# Modeling and Analyzing Software Behavior in UML

Ivan Porres

To be presented —with the permission of the Faculty of Chemical Engineering at Åbo Akademi University— for public criticisms in Auditorium 3102 at DataCity in Turku, Finland, on November 23, 2001, at 12 noon.

Department of Computer Science
Åbo Akademi University

## Supervised by

Professor Ralph Back
Department of Computer Science
Åbo Akademi University
Lemminkäisenkatu 14A
20520 Turku
Finland

## Reviewed by

Professor Michael Butler
Declarative Systems & Software Engineering Group
University of Southampton
Highfield Southampton SO17 1BJ
United Kingdom

Professor Wang Yi
Department of Computer Science
Uppsala University
Box 325, SE-75105 Uppsala
Sweden

ISBN 951-29-2169-3
ISSN 1239-1883

Painosalama Oy, 2001
Turku, Finland

To my family

The *Heart of Gold*'s Improbability Drive made it the most powerful and unpredictable ship in existence. There was nothing it couldn't do, provided you knew exactly how improbable it was the thing you wanted it to do would ever happend.

— Douglas Adams, The Hitch Hiker's Guide to the Galaxy

# Acknowledgements

This thesis is not the product of the work of a single person. After several years on the making, I am now really glad to be able to thank to everybody that has contributed in one or another way to this thesis.

I think of researching as a similar experience as steering Douglas Adam's *Heart of Gold*: There is nothing impossible, provided you know exactly what you really want to do. Fortunately, I had my supervisor Professor Ralph-Johan Back helping me on this task. He has been a constant source of advice, guidance and support during all this time.

I also want to thank Professor Michael Butler of the University Southampton, United Kingdom, and Professor Wang Yi of the University of Uppsala, Sweden, who kindly reviewed the thesis. They provided many constructive comments to improve it and I hope that I have been able to follow and integrate all their suggestions in this manuscript. Damon Tringham patiently corrected all the misuses of beautiful English language.

Most of the ideas presented in this thesis appeared in their initial form in several conference proceedings and a journal publication. These publications were coauthored by Professor Ralph-Johan Back and also by Professor Johan Lilius and Luigia Petre. My collaboration with them was not only scientifically fruitful but also a warm personal experience.

Other colleagues from Åbo Akademi University have also help me greatly in my research. Christina Cerschi, Professor Hannu Toivonen, Mauno Rönkkö and Professor Joakim von Wright provided many interesting comments in our enlightening discussions on the topics discussed in the thesis. Robert Gyllenberg and Marat Vagapov built the hardware of the digital sound recorder and made the validation of its design possible.

The Turku Centre for Computer Science and the Department of Computer Science provided an excellent working environment and generous financial support. I want to express my sincere thanks to all my colleagues for contributing to an inspiring working atmosphere. Dr. Mats Aspnäs provided reliable answers to the most diverse questions. Dr. Martin Büchi, Dr. Anna Mikhajlova and Dr. Leonid Mikhajlov were an excellent role model to follow. I can only wish to all my fellow PhD. students and specially to Dag Björklund, Luka Milovanov,

Franck Tetard, Vioerel Preoteasa, Seppo Virtanen and of course, Christina, Luigia and Mauno the best for their studies.

Finally, I want to thank my family and friends and specially to Åsa, for their continuous and unconditional love and support, wherever I am.

Turku, October 4 2001

Ivan Porres

# Contents


**1 Introduction** **1**

1.1 Introduction . . . . . 1
1.2 UML . . . . . 2
1.2.1 Semantics of UML . . . . . 3
1.2.2 Formal semantics . . . . . 5
1.3 The Quality of UML models . . . . . 6
1.4 Analyzing UML Models . . . . . 7
1.4.1 Analyzing Use Cases . . . . . 9
1.4.2 Analyzing Statecharts . . . . . 10
1.4.3 Hybrid Statecharts . . . . . 11
1.5 List of Publications . . . . . 11

**2 Modeling in UML** **13**

2.1 Introduction . . . . . 13
2.2 Object-Oriented Software Development . . . . . 14
2.3 UML . . . . . 16
2.3.1 OCL . . . . . 16
2.3.2 Tool Support . . . . . 17
2.3.3 Profiles . . . . . 18
2.3.4 UML is just a Modeling Language . . . . . 19
2.4 Requirements of the Sound Recorder . . . . . 19
2.5 Analysis: Object Structure . . . . . 24
2.5.1 Identifying Objects . . . . . 28
2.5.2 Building the Class Diagrams . . . . . 29
2.6 Analysis: Defining Object Behavior . . . . . 38
2.6.1 The User Interface . . . . . 39
2.7 Architectural Design . . . . . 39
2.7.1 Architectural Patterns . . . . . 42
2.7.2 Concurrency Design . . . . . 43
2.8 Mechanistic Design . . . . . 43

2.8.1 Collaboration Between the Hardware and the Reactive Objects. . . . 44
2.8.2 Collaboration Between the Reactive Objects and the User Interface . . . 44
2.8.3 Pattern: Reactive Subject . . . 44
2.8.4 Scheduler . . . 46
2.8.5 Changing the Time . . . 46
2.8.6 Playing a Message . . . 47
2.9 Detailed Design . . . 47
2.9.1 Objects Modeling the Hardware . . . 47
2.9.2 Sound Compression Algorithm . . . 49
2.9.3 Using Direct Memory Access Channels . . . 49
2.9.4 Memory Allocation . . . 52
2.10 Comments on the Implementation . . . 52
2.11 Conclusions . . . 53

**3 Analyzing Use Case Models as Contracts 57**
3.1 Introduction . . . 57
3.2 Use Cases . . . 59
3.2.1 Why Use Cases? . . . 60
3.3 Contracts . . . 64
3.4 Example: Library . . . 69
3.5 Analysis . . . 76
3.5.1 Proving properties . . . 77
3.6 Related Work . . . 83
3.7 Conclusions . . . 85

**4 Statecharts 87**
4.1 Introduction . . . 87
4.2 Structure of State Machines . . . 88
4.3 Dynamics of State Machines . . . 94
4.3.1 Calculation of the Step . . . 95
4.3.2 The Run-To-Completion Step . . . 99
4.3.3 Other model elements . . . 103
4.4 Interaction diagrams . . . 103
4.4.1 Particular Comments on the Semantics . . . 104
4.5 Conclusions and Related work . . . 105

**5 Verification of Collaborations of Statecharts 107**
5.1 Introduction . . . 107
5.2 Modeling Concurrent Systems with Statecharts . . . 108
5.2.1 Describing Properties of a Model . . . 111

5.2.2 Collaboration of objects . . . 113
5.3 The vUML Tool . . . 115
5.3.1 Open Models . . . 116
5.3.2 Structure of the PROMELA Model . . . 117
5.3.3 Verification with SPIN . . . 122
5.3.4 Representation of Counter-Examples . . . 123
5.4 Example: The Production Cell . . . 124
5.4.1 Modeling the Press . . . 126
5.4.2 Modeling the robot . . . 129
5.4.3 Verification . . . 130
5.5 Conclusions and Related Work . . . 133
5.5.1 Evaluation of the tool . . . 133
5.5.2 Related work . . . 135

**6 Real-Time and Hybrid Behavior in Statecharts 137**
6.1 Introduction . . . 137
6.2 Hybrid Statecharts . . . 138
6.3 Action Systems . . . 140
6.4 Continuous Action Systems . . . 143
6.4.1 Semantics of Continuous Action Systems . . . 145
6.4.2 Systems Using Differential Equations . . . 148
6.4.3 Real-Time Systems . . . 148
6.4.4 Composition of Continuous Action Systems . . . 148
6.5 From Statecharts to Continuous Action Systems . . . 149
6.5.1 Transitions . . . 151
6.5.2 Examples . . . 152
6.6 Safety Properties . . . 155
6.6.1 Proving the Safety of the Press . . . 156
6.7 Conclusions and Related Work . . . 161

**7 Conclusions 163**

# List of Figures


1.1 Example of the Concrete Syntax of Statecharts . . . . . 4
1.2 Example of the Abstract Syntax of Statecharts . . . . . 5

2.1 External Appearance of the Sound Recorder . . . . . 20
2.2 Context-Level Diagram . . . . . 22
2.3 Use Case Diagram . . . . . 22
2.4 User Interface Mock-up . . . . . 23
2.5 Playing a Message Scenario . . . . . 25
2.6 Alarm while Playing a Message Scenario . . . . . 26
2.7 Entering Stand-by Mode Scenario . . . . . 27
2.8 Sound Recorder Main Class Diagram . . . . . 30
2.9 Subsystems in the Sound Recorder . . . . . 31
2.10 Audio Subsystem Class Diagram . . . . . 32
2.11 Representation of Sound Messages . . . . . 33
2.12 Play Message Sequence Diagram . . . . . 34
2.13 Message Memory Class Diagram . . . . . 35
2.14 Alarm Clock Class Diagram . . . . . 36
2.15 User Interface Subsystem Class Diagram . . . . . 37
2.16 Behavior of an AudioController . . . . . 38
2.17 MenuUserMode Statechart . . . . . 40
2.18 EditClockUserMode Statechart . . . . . 41
2.19 Hardware Architecture of the Digital Sound Recorder . . . . . 42
2.20 Task Diagram . . . . . 43
2.21 Event Proxy Class Diagram . . . . . 45
2.22 Scheduler Class Diagram . . . . . 46
2.23 Collaboration between the Scheduler and its Clients . . . . . 47
2.24 Model-View-Controller Collaboration . . . . . 48
2.25 Collaboration between the User Interface and the Audio Controller 48
2.26 Recording an Audio Block . . . . . 50
2.27 Recording Compressed Audio . . . . . 51

3.1 Use Case Diagram . . . . . 61

3.2 Formal Specification Syntax . . . . . 69
3.3 Library Class Diagram . . . . . 70
3.4 Complete Definition of the Class Book . . . . . 73
3.5 Complete Definition of the Class LibraryCard . . . . . 74
3.6 Library Class Diagram with Operations . . . . . 74
3.7 Library Contract . . . . . 75
3.8 Weakest Precondition of the BorrowBook Use Case . . . . . 80
3.9 Result of the Weakest Precondition of the BorrowBook use case . 81
3.10 Result of the wp Computation, when M=L . . . . . 81
3.11 The Librarian Result of the wp Computation . . . . . 82

4.1 Example Statechart . . . . . 89
4.2 Using History States . . . . . 90
4.3 Replacing History States . . . . . 91
4.4 Translation of the State Machine in Figure 4.1. . . . . 92
4.5 The Translation of Join and Fork Vertices . . . . . 95
4.6 Translation of Junction Vertices . . . . . 96
4.7 First Version of the Run-to-Completion Step . . . . . 100
4.8 The Final Version of the Run-To-Completion Step . . . . . 101
4.9 Collaboration of Objects . . . . . 104
4.10 Machine $B$ is able to Observe the Execution Order of $A$. . . . . 105
4.11 Machine $A$ will Deadlock . . . . . 105

5.1 Using vUML . . . . . 109
5.2 Class Diagram for the Dining Philosophers . . . . . 109
5.3 Behavior of the Forks (top) and Philosophers (bottom) . . . . . 110
5.4 Temporal Requirements Embedded in the Statecharts . . . . . 112
5.5 Class Diagram for the Dining Philosophers with Constraints . . 113
5.6 A Dinner for Four . . . . . 114
5.7 After Sara's Move, the Dinner is Suddenly Over . . . . . 115
5.8 Example of Why it is Necessary to Separate Selection from Firing 121
5.9 vUML Window Showing a Counter-Example . . . . . 124
5.10 Icons Showing When an Event is Deferred and Dispatched Again 125
5.11 Top View of the Production Cell . . . . . 125
5.12 Robot and Press . . . . . 126
5.13 Class Diagram of the Press . . . . . 126
5.14 Controller of the Press . . . . . 127
5.15 Model for the Plant of the Press . . . . . 128
5.16 Press Collaboration Diagram . . . . . 129
5.17 Overall Behavior of the Robot Controller . . . . . 130
5.18 Detailed Behavior of the Robot Controller . . . . . 131
5.19 Detailed Behavior of the Robot Controller (cont.) . . . . . 132

6.1 Hybrid Statechart Saw . . . . . 140
6.2 Evolution of the Hybrid Statechart Saw . . . . . 141
6.3 Continuous Action System *Saw* . . . . . 144
6.4 Vertical Position of the Press . . . . . 153
6.5 Hybrid Statechart for the Press . . . . . 154
6.6 First Version of the Press as a Continuous Action System. . . . . 154
6.7 Final Version of the Press as a Continuous Action System. . . . . 155

# List of Tables

2.1 External Event List . . . . . 21
2.2 Operations to Play or Record a Message . . . . . 31
2.3 Classification of UML Diagrams . . . . . 54

3.1 Use Case Borrow Book . . . . . 62
3.2 Use Case Renew Loan . . . . . 62
3.3 Use Case Return Book . . . . . 63
3.4 Use Case Pay Fee . . . . . 63

5.1 Performance of the vUML Tool . . . . . 133

# Chapter 1

# Introduction

## 1.1 Introduction

Software is everywhere. Computers are an integral part of our society and economy and their software becomes larger and more complex every day. Software development is a difficult activity and many software projects take longer than expected, have large operational failures or are simply cancelled [39]. These problems are one of the main concerns of the software engineer community. It is proposed that one way to reduce the complexity of software development is to build software models.

A *model* is a simplification of the reality that still retains the elements relevant to our problem. The purpose of a software model is to represent and convey the most important requirements and design decisions of a software system. A good model should explain these concepts in an easy way and ignore all the unimportant details. A detail is unimportant if, for example, it is well known by all the members of the development team or if it can be easily inferred from the rest of the model.

A software model is described using a modeling language. Currently, the most widespread software modeling language is the Unified Modeling Language (UML) [77]. UML is widely used and it is accepted by the industry as the standard language for software analysis and design.

UML is a graphical language. Software concepts are represented visually as icons. Harel has presented interesting arguments in favor of using a visual representation in software development in [43]. A UML model is usually composed of several diagrams. This division of a model into different diagrams is an important mechanism to manage software complexity. Each diagram represents a different aspect or view of a system. We can see many examples of these diagrams in the sound recorder case study, a complete UML model that is presented in Chapter 2. This case study was performed with the help of Johan

Lilius and is based on publication number 1.

UML models are created and used during the whole software development process. We can start drawing UML diagrams during the initial planning and analysis stages, even when we still do not have a clear idea of how our system will look. We can refine and create more diagrams until we have a more concrete design. Then, the software is constructed according to the model. Thus, the quality of the final software depends on the quality of the model: if a model contains a wrong design decision, the actual program code will be built upon that decision.

In this thesis we present different analysis methods that can be used to assure the quality of a UML model. Analyzing a UML model is asking and answering questions about it. Safety critical systems, where the life of persons can be at risk if the software does not behave as expected, are obvious candidates for software analysis techniques. Nevertheless, there is also an economic argument that can be applied in any software development: UML models are created in the early stages of the development process and therefore the correction of mistakes during analysis and design phases can be achieved more cheaply and rapidly than during the implementation or when the software is already deployed. This principle still holds in evolutionary or spiral [15] software processes, since design always precedes coding. Also, UML models are usually simpler than their implementation. Hence, analyzing a UML model can be simpler and cheaper than analyzing its implementation as a C++ or Java program. Any development team concerned with the final quality of its software should be interested in verifying and validating its software models as early in the software process as possible.

## 1.2 UML

The Unified Modeling Language is a standardized notation for modeling and documenting object-oriented software and business processes; it is the result of the fusion of the best practices in software modeling and visual notations and is under constant evolution. Its origins are based on the previous work of Booch, Rumbaugh and Jacobson. The Object Management Group (OMG) is currently in charge of its maintenance and revision. A large group of companies and universities are members of the group and may propose and vote upon enhancements and corrections to the OMG specification.

The adoption of one single standard modeling language provides many benefits to software developers. Training is simplified since there is just one language to learn. It also simplifies the communication between development teams that now can exchange models using the common language. A large user base enables a market for sophisticated tools, training and consultancy services and

attracts the attention of the scientific community, which works on the improvement of the language. These are not benefits of UML by itself but of the idea of one standard language for all modeling needs.

However, a unified language also has an important shortcoming: UML is complex and bloated. Beginners can find UML too large and most of the tools do not support all the features of the language.

UML models are usually composed of different diagrams. Some UML diagrams show the static structure of a software system. For example, a class diagram shows a collection of classes, types and interfaces and their associations. A class diagram can show the operations of each class but it does not describe the actual behavior of these operations.

Behavioral diagrams, such as sequence diagrams and statecharts, can be used to describe the dynamics of a model. These diagrams can describe what happens when invoking an operation or the whole lifetime of an object.

A complete UML model always contains structural and behavioral diagrams describing both the static and dynamic aspects of a system. However, this thesis is focused on the study of behavioral diagrams.

### 1.2.1 Semantics of UML

The definition of UML is covered by a document named OMG Unified Modeling Language Specification [77]. This document contains the definition of all the elements that form the UML language. It is large: Version 1.3 from March 2000 has 1034 pages. The document is also more oriented towards tool developers and software scholars than towards the average UML practitioner.

UML, like other computer languages, is defined by three elements: its concrete syntax, its abstract syntax and its semantics. The UML notation guide, a chapter in [77], describes the concrete syntax or graphical representation of each model element.

The abstract syntax is described by the UML metamodel. The UML metamodel is a UML model describing what is a valid UML model. It enumerates all model elements, their attributes and relationships with other model elements. The UML metamodel only uses class diagrams and constraints in the Object Constrain Language (OCL). OCL [102] is a textual language for annotating and constraining model elements. It is an expression language, i.e. it is free of side effects. It uses a syntax similar to object oriented programming languages and it replaces mathematical symbols, like $\forall$ or $\exists$, for words in English, like forAll or exists.

The semantics of UML is given in natural language. The purpose and behavior of each model element is given using concise but some times confusing explanations in plain English.

As an example, Figure 1.1 shows the concrete syntax for some elements of

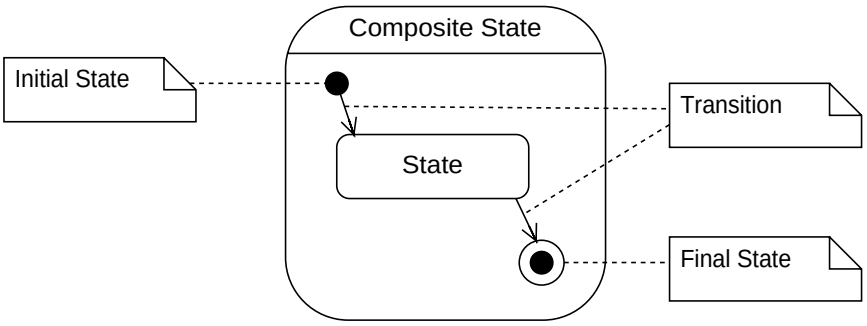



Figure 1.1: Example of the Concrete Syntax of Statecharts

the notation used in UML statecharts. A final state in a statechart is depicted as a black dot inside a circle. The UML metamodel for these elements is shown in Figure 1.2. This diagram shows that a state is a special kind of state vertex. Each state vertex can have many incoming and outgoing transitions. States can also have internal transitions. There are (at least) three kind of states: simple states, composite states that can contain other state vertices and final states. This diagram is also annotated in [77] using OCL expressions. In the case of final states there is a constraint indicating that final states cannot have outgoing transitions or in other words the number of outgoing transitions is zero. We write this constraint in OCL as:

```
context FinalState:
  outgoing->size=0
```

The meaning of each model element is defined in plain English. The semantics of a final state, according to the UML standard, is

> *a special kind of state signifying that the enclosing composite state is complete. If the enclosing state is the top state, then it means that the entire state machine has completed.*

The UML specification document is ambiguous since UML is explained in plain English and UML itself. It fails to give a precise meaning for each UML model element and it does not explain the interactions between model elements. The fact that it is possible to create a metamodel of UML using a subset of UML is interesting and it shows the power of the language, but it absolutely does not help to define its dynamics.

Ambiguous semantics lead to a situation where each member of a team is free to interpret a UML model in a different way. If a programmer interprets a model in a different way than a designer, the programmer will implement a system

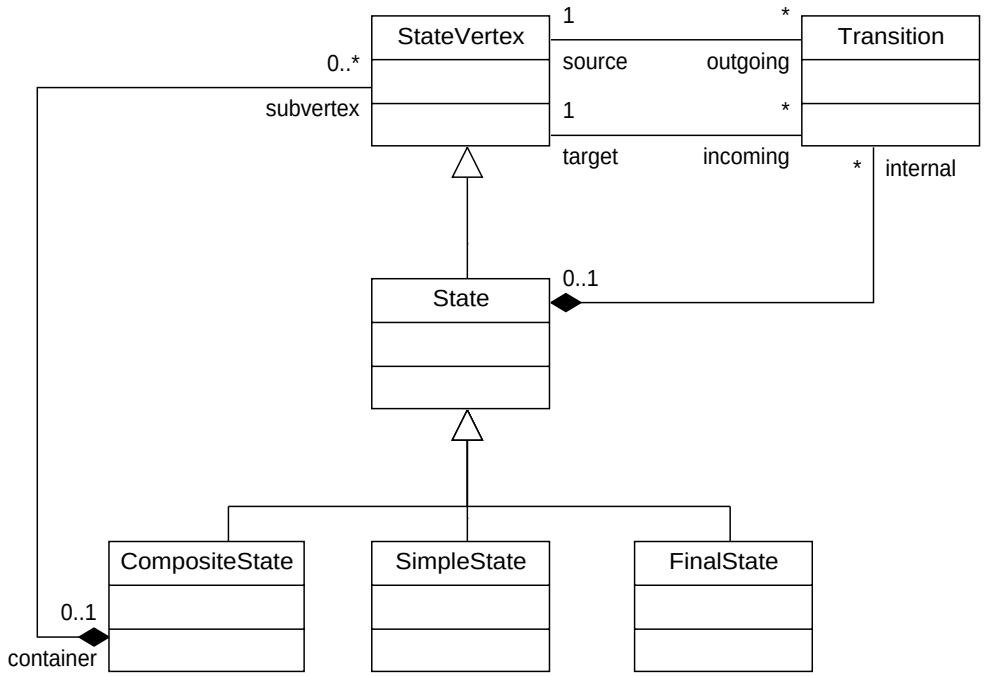



Figure 1.2: Example of the Abstract Syntax of Statecharts

that does not correspond to the design. All benefits of a standardized language are then lost. Ambiguous semantics also prevent the construction or utilization of advanced tools for code generation, model animation and validation.

### 1.2.2 Formal semantics

To remove ambiguities in the language and to remedy the previous situations, many authors are working towards a formal semantics for UML [59]. Formal semantics give an unambiguous interpretation to UML using a mathematical formalism and enable static and behavioral analysis of UML models.

We can give formal semantics to UML by introducing a mapping from UML diagrams to a well established formal method for software development. Examples of this approach are the formalizations of UML into Z [35], Object-Z [57, 58] or VDM [99]. Once a UML model is translated into a specification in a formal language, all the analysis and manipulations are performed on the formal specification. The clear advantage of this approach is that it is possible to reuse all the previous work on the formal method for analyzing UML models.

Other authors work on the formalization of UML directly in a mathematical theory, like set theory or category theory [93]. The advantage of this approach is that the underlaying mathematical theory is usually more abstract and imposes

fewer restrictions than a formal method that was not specifically designed for UML.

## 1.3 The Quality of UML models

Once we have a clear understanding of the UML language and UML models we can develop different techniques to assure and improve the quality of a model. These techniques are important for the UML user since the quality of her final software depends on the quality of the software models. We can classify these techniques as syntax checking, type checking, metrics, full behavioral verification and full validation.

### Syntax and Type Checking

The first issue we should address when analyzing a model is whether it is actually a UML model. A UML model is legal or syntactically correct if it conforms to the UML notation guide and the UML metamodel. We expect that tools that claim to be UML compliant only produce syntactically correct UML models.

Another interesting question is whether a UML model is type consistent. Clark presented in [22] a type system for class diagrams and OCL expressions that ensures that there are no illegal field references and that all values supplied to basic operations are contained within the operation's domain.

### Metrics and Critics

Some UML tools can calculate various metrics of a model. Metrics can provide hints about the quality of a model. As an example, a diagram with too many classes can be too difficult to understand while a class with too many attributes is a certain candidate for refactoring [34]. However, the designer is still responsible for deciding how many is too many.

Other tools, like Argo/UML [84], implement critics or automatic auditors. A critic is a program that can detect common design errors in a model, like a class without a constructor or a statechart without an initial state. A critic can also enforce style guidelines, like capitalization of the first letter of a class name or avoiding using program keywords as names.

Critics are based on heuristics and cannot ensure the final correctness and fitness of a design, but can help the designer to improve the overall quality of the model.

### Full Verification

The most demanding type of analysis of a UML model is a full formal verification. By formal verification, we mean a mathematical proof that a UML model *always* satisfies a certain property. If we are able to express our UML model and requirements in a mathematical formalism, we can use a theorem prover or a model checking tool to obtain such proof.

### Validation

Verification answers the question of whether we are building the model right. Validation answers the question of whether we are building the right model. Validation must be performed against the requirements of the final customer of the application. In this context, tools for animating and rapid prototyping of UML models can be of great help. USE [83] is an example of a tool for validating UML models. It can create object diagrams from class diagrams and evaluate their OCL constraints. Rhapsody from I-Logix is a UML tool that animates statecharts. According to Objexion Software, their Model Prototyper tool features automatic prototype generation from the UML models including persistence and user interface.

## 1.4 Analyzing UML Models

The main contribution of this thesis is a collection of verification techniques for use case diagrams and collaborations of statecharts. In order to develop these techniques we need to formalize some aspects of the UML language. For this purpose we have used two different formal methods: the refinement calculus and model checking.

### Refinement Calculus

The refinement calculus is a framework developed by Back and von Wright [12] for stepwise software development. The key characteristic of the refinement calculus is the formal refinement of specifications. Software is modeled in an imperative state-based specification language. An abstract specification can be stepwise refined into a better, more concrete specification that still preserves the original correctness requirements. By using refinement steps we can develop software from the initial requirements to the final implementation that is correct by construction.

The semantics of the specification language is based on Dijkstra's weakest precondition semantics for programs [29]. The weakest precondition predicate transformer computes the weakest initial predicate that guarantees that a given postcondition is always achieved by the specification.

The mathematical foundations of the refinement calculus are lattice theory and higher-order logic. This allows us to prove the correctness of the specifications in a rigorous manner. The refinement calculus has been formalized as a theory in the HOL theorem prover [101]. Hence, we can use this computer tool to perform proofs.

Higher-order logic is an extension of simply-typed lambda calculus. Functions are defined by $\lambda$-abstraction and can be used without explicit definition and naming. For example, the function that calculates the successor of a natural number is defined as $(\lambda n \cdot n + 1)$. We denote by $f.x$ the application of the function $f$ to the argument $x$, so that, e.g. $(\lambda n \cdot n + 1).1 = 2$. A binary relation $R \subseteq A \times B$ is here considered as a function $R : A \rightarrow \mathcal{P}B$ i.e. mapping elements in $A$ to sets of elements in B.

The refinement calculus defines a hierarchy of specification languages. In this thesis, we use two specification languages: action systems and contracts.

An action system is a construction in the refinement calculus used to model concurrent and distributed systems. The action system formalism was introduced by Back and Kurki-Suonio [7] in 1983. An action system is defined as a finite set of guarded actions that updates a finite set of shared attributes. Each action is atomic and it is executed as soon as its guard is enabled.

The most expressive specification language within the refinement calculus is the language of contracts. A contract is a specification binding the behavior of multiple agents. We define an agent as an entity with free will. A computation is seen as a number of agents (programs, modules, systems, users, etc...) that carry out actions according to a contract statement given in advance.

### Model Checking

Model checking [23] is the search of the complete state-space of a model for a given incorrect behavior. Model checking has been successfully applied in industrial verification projects, mainly in hardware and communication protocols. Nevertheless, nothing prevents us from applying the same technique in a UML model.

Model checking requires that a model is expressed as a finite automaton. Each possible behavior of our model corresponds to one possible run of the automaton. The property to be verified is expressed as a formula in a temporal logic. In this thesis we use propositional linear temporal logic (LTL) [80]. Linear temporal logic include operators such as $\Box$, read "always" and $\Diamond$, read "eventually" to model system executions. As an example, the LTL formulae $\Box(button1press \rightarrow \Diamond arrivefloor1)$ can be interpreted as "it is always true that when we press button 1 the elevator eventually arrives to the first floor". It is possible to convert a LTL formula into an automaton. The same applies to the negation of a LTL formula. If we intersect the automaton representing our

model with the automaton representing the negation of our property, we obtain a new automaton that represents all the runs that violate the property.

Using this approach, a model checking tool can verify 100% of the behaviors of a model automatically. If the model satisfies the property, i.e. it is correct, the result of the intersection is an empty automaton. If not, we can use the new calculated automata to produce counter-examples. A counter-example is a run of our model that violates the property to verify. By studying the counter-examples, we can fix our model and repeat the process until the model is correct.

