigger symbol if the latter exists.
Region		A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.
Simple state		A simple state is a state that does not have sub-states, i.e. it has no regions and it has no submachine state machine.
State list		The special case of the transition from the junction having a history as target may optionally be presented as the target being the state list state symbol.