The main challenge in the application of model checking is the state space explosion problem [98]. Modern model checkers can apply many optimizations, like partial order reduction [97] or on the fly verification, to reduce the memory needed and speed up the verification. It is also possible to perform an approximated verification, where not all possible behaviors are explored. In this case, we can still control and measure the coverage of the verification.

### 1.4.1 Analyzing Use Cases

The analysis of use case diagrams is the main topic of Chapter 3. A *use case* diagram specifies the required functionality of a system, showing the collaboration among a set of actors that are to perform certain tasks without revealing the internal structure of a system.

Use case diagrams are used to model both the context and the requirements of the system under development. The context of the system defines the environment that the system lives in and includes the actors of the system. The actors are entities with free will and have certain goals to fulfill. They impose some functional requirements over the system. We consider use case models as informal contracts between the system and the actors. If the actors behave as described in the use case, the system promises to deliver the functionality also described in the use case. A well-behaved system should carry out all its required use cases predictably and reliably.

We complement the use case diagrams by providing formal documents similar to specifications or programs, called contracts, which regulate the behavior of the actors involved. A contract is written in the refinement calculus and thus it can be analyzed mathematically. In particular, we can check whether a specific actor can achieve given goals with the contract, in spite of possibly conflicting behavior with other actors. In order to analyze a contract we define the precise meaning of each statement we use as weakest precondition predicate transformers [13, 12].

As an example, we show how to model and analyze a simple use case model for a library system. We create a contract for the use case model and analyze it to compute under which conditions a member of the library can use the system

to borrow a book.

This work has been performed with the collaboration of Ralph-Johan Back and Luigia Petre and it is based on publication number 3.

### 1.4.2 Analyzing Statecharts

A statechart describes the behavior of a model element as changes in its internal state as the result of receiving external stimuli. Statecharts were introduced by Harel in 1984 [42] and they were quickly adopted as a visual notation to design concurrent and reactive systems. Statecharts are based on the concept of finite state machine, but the notation includes many new features like hierarchical and concurrent states to model complex behaviors concisely.

UML statecharts are based on Harel's work but include some new concepts and some important differences in the dynamics of the models. The UML variant is geared towards a software only implementation in an object-oriented programming language. Also, it assumes that a large system is modeled as a collaboration of many objects defined using one or more statecharts.

We discuss a complete formalization of the semantics of UML statecharts in Chapter 4. The formalization is given in terms of an operational semantics and it can be used as the basis for code generation, simulation and verification tools for UML Statecharts diagrams.

We present the formalization in two parts: a formalization of the structure and of its behavior. There are two reasons for this distinction: Many model elements composing a statechart are syntactic sugar and can be explained in terms of simpler elements. Also, it is possible to define different semantics for different application domains based on the same structure for statecharts. In fact, we present in Chapter 6 a dynamic model that includes real-time and hybrid statecharts. Chapter 4 also discusses some of the problems associated with the standard definition of statecharts as presented in [77].

We use this formalization of statecharts in the vUML tool, a tool for model-checking UML models. The vUML tool supports concurrent and distributed models containing active objects and synchronous and asynchronous communication between objects. To use vUML, the behavior of each object must be specified using one statechart.

The vUML tool introduces new stereotypes and constraints to describe the behaviors to be satisfied by the model. The designer uses these stereotypes to add some extra information into the design so the tool can extract the properties that must be verified. These new stereotypes are not part of the UML standard, but they have been introduced to allow the designer to describe safety and liveness requirements without having to learn temporal logic.

Internally the tool uses the SPIN [49] model checker to perform the verification. In the design of the tool, emphasis has been put on its usability so that

the designer does not have to know how to use SPIN or the PROMELA [48] language. If an error is found during the verification, the tool creates a UML sequence diagram showing how to reproduce the error in the UML model.

We use the vUML tool to study a model of the production cell [66] case study. The production cell is a standard example for evaluating methodologies for designing embedded systems.

This work has been performed with the collaboration of Johan Lilius and it is based on publications 2, 4 and 5.

### 1.4.3 Hybrid Statecharts

A real-time system is a system that must perform the right thing at the right time. A hybrid system [100] models continuous processes controlled by discrete computations. Real-time and hybrid systems are useful to study systems where a computer supervises and controls a physical phenomena from the real world, like the movement of an elevator or a chemical plant.

The UML standard does not have support for real-time and hybrid systems. There is an ongoing effort to create an extension of the language for modeling real-time systems in UML [94]. However, at the time of writing this text, this extension presents many new model elements but it does not present a precise model of time for the models.

To solve this situation, we present in Chapter 6 an extension of UML statecharts that can be useful to model real-time and hybrid systems. This extension consists of just a new model element: the continuous action. A continuous action can be used to model a real-time clock or a physical behavior described as a function of time.

We formalize continuous actions using the action systems language of the refinement calculus. We show how to use this new formalism to analyze safety properties of hybrid statecharts. As an example, we show how to analyze a hybrid model of the press of the production cell and verify that the press never moves out of its bounds.

This work has been performed with the collaboration of Ralph-Johan Back and Lugia Petre and it is based on publication number 6.

## 1.5 List of Publications

This thesis is based on the following publications:

1. Ivan Porres and Johan Lilius, Digital Sound Recorder: A case study on designing embedded systems using the UML notation, TUCS Technical Report No. 234, January 1999.

2. Johan Lilius and Ivan Porres, vUML: a Tool for Verifying UML Models. In Proceedings of the Automatic Software Engineering Conference (ASE'99), October 12-15 1999, Cocoa Beach, FL, USA. IEEE Computer Society.

3. Ralph-Johan Back, Luigia Petre and Ivan Porres, Analysing UML Use Cases as Contracts. In R. France and B. Rumpe (eds.) UML'99 - The Unified Modeling Language, October 28-30 1999, Fort Collins, CO, USA. Lecture Notes in Computer Science vol. 1723, pp. 518-533.

4. Johan Lilius and Ivan Porres, Formalising UML state machines for model checking. In R. France and B. Rumpe (eds.) UML'99 - The Unified Modeling Language, October 28-30 1999, Fort Collins, CO, USA. Lecture Notes in Computer Science vol. 1723, pp. 430-445.

5. Johan Lilius and Ivan Porres, The Production Cell: An Exercise in the Formal Verification of a UML Model, In Proceedings of the 33th Hawaii International Conference on System Sciences (HICSS'33). January 4-7 2000, Maui, HI, USA. IEEE Computer Society.

6. Ralph-Johan Back, Luigia Petre and Ivan Porres, Continuous Action Systems as a Model for Hybrid Systems. In Nordic Journal of Computing, vol 8(1); pp. 2-21, Spring 2001.

# Chapter 2

# Modeling in UML

## 2.1 Introduction

This chapter is a brief introduction to the Unified Modeling Language (UML) [77], a standardized notation for modeling software intensive systems. We review the origins and goals of UML and we show how it can be applied in a software project through a complete case study using UML. UML is large and complex and it includes many different concepts and types of diagrams. In UML, it is the responsibility of the designer, and a source of confusion for the beginner, to decide what type of diagram fits best each design concept. We hope this case study will help us to understand how to apply UML in a project, from the requirements to the detailed design.

Our case study presents the complete design of a digital sound recorder or dictaphone. Our target application is simple: basically it just records and plays back speech. Nevertheless, it is a challenging exercise in UML. Data modeling plays a relatively small role in the design when compared with the importance of capturing real-time behaviors and user interaction.

There are several digital sound recorders available in the market. The model described here has been designed following the specifications of a commercial product from a well-known manufacturer. The design was validated by implementing it using the C++ programming language and a custom board based on a 32-bit embedded controller.

This design has been created following a software process targeted on real-time embedded systems [30]. This software process is iterative and, in fact, we ran through two iterations of the process before obtaining the current design. However, we present the design in a sequential fashion following the tasks described in a waterfall-like process. The reason for this is that we believe that it is easier to understand a finished design if it is presented in this way. Also, we felt it was unnecessary to show the mistakes produced during the first iteration

of the process.

We proceed as follows. The origins of UML and the current status of the language are reviewed in sections two and three. Section four introduces our case study that is then developed through the rest of the chapter. The fifth section discusses the object model of the system and presents the main class diagram. Section 2.6 continues the object-oriented analysis but now focusing on the internal behavior of each object. Section 2.7 deals with the architectural design. There we show the hardware architecture of the sound recorder and the concurrency model, where we assign each object to an execution thread. The design continues with the definition of the collaborations between the different objects. This is done in Section 2.8, where design patterns [37] are used to glue together the classes defined in the analysis phase. Specific design issues are discussed in Section 2.9. Finally, section 2.10 briefly discusses the implementation.

## 2.2 Object-Oriented Software Development

The roots of UML are found in the object-oriented software paradigm. The concept of object-oriented programming was introduced by Dahl and Nygaard with Simula [27] in 1966. But it was Alan Key who consolidated these ideas in the Smalltalk programming language [41]. Objects combine attributes for representing and storing data with operations to manipulate the attributes. Object-oriented programs are constructed as a collection of objects that collaborate by exchanging messages. Object-oriented programming languages (OOP) proliferated during the 80s with the introduction of C++, Objective C, Self, Eiffel, etc....

It was also during the 80s that the ideas introduced by OOP languages were generalized into software engineering principles. The term object-oriented design was first used by Booch [16, 17] in 1981. In 1986, Booch began to refer to his approach as "object-oriented development" [18].

Object-oriented software development advocates the use of objects during the whole software development cycle. The four main activities in software development are analysis, design, implementation and testing. We use object-oriented analysis to define *analysis objects* that capture the problem domain. A *problem domain* is a reusable description of the problem to solve and its context. Object-oriented design proposes a solution for our problem domain by creating and combining *design objects*. Finally, the design is realized into *program objects* using an object-oriented programming language.

These activities are orchestrated according to a software process. A *software process* describes how to manage a software project. It defines a series of tasks or phases to be performed in a given sequence. Large projects can run over years and require skilled planning. A software process contains some of the

knowledge needed to run such projects.

Each process or method presents a unique way of building software, trying to maximize the probability of success for the project by optimizing resources and minimizing risks. For example, a waterfall-like software process advocates a complete analysis of the problem domain before starting the design phase. Coding starts when the design is complete and the code is tested when the program is implemented. The main drawback of this idealized process is that we only validate our work when it is complete, e.g., we can only detect analysis errors during the final test step. By contrast, a spiral process [15] recommends interleaving the analysis, design, coding and testing in a succession of small steps, validating the project after each step.

Many complete software methods appeared during the period between 1985 and 1994, like Booch, Coad-Yourdon [24], Fusion [25] by Coleman, OMT [32] by Rumbaugh, OOSE [55] by Jacobson and Shlaer-Mellor [92]. Each method has its own unique strength but they also share many common concepts and practices.

Each method also introduces differences between the deliverables to be produced during the analysis and design phases. The deliverables are usually a combination of textual descriptions and graphical models. A *model* is a simplification of the reality that still contains the elements relevant to the problem. An analysis model represents a simplification of the problem domain while design model represents a particular solution to the problem.

Models are less detailed than programs. A program is an implementation of a design model, so detailed that even a computer can execute it. By contrast, a model is simpler and easier to understand. We can find an analogy between software models and architectural floor plans. A floor plan contains all the important directions needed to build a building, e.g. where to place the doors and windows, but it does not contains the small details, like the position of each individual brick. Software models show a general *floor plan* of a program, but do not show each local variable or program statement.

Unfortunately, each one of the previous software methods uses a different notation for documenting the project and describing the models. Even the most basic design concepts are represented in complete different ways depending on the process used to build the software. As an example, Booch represents a class as a "cloud", while OOSE represents it like a rectangle. These differences in the modeling notations restricted the use of these methods, fragmented the object-oriented market and complicated the development of computer-aided software engineering (CASE) tools.

Because of this situation, Booch, Rumbaugh and Jacobson started working together in 1994 towards the unification of their methods. In June 1996, they released version 0.9 of the Unified Modeling Language (UML), a language combining the strong points of their methods.

Surprisingly, the idea of combining the three methods probably does not come from the above authors. Rosenberg and Scott claim that they have been working in the ICONIX Unified Object Modeling approach [87], a methodology combining the strongest point of each of the three approaches, since 1993. Currently, ICONIX has also adopted the UML notation.

## 2.3 UML

The Unified Modeling Language is a standardized notation for modeling and documenting object-oriented software and business processes. It is the result of the fusion of the best practices in software modeling and visual notations.

UML is under constant evolution. The Object Management Group (OMG) is currently in charge of its maintenance and revision. A large group of software companies and universities are members of the group and may propose and vote on enhancements and corrections to the OMG specification.

The adoption of one single standard modeling language provides many benefits to software developers. Training is simplified since there is just one language to learn. It also simplifies the communication between development teams that now can exchange models using the common language. A large user base enables a market for sophisticated tools, training and consultancy services and attracts the attention of the scientific community, who works on the improvement of the language. These are not benefits of UML by itself but of the idea of one standard language for all modeling needs. However, a unified language has also an important shortcoming: since it has to satisfy the needs of all application domains the language becomes too large and difficult to learn.

### 2.3.1 OCL

Another element of the UML specification is the Object Constraint Language (OCL) [102]. OCL is a textual language for annotating and constraining model elements. It is an expression language, i.e. it is free of side effects. It was introduced by IBM and ObjecTime Limited in revision 1.1 of UML.

Some of the common uses of OCL include constraining associations between objects, defining the postcondition of an operation or defining class invariants. We can use OCL in any UML diagram. As an example, the following invariant states that it is not possible to have two employees with the same name in the same company:

```
context Company:
   inv:
    self.employee->forAll(e1,e2 |
     e1<>e2 implies e1.name <> e2.name)
```

This expression is an invariant for the model element Company. This is denoted by the `context` and `inv` keywords. An invariant is a condition that must hold always. The `->` operator is used to navigate an association, in this case a relationship between a company and its employees. `forAll` is the universal quantification, `implies` is the logical implication and `<>` stands for inequality. We can interpret the invariant in this way: for each two employees of a company, if the employees are not the same their names should be different.

### 2.3.2 Tool Support

UML can be used without any computer support, with paper and pencil, or with a generic drawing program. However, there is a need for tools with specific support for UML in large software projects. These tools not only allow us to draw fancy UML diagrams, but also often have additional features to support software development. There are many UML tools available on the market. Currently, there are more than 175 tools with prices that range from zero to more than 5000 USD per user. We can classify the functionality provided by UML-based tools in five large groups: diagram editing, code generation, model extraction, model verification and validation and model management. Most commercial UML tools implement some kind of functionality in all these areas.

A UML diagram editor helps the user to create, edit and print UML models. Ideally, a diagram editor would allow the user to create any legal UML diagram and only legal UML diagrams. By legal we mean diagrams that conform to the UML syntax specification. Unfortunately, at the time of writing there is no tool that has complete support for the whole UML standard [61]. This situation is aggravated when trying to use a specific UML extension. In this case, a generic drawing program can be more useful than a full-featured UML CASE tool. In this group we can also include drawing aids such as the electronic white board with handwriting recognition of UML symbols presented in [28].

A code generation module creates program source code based on a UML model. The power and utility of a code generator depends on the supported UML model elements and the target languages. Some code generation tools only produce the skeleton of a program based on the structure described in class diagrams. Other, more advanced, tools can generate full behavioral code based on statecharts. Almost all code generators support the C++ and Java programming languages, but there are tools that support scripting languages such as Python.

A model extractor creates a UML model by reverse-engineering program source code. Code generation combined with model extraction leads to *round-trip engineering*, where a model can be converted back and forth into a program. The most advanced form of conversion between models and code is *synchronized editing*, where UML diagrams and program code can be edited simultaneously

in two different windows of the same application.

Model verification and validation tools help the designer to ensure the final quality of a model. We have discussed some of these tools in Chapter 1 and we will present our own verification tool for statecharts in Chapter 5.

The last group comprises tools that are used for model management. The adoption of diagrammatic modeling languages means that ASCII text is no longer the primary form for describing software. Standard tools for manipulating program text, like diff and patch are not suitable for graphical models. In this group, we include tools for version control, visual differencing and web publishing UML models.

### 2.3.3 Profiles

UML is an expressive and rich language and it is used in many different application domains, from business software to embedded systems. Since the possible applications of UML are so varied, the language may not cover all the possible needs of a particular software designer. Therefore, there is a provision for extending the language by using stereotypes and tagged values.

A *stereotype* is a classification of model elements. We can mark a model element with an stereotype by using a special iconic representation of the stereotype or just by using the name of the stereotype between matching guillemets, i.e. ≪name≫. In Chapter 5, we use stereotypes to indicate that model elements represent disallowed behaviors. We mark those elements with the ≪invalid≫ stereotype.

A *tagged value* is a piece of additional data that we can add to each model element. A tagged value is a pair composed by a name or tag and a value string. UML does not give any interpretation to tagged values. As an example, a tool could use tagged values to track the author and revision number of each model element.

Stereotypes, tagged values and new model elements can be collected into a profile. A *profile* is a customization of UML for a specific application domain. We can think of a profile as an appendix to the UML standard. There are profiles for using UML with CORBA, scheduling and real-time applications, etc... There are also profiles for combining UML with other modeling languages, like the UML for SDL profile.

Given this variety of published and in-house profiles, some authors consider that UML is not a single language but a family of languages [26]. However, there is still a *standard UML*, maintained by OMG and common to all UML profiles.

### 2.3.4 UML is just a Modeling Language

It is important to realize that UML is not a software process nor a visual programming language.

UML specifies a notation for modeling software, but it does not describe how to actually build the models. This is one of the goals of UML, a standard language that can be used in any software process. However, UML is better suited towards a use-case driven, architecture-centric software process. The adoption of UML as a standard language has spawned a multitude of UML-based software processes and design methods. Rational Unified Process [62], Catalysis [31], ICONIX and Octopus/UML [3] are examples of post-UML software processes.

UML is not intended to be a visual programming language. Modeling is a completely different task than programming. A good model should describe the key elements of a system while abstracting from specific details. In theory, we can use any programming language to implement a UML model, but in practice we will obtain the best results and tool support by using a class-based object-oriented programming language, like C++ or Java.

## 2.4 Requirements of the Sound Recorder

In this section we introduct the digital sound recorder, a case study that we develop through the rest of the chapter. A digital sound recorder is a consumer electronic appliance designed to record and play back speech. The messages are recorded using a built-in microphone and stored in a digital memory. The user can quickly play back any message at any moment through a speaker placed in the front of the device. It should be small, light, easy to use and operated by a small battery. Figure 2.1 shows how our sound recorder could look like: a hand-held unit with a flat display and fairly large buttons.

These are the main features of the product that we consider in our design:

- Digital memory with ten different messages. The length of each message is limited by the available memory.
- Direct access to any message.
- Full function LCD Display. Easy to use with on screen menus. The display shows clear indications about how to use it and what it is doing.
- Alarm clock. The alarm beeps for 60 seconds or until the user presses the stop button. The current date and time is always shown on the display.
- Battery level indicator. The system beeps when the battery is low.

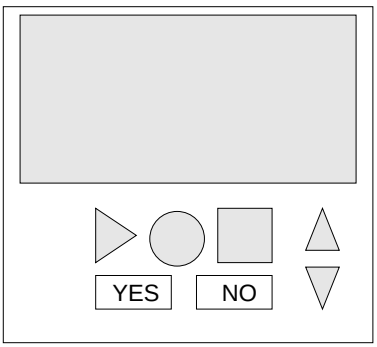



Figure 2.1: External Appearance of the Sound Recorder

- Stand-by mode. It economizes the battery power. Normal operation is resumed when the user presses a key.
- Good sound quality. Sound is processed at 6KHz using eight bits per sample.

**External Events**

An embedded system is constantly interacting with its environment. In this first stage of the analysis, we can consider our system as a black box reacting to messages from the environment. The environment is composed of several actors. Each actor interacts with our system with a different purpose and it exchanges a different set of messages. The context-level diagram on Figure 2.2 shows the actors that interact with the sound recorder. We have identified three actors: the user, the battery and the time. We consider the battery and the time as actors because we cannot control them. The battery can unexpectedly run out of power in the same way as the user can press a button at any moment. The time actor sends a message to our system whenever time passes. It represents the source of messages like "next second" or "next hour".

A context-level diagram also shows the interfaces, sensors and actuators that allow our system and the actors to exchange messages. These messages can be requests from the actors to the system or responses and services from the system to the actors. The sensors for the input from the user are the microphone and the buttons. The actuators for the user are the speaker and the display. The battery level meter senses the state of the battery while a crystal measures the time.

An event is a significant message from the environment. A real-time reactive system has to react to external events in a bounded time. We enumerate all external events that can occur in our system in Table 2.1. The direction of an

| Event | System Response | Dir | A | T |
|---|---|---|---|---|
| 1 Next second | Update internal clock<br>Check alarm<br>Update clock display<br>If playing or recording then<br>update progress | In | P | 0.5 s |
| 2 Next sample | Play or record next sample | In | P | 1/2 period |
| 3 Button press | Update user interface<br>Start user request | In | E | 0.5 s |
| 4 Low battery alarm | Warn the user and stop current task | In | E | 1 s |
| 5 Resume stand-by | Leave stand-by mode | In | E | 1 s. |

Table 2.1: External Event List

event can be *in*, from the environment to the system, or *out*, from the system towards the environment. The arrival pattern (A) can be *periodic*, if the event occurs on a fixed time basis, or *episodic*, if its arrival time is random. The Response time (T) sets an upper bound for the system response actions. The system will behave incorrectly if it does not react within the response time.

### Use Cases

Use cases describe the functionality of the system from the user's point of view. A user may be a person or another machine. Each use case is a different way of using the system and the completion of each use case produces a different result. In our system there are five main different use cases, represented in Fig. 2.3. In order to describe the use cases we also need to define the human-machine interface of the device. This includes its external appearance (Figure 2.1) and mock-ups of the graphical user interface (Figure 2.4). The interaction with the user is menu-driven and the display always shows the current menu. The top line shows the current time, date and the alarm time. At the bottom, there is a short help message for the user. The main use cases of our systems are: Select, play, record and delete a message, set time and date, display time and wake up alarm.

**Playback a Message** The user selects a message slot from the message menu and presses the play button. If the message slot contains a recorded message, then the system plays it through the speaker until its end or until the user presses the stop button. This use case includes the Select Message use case.

**Select Message** The user selects the Message option of the Main menu by using the up and down buttons. The user presses the yes button and the

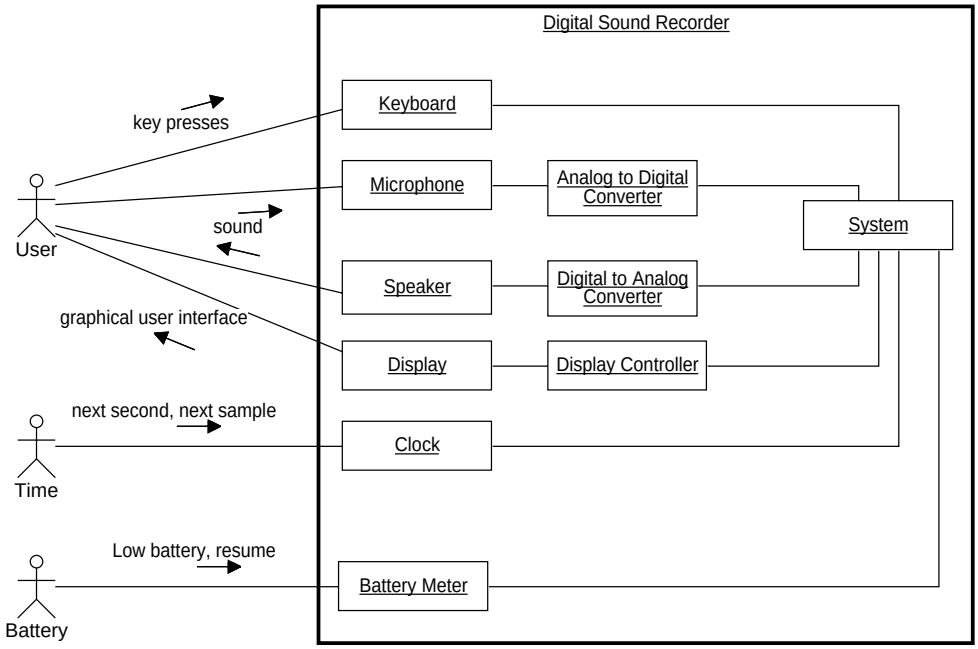



Figure 2.2: Context-Level Diagram

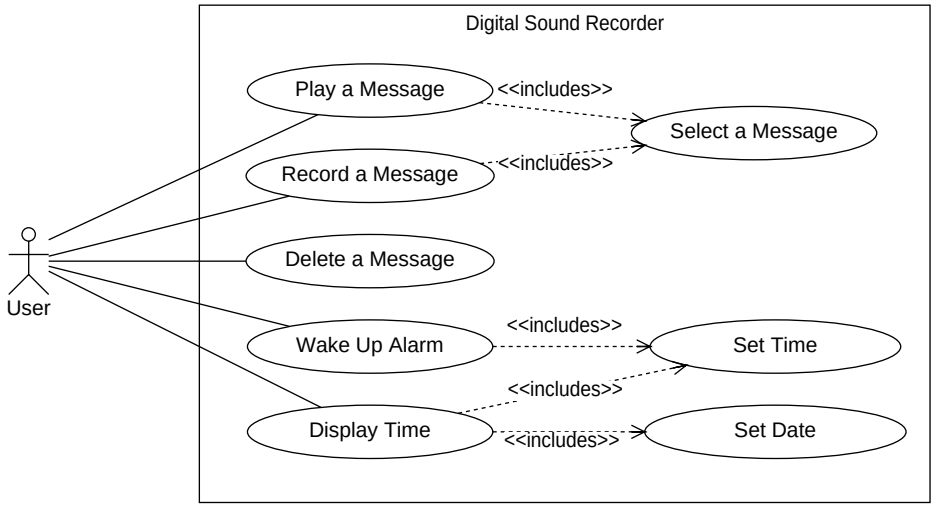



Figure 2.3: Use Case Diagram

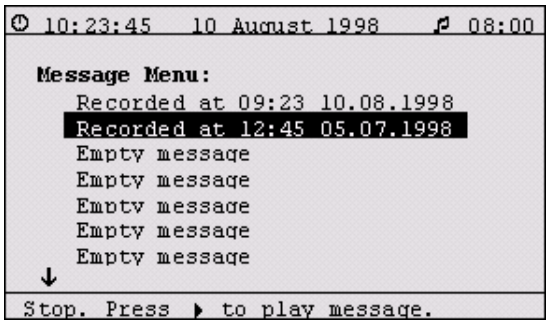



Figure 2.4: User Interface Mock-up

system shows the contents of message memory. The user selects a message slot with the up and down buttons.

**Record a Message** The user selects a message slot from the message menu and presses the record button. If the message slot already contains a message, it is deleted. The system starts recording the sound from the microphone until the user presses the stop button, or the memory is full. This use case includes the Select Message use case.

**Delete a Message** The user selects a message slot from the message menu and then presses the delete button. If the message slot contains a message then it is deleted from the memory. This use case includes the Select Message use case.

**Wake Up Alarm** The user selects the Alarm option of the Main menu by using the up and down buttons. The user presses the yes button and the system shows the contents of the alarm menu. The user can use the three options of the alarm menu to switch on and off the alarm and set the time when the alarm will sound. The wake up alarm beeps for 60 seconds or until the user presses the stop button. This use case includes the Set Time use case.

**Set Time** The user can set the hour and minute of a clock. This clock can represent the current time or the time of the wake up alarm. The yes button alternately selects between the hour and the minute field. The up button increases the value of the selected field while the down button decreases it. The user stops modifying the time by pressing the stop or no buttons.

**Display Time** The system constantly shows the current time on the display. The user can set the clock time and date using the two options of the

clock menu. This use case includes the Set Time and Set Date use cases.

**Set Date** The user can set the day, month and year of the system date. The yes button alternatively selects between the day, month and year fields. The up button increases the value of the selected field while the down button decreases it. The user quits this use case by pressing the stop or no buttons.

#### Usage Scenarios

Scenarios describe the interaction between the active external actors (the user, the battery and the time) with the system. Each scenario shows what happens when the actors behave in a given way. As an example, Figure 2.5 shows an scenario for the Play Message use case. The scenario starts with the user pressing the play button to play a message. In this scenario the battery does not interact with the system. This is an example of a scenario describing the main course of action of a use case. In this scenario, everything happens as expected by the user. However, each use case includes many alternative scenarios, describing other possible outcomes of the interaction with the system.

Describing all possible scenarios can be a complex and cumbersome task. Even if the role of each actor is usually clear, it can be difficult to study all the possible interactions between all the actors and the system. However, it is better to resolve these interactions as early in the design process as possible. In our example, we have to study what happens when the battery goes low while the system is playing a message or what to do if the alarm sounds while the system is recording a message.

Figure 2.6 shows what should happen when the alarm sounds while the user wants to play a message. This is a conflict between two uses cases. It is not possible to carry out the Play a Message and the Wake Up Alarm simultaneously. We have decided to give priority to the alarm sound.

Figure 2.7 shows a scenario where the system enters stand-by mode, then it is woken up by the alarm clock. After another battery warning, it enters again stand-by mode. The system can switch on and off the screen back-light, the microphone and the speaker. These elements use a considerable amount of battery power and the system can save energy and increase the battery life by switching them off. The battery meter can also warn the system when it is almost out of energy.

## 2.5 Analysis: Object Structure

After the requirement analysis, Seidewitz and Stark [89] propose continuing with the domain analysis phase. In this phase, we analyze the requirements and

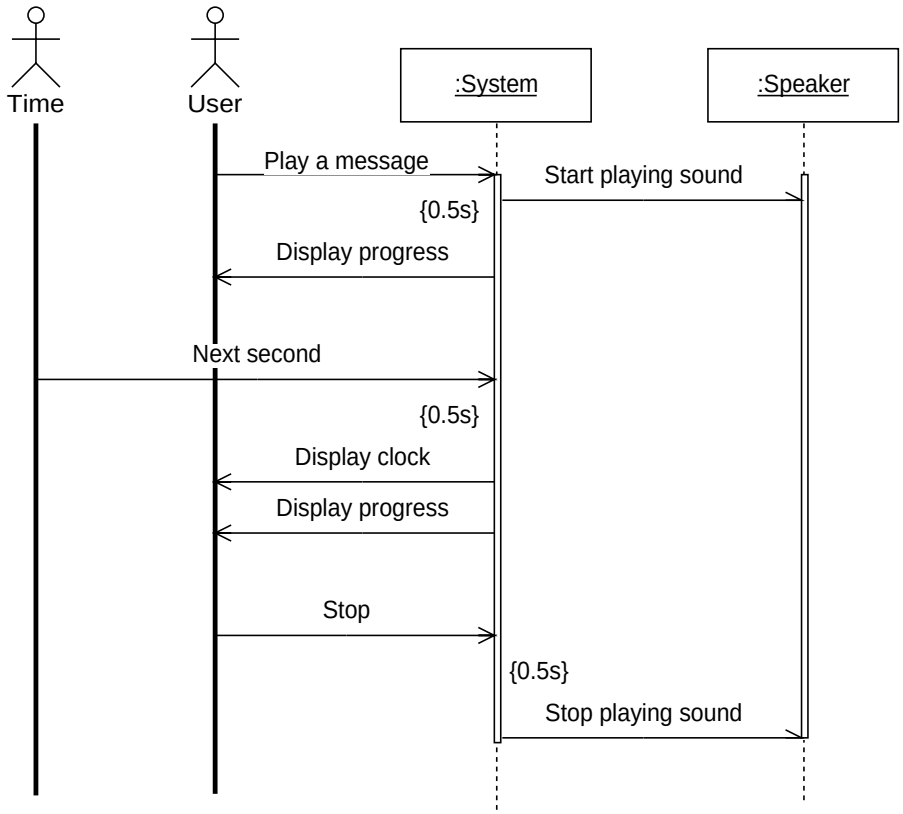



Figure 2.5: Playing a Message Scenario

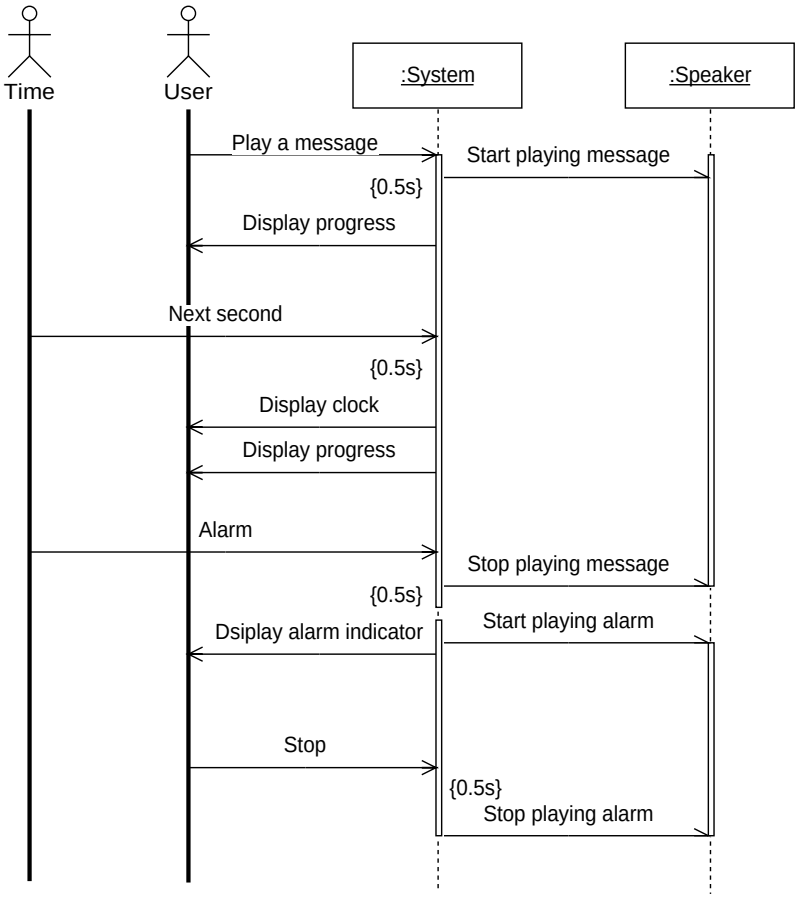



Figure 2.6: Alarm while Playing a Message Scenario

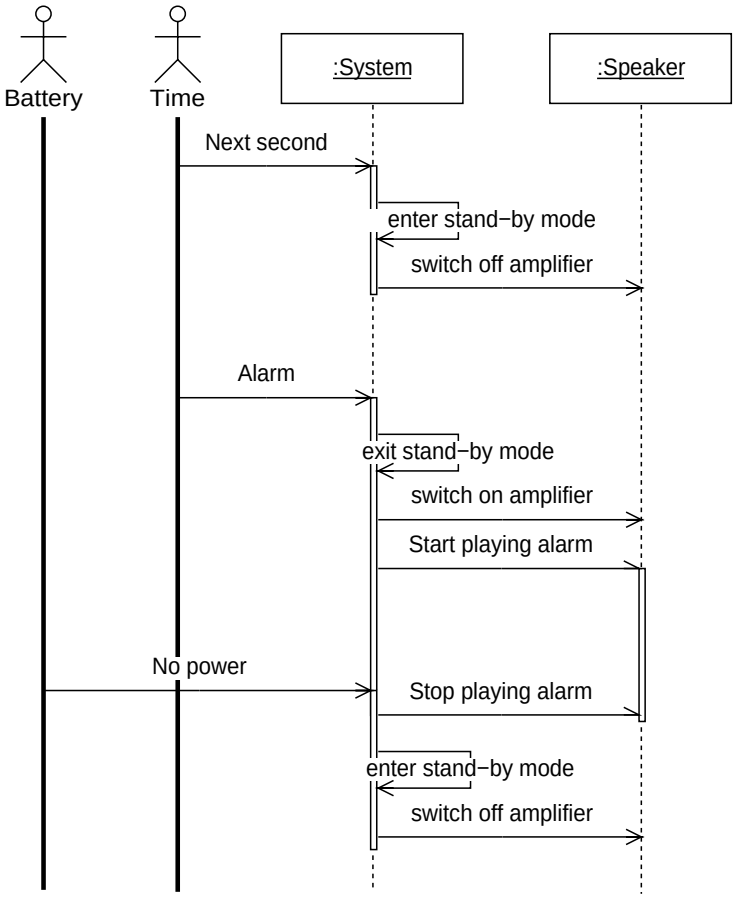



Figure 2.7: Entering Stand-by Mode Scenario

present a class diagram as a general description of our problem. The domain diagram shows the main classes of the system and their relationships, but it omits their interfaces. The first step in building the class diagram is identifying the objects involved in our system and their classification into classes.

### 2.5.1 Identifying Objects

In this task, we enumerate objects or "things" related to our project. We are only trying to discover the objects participating in our model and we do not yet need to describe the exact properties of these objects.

The objects identified in this section are analysis objects. Most of these objects will be implemented as objects in a programming language. However, some objects will be implemented as hardware and other objects are just intellectual artifacts needed during the analysis and design which will not be realized in the program code.

| | |
|---|---|
| Active Objects | Message player and recorder<br>Alarm clock |
| Services | Battery level sensor<br>Measures the remaining battery power level<br>Analog to digital converter<br>The interface between the microphone and the processor<br>Digital to analog converter<br>The interface between the processor and the speaker<br>Screen controller<br>It generates the image on the LCD screen. |
| Real-World Items | Date, time<br>Speech, sound |
| Physical Devices | Microphone, speaker<br>Battery<br>Keyboard buttons<br>Screen |
| Key Concepts | Speech, sampling, sound sample<br>We must study how a discrete digital system can capture and process an analog and continuous sound signal<br>Sound message |
| Persistent Objects | Collection of recorded messages<br>Current time, date, alarm time |
| Visual Elements | Menu, menu option, menu selection<br>Clock, date, alarm on/off indicator<br>Status bar<br>Task progress indicator |

### 2.5.2 Building the Class Diagrams

#### The Sound Recorder Class Diagram

The user interacts with the system through the display and the keyboard. These are passive objects, so we decided to add the User Interface object that manages the interaction with the user. The User Interface class relies on the Audio Controller to record and play sound. The Audio Controller is the core of the sound subsystem and it executes the tasks suggested by the User Interface. The Audio Controller uses the Audio Input and Audio Output classes to process the sounds. These classes are related to a microphone and speaker hardware wrapper. Hardware wrappers [3] are simple objects that abstract a piece of hardware

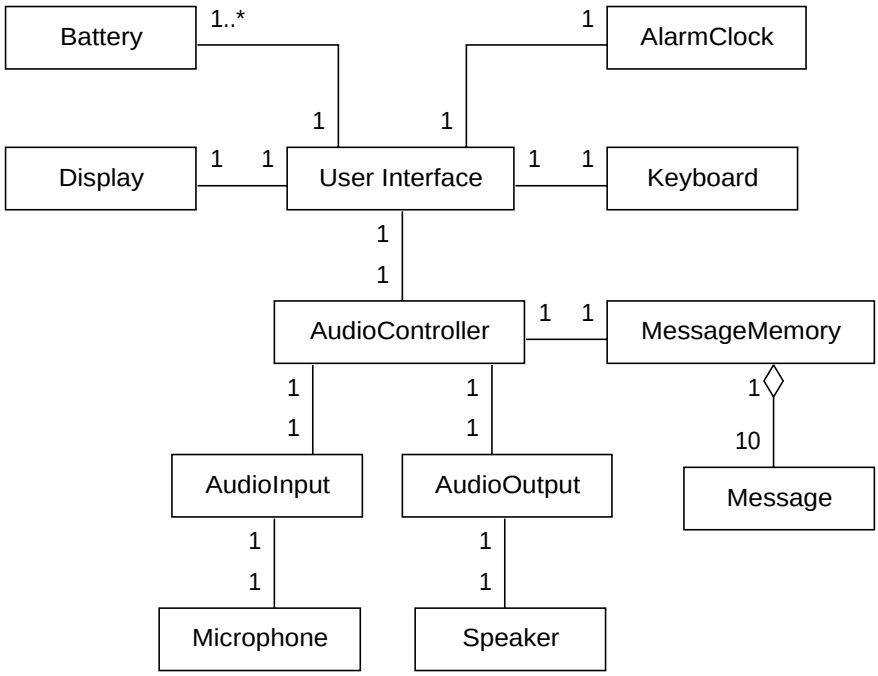



Figure 2.8: Sound Recorder Main Class Diagram

as a design or program object.

Sound messages are stored in the Message Memory. This class keeps a directory containing the recorded messages, allocates space for new messages, and deletes the old ones. The Alarm Clock updates the internal clock and checks for the alarm. In this case, it notifies the event to the User Interface. At this time, the User Interface is responsible for showing an indication on the display and playing an alarm sound with the help of the Audio Controller. The Battery periodically measures the battery power level. When the battery goes low then it reports the event to the User Interface.

Figure 2.8 shows the main class diagram of the digital sound recorder. This class diagram provides a general overview of the whole system by abstracting from the details of each class. We have divided the class diagram into five different subsystems: the alarm clock, the battery, the user interface, the memory and the audio subsystem. This division is represented in Figure 2.9. The following sections study and develop each subsystem in detail.

### The Audio Subsystem Class Diagram

The Audio Controller is the main element in the audio subsystem. It plays and records complete messages. Each message is composed of several Audio

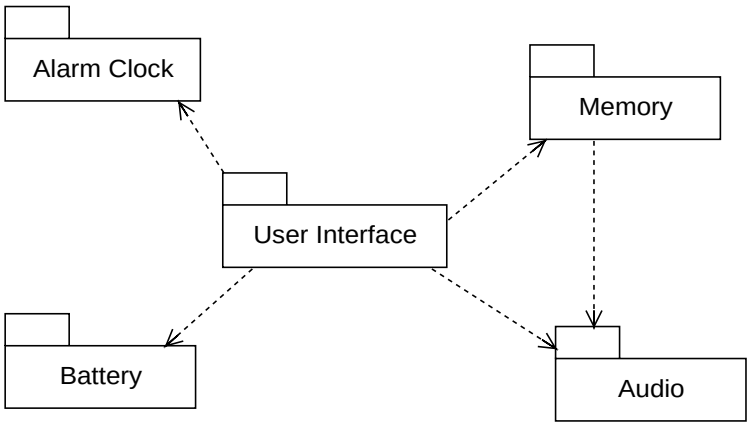



Figure 2.9: Subsystems in the Sound Recorder

| Granularity | Play | Record |
|---|---|---|
| Sound sample<br>1/6000th of second | Speaker<br>playSample() | Microphone<br>.recordSample() |
| Block<br>1/3th of second | AudioOutput<br>.playAudioBlock() | AudioInput<br>.recordAudioBlock() |
| Message<br>variable length | AudioController<br>.startPlayingMessage() | AudioController<br>.startRecordingMessage() |

Table 2.2: Operations to Play or Record a Message

Block objects and each block is composed of many sound samples. The Audio Controller uses the help Audio Input and Audio Output to process the messages. The Audio Input class knows how to record a single Audio Block. It uses the Microphone class, that is a hardware wrapper for the physical microphone. A Microphone can record one sound sample each time. We use the Audio Output and Speaker classes to play a block through the speaker in a similar fashion.

Figure 2.11 shows the breakdown of a message into audio blocks and sound samples. It also shows what class is responsible for playing each element. Table 2.2 shows the responsibilities of the operations of these classes.

It could be questioned why we should need three different classes just to play a message. The answer is that playing or recording a message is a complex task that requires precise timing and interaction with the hardware. We also think that by splitting the representation of the messages from their streaming, we increase the flexibility of the design. This design can be easily extended to stereo messages with two channels of audio blocks or pipelined compression, where the sound is compressed while it is recorded.

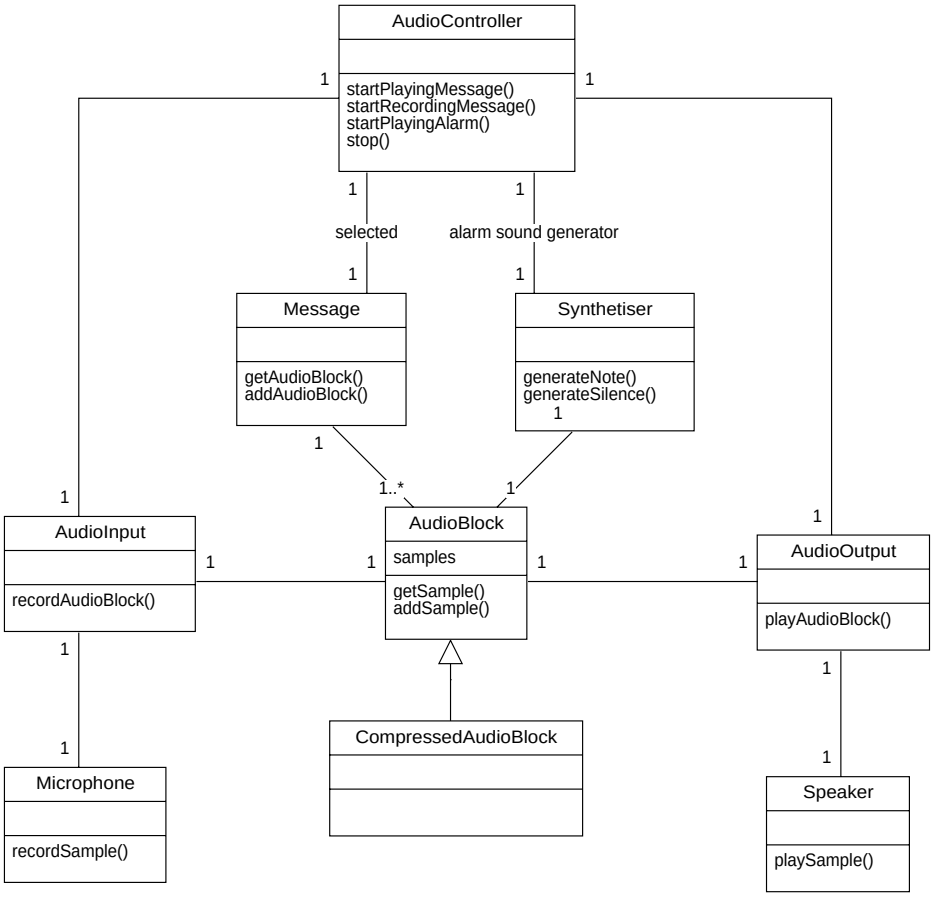



Figure 2.10: Audio Subsystem Class Diagram

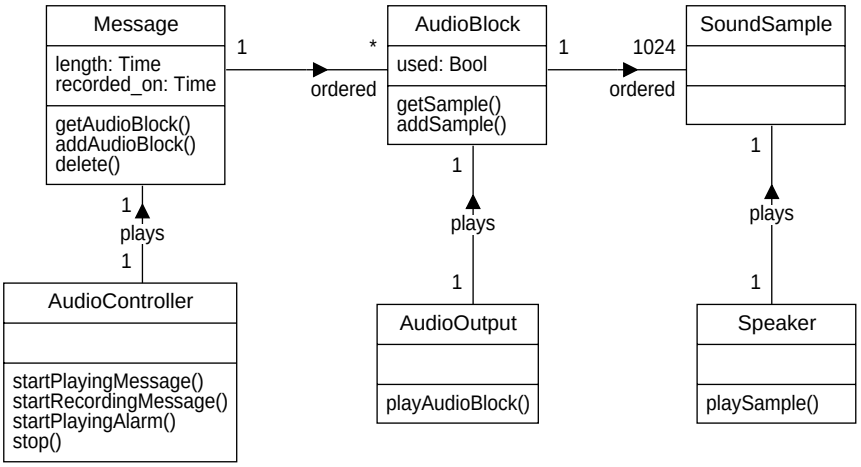



Figure 2.11: Representation of Sound Messages

Figure 2.12 shows the messages exchanged in order to play a message. It also shows three real-time constraints in the audio subsystem. The *play message response time* is the time period that elapses from when the user presses the play button until the sound starts playing. The *stop response time* is the period that elapses from when the user presses the stop button until the sound actually stops. Since the Audio Output always plays a whole block, the user can detect a delay if the blocks are too large. Finally, the *block gap* is the period that elapses between two consecutive audio blocks. This time constraint affects the sound quality. If the gap is too large, it will be audible during the reproduction of a message as silences and "clicking" sounds.

### The Message Memory Subsystem Class Diagram

The Message Memory class manages the storage space of the sound recorder, it keeps a directory of recorded messages and it allocates space for new messages. The class diagram for the memory subsystem is shown in Figure 2.13

### The Alarm Clock Class Diagram

The Alarm Clock keeps the current time and date and the alarm time. It uses a timer to measure the passage of gtime. It has the responsibility if updating the internal representation of the time each second. For this purpose, it uses the nextSecond operation of Time class. This operations adds a second to a time object with carry. For example, nextSecond of 23:59:59 is 00:00:00. When the now object wraps around, the date is updated using the nextDay method.

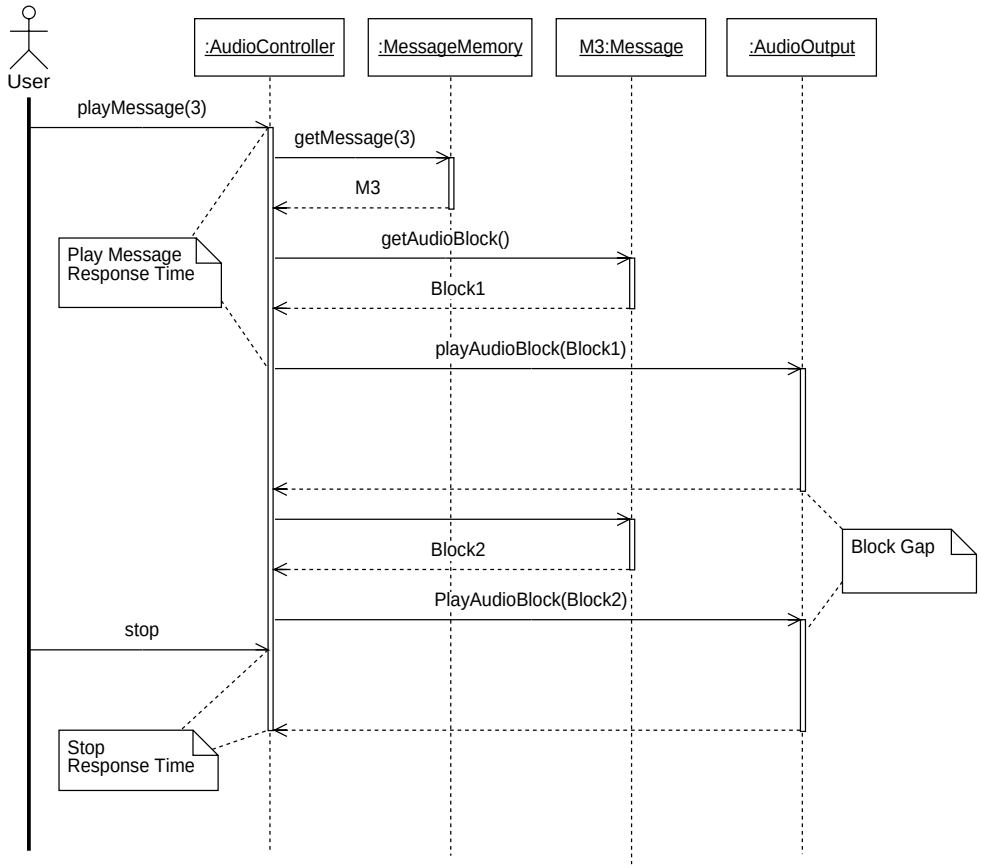



Figure 2.12: Play Message Sequence Diagram

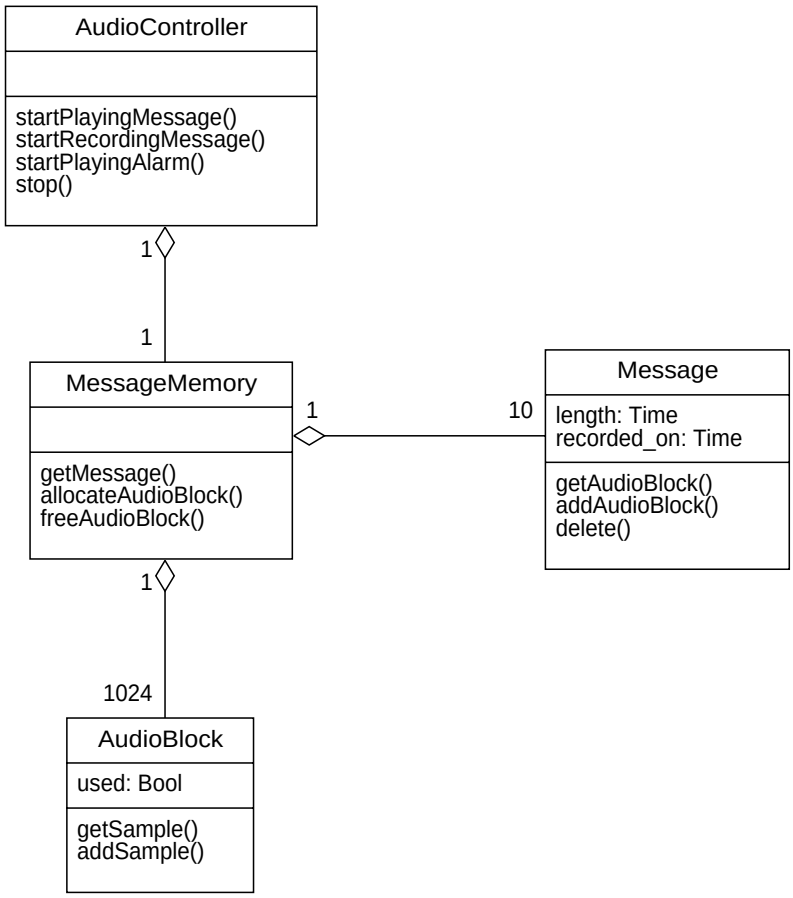



Figure 2.13: Message Memory Class Diagram

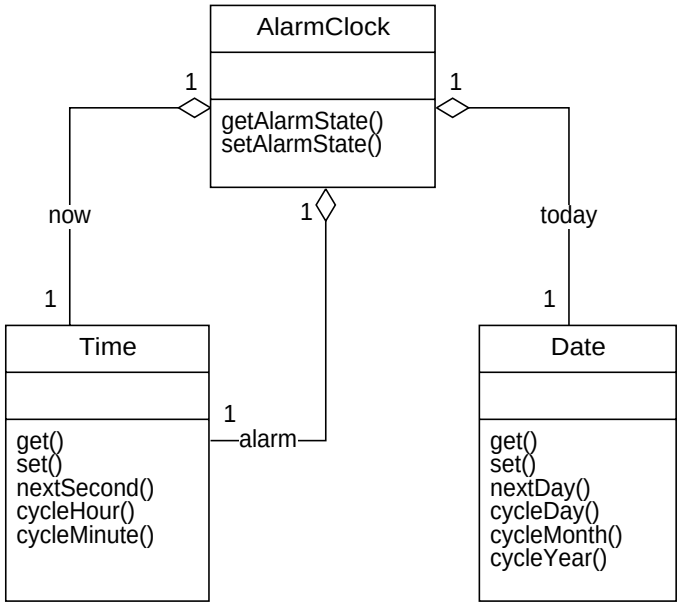



Figure 2.14: Alarm Clock Class Diagram

The cycleHour and cycleMinute methods are used to edit the time. These methods increase the current hour or minute without carry. For example, cycleMinute of 23:59:59 is 23:00:59. The class diagram for the alarm clock subsystem is shown in Figure 2.14.

### The User Interface Class Diagram

The User Interface class manages the interaction with the actors. It receives input events from the user via the Keyboard, from the time actor via the Alarm Clock and from the battery via the Battery hardware wrapper.

There are two kinds of events: modal and modeless. The system always reacts in the same way whenever a modeless event is present while the interpretation of a modal event depends on the sate of the system. As an example, keyboard events are modal. When the user presses the down button, the system can select the next option in a menu if the system is in the menu mode or it can decrease the current time if it is in the edit clock mode. Modeless events are managed directly by the User Interface. Modal events are forwarded from the generic User Interface event handler to a more concrete User Mode object.

The User Interface provides feedback to the user using the graphic display. The Display class is the interface with the hardware display. It can be switched on or off in order to save energy. The Graphic Context abstraction is used to draw on a Display. It provides some basic drawing primitives like drawing points,

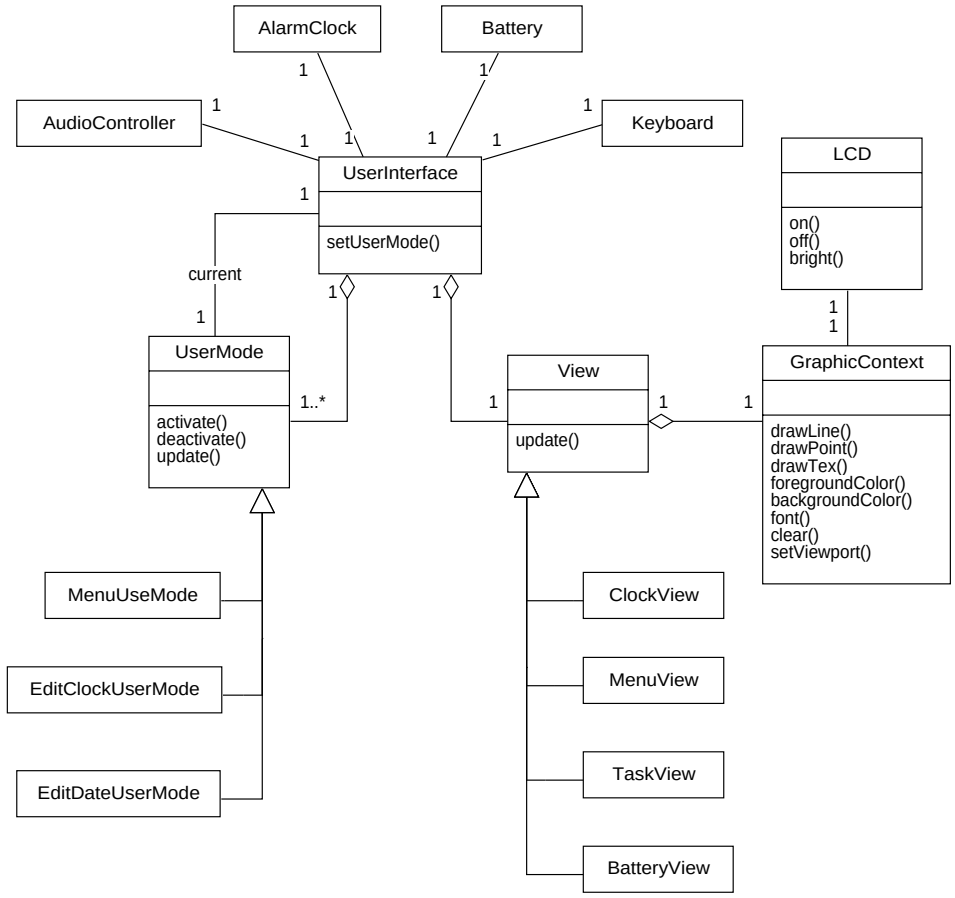



Figure 2.15: User Interface Subsystem Class Diagram

lines, text strings or filling rectangular regions with a flat color. Each Graphic Context represents a rectangular area of a Display. The Graphic Context manages the geometry transformation from its local coordinate system to the global one. The View classes use these graphic primitives to render the application objects like the current time or the user menu. Each View uses a different Graphic Context to draw on.

The Menu User Mode and Menu View classes allow the user to choose from different options on a screen menu. The Edit Date and Edit Clock user modes allow the user change the current date and time. They use Date View and Time View to render the date and time on the screen.

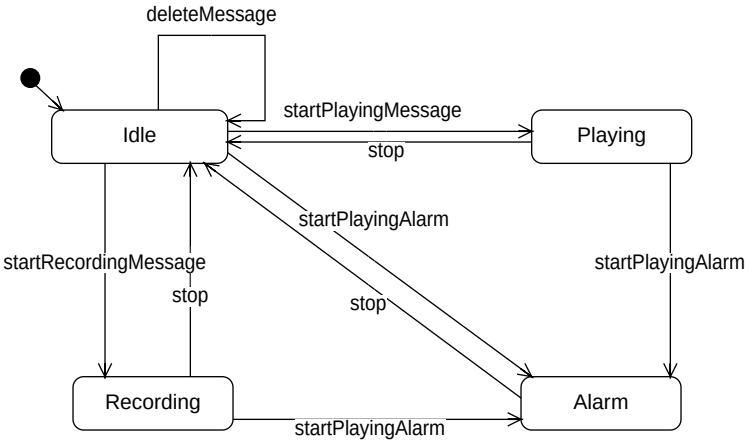



Figure 2.16: Behavior of an AudioController

## 2.6 Analysis: Defining Object Behavior

A class diagram presents a static view of our system. To understand the behavior of our system we can create new diagrams showing the dynamic aspects of our design. For this task, we can any of the UML behavioral diagrams: use activity, statechart, collaboration and message sequence diagrams.

### Audio Controller

An Audio Controller object is the referee for the sound channels. A sound channel can be used to record a message, to play a message or to play an alarm sound. Playing the alarm has priority over playing or recording sound messages. This fact is represented in the statechart shown in Figure 2.16. There we can see that the Audio Controller has four main states: playing a message, playing the alarm, recording a message and idle. The initial state is idle. If the Audio Controller is playing or recording message it can switch to the alarm state if the startPlayingAlarm operation is invoked. There we can also see that we can only delete messages when the Audio Controller is idle. This design decision has been taken to simplify the design of the audio subsystem. Otherwise, it would had been necessary to use a mutal exclussion mechanism to prevent a possible race condition when the user interface tries to delete a message that it is being played by the Audio Controller. We detected this situation while performing the concurrency design in Section 2.7.2.

### 2.6.1 The User Interface

#### Menu User Mode

The menu user mode is the main mode for the user interface. The buttons allow the user to navigate through the menus and invoke the desired menu option. The up and down buttons select the next and previous menu option. The yes button invokes an option while the no button is used to go to the previous menu. The behavior of this user mode is represented in the statechart shown in Figure 2.17.

#### Edit Clock User Mode

The edit clock user mode is used to change the current time or the alarm time. The yes buttons switches the active field between the minute and the hour field. The up and down buttons modify the active field. The behavior of this user mode is represented in the statechart from Figure 2.18.

#### Edit Date User Mode

The edit date user mode is used to change the current date. The yes button switch the active field between the day, month and year field. The up and down buttons modify the active field. Its behavior is similar to the EditCLockUser-Mode.

## 2.7 Architectural Design

In this section, we describe the hardware resources allocated for the design. In our final product, the design of the hardware is as important as the software. A hypothetical customer will not buy a program or a hardware device but a shrink-wrapped product containing both elements. However, we did not try to optimize the hardware of the sound recorder. For example, the processor we are currently using is more powerful than it is needed. It is probably more expensive, bigger and it needs more energy than a less powerful processor. Nevertheless, this extra processor power allows us to focus more on the design and ignore the implementation issues and optimization techniques of a particular architecture.

The main processor is a Hitachi SH7032 running at 16MHz with 8KB of on-chip RAM. This embedded processor is built around a processor core and several built-in peripherals. The processor core is a conventional RISC integer pipeline and its task is to execute the machine code instructions. The built-in peripherals consist of an analog to digital converter, several timer and counter units, DMA channels and a watchdog. The manual [70] describes the instruction set of the processor core. The hardware and built-in peripherals are described

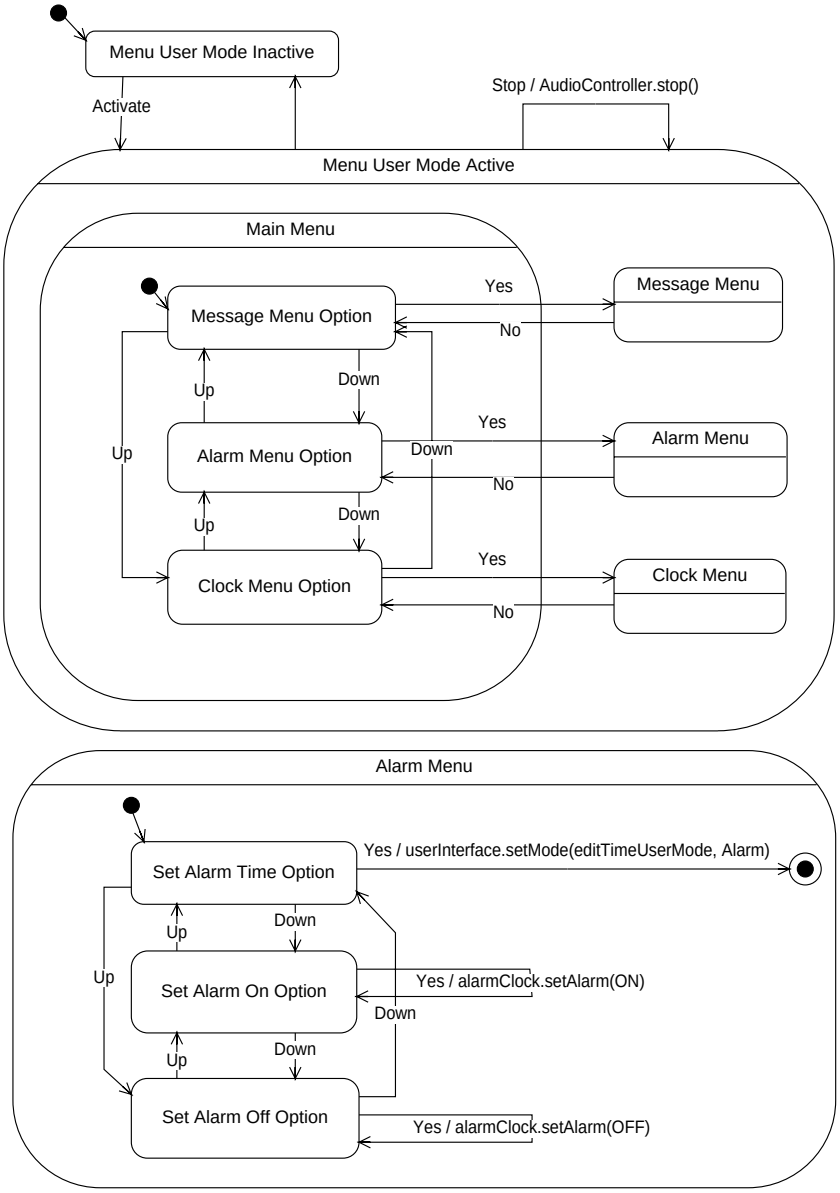



Figure 2.17: MenuUserMode Statechart

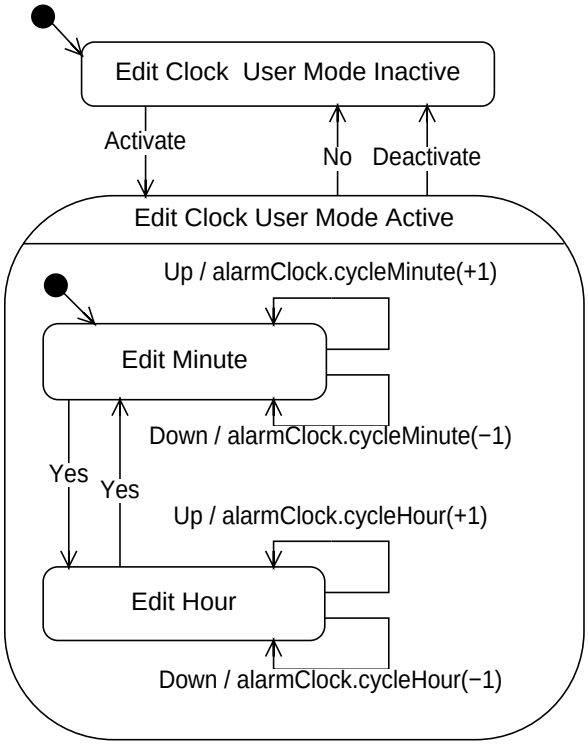



Figure 2.18: EditClockUserMode Statechart

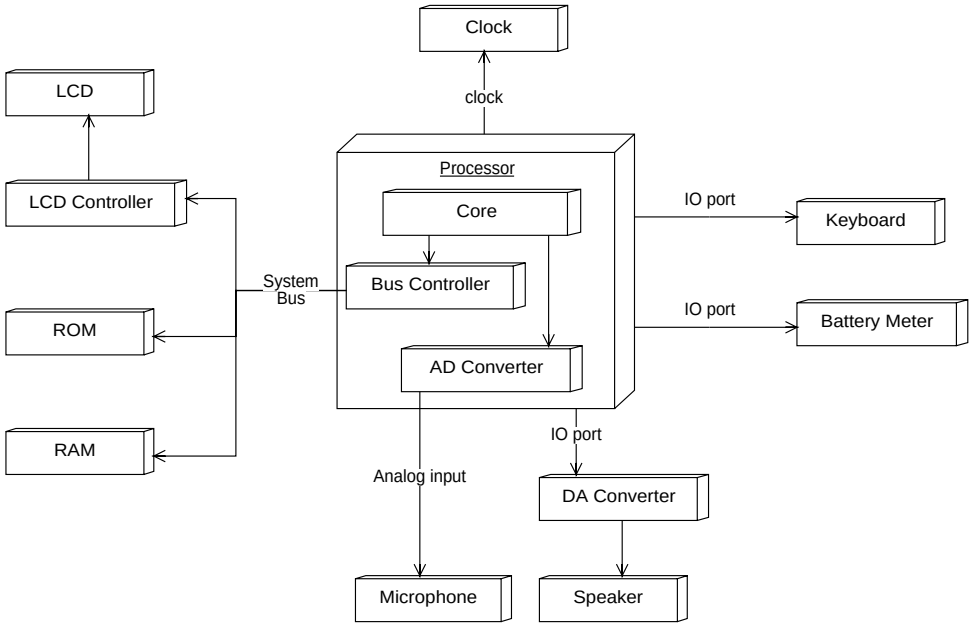



Figure 2.19: Hardware Architecture of the Digital Sound Recorder

in [69]. Figure 2.19 shows a deployment diagram with the proposed architecture. The prototype version also contains an EPROM memory with a debugger and two serial ports for downloading and debugging the software.

### 2.7.1 Architectural Patterns

The software of the sound recorder does not have special safety or reliability requirements. The architecture will not support hardware fault tolerance. Probably, the user will damage the unit, by dropping it on the floor or by spilling some liquid into it, before the hardware wears out.

A watchdog will restart the system in case of a sporadic fault. When the watchdog resets the processor, the message memory will be lost. Since it is a wrapped and embedded solution, the sound recorder does not have any communication link with other systems. It is not necessary to provide a communication link for testing and diagnosing faults because the system is quite simple and it can be tested following indications on its own display. It is not designed to be repaired. All peripherals are accessed through the memory address space of the processor. Since they are tightly coupled, they do not require the use of special communication patterns.

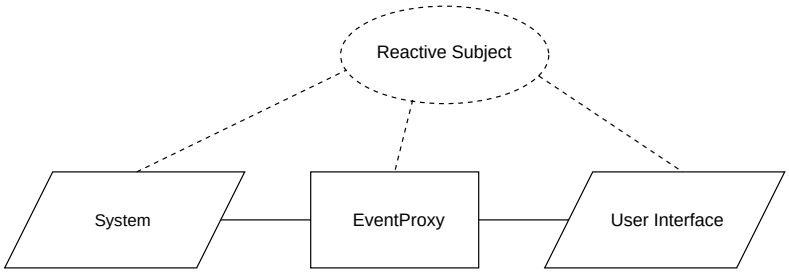



Figure 2.20: Task Diagram

### 2.7.2 Concurrency Design

The concurrency model specifies the different execution threads and the communication mechanism between threads. In this design, there are two execution threads: the user and the system thread. Figure 2.20 shows an interaction diagram with the tasks of the system.

The user thread is an interactive thread and manages the interaction with the user trough the display and the buttons. This thread does not have real-time requirements. The system thread manages the real-time objects of the system. It is activated whenever the processor acknowledges an interrupt request. The system thread has priority and it can preempt the user thread. There is a Scheduler object running in the context of the system thread that schedules the execution of the reactive objects, like the AudioController or the Battery. These objects share the same thread, so they must return the control to the scheduler as soon as possible. Two objects running in different threads communicate by using the Reactive Subject pattern, described below.

## 2.8 Mechanistic Design

In this section, we discuss how the different software objects collaborate to achieve their objectives. Section 2.6 presented the internal behavior of each individual object. There we described how the state of each object changes when it receives a message. Now, we focus on how and when the objects exchange messages and we describe it using software patterns.

A software pattern is a description of a design problem and its general solution in a given context. Software patterns allow us to reuse some of the knowgledge needed to put a large software system together. Patterns are important because they capture and expose important design decissions that otherwise will be hiden between the object structure of a system. In this section we use some of the patterns presented by Gamma et al. in [37] but we also introduce

a new pattern to deal with some forces only presented in embedded systems.

### 2.8.1 Collaboration Between the Hardware and the Reactive Objects.

A hardware device requests an interrupt whenever it wants to communicate an event to a running program. At some point, the processor will acknowledge the interrupt, stop the current program flow and call an interrupt service routine (ISR). This routine should process the hardware request and return as soon as possible to allow the normal program execution to continue.

Unfortunately, an ISR cannot be an object's method. An ISR does not have any context information associated and thus can not carry a this or self pointer as is required in most object-oriented languages. Therefore, we need to build a mechanism that transforms hardware interrupt requests into messages to objects. We have wrapped this mechanism in an abstract class called ISR. A subclass of ISR can implement interrupt service routines as a normal method.

### 2.8.2 Collaboration Between the Reactive Objects and the User Interface

The keyboard, the battery level meter, the alarm clock and the audio controller collaborate with the user interface by using the Reactive Subject pattern. The reactive objects send events to the event proxy but do not wait for the user interface to acknowledge them. The user interface constantly checks for new events in the event proxy. When it finds a new event, it delegates the responsibility to the view and controller objects.

### 2.8.3 Pattern: Reactive Subject

**Problem** A reactive real-time object wants to communicate an event to an interactive non real-time object.

**Context** We are combining both interactive and reactive objects in the same program and an interactive object needs the most updated state information from a reactive object. Reactive objects usually implement the core of an embedded system. Interactive objects can implement the user interface, system monitoring or data logging tasks.

**Forces**

1. Embedded real-time systems must complete some tasks within strict deadlines. Interactive systems do not have these restrictions. They are usually easier to design and implement.

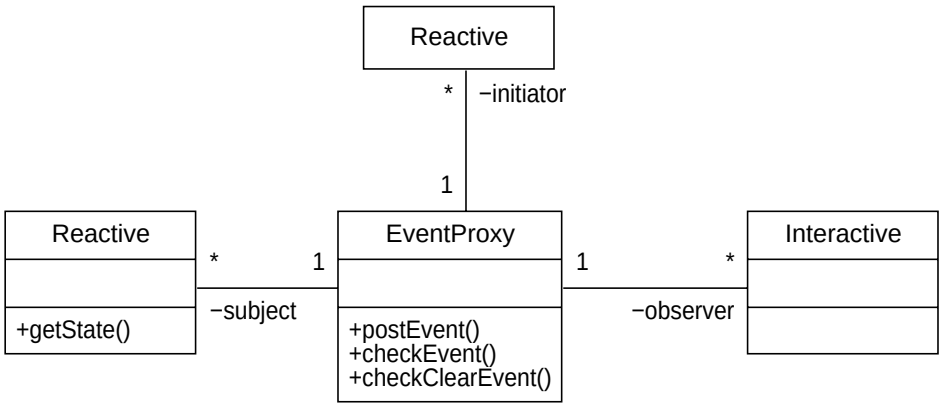



Figure 2.21: Event Proxy Class Diagram

2. The reactive part cannot make any assumptions about the response time of the interactive part.

3. In some applications the interactive part needs to process the full sequence of events sent by the reactive part, but in other cases the interactive part just needs the last produced event that carries the most updated information.

**Solution** The reactive object posts the events to an Event Proxy object. The Event Proxy stores the presence of the event plus a reference to an object. This operation must be done without blocking the reactive object. The interactive object periodically polls the Event Proxy to check the presence of a given event by using the checkEvent method of the Event Proxy. If the event is present, the Event Proxy returns the stored reference to the interactive object. If the event is not present then the Event Proxy returns a null reference. The interactive object uses the returned reference to query the state of the object by using the getState method. In this step, the interactive object can be blocked when trying to get the state information. The Event Proxy provides a checkClearEvent method that checks for an event and erases it from the event table if present.

**Rationale** By using this pattern extensively in a system, we are decoupling the reactive, control oriented part of our software from the interactive objects. These allow several benefits to our design: First, the objects do not know the address or even the existence of their partners. It can be possible for a reactive object (the initiator) to post an event containing a reference to another reactive object (the subject) with the state information. Second, the objects do not share an execution thread neither do they need to block or interfere with each

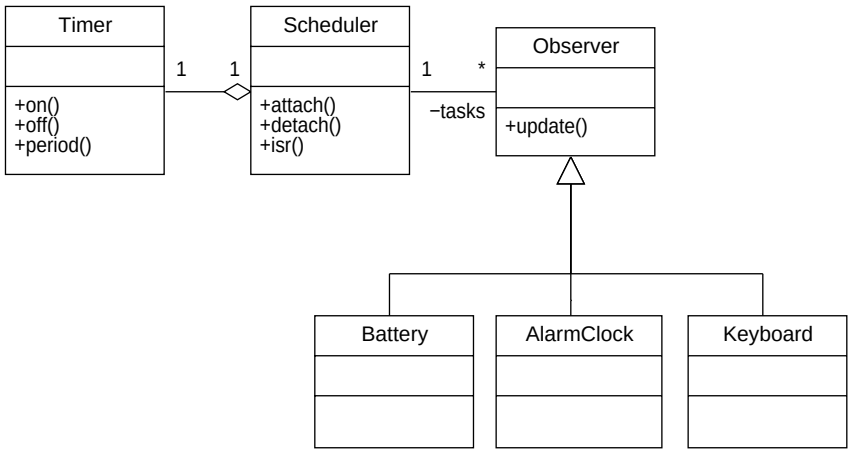



Figure 2.22: Scheduler Class Diagram

other's execution. It is possible to implement the Event Proxy in a way that does not need it to use any mutual exclusion mechanism while preventing race conditions. This requires the system to be able to write or read a reference to an object atomically. The simplest way to implement this is by storing the event table in an array of references to objects indexed by the event type.

**Example** The collaboration of the Alarm Clock and the Time View described above is an example of the Reactive Subject pattern.

### 2.8.4 Scheduler

A Scheduler provides accurate timing for the time-dependent objects, such as the Alarm Clock, the Keyboard and the Battery Meter. The Alarm Clock subscribes to the Scheduler during its initialization. Every second, the Scheduler notifies to the Alarm Clock that a second has elapsed. The Keyboard uses the Scheduler to poll the physical keyboard ten times per second. The Battery Level Meter measures the voltage supplied by the battery every five seconds. Figure 2.23 shows how the Scheduler periodically wakes up the reactive objects of the system.

### 2.8.5 Changing the Time

The Edit Time User Mode, the Alarm Clock, the Keyboard and the Clock View objects follow the Model-View-Controller pattern [37]. The Alarm Clock provides a model for the Clock View object, which renders the time into the display. The User Mode objects control the interaction with the user. Since the Alarm Clock

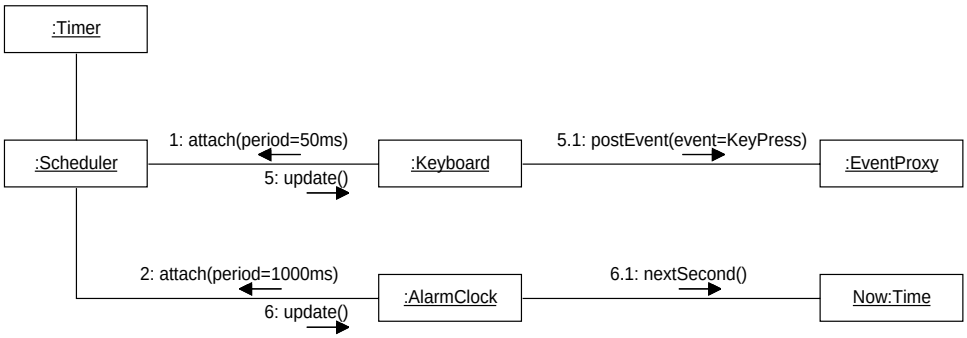



Figure 2.23: Collaboration between the Scheduler and its Clients

is a reactive object and the Clock View an interactive one, they also collaborate using the Reactive Subject pattern. The Keyboard object also uses the Reactive Subject pattern to report the key presses. Figure 2.24 shows the sequence of messages produced when the user presses the up button while the current user mode is the EditTimeUserMode.

### 2.8.6 Playing a Message

The User Interface and the Audio controller use the Reactive Subject Pattern. The Audio Controller, Message, Audio Input and Audio Output collaborate using the observer Pattern. Figure 2.25 shows the sequence of messages sent to play a message. In order to simplify the diagram, the message is composed of just one audio block. This collaboration is rather complex, but it supports playing and recording two different messages simultaneously or stereo sound, where each message is composed of two streams of audio blocks.

## 2.9 Detailed Design

### 2.9.1 Objects Modeling the Hardware

A hardware wrapper is a software object representing a hardware device. It is an interface between the application objects and the physical devices. The constructor method of a hardware wrapper initializes the hardware device. The object methods of the wrapper configure and operate the device by reading and writing directly into the registers of the device. Usually a wrapper has attributes, because the state of the wrapper is a function of the state of the hardware device. The detailed design and implementation of a hardware wrapper requires a precise knowledge of the underlaying hardware device. In this design, the speaker, microphone, timer, battery meter and keyboard objects

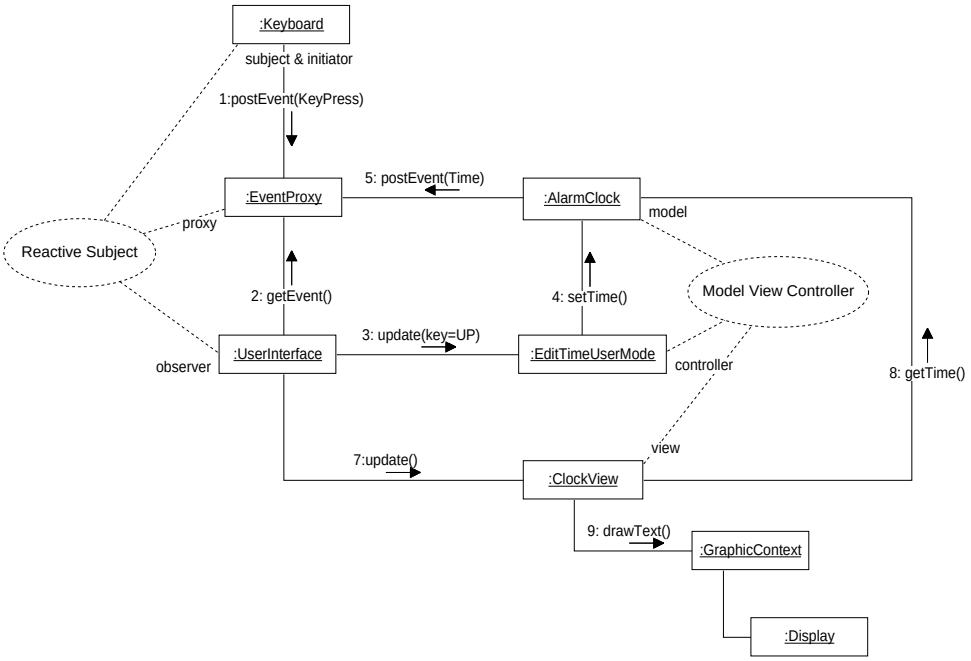



Figure 2.24: Model-View-Controller Collaboration

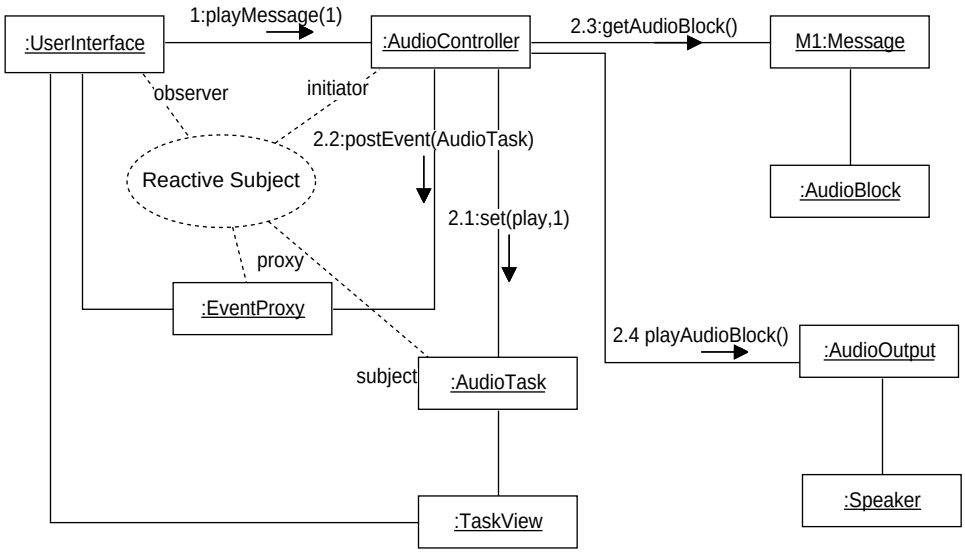



Figure 2.25: Collaboration between the User Interface and the Audio Controller

are examples of hardware wrappers.

### 2.9.2 Sound Compression Algorithm

One of the requirements specifies that the sound must be recorded at 6KHz using eight bit samples. This means that we need almost 6KB of memory for storing each second of sound. In order to reduce the amount of memory used we can apply a compression algorithm to the input signal. We have chosen the Adaptive Delta Pulse Code Modulation (ADPCM) [51] compression algorithm since its implementation is rather simple. ADPCM has also a low CPU overhead.

### 2.9.3 Using Direct Memory Access Channels

To record sound, an Audio Input object needs to move 6000 samples from the microphone to an audio block each second. There are several ways to perform this task. One option is to use the scheduler to time the task. A timer activates the scheduler 6000 times per second and then the scheduler activates the audio input object that transfers the sample. Clearly, many CPU cycles are wasted just for transferring a sample. A second option is to use a new timer to activate directly the audio input object. In this case, the CPU usage is lower but still considerable, since interrupt service routines have a great impact upon the performance of most RISC processors. The third alternative is to use a direct memory access channel (DMAC). A DMAC is a hardware component that can transfer several words between memory positions or devices without CPU intervention. In our design, we use a DMAC to transfer the samples from the microphone into an audio block and from an audio block to the speaker. A DMAC unit has several registers that control its behavior. The most important registers are the source address, target address, transfer count and operation flags. The flags indicate, among other things, if the source or the target addresses should be incremented or not after each transfer or if the transfer unit is a word or a byte. It is also possibile to perform the transfers as fast as possible or to synchronize it with the activity of a timer.

To record the sound, we program the DMAC with the address of the A/D converter as the source address. The target address points to an internal buffer and we set a flag to auto-increment the target address after each transfer. The transfer unit should be a byte, the size of a sample. We use a timer, programmed to generate an interrupt 6000 times per second, to initiate each transfer. The DMAC unit will start a byte transfer each time that the timer requests an interrupt.

After the samples have been recorded into the buffer, the DMAC unit generates an interrupt request. The Audio Input attends it and then compresses

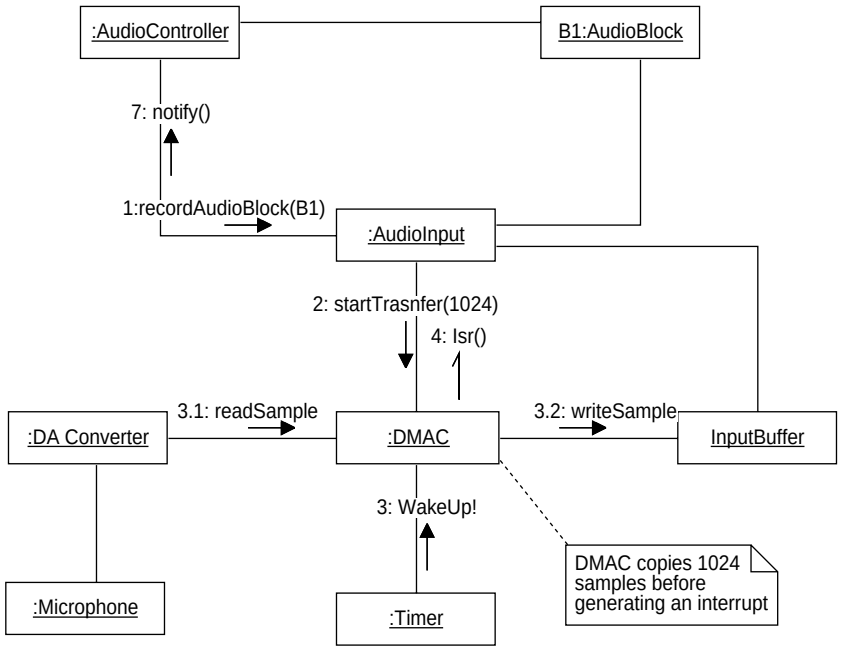



Figure 2.26: Recording an Audio Block

and stores the input buffer into a block. Figure 2.26 shows the sequence of messages needed to record a block. This sequence is repeated for each block of a message.

This mechanism is only acceptable if the processor can compress and expand the audio blocks very quickly. If not, the user will be able to hear a glitch between the reproduction of two audio blocks. This problem can be solved not only by optimizing the transfer of a single audio block but also by optimizing the streaming of blocks. Since the CPU and the DMAC can work in parallel, we can compress an audio block while recording the next one. Figure 2.27 shows this mechanism. In this diagram, the AudioInput starts recording the second block before compressing the first one.

Another design issue is the size of the audio blocks. If the blocks are too small, the CPU will expend a significant amount of time dispatching methods between the objects and synchronizing its activity with the DMAC. All the messages are composed by an integer number of audio blocks. The length in seconds and the size in bytes of a message are a multiple of the length and size of the audio blocks. If the blocks are too big, then memory is wasted and the system loses responsiveness. There is also another constraint in the size of the blocks imposed by the memory manager and explained below.

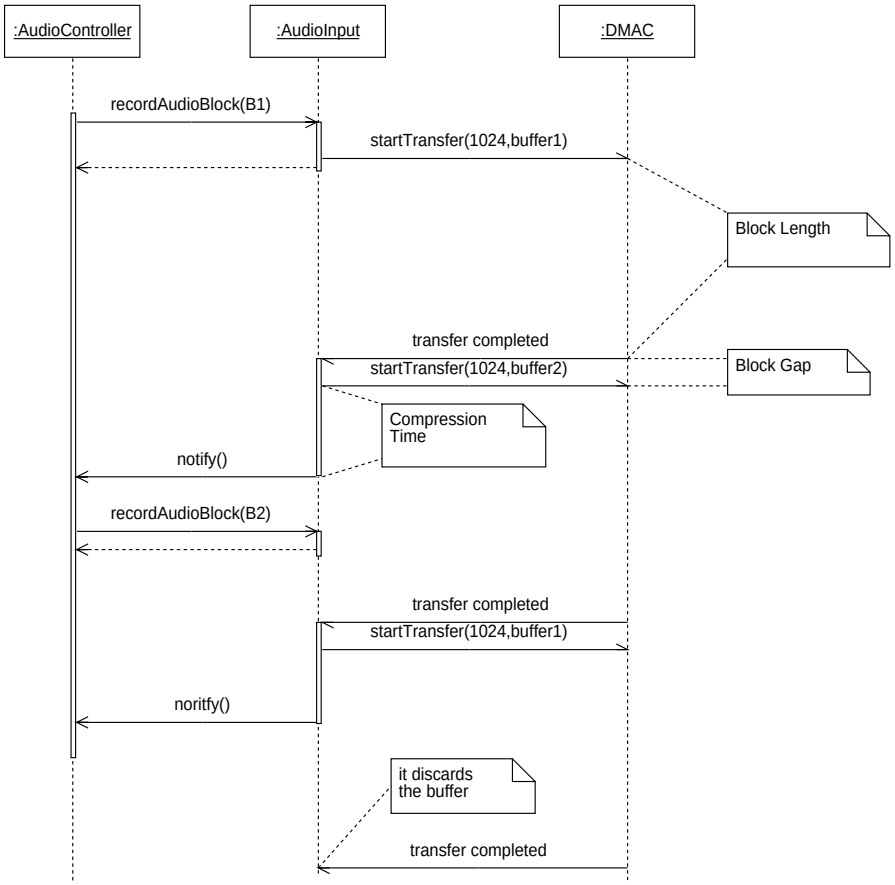



Figure 2.27: Recording Compressed Audio

### 2.9.4 Memory Allocation

The Message Memory is responsible for allocating and freeing the memory space used to store the messages. Since the available memory is quite limited and the system lacks virtual memory, we have to design a mechanism that ensures an optimal use of the system memory. This includes avoiding wasting space by memory fragmentation and object aligning. When a Message Memory object is created, it pre-allocates an array of Audio Block objects. We avoid memory fragmentation by pre-allotting and reusing the blocks instead of creating and destroying them every time a message is recorded or deleted. Usually, the memory allocation function is only able to allocate memory blocks aligned to a certain value. The size of an Audio Block object is a multiple of the alignment factor of the memory allocation function of our system which ensures that no single byte is wasted.

## 2.10 Comments on the Implementation

We can consider that the final deliverable for an embedded system is not the application source code but a non-volatile memory containing the program. The linker must statically link all the libraries into the program and allocate all the program symbols into absolute memory addresses. The program must include some code to initialize and check the hardware and rearrange the executable program into the RAM. That includes initializing the processor stack, copying the initialized variables from the read-only memory to the RAM and setting the uninitialized variables to zero. A prototype version of the target usually has a ROM memory with a small debugger. The debugger is activated when the system is switched on and it waits for the host computer to upload the main program using a communication link, such as a serial cable or a network connection. When the program is downloaded, it can be executed and debugged. When the software has been completely developed and tested, it can be recorded into a non-volatile memory. This memory can be inserted into the target and replace the debugger. When the stand-alone target is switched on, it then executes the application directly.

We have chosen the C++ programming language for implementing our system. The choosing of a programming language for an embedded system can be affected by the availability of tools for the target architecture.

The current implementation of the software comprises 3500 lines of C++ code and 2000 lines of legacy C code for the LCD display. The software is built in the host computer by a cross-compiler and it can be downloaded into the prototype or recorded into a ROM or EPROM. The code uses 40KB of ROM and 10KB of RAM plus the storage space for the sound. We have also implemented a prototype version of the hardware of the digital sound recorder.

The board includes several analog components, like the amplifier for the speaker or the microphone low-pass filter. The description of the hardware components is beyond the scope of this chapter and the UML notation. However, they play a big role in the quality of the sound and therefore the user's appreciation of the system.

## 2.11 Conclusions

In this chapter we have presented the origins and objectives of the Unified Modeling Language. We have shown how to apply the language through a complete case study. In this the case study, we have identified, analyzed and designed the main elements of a small embedded system. We have used UML as a graphical notation to document the artifacts of our design and we have shown how different types of UML diagrams can be used to describe different aspects of our system.

We can now summarize all the different kind of diagrams and their main purpose in UML. Probably, the most popular UML diagram is the class diagram. A class diagram shows the structure of a system as a collection of classes and their relationships. By contrast, an object diagram shows an instance of a class diagram. We can build and deploy binary components using component and deployment diagrams.

Behavioral diagrams show the dynamics of a system. Use case diagrams show the functionality of a system as a black box, without revealing its internal structure. They were introduced in UML by Jacobson. The semantics and analysis of use case diagrams is the main topic of the following chapter.

Interaction diagrams show how different objects exchange messages in order to perform a task. There are two kinds of interaction diagrams: sequence and collaboration diagrams.

Sequence diagrams focus on the ordering of the exchanged messages. There is an implicit time line that flows from the top to the bottom of the diagram. Sequence diagrams are similar to the Message Sequence Charts [53] used in other graphical languages like SDL [52]. Collaboration diagrams are interaction diagrams that focus on the relationships between objects. Messages can include sequence numbers, so it is still possible to order the sequence in time, although this requires some work from the reader. UML adopted the collaboration diagrams from the Booch and OOSE methods.

Statecharts model the behavior of a single model element as a reaction to external events. Harel introduced Statecharts in 1984 [42] and they were quickly adopted as a visual formalism to design concurrent and reactive systems. We study UML statecharts thoroughly in chapters 4, 5 and 6.

Finally, activity diagrams model a workflow, a series of steps to follow in

| Diagram | Kind | Problem Domain | Use Case Model | Design Model | Detailed Design |
|---|---|---|---|---|---|
| Use Case | Behavioral | | √ | | |
| Class | Structural | √ | √ | √ | √ |
| Object | Structural | | | √ | √ |
| Sequence | Behavioral | | √ | √ | √ |
| Collaboration | Behavioral | | | √ | √ |
| Statechart | Behavioral | | | √ | √ |
| Activity | Behavioral | | √ | √ | |
| Component | Structural | | | | √ |
| Deployment | Structural | | | | √ |

Table 2.3: Classification of UML Diagrams

order to perform a task or an operation. They are based on data flow diagrams (DFD) [71, 38].

Table 2.3 sumarizes the different types of UML diagrams. The first column of the table indicates whether a diagram shows a structural (static) view of the model or a behavioral (dynamic) view of the model. The following columns indicate in which development artifact the diagram is most likely to be created. Once we create a diagram, it is likely to be useful and used during the remainder of the project.

This table is not an absolute classification. Nothing prevents using any type of diagram in any stage of a software project. Even the distinction between behavioral and structural diagrams is not categorical: Class diagrams, although they are mainly structural diagrams, can also describe the behavior of the methods if they are properly annotated.

Finally, we should remark that although UML includes many types of diagrams, not all of them are equally rich and expressive. We think that component and deployment diagrams are poor and of little use and that activity diagrams are superseded by statecharts. Reducing the number of diagrams could simplify training and tools. However, UML can be used in many application domains and there are situations where these diagrams are of great importance for the designer.

UML is built based on the best practices in software engineering, but it is not perfect. The language has many different model elements that are used seldom and can confuse the beginner. Also, a good UML tool is needed to keep source code, UML models and documentation (such as this text) always synchronized since manual synchronization is cumbersome and unproductive.

We like to split our model into several diagrams to make the size more manageable while maintaining consistency. However, the UML semantics does

not provide a mechanism to ensure that all diagrams are consistent and actually describe the same system. Another drawback of UML is that there is no way to enforce a design. There is no a systematic approach for translating all diagrams of a UML design into an executable program.

We would hope that future designs would contain more UML diagrams and fewer textual descriptions. To achieve this, we need more ready-to-use software patterns for UML, including patterns for real-time embedded systems. We would like to use textual descriptions just to explain how we tailor certain software patterns to build our design.

UML support for real-time systems is quite poor. At the time of writing, there is an ongoing effort to create a standard profile for modeling real-time systems in UML [94]. We study this problem in Chapter 6, where we propose a model of time for UML statecharts. It can be used to model real-time and hybrid behaviors, like the real-time constraints present in the audio subsystem of the digital sound recorder.

Still we consider that the main drawback of UML is the lack of a systematic method to validate the models. Any good UML tool should allow us to create only models that are syntactically correct, i.e. well-formed models that only use the elements of the UML language. But in the case of the UML language this is actually of little importance. The fact that a model is syntactically correct does not mean that it can be actually implemented, contains all user requirements or that it actually tries to solve the user's problem.

In the rest of the thesis we try to overcome some of these problems by proposing rigorous anlysis methods that can be used to study and assure the quality of a UML model.

In the case study, we describe the functionality of our system using use case diagrams. We present several use cases and describe what we can achive with them. In the following chapter, we show how to formalize and analyze use cases and mathematically prove that the actors of a system can achieve their goals. This is important since, in a use-case driven software process, the rest of the UML models and the source code are created in order to satisfy what is stated in the use case models. If the use case model is inconsistent or incomplete, the final software will not satisfy the needs of the users.

The design of the sound recorder also uses statechart and collaboration diagrams to describe the behavior of the objects. In these diagrams, we describe the expected behavior of the system. However, in Chapter 5 we show that a statechart can easily describe undesired behaviors in a system, for example deadlocks. In that chapter, we will show how to detected these kinds of errors automatically using a verification tool.

# Chapter 3

# Analyzing Use Case Models as Contracts

> A towel, it says, is about the most massively useful thing an interstellar hitch hiker can have. Partly it has great practical value - you can wrap it around you for warmth as you bound across the cold moons of Jaglan Beta; you can lie on it on the brilliant marble-sanded beaches of Santraginus V, inhaling the heady sea vapours; you can sleep under it beneath the stars which shine so redly on the desert world of Kakrafoon; use it to sail a mini raft down the slow heavy river Moth; wet it for use in hand-to-hand combat; wrap it round your head to ward off noxious fumes or avoid the gaze of the Ravenous Bugblatter Beast of Traal (a mindbogglingly stupid animal, it assumes that if you can't see it, it can't see you - daft as a brush, but very very ravenous); you can wave your towel in emergencies as a distress signal, and of course dry yourself off with it if it still seems to be clean enough.
>
> —Douglas Adams, The Hitch Hiker's Guide to the Galaxy

## 3.1 Introduction

A UML use case diagram shows the interaction of a system with external entities called *actors*. They describe the functionality of the system as a black box, i.e. without revealing its internal structure. Use cases describe *what* the system supposed is to do, without revealing *how* the system does it.

UML is often applied on a use case-driven software engineering process [19]. In this kind of process, use cases are a primary artifact for capturing functional requirements and establishing the desired external behavior of the system, for the verification and validation of the system design and implementation and as

a communication device among the different stakeholders of the project. The Rational Unified Process [62] and Catalysis [31] are two examples of use case driven software engineering processes based on UML.

Use cases were introduced by Jacobson in [54] and further developed in [21, 55, 56, 19]. At the time of their introduction they were considered too informal and simple. Yet, it is this simplicity that makes them popular and accessible.

Use case diagrams are used to model both the context and the requirements of the system under development. The context of the system defines the environment that the system lives in and includes the actors of the system. The actors are entities with free will and have certain goals to fulfill. They impose some functional requirements upon the system. Use case diagrams can be considered as informal contracts between the system and the actors. If the actors behave as described in the use case, the system promises to deliver the functionality also described in the use case. A well-behaved system should carry out all its required functionality predictably and reliably.

The notion of contract has been used in the literature before. The paradigm 'Design by Contract' [72], where a contract is attached to each class and consists of a class invariant together with pre- and postconditions for each method of the class, is a well known example of using contracts in software engineering. This paradigm comes from work on program correctness [33, 46, 29], and is based on using and checking assertions within programs. The contract as a specification binding several parties was first introduced in [82], to constrain the collaborations among different objects. This idea was further developed, for instance in [36]. Contracts are also used in the context of software components [95]. Since the components and their clients are developed in mutual ignorance, the contract is the common ground for successful interaction. Many researches have studied the contracts of components, e.g. in [103, 50, 14, 40].

We will take here the view of contracts proposed by Back and von Wright. According to [12], a computation involves a number of actors who carry out actions according to a formalized contract that has been laid out in advance. A contract is defined by a contract statement. A contract statement describes in what ways the actors can modify the state of the system. Both the properties of contract statements and the state itself are formalized within the *refinement calculus* [4, 12]. It provides a rigorous mathematical foundation for contracts as well as the tools to analyze and manipulate the contracts. In particular, a contract can be analyzed to check whether an actor can achieve a specific goal by using the contract. The state of the contract models the problem domain of the system while the contract itself expresses the functional requirements for the same system.

The refinement calculus notion of contracts gives a formal semantics for UML use case diagrams and use case models. The contracts are not meant as a replacement of the use case diagrams as they are currently used, but as a formal

complement of them. Use cases and contracts are created in two different phases of the software engineering process. While use cases are developed during the analysis phase, contracts are used as an intermediate step between the informal specification of the system and its detailed design and implementation. While use cases present an informal view of the functional requirements of the system, the contracts express the same requirements in a precise form.

This chapter is structured as follows. In Section 3.2 we review the use case notation using an example that we then develop throughout this chapter. In Section 3.3 we present the notion of contracts provided by the refinement calculus. The formal specification of the case study using contracts is described in Section 3.4. Section 3.5 presents the analysis method that we propose for use cases and applies it to the example. Related research is discussed in Section 3.6. We summarize the main contributions of this chapter in Section 3.7.

## 3.2 Use Cases

Use cases were introduced by Jacobson in [54] as follows: "A use case is a sequence of transactions in a system, whose task is to yield a measurable value to an individual actor of the system". In addition, he wrote that "the set of use case descriptions specifies the complete functionality of the system".

An actor is a coherent role played by a user of the system. Each actor has certain goals to fulfill by using the system. However, a system can only be used as described in the use cases. The collection of use cases logically decomposes the functionality of the system; each use case involves certain requirements on the system.

The behavior of a use case can be described using plain English, interaction diagrams, unstructured text, expressions in a formal language or other layouts, like tables [3]. In UML, the relationships among actors and use cases are represented with association relationships. Different use cases can be related to each other by 'generalization', 'include' and 'extend' dependencies.

Figure 3.1 shows the use case diagram for the example we will develop throughout this chapter. The example is a system for managing the loans of a library. The members of a the library can borrow books, renew loans and return the books by a deadline. There is a fee to be paid if a book is returned after its deadline. The fee is 1 unit per each day late. Members cannot borrow another book or renew an existing loan if they have debts to pay. Most books are freely available but some specials books, mainly reference books, are restricted. Usually, restricted books cannot be borrowed and should remain within the library premises. However, in some special cases such as before an extended holiday, the librarian can allow reliable members to borrow restricted books. In any case, the librarian is who decides when it is possible to borrow restricted

books and by whom. It is not possible to renew a loan of a restricted book.

For simplicity, we assume that the set of members and the set of books is constant in the system.

The requirements of this system are illustrated in Fig. 3.1 by the four use cases. The members of the library and the librarian, who is in charge of the system, form the context of the system. Two actors represent these two user roles. A member can borrow a book, renew a loan, or return a book. These tasks can include the payment of a fee and are performed with the help of the librarian.

We describe the behavior of the use cases of the library example as tables in Tables 3.1 to 3.4. Each section of the table describes one aspect of a use case. The 'Use Case' section shows the name from the diagram and the 'Actors' section describes the actors involved. The 'Summary' row gives a brief description of the use case. The 'Precondition' section describes some conditions that must be fulfilled by the actors before attempting the use case. The effect of the use case in the system is described in the 'Postcondition' row. The 'Description' section contains a more detailed description of the sequence of actions that can take place during the use case. The 'Exceptions' row lists all the possible deviations from the behavior presented in the 'Description' row. Finally, the 'Includes' section lists the use cases that are included in the use case described by the table.

The heuristics for describing the behavior of a use case are rather interesting. For instance, it is not obvious what determines the separation between the precondition and the description of the use case. For example, in the Borrow Book a book can be borrowed by a member only if she has no debts to the library. Other logical conditions can be stated, i.e. that the book should not be already lent or that is not restricted. Thus, three conditions must be fulfilled in order to borrow a book. The reason that they are not stated in the precondition section is that they can only be checked while carrying out the use case.

### 3.2.1 Why Use Cases?

Constructing use cases has many advantages. When we create use cases, we:

- Identify the different goals for individual actors.
- Capture the context that the system is supposed to live in and also the functionality required by the external actors.
- Understand the problem domain, the candidate objects for the problem domain, and also of the proposed solution, and
- Provide a model against which the final system design can be checked.

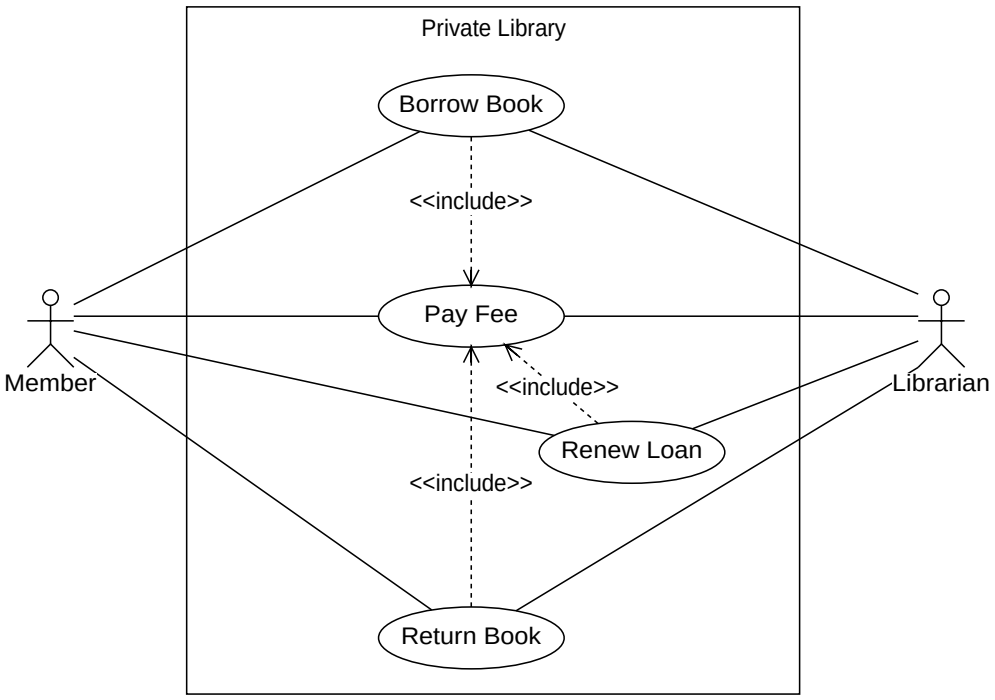



Figure 3.1: Use Case Diagram

| Use Case | Borrow Book |
|---|---|
| Actors | Member, Librarian |
| Summary | The Member borrows a book from the library. |
| Precondition | |
| Description | The Member chooses a book that is not already lent. The Librarian updates the library card of the member: assigns the Member as the borrower of that book and sets deadline for returning the book. |
| Postcondition | The Member has successfully borrowed the book. |
| Exceptions | 1. If the Member has debts to pay, the Member can pay them (see Pay Fee use case) and try again.<br>2. The book is already lent.<br>3. The book is restricted and the librarian does not allow to borrow it. |
| Includes | Pay Fee |

Table 3.1: Use Case Borrow Book

| Use Case | Renew Loan |
|---|---|
| Actors | Member, Librarian |
| Summary | The Member renews the loan of a book. |
| Precondition | |
| Description | The Member chooses a book that is already borrowed by her. The Librarian assigns a new deadline for returning that book. |
| Postcondition | The Member has successfully renewed the loan. |
| Exceptions | 1. If the Member has debts to pay, the Member can pay them (see Pay Fee use case) and try again.<br>2. The Member did not borrow the book before.<br>3. The book is restricted. |
| Includes | Pay Fee |

Table 3.2: Use Case Renew Loan

| Use Case | Return Book |
|---|---|
| Actors | Member, Librarian |
| Summary | The Member returns a book to the library. |
| Precondition | The Member has borrowed the book before. |
| Description | If the Member has debts to pay, the Member can pay them (see Pay Fee use case). The book is returned and the Member is no more the borrower of the book. |
| Postcondition | The Member has successfully returned the book. |
| Exceptions | 1. The Member did not borrow the book before.<br>2. The book is already returned. |
| Includes | Pay Fee |

Table 3.3: Use Case Return Book

| Use Case | Pay Fee |
|---|---|
| Actors | Member, Librarian |
| Summary | The Member pays a certain amount of her debts to the library. |
| Precondition | The Member has debts to the library. |
| Description | The Member chooses a certain amount of the debt to pay. The Librarian subtracts the sum from the Member's debts. |
| Postcondition | The Member has successfully paid the sum. |
| Exceptions | The Member has no debts to pay. |

Table 3.4: Use Case Pay Fee

- Create an informal contract between the users and the developers.

Still, use cases have also a number of shortcomings:

- They are informal. This is an advantage during the beginning of the development process, but later on, informal requirements can be easily misinterpreted.
- It is difficult, if not impossible, to check whether the system provides the functionality expected by the actors, that is, it is difficult to ensure that the actors can achieve their goals by using the system.
- They are essentially functional in character, even though they are used in UML to develop object-oriented systems. There is a missing connection between the control-oriented use case diagrams and object-oriented class diagrams.

These missing features constitute the starting point for our approach. We want to express the information captured by use cases more precisely, using a consistent syntax with well defined and precise semantics for regulating the allowed activities between the actors and the system. We propose to complement the use case diagrams using formal *contracts* within the refinement calculus. Contracts have the following advantages:

- They are based on a rigorous formalism. The interpretation of a contract is unambiguous.
- They can be analyzed mathematically. We can prove achievability and correctness of contracts.
- The state space of a contract can be described in terms of a UML class diagram. Contracts can connect the functional requirements of a system with its object-oriented description of its problem domain.

As the main contribution of this chapter, we will show how UML use case diagrams, such as the one in Fig. 3.1, can be completely described using a *contract* in the refinement calculus. Moreover, we show how to analyze a contract in order to check whether an actor can achieve its goals. The following section describes contracts in more detail.

## 3.3 Contracts

A use case diagram is, in fact, an informal model for the computation required of the system under development. This computation is carried out by a collection

of actors and the system. We identify the notions of 'actor' and 'system' in UML with the notion of agent in the refinement calculus formalism. An *agent* is an entity identified by a given name and it has free will. A computation is seen as a number of agents (programs, modules, systems, users, etc...) that carry out actions according to a document (specification, program) given in advance. When reasoning about a computation, we can view this document as a *contract* between the agents involved. Below we describe the notion of a contract as put forward by Back and von Wright in [13, 12].

The refinement calculus is based on Higher-order logic, an extension of simply typed lambda calculus. Functions are defined by $\lambda$-abstraction and can be used without explicit definition and naming. For example, the function that calculates the successor of a natural number is defined as $(\lambda n \cdot n + 1)$. We denote by $f.x$ the application of the function $f$ to the argument $x$, so that, e.g.$(\lambda n \cdot n + 1).1 = 2$. A binary relation $R \subseteq A \times B$ is here considered as a function $R : A \to \mathcal{P}B$ i.e. mapping elements in $A$ to sets of elements in B.

The world that a contract talks about is described as a state $\sigma$. The state space $\Sigma$ is the set of all possible states $\sigma$. The state is observed via a collection of *attributes* $x_1, ..., x_n$, each of which can be observed and changed independently of the others. Each attribute is associated with a non-empty set of *values*, called the *type set* of the attribute. If the attribute $x$ takes values from $\Gamma$, we say that $x$ has the type $\Gamma$ and we write it as $x : \Gamma$. We consider several predefined types, like $\mathsf{Real}_+$ for the set of non-negative real numbers, $Bool$, for the truth values, $Date : 2001/07/0109 : 36 : 34$, $Integer$, $Natural$ and $String$.

An attribute $x$ of type $\Gamma$ is really a pair of two functions, the *value function* $valx : \Sigma \to \Gamma$ and the *update function* $setx : \Gamma \to \Sigma \to \Sigma$. The function $valx$ returns the value of the attribute $x$ in a given state, while the function $setx$ returns a new state where $x$ has a specific value, while the values of all other attributes are unchanged.

We will assume here that attributes are partitioned into *objects*. For instance, if we have the attributes $o1.name$, $o2.name$, $o1.age$, $o2.age$, $o3.author$, $o3.title$, we group these attributes into three objects. Object $o_1$ has attributes $o_1.name$, $o_1.age$, object $o_2$ has attributes $o_2.name$, $o_2.age$, and object $o_3$ has attributes $o_3.title$, $o_3.author$. There may also be attributes that are not part of any object: they represent local information for the contract.

We simplify our presentation by assuming a constant collection of objects for the systems we model. This restriction excludes the dynamic creation and destruction of objects. Our formalism can be extended to handle also these issues, but since they are not the focus of the analysis of use cases, dynamic collections of objects have been excluded. The usual rules for arrays can then be used to reason about dynamic collections of objects as well [12]. As the grouping of attributes into objects is just a logical partition of the set of attributes, we can reason in the sequel as if the state has a flat collection of attributes.

The state $\sigma$ can be also observed via different combinations of attributes. An *expression* like $x + y$ is a function on states. Evaluating an expression in a state gives a value: $x + y.\sigma = valx.\sigma + valy.\sigma$. We thus use expressions to observe properties of the state. The expressions are also used to modify the state, in assignments like $x :=\, x + y$. This assignment denotes a state changing function that updates the value of $x$ to the value of the expression $x + y$, i.e.

$$(x :=\, x + y).\sigma = setx.(valx.\sigma + valy.\sigma).\sigma$$

A function $f : \Sigma \to \Sigma$ that maps states to states (like the assignment) is called a *state transformer*. We also make use of predicates and relations over states. A *state predicate* is a boolean function $p : \Sigma \to Bool$ on the state. A *boolean expression* is an expression that ranges over truth values $Bool$, and is used to describe predicates on the state. For instance, $x \leq y$ is a boolean expression stating that $valx.\sigma \leq valy.\sigma$ in a given state $\sigma$. A *state relation* $R : \Sigma \to \Sigma \to Bool$ relates the state $\sigma$ to a state $\sigma'$ whenever $R.\sigma.\sigma'$ holds. We permit a generalized assignment notation for relations. For instance, $[x :=\, x' | x' > x + y]$ relates state $\sigma$ to state $\sigma'$ if the value of $x$ in $\sigma'$ is greater than the sum of the values of $x$ and $y$ in $\sigma$ and all other attributes are unchanged. This notation generalizes ordinary assignment: we have that $\sigma' = (x :=\, e).\sigma$ if and only if $(x :=\, x' | x' = e).\sigma.\sigma'$.

The updates of the state are usually performed from behalf of a certain agent. Their allowed behavior is restricted by a contract, that is described by a *contract statement*. The syntax of contract statements is as follows:

$$S ::= x :=\, e \mid \{p\}_a \mid [x :=\, x' \mid b]_a \mid S_1 \,;\, S_2 \mid S_1 \sqcup_a S_2 \mid (\mathsf{rec}_a\ X \cdot S_1) \qquad (3.1)$$

Here $x$ is a list of attributes, $e$ is a list of expressions, $p$ and $b$ are state predicates, $x'$ is a list of values, $a$ is an agent name and '$\mathsf{rec}_a\ X$' denotes the recursion over the variable statement $X$. The agents carry out a contract statement as follows.

The *assignment statement* '$x :=\, e$' sets simultaneously the values of the attributes in the list $x$ to the values of the expressions in the list $e$, evaluated in the initial state.

The *assertion* '$\{p\}_a$' is a requirement that agent $a$ must satisfy in a given state. E.g., '$\{x + y = 0\}_a$' expresses that the sum of (the values of the attributes) $x$ and $y$ in the state must be zero and that this should be guaranteed by the agent $a$. If the assertion holds when the contract is carried out, then the state is unchanged. If the assertion does not hold, then the agent $a$ has *breached* the contract and the execution of the contract is aborted.

The *non-deterministic assignment* '$[x :=\, x' \mid b]_a$' is a contract statement that permits the agent $a$ to choose between all final values $x'$ related by $b$ to the

initial value of $x$. If no such final value exists, then the agent $a$ has breached the contract. For instance, in the contract statement '$[x := x' \mid x < x']_a$' agent $a$ chooses for $x$ a value $x'$ that is larger than the current value of $x$, without changing the values of any other attributes.

In the *sequential composition* '$S_1 \,;\, S_2$' the statement $S_1$ is first carried out, followed by $S_2$. A *choice* '$S_1 \sqcup_a S_2$' obliges agent $a$ to choose between carrying out $S_1$ or $S_2$. The agent $a$ is free to choose either alternative. This binary choice extends directly to a choice among a finite set of alternatives.

The *recursive contract* '$(\mathsf{rec}_a\ X \cdot S)$' is interpreted as the contract statement $S$, but with each occurrence of the variable statement $X$ in $S$ treated as a recursive invocation of the whole contract. We assume that the recursion is done on behalf of the agent $a$, who is responsible for the termination of the recursion. Nontermination (infinite unfolding) means that the agent $a$ breaches the contract.

Programs can be seen as special cases of contracts where exactly two agents are involved: the *user* and the *computer system*. In simple batch-oriented programs, choices are only made by the computer system, which resolves any internal choice in a manner that is unknown to the user of the system (non-determinism). In general, contracts can be given a game interpretation. The operational semantics of contract statements [12] encodes the rules of a game. The contract statement encodes a play (execution) of the game, while the agents involved are the players. They will try to achieve each a goal by following the rules of the game. That is, each agent will try to win the game. If an agent breaches the contract, then the contract is aborted and all the other agents are *released* from carrying out their obligations stated by the contract statement [12].

We can easily extend the simple language (3.1) of contracts to include other program constructs, such as 'abort' or 'skip' statements, conditionals and iteration. The $\mathsf{abort}$ statement can be explained as $\mathsf{abort} = \{\mathsf{false}\}_{user}$. If executed, it means that the user has breached the contract, releasing the other agents (the system) from any obligations to carry out the rest of the contract. We can also introduce the contract $\mathsf{skip}$ which leaves the state unchanged. We define $\mathsf{skip} = \mathsf{id}$ where '$\mathsf{id}$' is the identity function. The *conditional contract*

$$\mathsf{if}\ x \geq 0\ \mathsf{then}\ x := x + 1\ \mathsf{else}\ x := x + 2\ \ \mathsf{fi}$$

stands for the contract

$$(\{x \geq 0\}_{system}\ ;\ x := x + 1) \sqcup_{system} (\{x < 0\}_{system}\ ;\ x := x + 2)$$

The computer system can choose here between two options. We assume that an agent does not want to breach a contract (it wants instead to win the game). The agent will therefore always choose the alternative for which

the guarding assertion is true; choosing the other alternative would breach the contract. The contract statement '$\mathsf{if}\ p\ \mathsf{then}\ S\ \mathsf{fi}$' is interpreted as a shortened form for '$\mathsf{if}\ p\ \mathsf{then}\ S\ \mathsf{else\ skip\ fi}$'.

*Iteration* is defined in terms of recursion:

$$\mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}\ = (\mathsf{rec}_{user}\ X \cdot \mathsf{if}\ g\ \mathsf{then}\ S\ ;\ X\ \mathsf{else\ skip\ fi})$$

This interpretation means that we consider nontermination of the loop as an error which the user should try to avoid.

As a final extension of the language for contracts, we introduce procedures:

$$S ::= ... \mid q_a.$$

Here $q_a$ stands for a procedure call done by agent $a$. A procedure declaration

$$\mathsf{proc}\ q : \mathsf{pre}\ b\ ;\ S$$

introduces a procedure named $q$, with precondition $b$ and body $S$. When a procedure is called, there is usually an agent $a$ responsible for the call. The procedure call is then interpreted as '$\{b\}_a\ ;\ S$', i.e. the agent is responsible for the precondition being satisfied upon entry to the procedure. If agent $a$ has called the procedure in a state that does not satisfy its precondition, then $a$ has breached the contract. A procedure without precondition, '$\mathsf{proc}\ q : S$' is interpreted as '$S$', i.e. as if the precondition would be '$\mathsf{true}$'. Value and variable parameters for procedure calls are specified using the keywords $\mathsf{val}$ and $\mathsf{var}$ respectively; when a procedure with parameters is called, then the formal parameters are substituted for the actual parameters in a way that takes the parameter passing mechanisms into account [12]. Preconditions can be used to verify the parameters passed to a procedure body. E.g., if a procedure requires that two of its parameters $p_1$ and $p_2$ refer to two different objects (no aliasing), we can add the precondition $\mathsf{pre}\ p_1 \neq p_2$.

When several objects have the same structure and behavior, they belong to the same *class*. For instance, objects $o_1$ and $o_2$ mentioned before belong to a class *Person* with attributes *name* and *age*. An agent can modify the attributes belonging to an object by invoking a method of the object. Fig. 3.2(a) shows the specification of a class where $x$ denotes a list of public attributes, declared with their types, $x_0$ a list of initial values and $p$ a list of operations. The initial values can be specified for each attribute in the $\mathsf{init}$ section; if the initialization is missing, then the attribute gets an arbitrary value in its type set. The operations of the class are public procedures. They act only over the attributes of the class and over their own parameters, if any.

This basic model of classes and objects is enough for describing simple use case models but it lacks many advanced features like inheritance. However, it

```
class    ClassName
  attr   x
  init   x := x_0
  proc   p
end
```

(a) Syntax of Class

```
contract   ContractName
  agent    a
  var      y
  proc     q
begin
  S
end
```

(b) Syntax of Contract

Figure 3.2: Formal Specification Syntax

is not our intention to formalize all the features present in UML class diagrams but to analyze use cases.

Fig. 3.2(b) shows the specification of a contract where $a$ stands for the list of the agents involved, $y$ for the local attributes, $q$ for the list of procedures used in the contract, and $S$ for the body of the contract. We use these specification forms from Fig. 3.2 in the next section.

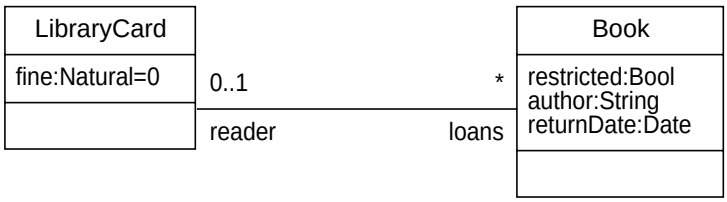
## 3.4 Example: Library

In this section, we show how to express the Library use case model from Section 3.2 as a contract. There are two main tasks in this process. First, we must model the state of the contract. That is, we must create a class diagram describing the objects involved in the use cases. After this step, we should translate the informal description of the behavior of each use case into a series of contract statements.

The state of a contract models the problem domain of the system, graphically represented in UML as a class diagram. The class diagram for the Library example is shown in Fig. 3.3. We essentially need to model two notions within our system: library cards and books. The identification and construction of the classes is based on the description of the use cases.

Classes and their relationships can also be expressed using the format shown in Fig. 3.2(a). These two representations of the problem domain are equivalent, except that the refinement calculus description allows the behavior of the operations of the classes to be defined in a more precise way.

Figure 3.3 shows the attributes of the Book and LibraryCard. When a book is borrowed, the attribute returnDate stores the deadline for returning it. The name of the book's author is stored in the attribute author. The other three attributes are restricted, status and reader. Restricted stores whether the book is restricted or not, status whether is Free or Lent, while reader denotes the




class $LibraryCard$
  attr $loans$ : set of $Book$ ; $fine$ : $Natural$
  init $loans, fine := \emptyset, 0$

class $Book$
  attr $returnDate : Date$;
    $author : String$;
    $restricted : Bool$;
    $status : \{Lent,\ Free\}$;
    $reader : LibraryCard \cup \{\text{None}\}$
  init $status, reader := Free, \text{None}$
end

Figure 3.3: Library Class Diagram

instance of library card that has been used to borrow the book. Initially, a book is not lent (status=Free) and the reader attribute has the value None since there is no reader of the book yet. This is stated in the init section of the class.

The class LibraryCard has two attributes. The attribute loans stores the references to the books that are borrowed by the person. The attribute fine represents the debts that the member has to pay to the library. Initially there are no loans and the member has nothing to pay.

There are two agents involved in this contract: a Member and a Librarian. The former represents the actions of a member of the library, while the latter models the reaction of the computerized library system. The use cases involve two objects, the library card of the member and the book that the member wants to borrow or return. We refer to these specific objects as card and book.

Creating a formal contract from the informal use case is not a always a trivial task. As an example, we show how to translate step by step the Borrow Book use case into a contract.

The first sentence of the use case states that "*The Member chooses a book that is not already lent*". We can model the selection of one object from a set with a non-deterministic assignment. In the example, the agent Member can choose any book that is not restricted and its status attribute is free:

$$[book :=b' \mid b' \in Book \wedge \neg b'.restricted \wedge b'.status = Free]_M$$

However, we know that sometimes the librarian allows to borrow a restricted book. We specify this choice with the binary choice statement $\sqcup$:

$$\begin{array}{l} [book :=b' \mid b' \in Book \wedge \neg b'.restricted \wedge b'.status = Free]_M \\ \sqcup_L \\ [book :=b' \mid b' \in Book \wedge b'.status = Free]_M \end{array}$$

The second sentence indicates that "*... assigns the Member as the borrower of that book and also states a deadline for returning the book*". The first part of the sentence states that we should create the link between the book and the card objects. We can perform this with two assignments:

$$book.reader = card \;\; ; \;\; card.loans = card \cup \{book\}$$

We can then update the status of the book and its deadline:

$$book.status = Lent \;\; ; \;\; book.returnDate = Today() + 4\ weeks$$

The 'Exceptions' section of the table states that members cannot borrow a book if they have debts to pay. We can group the previous assignments under an if - then -fi statement so they only occur if the debt annotated in the card is zero:

$$
\begin{array}{l}
\mathsf{if}\ \mathit{card.fine} = 0\ \mathsf{then} \\
\quad \mathit{book.reader} = \mathit{card}\ \ ;\ \ \mathit{card.loans} = \mathit{card} \cup \{\mathit{book}\}; \\
\quad \mathit{book.states} = \mathit{Lent}\ \ ;\ \ \mathit{book.returnDate} = \mathit{Today}() + 4\ \mathit{weeks} \\
\mathsf{fi}
\end{array}
$$

However, the member has the opportunity to pay the fee:

$$\mathsf{if}\ \mathit{card.fine} \neq 0\ \mathsf{then}\ \mathit{PayFee}\ \mathsf{fi}$$

We will define the PayFee use case later. By now, we have all the statements that involve the Borrow Book use case, but we must order them in an exact sequence. This step is important because it is not always clear when to check for the possible exceptions to the normal behavior of a use case. In our example we have decided to arrange the contract as follows:

$$
\begin{array}{l}
\quad [\mathit{book} :=_{-} b' \mid b' \in \mathit{Book} \wedge \neg b'.\mathit{restricted} \wedge b'.\mathit{status} = \mathit{Free}]_M \\
\sqcup_L \\
\quad [\mathit{book} :=_{-} b' \mid b' \in \mathit{Book} \wedge b'.\mathit{status} = \mathit{Free}]_M; \\
\mathsf{if}\ \mathit{card.fine} \neq 0\ \mathsf{then}\ \mathit{PayFee}\ \mathsf{fi}; \\
\mathsf{if}\ \mathit{card.fine} = 0\ \mathsf{then} \\
\quad \mathit{book.reader} = \mathit{card}\ \ ;\ \ \mathit{card.loans} = \mathit{card} \cup \{\mathit{book}\}; \\
\quad \mathit{book.states} = \mathit{Lent}\ \ ;\ \ \mathit{book.returnDate} = \mathit{Today}() + 4\mathit{weeks} \\
\mathsf{fi}
\end{array}
$$

This contract formalizes the behavior of the Borrow Book use case. Because a large design involves many use cases, it is a good idea to encapsulate the contract into a procedure. It also a good idea to avoid updating the attributes of the objects directly. Instead, we can create new operations for the classes representing the state of the contract. The final version of the Borrow Book contract is as follows:

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{BorrowBook} : \\
\quad [\mathit{book} :=_{-} b' \mid b' \in \mathit{Book} \wedge \neg b'.\mathit{restricted} \wedge b'.\mathit{status} = \mathit{Free}]_M \\
\sqcup_L \\
\quad [\mathit{book} :=_{-} b' \mid b' \in \mathit{Book} \wedge b'.\mathit{status} = \mathit{Free}]_M; \\
\quad \mathsf{if}\ \mathit{card.fine} \neq 0\ \mathsf{then}\ \mathit{PayFee}_M\ \mathsf{fi}; \\
\quad \mathsf{if}\ \mathit{card.fine} = 0\ \mathsf{then} \\
\quad\quad \mathit{book.SetLoan}(\mathit{card})_L\ ;\ \mathit{card.BorrowBook}(\mathit{book})_M \\
\quad \mathsf{fi}
\end{array}
$$

We proceed in a similar way until we have formalized all the use cases as contracts. We formalize the operations of the objects by observing how the different use cases manipulate the attributes of the objects. In our example,

```
class Book
   attr returnDate : Date ; author : String ; restricted : Bool;
      status : {Lent, Free} ; reader : LibraryCard ∪ {None}
   init status, reader : = Free, None
   proc SetLoan(val card : LibraryCard) :
     pre status = Free;
     reader, status, returnDate : = card, Lent, Today() + 4 weeks
   proc ResetLoan :
     pre status = Lent;
     reader, status : = None, Free
   proc NewDeadLine :
     returnDate : = Today() + 4 weeks
   proc Price(var sum : Natural) :
     if returnDate ≥ Today() then sum := 0
     else sum : = (Today() − returnDate) fi
end
```

Figure 3.4: Complete Definition of the Class Book

we define the operations SetLoan and ResetLoan to set the attributes of a Book object when a loan is performed or canceled. If an existing loan is extended, the attribute returnDate is modified by the method NewDeadLine. A fee on the book is computed by the method Price: the fee is zero only if the deadline is still in the future or is the same day as the day of the transaction. Otherwise, the fee is computed as described before.

The debt on a LibaryCard object is modified using the method ModifyFine of that class. There are also two operations for properly setting the attributes of a LibraryCard object when a loan is performed or canceled: BorrowBook and ReturnBook. Figure 3.4 shows the complete definition of the Book class while Figure 3.5 shows the LibraryCard class.

Finally, we describe the main statement of the use case, where we allow a member to choose her library card and invoke one of the available use cases:

$$[card : = c' \mid c' \in LibraryCard]_M \ ;$$
$$BorrowBook \sqcup_M RenewLoan \sqcup_M ReturnBook \sqcup_M PayFee$$

Figure 3.7 shows the complete contract for the Library system. There is another local attribute for the contract, amount, used in ReturnBook and PayFee use cases. The main statement of the contract is shown in the begin...end clause of the contract.

```
class LibraryCard
   attr loans : set of Book ; fine : Natural
   init loans, fine : = ∅, 0
   proc BorrowBook(val book : Book) :
     pre book ∉ loans;
     loans : = loans ∪ {book}
   proc ReturnBook(val book : Book) :
     pre book ∈ loans;
     loans : = loans \ {book}
   proc ModifyFine(val diff : Integer) :
     pre fine + diff ≥ 0;
     fine : = fine + diff
end
```

Figure 3.5: Complete Definition of the Class LibraryCard

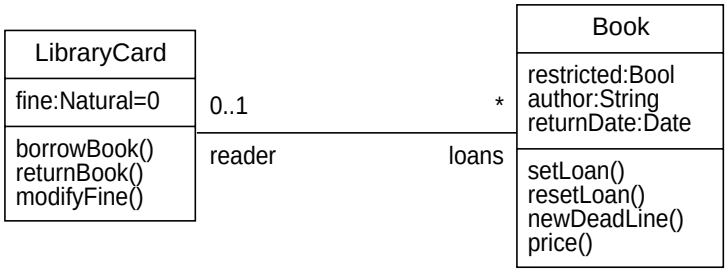



Figure 3.6: Library Class Diagram with Operations

```
contract Library
  agent M, L
  var book : Book, card : LibraryCard, amount : Natural;
  proc BorrowBook :
      [book :∈ b' | b' ∈ Book ∧ ¬b'.restricted ∧ b'.status = Free]_M
    ⊔_L
      [book :∈ b' | b' ∈ Book ∧ b'.status = Free]_M
    if card.fine ≠ 0 then PayFee_M fi;
    if card.fine = 0 then
      book.SetLoan(card)_L ; card.BorrowBook(book)_M fi
  proc RenewLoan :
    [book :∈ b' | b' ∈ Book ∧ ¬b'.restricted ∧ b' ∈ card.loans]_M ;
    if card.fine ≠ 0 then PayFee_M fi;
    if card.fine = 0 then book.NewDeadLine()_L fi
  proc ReturnBook :
    [book :∈ b' | b' ∈ Book ∧ b' ∈ card.loans]_M ;
    book.Price(amount);
    card.ModifyFine(amount)_L;
    if card.fine ≠ 0 then PayFee_M fi;
    card.ReturnBook(book)_M;
    book.ResetLoan()_L
  proc PayFee :
    {card.fine ≠ 0}_M;
    [amount :∈ pay | 0 ≤ pay ≤ card.fine]_M;
    card.ModifyFine(−amount)_L
begin
  [card :∈ c' | c' ∈ LibraryCard]_M ;
  BorrowBook ⊔_M RenewLoan ⊔_M ReturnBook ⊔_M PayFee
end
```

Figure 3.7: Library Contract

The preconditions of the operations of a class constrain (some of) the attributes that are about to be modified or the values that are to be assigned to these attributes. These preconditions refer exclusively to that respective class and do not express any other logical preconditions of the requirements of the use cases that would call these operations. We therefore use these class descriptions as the domain of our case study and we specify the entire functionality of the system in the contract in Fig. 3.7. It embeds all the use cases as pictured by the diagram in Fig. 3.1 and also captures the requirements described in Section 3.2. As the use cases involve both libray cards and books, we express the requirement in a use case using operations from both classes LibrayCard and Book. These operations treat the use case requirement from the perspective of the correspondent class.

The formal model of contracts that we use for describing the functionality of the system suits the object-oriented methodologies for developing systems. The use case diagrams are created during the first phase of a typical development process - the *analysis* phase. The class diagram and the classes as specified in Figures 3.4 and 3.5 are based on the use cases and so is the contract in Fig. 3.7. These two artifacts belong to the *design* phase. Moreover, the contract embeds both the use case diagram and the class diagram within its precise form, allowing for analysis on the system to be performed. We can verify whether an actor can meet its goals or whether the precondition it has to fulfill is sufficient for meeting its goals. These aspects form the subject of the following section.

## 3.5 Analysis

A contract describes in a precise manner the information contained in a use case diagram. We can therefore think of contracts as a formalization of use case diagrams. We intend to use this formalization to prove the achievability of the actors' goals when using a certain use case, by analyzing the contract.

In order to analyze a contract we need to define the precise meaning of each statement we use, i.e. we need to define the semantics of contract statements. For our basic language (3.1), the semantics is given within the refinement calculus as weakest precondition predicate transformers [13, 12]. For each statement $S$ and postcondition $q$, the predicate $wp_a.S.q$ denotes the weakest precondition for agent $a$ to achieve the goal $q$ by executing $S$:

$$
\begin{array}{lcl}
wp_a.(x := e).q & = & q[x/e] \\
wp_a.\{p\}_b.q & = & \begin{cases} \neg p \vee q, & a \neq b, \\ p \wedge q, & a = b. \end{cases} \\
wp_a.[x := x' \mid r]_b.q & = & \begin{cases} \forall x' \in r.x \cdot q[x/x'], & a \neq b, \\ \exists x' \in r.x \cdot q[x/x'], & a = b. \end{cases} \\
wp_a.(S_1 \,;\, S_2).q & = & wp_a.S_1.(wp_a.S_2.q) \\
wp_a.S_1 \sqcup_b S_2.q & = & \begin{cases} wp_a.S_1.q \wedge wp_a.S_2.q, & a \neq b, \\ wp_a.S_1.q \vee wp_a.S_2.q, & a = b. \end{cases}
\end{array}
$$

From this basic set of preconditions we can deduce formulas for other statements. E.g., we have:

$$
\begin{array}{lcl}
wp_a.\mathsf{abort}_b.q & = & \begin{cases} \mathsf{true}, & a \neq b \\ \mathsf{false}, & a = b. \end{cases} \\
wp_a.\mathsf{skip}.q & = & q \\
wp_a.(\mathsf{if}\ p\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}).q & = & p \wedge wp_a.S_1.q \vee \neg p \wedge wp_a.S_2.q \\
wp_a.p_b.q & = & \begin{cases} \neg r \vee wp_a.P.q, & a \neq b, \\ r \wedge wp_a.P.q, & a = b. \end{cases}
\end{array}
$$

Here '$\mathsf{proc}\ p : \mathsf{pre}\ r \,;\, P$' is a procedure. If the procedure has any parameters, the actual values are substituting the formal parameters in the body $P$. A formula for the weakest precondition for the recursive statement can be expressed using a fixpoint notation. The details of the above definitions are studied in [12].

### 3.5.1 Proving properties

Using the $wp$ predicate transformer we can verify whether an agent can achieve its goal or not by using a certain contract. The goal is modeled by the predicate $q$ while the contract is given as a contract statement. The weakest precondition determines the initial states from which the agent can reach the goal, i.e. $wp_a.S.g$ calculates the conditions to be fulfilled by the agent $a$ in order to achieve the goal $g$ using contract $S$.

A contract can also be interpreted in terms of games. Thus, an agent $a$ achieves a goal $q$ with the contract $S$ if there is a *winning strategy* for $a$ to establish $q$ with the contract $S$, no matter how the other agents will behave.

As an example, we analyze the conditions in which a member of the library can borrow a book written by a certain author. For this, the agent Member gets her library card and invokes the use case BorrowBook. The agent can successfully borrow a book written, say, by Steven King, with a given library

card, i.e. if $\exists b \in Book \cdot b.author = \text{`StevenKing'} \wedge b \in card.loans$ holds after invoking the use case. We can determine the weakest precondition to achieve this goal by computing:

$$wp_M.BorrowBook_M.\exists b \in Book \cdot b.author = \text{`StevenKing'} \wedge b \in card.loans$$

If we obtain a predicate different from false, then we have shown that with our contract the member can borrow a new book authored by Steven King with a specific library card under a certain condition. Applying the rules of the weakest predicate transformer given above, the following result is obtained:

$$\begin{array}{l} wp_M.BorrowBook_M.\exists b \in Book \cdot b.author = \text{`StevenKing'} \wedge b \in card.loans = \\ \quad (\exists b' \in Book \cdot \neg b'.restricted \wedge b'.status = Free \\ \qquad\qquad b'.author = \text{`}StevenKing\text{'}) \vee \\ \quad (\exists b \in Book \cdot b.author = \text{`StevenKing'} \wedge b \in card.loans) \wedge \\ \qquad (\exists b' \in Book \cdot \neg b'.restricted \wedge b'.status = Free) \end{array}$$

We can interpret this result in the following way. From the member's perspective, to borrow a book written by a certain author, with a certain library card, there should exist in the library a free non-restricted book $b'$ and written by that author ($\neg b'.restricted \wedge b'.status = Free \wedge b'.author = \text{`StevenKing'}$).

There is also an alternative way to satisfy the goal: if a book written by that author is already being borrowed by the member before executing the use case. However, in this case there should also exist another free book in the library. This latter condition was not intended. After translating the use cases and designing the classes and the contract, we can get such unexpected results, due to some errors in analysis, design or proof. Here the error is easy to detect: the member is always forced to choose a book. An improved design of the use case, which avoids this error, is given below:

$$\begin{array}{l} \textsf{proc } BorrowBook: \qquad // \textit{ use case } \textbf{Borrow Book} \\ \quad \textsf{if } card.fine \neq 0 \textsf{ then } PayFee_M \textsf{ fi}; \\ \quad ( \\ \quad \{card.fine = 0\}_M; \\ \qquad [book :{=} b' \mid b' \in Book \wedge \neg b'.restricted \wedge b'.status = Free]_M \\ \quad \sqcup_L \\ \qquad [book :{=} b' \mid b' \in Book \wedge b'.status = Free]_M \\ \quad book.SetLoan(card)_L \; ; card.BorrowBook(book)_M \\ \quad ) \\ \sqcup_M \\ \quad \textsf{skip} \end{array}$$

We show a detailed calculation of the *wp* for the revised contract in Fig. 3.8. In the following figures, we use a new format for performing the derivations. This format is based in the structured calculational proof [6, 11]. Basically we use an outline for showing the formulas and the statements, thus avoiding the parentheses, the sequential composition, the conjunction and the disjunction signs. More precisely, the predicate *wp* has two subformulas, namely the statement and the postcondition. The statement is shown in bold face and the formulas in normal face. Subsequent lines on the same indentation level denote sequential composition in the statement or a conjunction in formulas. A disjunction in formulas or a non-deterministic choice in the statement is preceded by the keywords *either* and *or*, indexed in the latter case with the name of the agent responsible for the choice. The conditional is shown using the format *if condition: ...* and the implication using the keywords *if ... then.* Existentially and universally quantified formulas have a header, the quantification, with the body written as a subformula. Each subformula is indented with respect to its parent. This format can be extended to cover all the constructs of the language of contracts.

After applying the weakest precondition rules to the revised *BorrowBook* use case and the postcondition ($\exists b \in Book \cdot b.author = \text{'StevenKing'} \wedge b \in card.loans$), a structure similar to the initial contract is obtained, as shown in Fig. 3.8. The result of the computation, after further simplifications, is shown in Fig. 3.9.

This result can be interpreted as follows. From the member's perspective, in order to borrow a book written by Steven King, there should exist a non-restricted free book in the library, which is written by that author ($\neg book.restricted \wedge book.status = Free \wedge book.author = \text{'StevenKing'}$). Alternatively, a book written by the respective author should already be being borrowed by the member before executing the use case.

We have also computed the weakest precondition for reaching the same goal as above, with the same contract, when the two agents are not against each other, but instead co-operate. The agents are then said to form a coalition [12], and can be represented by the same agent name, say $M$. The result of this computation is presented in Fig. 3.10. We note that in this case it is possible to achieve the goal by borrowing a restricted book. This is no surprise since the member and the librarian are allies (or even the same person) and the librarian will always allow the member to borrow restricted books.

Finally, it is interesting to also check the weakest precondition for reaching the same goal as above, with the same contract, but now from the librarian's perspective. That means that we compute the conditions in which a book written by a certain author can be borrowed by a member even when the latter might be against this action. The question is interesting by itself. How can a librarian manage to lend a book by Steven King to a member, even if the

$wp_M$
  **if card.fine** $\neq$ **0** :
    **PayFee**$_{\mathbf{M}}$
  **either**$_{\mathbf{M}}$
    $\{$**card.fine** $=$ **0**$\}_{\mathbf{M}}$
    **either**$_{\mathbf{L}}$
      $\{$**book** $:=$ **b**$|$**b** $\in$ **Book** $\wedge$ $\neg$**b.restricted** $\wedge$ **b.status** $=$ **Free**$\}_{\mathbf{M}}$
    **or**$_{\mathbf{L}}$
      $\{$**book** $:=$ **b**$|$**b** $\in$ **Book** $\wedge$ **b.status** $=$ **Free**$\}_{\mathbf{M}}$
    **book.SetLoan**(**card**)$_{\mathbf{L}}$
    **card.BorrowBook**(**book**)$_{\mathbf{M}}$
  **or**$_{\mathbf{M}}$
    **skip**
  $\exists b \in Book \bullet$
    $b.author =$ *'StevenKing'*
    $b \in card.loans$

= {applying the $wp$ rules for the above contract}

*either*
  $card.fine \neq 0$
  $\exists book \in Book \bullet$
    $\neg book.restricted$
    $book.status = Free$
    $book \notin card.loans$
    $\exists b \in Book \bullet$
      $b.author =$ "$StevenKing$"
      $b \in card.loans \cup book$
*or*
  $card.fine = 0$
  $\exists book \in Book \bullet$
    $\neg book.restricted$
    $book.status = Free$
    $book \notin card.loans$
    $\exists b \in Book \bullet$
      $b.author =$ "$StevenKing$"
      $b \in card.loans \cup book$
*or*
  $\exists b \in Book \bullet$
    $b.author =$ "$StevenKing$"
    $b \in card.loans$

Figure 3.8: Weakest Precondition of the BorrowBook Use Case

$$
\begin{array}{l}
\mathit{either} \\
\quad \exists \mathit{book} \in \mathit{Book} \bullet \\
\qquad \neg \mathit{book.restricted} \\
\qquad \mathit{book.status} = \mathit{Free} \\
\qquad \mathit{book.author} = \text{"}\mathit{StevenKing}\text{"} \\
\mathit{or} \\
\quad \exists b \in \mathit{Book} \bullet \\
\qquad \mathit{b.author} = \text{"}\mathit{StevenKing}\text{"} \\
\qquad b \in \mathit{card.loans}
\end{array}
$$

Figure 3.9: Result of the Weakest Precondition of the BorrowBook use case

$$
\begin{array}{l}
wp_M \\
\quad \textbf{if card.fine} \neq \textbf{0} : \\
\qquad \textbf{PayFee}_{\mathbf{M}} \\
\quad \textbf{either}_{\mathbf{M}} \\
\qquad \{\textbf{card.fine} = \textbf{0}\}_{\mathbf{M}} \\
\qquad \textbf{either}_{\mathbf{M}} \\
\qquad\quad \{\textbf{book} :=_{\mathbf{S}} \textbf{b} | \textbf{b} \in \textbf{Book} \wedge \neg \textbf{b.restricted} \wedge \textbf{b.status} = \textbf{Free}\}_{\mathbf{M}} \\
\qquad \textbf{or}_{\mathbf{M}} \\
\qquad\quad \{\textbf{book} :=_{\mathbf{S}} \textbf{b} | \textbf{b} \in \textbf{Book} \wedge \textbf{b.status} = \textbf{Free}\}_{\mathbf{M}} \\
\qquad \textbf{book.SetLoan}(\textbf{card})_{\mathbf{M}} \\
\qquad \textbf{card.BorrowBook}(\textbf{book})_{\mathbf{M}} \\
\quad \textbf{or}_{\mathbf{M}} \\
\qquad \textbf{skip} \\
\quad \exists b \in \mathit{Book} \bullet \\
\qquad \mathit{b.author} = \mathit{'StevenKing'} \\
\qquad b \in \mathit{card.loans} \\
= \{\text{applying the } wp \text{ rules for the above contract}\} \\
\mathit{either} \\
\quad \exists \mathit{book} \in \mathit{Book} \bullet \\
\qquad \mathit{book.status} = \mathit{Free} \\
\qquad \mathit{book.author} = \text{"}\mathit{StevenKing}\text{"} \\
\mathit{or} \\
\quad \exists b \in \mathit{Book} \bullet \\
\qquad \mathit{b.author} = \text{"}\mathit{StevenKing}\text{"} \\
\qquad b \in \mathit{card.loans}
\end{array}
$$

Figure 3.10: Result of the wp Computation, when M=L

$wp_L$
  **if card.fine** $\neq$ **0** :
    **PayFee**$_{\mathbf{M}}$
  **either**$_{\mathbf{M}}$
    $\{$**card.fine = 0**$\}_{\mathbf{M}}$
    **either**$_{\mathbf{L}}$
      $\{$**book** : $=$ **b**$|$**b** $\in$ **Book** $\wedge$ $\neg$**b.restricted** $\wedge$ **b.status = Free**$\}_{\mathbf{M}}$
    **or**$_{\mathbf{L}}$
      $\{$**book** : $=$ **b**$|$**b** $\in$ **Book** $\wedge$ **b.status = Free**$\}_{\mathbf{M}}$
    **book.SetLoan**(**card**)$_{\mathbf{L}}$
    **card.BorrowBook**(**book**)$_{\mathbf{M}}$
  **or**$_{\mathbf{M}}$
    **skip**
  $\exists b \in Book \bullet$
    $b.author$ = *'StevenKing'*
    $b \in card.loans$

= {applying the *wp* rules for the above contract}
$\exists b \in Book \bullet$
  $b.author =$ "$StevenKing$"
  $b \in card.loans$

Figure 3.11: The Librarian Result of the wp Computation

member does not want it?. The result of this computation is shown in Fig. 3.11, and can be interpreted as follows: the librarian cannot really force the member to borrow a given book since the member has always the option to not borrow a book thanks to the skip statement. The only way to ensure the librarian's goal is if the member already has the given book.

In this example we have shown how contracts can be used not only as a means for precisely describing use cases and for connecting the functional and object-oriented views of a system (Sections 3.3 and 3.4), but also as an analysis tool. By computing the weakest precondition for an agent to achieve a certain goal with a contract we *prove* that once that condition is fulfilled by the corresponding agent, then the goal can be achieved, even if the other actors do not collaborate in the task. By computing the weakest precondition for different agents, we check how the system will be used by different actors and also the initial conditions of reaching certain goals for these actors.

Using only the formulas of the weakest precondition predicate transformers, the derivations become rather long and complex. An example of such a derivation is shown in [8]. Switching to an outline format makes it easier to carry out the derivations by hand and also shows the similarity between the structure of the contract and the structure of the computed weakest precondition.

Besides the achievability analysis, other properties can be stated and proved

in a similar manner. We can state some properties as invariants for different classes or for the overall contract. Candidate conditions for the invariant can be, e.g., the cardinalities of the relationships between classes. We can state as an invariant of the class Book that $status = Free \equiv reader = None$. By calculating the weakest preconditions, we can check whether these invariants are preserved by the contract statements. All in all, the weakest precondition semantics of contract statements proves to be a powerful tool for analyzing the properties of a system under development.

## 3.6 Related Work

The roots of contracts as formal expressions for analyzing the software consist in the notion of *assertions*. This notion had been thought of ealier by, e.g. Alan Turing in a conference at Cambridge, in 1950 [47]: "How can one check a large routine in the sense that it's right? [...] the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows". These ideas were developed by Floyd [33], Hoare [46], Dijkstra [29] and many others. Continuing in this direction, a notion of contracts is used by Meyer in his paradigm 'Design by Contract' [72]. In his approach, a contract consists of a set of assertions - contract clauses - attached to a class: a class invariant and pre- and post-conditions attached to each method of the class. This paradigm builds on the principle that "in software development, reliability should be built-in, not an afterthought" [72]. Our notion of contracts also presumes that class methods are equipped with preconditions, but rather than using postconditions, the effect of a method call is described by a statement (which can be as abstract as a postcondition, by using non-deterministic assignments). We can also add an invariant to each class and check that each state change induced by a procedure call will keep the invariant true.

An important feature of object-oriented systems consists in the collaboration of distinct objects in order to accomplish certain tasks or goals. This feature was captured in [82] by Helm et. al, with their notion of contracts, that proposes a shift from class based design to interaction, or use case based design. The contracts in [82] specify in a precise manner the collaborations and the obligations of participant objects. The behavior specified in one contract can be further used by other contracts using relationships like refinement and inclusion. The way a class implementation meets the obligations of objects participating in a contract is specified by a 'conformance' declaration. The instantiation of contracts is also discussed. Our approach in this chapter resembles the contracts of Helm et al., but we emphasize the collaborations of *agents* and their obligations rather than the collaborations of objects. Thus,

we treat the problem domain as the vocabulary to work with and we constrain the ways it is used, by formalizing the behavior of UML actors, as captured by UML use cases. In our approach we specify use cases as procedures, while procedure calls in the body of the contract model use case *instances* as used by the involved agents.

Another way of using contracts as formal assertions for specifying collaborations of objects is presented in [36]. The methodology of Di Marzo et al. refers to an entire software development process in which a two-language framework is used. A logical language expressed in terms of temporal logic formulas denotes the properties or requirements of the system while a model-oriented language expressed as synchronized Petri nets describes the models or implementations of the system. A contract expressed in terms of logical properties is added to a model-oriented specification and the former describes the essential properties of the latter. As the software development process evolves, new models are derived from previous (and more abstract) ones, based on the refinement of contracts. In order for the new model to be a correct refinement of the old one, the contract attached to the old model should be refined by the contract attached to the new model. This approach resembles our approach to contracts in its emphasis on refinement. We intend our contracts to be stepwise refined to a correct implementation of the system, using the basic methods of the refinement calculus. The evolution from one contract to another is done by verifying certain conditions that relate the old and the new contract. These conditions are expressed within the same framework of the refinement calculus as the contracts. In particular, we use the *same* weakest precondition mechanism for refining our contracts as we use for analyzing them in contrast to the two-language framework used in [36].

More recently, contracts have been used as software engineering terms in the context of component software. Thus, components have *contractually* specified interfaces that are provided for clients [95]. Contracts specify the obligations that component providers must meet and the expectations component clients may have [103]. That is, one must specify for each component the services it offers to other components and also the services it requires from other components. These services are usually specified as contracts between the parts. Reuse contracts [50], an extension of Lamping's specialization clauses [63], were defined and developed as enhanced interface descriptions for sets of collaborating participant components. Hierarchies of components contracts were defined [14] in order to check a component behavior before using it. This feature is important due to the late integration of components. Since the context in which a component might be used is unknown, their integrity has to satisfy high standards. Differentiating between basic, behavioral, synchronization and quality of service contracts helps in improving the integrity of the components. Another method for designing contract-based components was presented in [40]. The

method proposes to regard contracts as stereotyped UML interfaces, extended with protocols for modeling the coordination and synchronization aspects during the design phase. While the UML interfaces specify the interaction steps, the protocols describe the supported interaction orders. This approach and our notion of contracts are similar in that they both describe the 'active' behavior of the components, in contrast to the (purpose of) UML interfaces.

The formalization of the structure of UML use cases is discussed in [78], but the authors do not show how to analyze the use cases. OCL is also a textual language that is part of the UML standard and it is used to constrain the behavior of UML elements. Unfortunately, OCL specifications cannot capture the interaction with the actors and, therefore, they are not as expressive as contracts.

## 3.7 Conclusions

In this chapter we have presented an analysis method for the UML use cases. The analysis method is based on the notion of a contract, understood as a formal specification of the behavior of a UML use case. The interpretation of a contract is given in terms of the weakest precondition operator and it is unambiguous. Our intention is to enhance the (informal) use case diagrams by providing contracts as their formal counterpart. Thus, a contract is a rigorous and analyzable description of a use case diagram while a use case diagram is a diagrammatic representation of a contract. We have also shown that using our approach, the functional and the object-oriented views of the system can be integrated and analyzed in a uniform way, as the goals of the agents and the preconditions of the use cases are both described in terms of objects of the problem domain. Finally and most importantly, we have shown how to analyze use case diagrams. In particular we check whether the system can provide the functionality described in the use cases.

The formal approach we proposed is especially helpful in the development of systems where correctness is a central concern. A contract is an intermediate document between informal specifications, such as the use case diagram, design artifacts, such as the class diagram, and the final program. The novelty of our approach is the fact that a contract allows both an analysis of the future implementation and an analysis of the ways in which that program will be used by potential users (the agents). Moreover, the fact that contracts *look like* programs makes them easier to use by programmers, while the preciseness of contracts allows rigorous analysis of their properties.

# Chapter 4

# Statecharts

## 4.1 Introduction

A statechart describes the behavior of a model element as changes in its internal state as the result of receiving external stimuli. Statecharts were introduced by David Harel in 1984 [42] and they were quickly adopted as a visual notation to design concurrent and reactive systems. Statecharts are based on the concept of the finite state machine, but the notation includes many new features like hierarchical and concurrent states to model complex behaviors concisely.

UML includes a variant of statecharts as one of its behavioral diagrams. In the UML terminology, a statechart is a graphical representation of a state machine. A UML state machine can be used to describe the behavior of a classifier (a class) or a behavioral feature, for example, an operation of a class. A statechart associated to a classifier defines the whole life of any of its instances, from their creation until their destruction. A statechart associated to an operation only describes the execution of that operation.

The UML notation for statecharts and semantics for state machines are described in [77]. The semantics in this context is slightly misleading: the UML semantics is given in natural language and UML itself. Clearly it is not possible to analyze a UML model containing statecharts without a more concrete and detailed semantics.

In this chapter we discuss the formalization of UML state machines and we present a precise semantics of statecharts that is compatible with the UML metamodel. Although we develop this semantic interpretation for our verification tool, we believe that such a description is needed as a reference model for implementing tools for code generation, simulation and analysis of UML Statecharts.

Our formalization consists of two parts: the formalization of the structure of the state machines and the formalization of its semantics. There are two

important reasons for presenting the semantics in this way. First, many model elements composing a statechart are syntactic sugar and can be explained in terms of simpler elements. Also, it is possible to define different semantics for different application domains based on the same structure for statecharts. In this chapter we present a semantics of state machines for discrete systems while in Chapter 6 we present an extension of the semantics suitable for real-time and hybrid systems.

Another important issue when defining the semantics of UML statecharts is how to define actions and guards. In contrast with Harel's statecharts, UML does not define a standard language for the guards and actions. In this chapter, we just assume that such languages exist and their semantics are defined independently of the semantics of the statecharts. We require that the language of guards is free of side effects but it can be a subset of the language of actions. Again, we consider that different application domains may require different action languages so their definitions should be independent of the definitions of the statecharts.

We proceed as follows. Section 4.2 shows the formalization of the structure of state machines while Section 4.3 discusses their dynamics. Section 4.4 discusses the collaboration of objects defined using statecharts. Conclusions and related work are presented in Section 4.5.

## 4.2 Structure of State Machines

A UML Statechart diagram represents a hierarchical state machine. A hierarchical state machine includes composite states that can contain other states, creating a tree structure. If a child state is active, its parent state is also active. Composite states can also contain orthogonal regions. All orthogonal regions directly below a composite state are active whenever the container state is active. This allows us to model concurrent behaviors where several states are active simultaneously. We can use hierarchical states and orthogonal regions to simplify the representation of a large state machine into a simpler statechart using less visual elements.

### State configuration

Let us assume that we are given a state machine $\mathsf{SM}$. The set of states $\Sigma$ of the state machine can be partitioned into four disjoint sets: *composite states* ($\Sigma_{cs}$), *simple states* ($\Sigma_{ss}$), *final states* ($\Sigma_{fs}$), and *history states* ($\Sigma_{his}$), e.g. $\Sigma = \Sigma_{cs} \cup \Sigma_{ss} \cup \Sigma_{fs} \cup \Sigma_{his}$. Some of the these states can be marked as initial. We call the set of initial states $\Sigma_{is}$ and it is subset of $\Sigma$.

Final and history states do not usually have names in the graphical representation but we shall use a default naming scheme $ps_1, \ldots, ps_n$ where $n =$

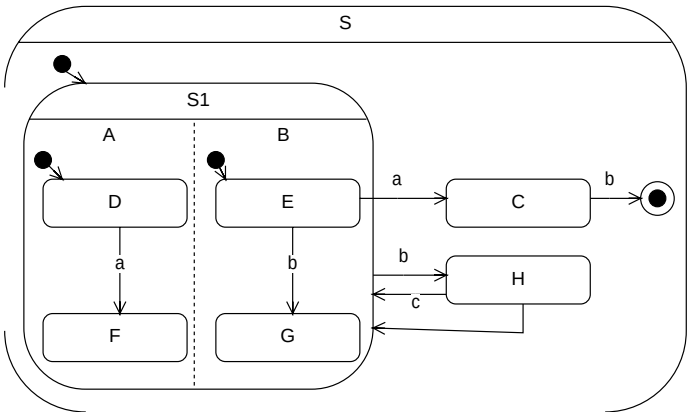



Figure 4.1: Example Statechart

$|\Sigma_{fs} \cup \Sigma_{his}|$.

A composite state contains other states and orthogonal regions. We assume that there is a composite state that contains all other states of the machine. We call this state the top state.

Each state in a state machine may have an *entry*, *exit*, and *activity* action associated with it. We shall assume that these are defined in some language $\mathcal{L}_a$. The only requirement is that the language of actions $\mathcal{L}_a$ must include a 'skip' action. These concepts will be discussed more in detail in the context of the operational semantics below.

The state machine represented in Figure 4.1 will be our running example. The sets are: $\Sigma_{ss} = \{C, D, E, F, G, H\}$, $\Sigma_{cs} = \{S, S_1, A, B\}$, $\Sigma_{fs} = \{ps_1\}$, and $\Sigma_{his} = \emptyset$ while $\Sigma_{is} = \{S, S_1, D, E\}$. The top state of the machine is $S$.

### State configuration

The current state of a UML state machine is defined as a tree of states starting with the single top state at the root down to individual simple states at the leaves. Such a state is called a *state configuration*. We will encode state configurations as terms over a signature, where we will interpret the sets $\Sigma_{cs}$, $\Sigma_{ss}$ and $\Sigma_{fs}$ as operator symbols. The *arity* : $\Sigma\mathbb{N}$ of a composite state is defined as the number of orthogonal regions contained in it. The arity of a state that is not a composite state is 0. Thus in our example in Figure 4.1 we have:

| $s$ | S | $S_1$ | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|---|
| $arity(s)$ | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Let $X$ be a countable set of variables s.th. $X \cap \Sigma = \emptyset$. Then given the set of symbols $\Sigma$, we can form the set of *linear terms* $T_\Sigma(X)$ as the smallest set

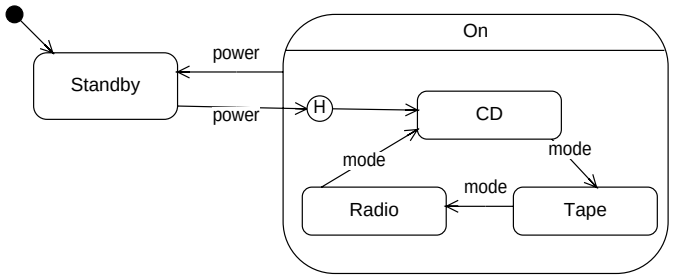



Figure 4.2: Using History States

satisfying:

1. $\Sigma_{ss} \cup \Sigma_{fs} \cup X \subseteq T_\Sigma(X)$
2. $op(t_1, \ldots, t_n) \in T_\Sigma(X)$, where $op \in \Sigma_{cs}$, $n = arity(op)$ and $t_1, \ldots, t_n \in T_\Sigma(X)$
3. $\forall t \in T_\Sigma(X), \forall x \in Var(t), \#(x, t) = 1$

The last condition restricts the set of terms to those where each variable occurs only once. We can now define a *state configuration* of a state machine SM as a term in $T_\Sigma$. $T_\Sigma(\emptyset)$ is called the set of *linear ground terms* and simply denoted $T_\Sigma$. From now on, we shall use 'term' as a shorthand for 'linear term'.

We will need to reference the set of symbols $symb(s)$ of a term $s \in T_\Sigma$.

$$\begin{aligned} symb(s) &= \{s\} \text{ iff } s \in T_{\Sigma_{ss}} \cup T_{\Sigma_{fs}} \\ symb(s(s_1, \ldots, s_n)) &= \{s\} \cup_{i=1..n} symb(s_i) \end{aligned}$$

History states allow us to return to the previous configuration of a composite state. History states have two special attributes. The *deep* attribute indicates whether the state is a deep or a shallow history state. Deep history states remember the active configuration of a composite state and all its nested substates while shallow history states remember only the states directly below the composite state. The $defaultHistory$ stores the transition to be taken in the event that there is no previous history information in the history state.

Figure 4.2 shows a statechart containing a history state. The statechart describes the operation modes of a home stereo. There are three modes: CD player, tape and radio. Initially, the stereo is switched off and it can be turned on by pressing the power button. In this case, the stereo will return to the previous mode thanks to the history state. If there is no information about the previous state the default transition will select the CD state.

History states are not absolutely necessary. We can obtain the same behavior by introducing new simple states that act as history information. For example, Figure 4.3 shows an equivalent statechart for the home stereo system

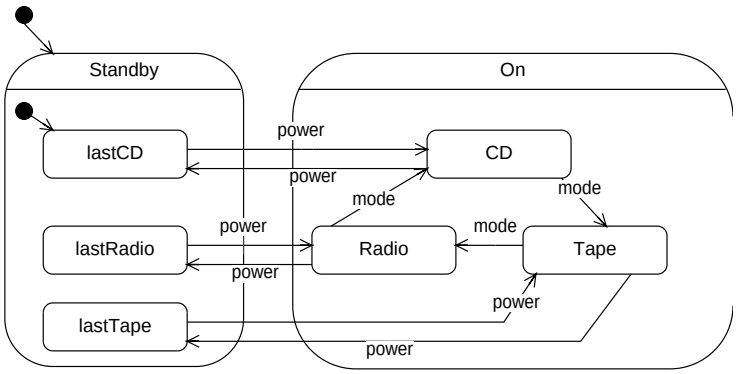



Figure 4.3: Replacing History States

that does not use history states. However, this statechart contains more states and transitions and is probably more difficult to understand. We can find the same pattern in other features of a statechart. For example, it is possible to remove an orthogonal state by performing the crossproduct of all its regions. Again, this can cause an explosion in the number of model elements of the statechart and affect the legibility of the model.

### Transitions

A transition in a UML state machine consists of the following elements:

source: the originating state vertex,

target: the target state vertex,

trigger: the event that fires the transition,

guard: a predicate that must be true in order to fire the transition. It is defined using the language of predicates $\mathcal{L}_p$, and

effect: an optional action to be performed when the transition fires. It is defined using the language of actions $\mathcal{L}_a$.

State machines communicate by sending events to each other. We shall assume a global set of events $E$. The set of trigger events $E_t \subseteq E$ of the state machine $\mathsf{SM}$ is the union of all trigger events appearing on the transitions of the state machine. The UML standard also allows triggerless transitions. Such transitions, called *completion transitions* have an implicit trigger, the *completion event*, which is generated when all transition and entry actions and activities in the currently active state are completed. Completion events are

$$
\begin{aligned}
t_0 = S(X) &\xrightarrow{\ll create \gg} S(S_1(A(D), B(E))) \\
t_1 = S(S_1(A(D), B(X))) &\xrightarrow{a} S(S_1(A(F), B(X))) \\
t_2 = S(S_1(A(X), B(E))) &\xrightarrow{b} S(S_1(A(X), B(G))) \\
t_3 = S(S_1(A(X), B(E))) &\xrightarrow{a} S(C) \\
t_4 = S(S_1(X, Y)) &\xrightarrow{b} S(H) \\
t_5 = S(H) &\xrightarrow{c} S(S_1(A(D), B(E))) \\
t_6 = S(H) &\xrightarrow{H_c} S(S_1(A(D), B(E))) \\
t_7 = S(C) &\xrightarrow{b} S(ps_1)
\end{aligned}
$$

Figure 4.4: Translation of the State Machine in Figure 4.1.

special because they have priority over all other events, and they will be implemented in our semantics by assigning a special completion trigger to such transitions. The set of completion trigger events is $E_c$, and the set of events of a state machine $E_{SM} = E_t \cup E_c$. Figure 4.1 shows a transition without trigger emanating from state $H$ that will be labelled $H_c$ in our example. We will discuss the generation of completion events in more detail below. From now on, we shall assume that all completion transitions have been labelled with the corresponding completion event.

We define a transition as follows: A *transition* is a tuple $(s, s', e, g, a)$, where $s, s' \in T_\Sigma(X)$ are the source and target states, $e \in E_t$ is the event trigger, $g \in \mathcal{L}_p$ is the guard and $a \in \mathcal{L}_a$ is an action to execute when firing the transition.

The set of transitions of a state machine is denoted by $\Phi$. Assume that $t \in \Phi, t = (s, s', e, g, a)$, we can define the following projections $source(t) = s$, $target(t) = s'$, $trigger(t) = e$, $guard(t) = g$, $effect(t) = a$. We will use the following notation for transitions:

$$t = s \xrightarrow{e[g]/a} s',$$

We usually do not represent the guard if it is true or the action if it is skip.

Figure 4.4 shows the encoding of the transitions in our example. The transition $t_0$ is associated with the initialization of the state machine and will be explained later. The completion transition $t_6$ has the completion event $H_c$ as its trigger, to mark the fact that it will be fired when the activities in state $H$ have been finished. Our notation for states makes the hierarchical relationship of states explicit. The fact that $t_1$ lives in an orthogonal component is made explicit through the fact that the variable $X$ appears on both the left-hand-side

and the right-hand-side in the same position $B(X)$. This means that firing $t_1$ does not change the state of the $B$ component in the orthogonal region $S_1$.

**Definition of State Machine**

We can now summarize the different elements that compose a state machine:

Assume that we are given a language for describing actions $\mathcal{L}_a$, a language for describing predicates $\mathcal{L}_p$ and a set of events $E$. A UML state machine is a tuple

$$\begin{array}{ll}( & \Sigma, \mathit{entryAction}, \mathit{exitAction}, \mathit{activity}, \mathit{defaultHistory}, \\ & \mathit{deep}, \Phi, \mathit{source}, \mathit{target}, \mathit{trigger}, \mathit{guard}, \mathit{effect} \qquad\qquad )\end{array}$$

where

$\Sigma$ is the set of states of the state machine.

*entryAction* the entry action associated with each state, is a function $\Sigma \to \mathcal{L}_a$.

*exitAction* the exit action, is a function $\Sigma \to \mathcal{L}_a$,

*activity* the activity, is a function $\Sigma \to \mathcal{L}_a$,

*deferred* the set of deferred events $\Sigma \to \mathcal{P}(E_t)$,

*defaultHistory* is the default transition of a history state $\Sigma_{his} \to T_\Sigma$,

*deep* if the state is a deep history $\Sigma_{his} \to \mathsf{Bool}$,

$\Phi$ is the set of transitions $\Phi \subseteq T_\Sigma \times T_\Sigma \times E_{SM} \times \mathcal{L}_p \times \mathcal{L}_a$,

*source* is the left projection $\Phi \to T_\Sigma$,

*target* is the right projection $\Phi \to T_\Sigma$,

*trigger* is the trigger event $\Phi \to E_t$,

*guard* is a guard of the transition $\Phi \to \mathcal{L}_p$ (the default guard is $\mathsf{true}$), and

*effect* is an action or side-effect of the transition, a function $\Phi \to \mathcal{L}_a$.

The UML standard defines other *pseudostates*, e.g. initial, join vertices, fork vertices, and junctions. These are vertices in the statechart that typically are used to connect multiple transitions into more complex state transition paths. These are extensions to the UML state machine syntax, because the primitive constructs are not rich enough to express the desired behavior. However, it turns out that our formalization allows us to directly express most of these constructs without any extensions. So we will not give explicit translation of

these features but instead just indicate how they can be directly expressed in our formalization.

**Initial states:** We treat initial states by requiring that transitions that go to the boundary of composite states are translated in a way that they explicitly reference the initial state. As an example, see transitions $t_5$ and $t_6$ in Figure 4.4.

Initial states are also used during the initialization of a state machine. In our translation, we assume that the top state is always active and we add a special transition called the initial transition. An initial transition goes from the top state of the machine to its initial configuration. According to the UML standard, the trigger of an initial transition is the $\ll create \gg$ call event if the context of a statechart is a classifier. If the statechart describers a behavioral feature (an operation), the initial transition is triggered by the invocation of the feature. Figure 4.4 shows the creation transition of our example as transition $t_0$.

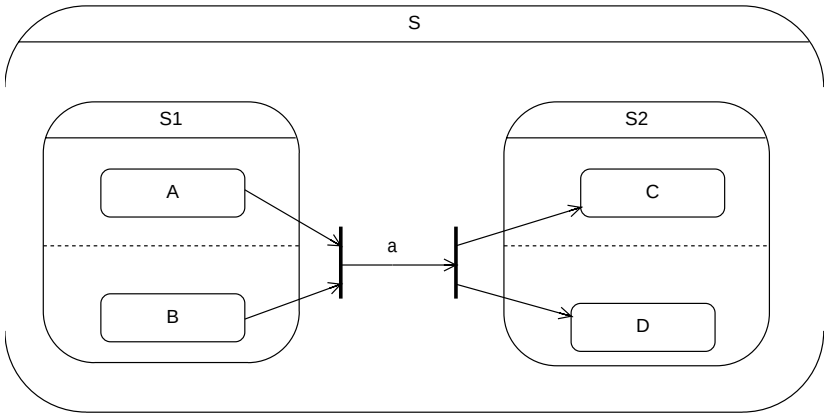
**Join and fork vertices:** A join vertex is used to merge several transitions emanating from the source vertices in different orthogonal regions causing the execution to synchronize, while fork vertices serve to split an incoming transition into two or more transitions terminating in orthogonal target vertices. Join and fork vertices can be translated by making the source and target vertices explicit in the translation. An example is given in Figure 4.5.

**Junction vertices:** These are a generalization of join and fork vertices, where any number of transitions may enter or exit the vertex. The transitions may also contain guards. We will not deal with junction vertices explicitly, since they can always be translated into simpler constructs by logically combining the optional guards on the transitions (cf. Figure 4.6).

## 4.3 Dynamics of State Machines

In this section we attempt to formalize the informal description of the execution semantics of statecharts given in section 2.12 of [77]. We assume that a statechart is used to describe the behavior of a classifier. The execution semantics of a UML state machine is defined in terms of a hypothetical machine whose key components are:

- an *event queue* that holds incoming event instances until they are dispatched,
- an *event dispatch mechanism* for selecting and de-queueing event instances from the event queue, and
- an *event processor* that processes dispatched events.




$$S(S_1(A, B)) \xrightarrow{a} S(S_2(C, D))$$

Figure 4.5: The Translation of Join and Fork Vertices

Each object will execute an algorithm, the *run-to-completion* (RTC) step, that dispatches and executes events on its event queue until the top state generates a completion event and the state machine finishes. The standard does not define the semantics of the input queue, however, it stated that some kinds of events, i.e. completion events have priority. We shall implement the event queue as an abstract data type with four operations: *enqueue*(*element*), *priorityEnqueue*(*element*), *dequeue*, *priorityInsert*(*queue*). Priority elements are always dequeued first from the queue. We denote an empty queue as $\emptyset$.

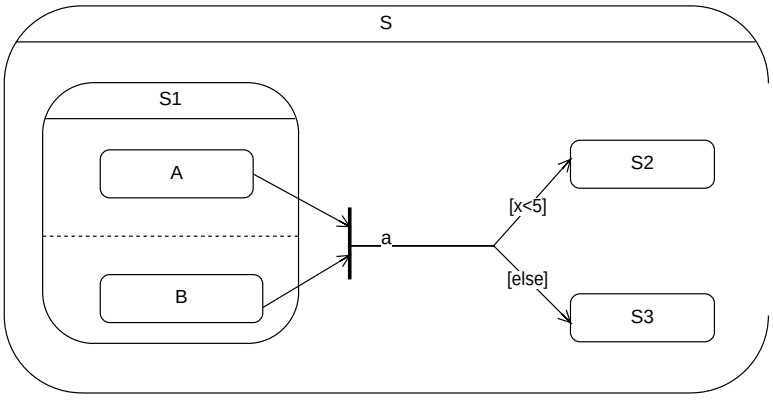
The main task of the event processor is to find out which transitions are *enabled*, e.g. can be fired based on the first event in the queue. The set of transitions is then grouped into a step. The transitions in the step are executed in sequence, but the event processor will only consider the next event in the queue after all transitions in the step have been fired.

### 4.3.1 Calculation of the Step

First, we need to determine the transitions whose source term matches the current state configuration. We call these transitions *active*:

$$active(s) \triangleq \{t \in \Phi : \exists \rho : X \to T_\Sigma : \rho(source(t)) = s\}$$

The substitution $\rho$ is called the *mode* of the transition $t$. We use an auxiliary function $mode(t, s)$ to denote it. A pair $(t, \rho)$ is called a *transition instance* of $t$.




$$t_1 = S(S_1(A, B)) \xrightarrow{a[x<5]} S(S_2)$$
$$t_2 = S(S_1(A, B)) \xrightarrow{a[x\geq 5]} S(S_3)$$

Figure 4.6: Translation of Junction Vertices

For example, lets assume that the active configuration in the statechart shown in Figure 4.1 is $s = S(S_1(A(D), B(E)))$. In this case, transition $t_1$ is active since it exists a substitution $\{X \mapsto E\}$ so that the source state of the transition $S(S_1(A(D), B(X)))$ equals the active state configuration $s = S(S_1(A(D), B(E)))$. In the same example, $mode(t_3, S(S_1(A(D), B(E)))) = \{X \mapsto D, Y \mapsto E\}$. We use these definitions to determine conflicts between transitions in the following section.

The last event dequeued is called the *current event*. Given a current event $e$ we need to find those transitions $t$ in $active(s)$ whose trigger matches $t$ and whose guard evaluates to true.

A transition is enabled if the current event matches its trigger and the guard holds:

$$enabled(t, e) \stackrel{\triangle}{=} e = trigger(t) \wedge guard(t))$$

The set of transitions enabled by the current event $e$ is denoted by the function $enabled(s, e)$. If the current event is clear from the context we write $enabled(s)$.

**Conflicts**

Two transitions in a UML state machine that are enabled at state $s$ are in *conflict* iff the intersection of the set of states that their corresponding transition instances exit is non-empty. For example, in Figure 4.1 transitions $t_1$ and $t_3$ are in conflict at state $S(S_1(A(D), B(E)))$, because both transition instance $(t_3, \{X \mapsto D, Y \mapsto E\})$ and transition instance $(t_1, \{X \mapsto E\})$ in state $S(S_1(A(D), B(E)))$ exit the simple state $D$. To define the conflict relation we need to calculate the set of states that is exited by firing a transition instance.

Assume that transition $t = (s_1, s_2, e, g, a)$ is enabled at state $s$ with mode $\sigma$. Then the set of states that are exited (and the set of states that are entered) in this firing mode is given by:

$$\begin{aligned}
exit(t, \sigma) &\stackrel{\wedge}{=} symb(\sigma(s_1)) - symb(\sigma(s_2)) \\
entry(t, \sigma) &\stackrel{\wedge}{=} symb(\sigma(s_2)) \cup \\
&\qquad \{s \in targetH(\sigma(s_2)) : history(s)\} - symb(\sigma(s_1)) \\
targetH(s) &\stackrel{\wedge}{=} \{s\} \text{ iff } s \in T_{\Sigma_{his}} \\
&\qquad \emptyset \text{ otherwise} \\
targetH(s(s_1, ..., s_n)) &\stackrel{\wedge}{=} \cup_{i=1..n} targetH(s_i)
\end{aligned}$$

The conflict operation between enabled transitions is defined as:

$$conflict(t, t', s) \stackrel{\wedge}{=} exit(t, mode(t, s)) \cap exit(t', mode(t', s))$$

Two transitions are in conflict iff $conflict(t, t', s) \neq \emptyset$.

In Figure 4.1, at $S(S_1(A(D), B(E)))$, transitions $t_1$ and $t_3$ are in conflict, because

$$conflict(t_1, t_3, S(S_1(A(D), B(E)))) = \{D\}$$

as are transitions $t_2$ and $t_3$, because

$$conflict(t_2, t_3, S(S_1(A(D), B(E)))) = \{E\}$$

.

Given two transitions that are in conflict at a state, we can sometimes resolve the conflict with the help of a priority relation. The UML standard defines the priority of a transition based on its source state. If $t_1$ is a transition with $source(t_1) = s_1$ and $t_2$ a transition with $source(t_2) = s_2$, then:

- if $s_1$ is a direct or transitively nested substate of $s_2$, then $t_1$ has higher priority than $t_2$,
- if $s_1$ and $s_2$ are not in the same state configuration, then there is no priority difference between them.

The intuition is to give priority to transitions that start "lower" in the hierarchy. For example, in Figure 4.1 the conflict between $t_2$ and $t_4$ can be resolved in favor of $t_2$ because $t_4$ starts at the border of $S_1$ which is clearly higher up in the hierarchy than $E$. On the other hand, the conflict between $t_1$ and $t_3$ cannot be resolved, because they start in states that are not substates of each other.

The relation "direct or transitively nested substate" can be translated into a notion of covering of terms:

$$cover(a, b) \triangleq \exists \rho : X \mapsto T_\Sigma(X) : \rho(a) = b,$$

i.e. $a$ covers $b$ iff we can find a substitution $\rho$ that allows us to transform $a$ into $b$. For example, $S(S_1(X, Y))$, the source state of $t_4$, covers state $S(S_1(A(X), B(E)))$, the source of $t_2$, by substituting $\{X \mapsto A(X), Y \mapsto B(E)\}$ so we can transform $S(S_1(X, Y))$ into $S(S_1(A(X), B(E)))$. On the other hand if we look at the conflict between $t_1$ and $t_3$ we see that $S(S_1(A(D), B(X)))$ covers $S(S_1(A(X), B(E)))$ and the term $S(S_1(A(X), B(E)))$ covers the term $S(S_1(A(D), B(X)))$, e.g. the conflict cannot be resolved. Thus the relation "has-priority-over", denoted by $\succ$ is given by:

$$a \succ b \triangleq cover(b, a) \wedge \neg cover(a, b).$$

Finally, we define the static priority relation on transitions $t = (s_1, s_2, e, g, a)$, and $t' = (s'_1, s'_2, e', g', a')$ as:

$$t \succ t' \triangleq s_1 \succ s'_1$$

In our running example, when the current state is $S(S_1(A(D), B(E)))$ and the current event is $b$, then transitions $t_2$ and $t_3$ are enabled and in conflict. But since $t_3$ covers $t_2$, we will only fire $t_2$. On the other hand since neither $t_1 \succ t_3$ nor $t_3 \succ t_1$, we cannot resolve the conflict between $t_1$ and $t_3$.

We can now define the rule for calculating a step $\Gamma$ of transitions to fire from a state $s$ assuming a given current event $e$. Given a state $s$ the set $conflict(s) \subseteq enabled(s)$ is the set of transition instances that are in conflict at $s$. The set of transitions can be ordered by the relation $\succ$ to resolve some of the conflicts by priority. We denote the set of conflicting transition instances that are less priority by $conflict_\succ(s)$. More formally:

$$\begin{aligned} conflict(s) &\triangleq \{t \in \Phi : \exists t' \in \Phi : conflict(t, t', s)\} \\ conflict_\succ(s) &\triangleq \{t \in conflict(s) : \exists t' \in conflict(s) : t' \succ t\} \end{aligned}$$

The set of fireable transition instances is obtained by deleting all those transitions that cannot fire because of the priority relation:

$$\mathit{fireable}(s) \stackrel{\wedge}{=} \mathit{enabled}(s) - \mathit{conflict}_{\succ}(s)$$

A step $\Gamma$ is a maximal conflict-free set of transitions $\Gamma \subseteq \mathit{fireable}(s)$:

$$\forall t, t' \in \Gamma : \mathit{conflict}(t, t', s) = \emptyset \wedge t \in \mathit{fireable}(s) \\ \wedge (\exists t' \in \Gamma : \mathit{conflict}(t, t', s) = \emptyset) \implies t \in \Gamma$$

### 4.3.2 The Run-To-Completion Step

The idea of the run-to-completion step is to ensure that an event can only be dequeued and dispatched after the previous current event has been fully completed. The next configuration of a state machine given a step is computed as the sequential composition of the *nextstate* function, which is defined as

$$\mathit{nextstate}(t, s) = \mathit{mode}(t, \mathit{target}(t))(s)$$

Given the above mathematical tools we can now proceed to give a first version of the algorithm that implements the run-to-completion step as defined in the UML semantics. This algorithm is shown in Figure 4.7. In this version we wait for an event to arrive in the queue, calculate the enabled transitions, and execute a step. If no transitions are enabled then the step will be empty, which implements the "implicit consumption" semantics as mandated by the UML semantics.

However, the UML semantics defines a number of notions that we have overlooked in the previous discussion. These include the different actions associated with a state: entry actions, exit actions and activities; different events: completion events, deferred events, call events and time events; internal transitions and history states. The full RTC algorithm is given in Figure 4.8. The RTC algorithm is a while loop that is executed until the top state of the state machine is completed. We shall discuss the extensions individually.

**State actions, activities and completion events:** Whenever a state is entered, the *entry action* is executed *prior* to any other action (l. 28). Analogously, when the state is exited, the *exit action* is executed as the final step before leaving the state (l. 21). In the case of nested states the entry actions are executed outside-in, e.g. the entry action of the outermost state is executed first. The exit actions are executed in the opposite order, e.g. inside-out.

A state may also include an *activity*. This is an action that is forked as a concurrent task just after the entry action has been completed (l. 29). The

```
SimpleRTC()
 1  while true
 2  do
 3     if queue ≠ ∅
 4        then currentevent ← queue.dequeue()
 5             enabled ← enabled(currentstate, currentevent)
 6             Select a step Γ based on Def. 4.3.1
 7             newstate ← currentstate
 8             for  each γ ∈ Γ
 9             do
10                newstate = nextstate(γ, newstate)
11                execute effect(γ)
12             currentstate ← newstate
```

Figure 4.7: First Version of the Run-to-Completion Step

activity continues executing while the object remains in the same state. We assume that the last action executed by an activity is the generation of a completion event. This event has priority and it should be added to the input queue using the *priorityEnqueue* operation.

If a state associated with an activity is exited before the activity ends the activity is aborted before the exit action is executed (l. 20). In this case, no completion event is generated.

Completion events can also be generated when all subregions of a composite state reach a final state. We formalize this condition using the function $finished$:

$$finished(cs) \quad \triangleq \quad \forall s \in symb(cs) : s \in \Sigma_{fs}$$

After each execution of a step, we check if any subregions have reached a final state and we push the corresponding completion event in front of the queue. Finally, we start the RTC loop again until the top state is finished (l. 1).

**Deferred events:** Usually an event that does not enable any transition is discarded. However, sometimes it is useful to keep this event waiting until the next state. Such an event is called a *deferred event*. We can define a set of deferrable events for each individual state of a machine. The set of deferrable events of a state configuration is the union of the deferrable events of the active states. We modify the transition selection mechanism so it does not necessarily discard the current event. If the current event does not enable any transitions,

```
RTC()
 1  while ¬finished(top)
 2  do
 3      currentevent ← queue.dequeue()
 4      enabled ← enabled(currentstate, currentevent)
 5      if enabled = ∅ and currentevent ∈ deferred(currentstate)
 6        then deferredevents.enqueue(currentevent)
 7      Select a step Γ based on Def. 4.3.1
 8      newstate ← currentstate
 9      for  each γ ∈ Γ
10      do
11          for  each s ∈ exit(γ)  in "outside-in" order
12          do
13              if s is a composite state
14                then for  each h history state in s
15                       do if currentevent  is not a completion event
16                             then history(h) ← configuration(s, deep(h))
17                             else  history(h) ← defaultHistory(h)
18          for  each s ∈ exit(γ)  in "inside-out" order
19          do
20              abort activity(s)
21              execute exitAction(s)
22          newstate = nextstate(γ, newstate)
23          execute effect(γ)
24          if effect(γ) is a method call
25            then wait for return of corresponding call event
26          for  each s ∈ entry(γ)  in "outside-in order
27          do
28              execute entryAction(s)
29              start activity(s)
30      if currentevent is a call event
31        then send return event
32      currentstate ← newstate
33      if any subregion have reached a final state
34        then push the corresponding completion event in front of the queue
35      if enabled ≠ ∅
36        then queue.priorityInsert(deferredevents)
37             deferredevents ← ∅
38      if enabled = ∅ and currentevent is a call event and this is a protocol statechart
39        then send return event
```

Figure 4.8: The Final Version of the Run-To-Completion Step

but belongs to the deferrable events of the current state configuration (l. 5), it is pushed onto a special list (l. 6), and the main loop is run through once. At the end of the loop (l. 36-37), we insert all the deferred event back into the input queue. But we can only do this when we have actually fired an event, e.g. changed the object state. Otherwise we would go into an infinite loop, because the deferred event which does not enable any events will be put in the input queue and dispatch it again and again. We can check this condition by testing if the set *enabled* is nonempty (l. 35).

**Internal transitions:** Each state may also have a number of internal transitions, i.e. transitions where their source and target is the same state. According to the definitions of the *entry* and *exit* functions, firing an internal transition does not cause exit and entry actions to be executed, nor does it cause the state activity to be restarted.

**History states:** The history information of a history state is stored in an attribute named *history*. It is updated every time that the composite state containing the history state is exited. If it is exited using a completion transition, then the history information should be reset to the default history (l. 17). Otherwise, we store the current configuration of the state in the history attribute (l. 16).

The definition of *entry* has a special case to handle history states, so a transition leading to a history state also enters the states of the history.

**Call events:** The final touch to the RTC step is the treatment of call events (lines 24-25 and lines 30-31). Call events are special events that represent synchronous invocation of a specific operation. A call event is generated by a method call action. Although the standard does not make this explicit, we have assumed that each call event has a corresponding return event. On the level of a single state machine, when a method call is executed as an action, the whole transition step is only completed when the invoked objects complete their own run-to-completion steps. Thus at line 25 the caller waits for each of the method calls to end. In line 31 the callee has finished its own RTC step, and must now send the corresponding return events.

The UML semantics define the implicit consumption of events if they cannot fire a transition. However, in the UML semantics, there is a special rule concerning call events in a statechart describing a protocol. A *protocol statechart* describes in which states is acceptable to invoke an operation of a classifier, but it does not describe the outcome of the operations. In this case, the event is consumed implicitly, but the caller can continue its operation. We implement this behavior in our RTC algorithm in the last two lines of the RTC loop.

### 4.3.3 Other model elements

There are two model elements that are not directly supported by the semantics presented in this chapter: synch states and choice points.

A synch state is a state that has incoming arrows from one region and outgoing arrows into *another* orthogonal region. The synch state itself belongs to the least common ancestor of the source and target regions. Each synch state has an associated attribute named *bound*. This attribute is defined in the UML semantics as the maximal difference between the number of times the incoming and outgoing transitions of the synch state can fire. It is a positive integer or $\infty$, for an unlimited bound.

In [68] we present synch states as elementary model elements and include them in the RTC algorithm. However, this approach presents two problems. First, it clutters the exposition explaining a feature that it is seldom used. Second, unbound synch states have, by definition, an infinite state space and they cannot be verified using a model checker.

In this chapter, we prefer to consider a synch not as an special state but just a simple state with special entry and exit actions. We define for each synch state $s$ two attributes *count* and *bound*. The entry action associated with the state $s$ is $count_s := min(count_s + 1, bound_s)$. The outgoing transitions from a synch state are guarded by this condition: $[count_s \leq 0]$ and the exit action of the state is $count_s := count_s - 1$. The exact definition of these actions depends on the action language.

Another model element that is not supported directly is the choice point. A choice point introduces an intermediate edge in a transition so several actions and decisions can be taken while firing a transition. The formalization of choice points would require the introduction of macro and micro steps in the RTC algorithm. It also complicates the implementation of the algorithm since we do not know which states we exit or enter when firing a transition until we have finish firing it.

## 4.4 Interaction diagrams

In the previous sections, we have studied the semantics of a single statechart. However, in non-trivial designs we are interested in studying the interaction between many statecharts, or more precisely, the interaction of many model elements whose behavior is described by a statechart. To simplify our exposition, we assume that each statechart describes the behavior of an object.

We assume that each object in a interaction diagram runs on its own process. Each *process* is an instance of the run-to-completion step, and it is scheduled asynchronously. Objects communicate by sending events through links. Events are always directed to a single and specific object: there are no broadcast events

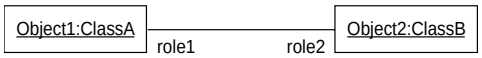



Figure 4.9: Collaboration of Objects

or events sent to the environment. A link can introduce arbitrary delays when transmitting an event, but it cannot lose events.

The main design decision for the semantics of interaction diagrams, is what to consider to be an atomic step of the RTC algorithm. Ideally, each step of the RTC algorithm should be atomic, but call events require synchronization between two processes. We have chosen to split the RTC algorithm into three atomic blocks. The first block contains lines 4-22 and it deals with the selection of transitions and exiting of current states. The second block contains lines 23-26 and deals with the execution of the action. The third block contains lines 26-32 and deals with executing entry actions.

### 4.4.1 Particular Comments on the Semantics

We would like to point out a few problematic situations that may arise due to the way the UML standard has decided to interpret orthogonality.

Our first example concerns the situation in Figure 4.10. We have two objects Object1 and Object2 that are linked as shown in Figure 4.9. Assume that Object1 somehow receives an event $a$. This enables both transitions in the orthogonal region and the corresponding effects are executed. Because the order of execution of an orthogonal region in the RTC step is not defined but is sequential, the object Object2 is able to distinguish between different implementations of Behavior ClassA. In the example, if the order of execution is such that Behavior ClassA fires the left transition first, machine Behavior ClassB will end up in the state, Result1 while in the other case it will end up in the state Result2. This problem may arise if one uses orthogonality as a synonym for concurrency.

The important distinction between orthogonality and concurrency is that for a true concurrent system we must be able to observe both interleavings, while for orthogonality it is enough to observe only one. Interestingly this is one of the places where UML state machines differ substantially from Harel's statecharts. In Harel's statecharts all events generated by a step are available to the receiver at the same time, and the problem does not occur. This may cause confusion to users who are used to Harel's Statecharts, because the model may contain implicit dependencies on the order of evaluation, that are extremely difficult to find by simulation. Fortunately, it is possible to detect this situation using a verification tool.

Our second example concerns call events. If there is a circular chain of

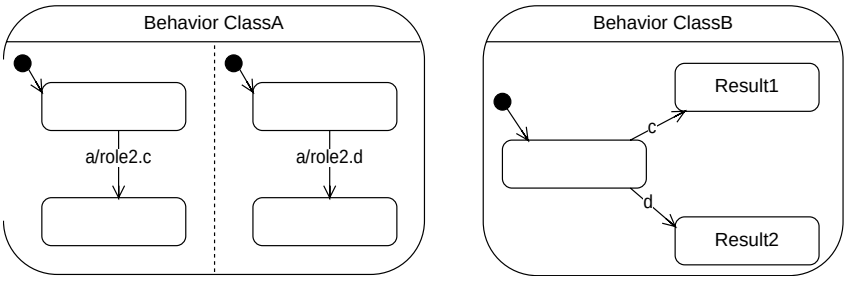



Figure 4.10: Machine $B$ is able to Observe the Execution Order of $A$.

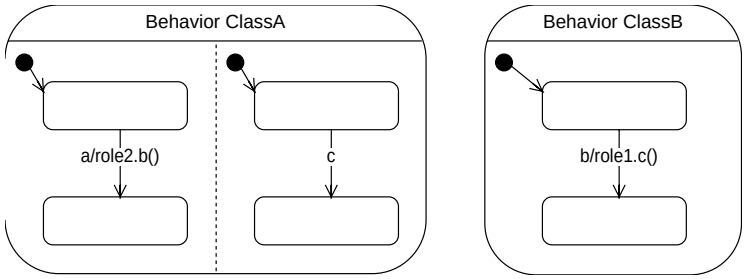



Figure 4.11: Machine $A$ will Deadlock

call events from an object back to itself, the object may block. The trivial case of this situation is when an object wants to execute a call action on itself. To illustrate the more complex situation, consider the example in Figure 4.11. Assume that object Object1 receives an event $a$. This triggers the sending of a call event $b$ to Object2, which sends the call event $c$ back to $A$. Now, even though the event $c$ could be acted upon in the second orthogonal region, the semantics of UML forbids this, and thus we have a *deadlock*. A solution to this problem would be to interpret the RTC step recursively and instantiate concurrent instances of the RTC loop for each orthogonal region. Then the call to $c()$ in figure 4.11 could finish and the system would not block.

Finally, we would like to point out that since actions contain side-effects and are allowed to change all the attributes of an object, orthogonal regions may not necessarily be independent since they can interact through local attributes.

## 4.5 Conclusions and Related work

In this chapter we have proposed a formalization of the structure and behavior of UML state machines. This formalization attempts to clarify some of the ambiguities in the definition of statecharts in the UML specification. It can

be used as the foundation for the construction of tools for code generation, animation and verification of UML statecharts.

UML statecharts are based on classical statecharts as presented by Harel, but they include many additions from ROOMcharts, an object-oriented variant of statecharts defined in the ROOM modeling language [91]. It is important to realize that UML statecharts have substantially different dynamics than Harel's statecharts: In UML, events are queued and dispatched one by one. Events can carry parameters. There are no negated events: a transition cannot be fired because an event is not being dispatched. There is no zero-time assumption, an event can wait in the queue an undefined amount of time until it is processed. Finally, transitions in different orthogonal regions are fired sequentially. These changes allow us to use statecharts to describe the behavior of objects rather than systems and facilitate the implementation of statechart designs in software.

Several semantics for statecharts have been proposed in the literature, for example the STATEMATE semantics proposed by Harel [44], compositional semantics by Uselton and Smolka[96], Pnueli and Shalev [81],and the approaches focused on verification by Levi [65],Mikk et al. [75] and Latella et al. [64]. Most of these are concerned with defining the semantics of Harel's statecharts as presented in the STATEMATE tool. From our point of view, the work of [64] is interesting to compare with ours. There the authors discuss a formalization of UML state machines that takes hierarchical automata [74] as its basis. However, the authors ignore many of the particular features of UML state machines.

In the following section we present vUML, a tool for model checking UML models based on this formalization of statecharts. The vUML tool is based on the SPIN model checker [49]. Sara used this formalization in his thesis on verification of UML State Machines [88]. He implemented a verification component that implements this semantics using the ACF framework [67].

# Chapter 5

# Verification of Collaborations of Statecharts

## 5.1 Introduction

Statecharts is an expressive and concise notation for describing the behavior of reactive objects. In the previous chapter we have studied the structure and dynamics of statecharts and presented a formalization of their behavior as a run-to-completion algorithm. We can now apply this knowledge in the construction of methods and tools for analyzing the behavior of objects and collaborations of objects defined using statecharts.

A statechart defines the behavior of a single object (or a single behavioral feature of an object). In UML we can model complex systems as a collaboration of objects. The behavior of each object is then defined independently, using a different UML diagram. A collaboration can also model concurrent systems containing active objects. An active object has its own thread of control and it evolves independently of other objects. Some tools, like Rhapsody from I-Logix, can automatically generate a complete implementation of collaborations of objects defined using statecharts in the C++ and Java programming languages.

UML models based on statecharts are highly descriptive and easy to understand, but UML models should still be verified since they may exhibit behaviors not expected by the designer. We propose to use a model checker to verify exhaustively collaborations of objects whose behavior is described using statecharts. A model checker is a tool that automatically searches the complete set of states of a model for a given incorrect behavior. Model checking (cf. [23] for a survey) has been successfully applied in large industrial verification projects. By verifying a UML model, we can ensure that the software is free of design errors before implementing it.

In this chapter we present vUML, our first realization of such a tool. The target user of the tool is the designer of object-oriented software systems. The main goal of vUML is to be as automatic and easy to use to as possible. We try to achieve this by hiding the inner workings of the tool from the designer.

Internally the tool uses the SPIN model checker by Holzmann [49] to perform the verification. However, the designer does not have to know how to use SPIN or the PROMELA [48] language. In model checking it is customary to use some kind of temporal logic to describe the properties to verify. In vUML we take an alternative approach. The vUML tool introduces new stereotypes and constraints to describe the behaviors to be satisfied by the model. The designer uses these stereotypes to add some extra information to the design so the tool can extract the properties that must be verified. These new stereotypes are not part of the UML standard, but they have been introduced to allow the designer to describe safety and liveness requirements without having to learn temporal logics.

Figure 5.1 shows a normal session with vUML. After creating the model, the designer invokes vUML to verify it. The tool automatically converts the UML model into a PROMELA specification and then invokes SPIN. The verification is performed by SPIN without interaction from the designer. If the verification fails, it generates a counter-example that shows how to reproduce the error in the model. vUML converts the counter-example into a new UML sequence diagram. The designer can study this diagram, fix the error, and verify the model again. Once the model is correct, the designer can try to introduce new correctness requirements into the model and verify them, until he is confident enough with the model.

We proceed as follows. Section 5.2 shows how to model concurrent systems in UML and introduces the new stereotypes used to model correctness requirements. In Section 5.3 we explain how the vUML tool works and we demonstrate it with a case study in Section 5.4. We discuss the conclusions and related work of this chapter in Section 5.5.

## 5.2 Modeling Concurrent Systems with Statecharts

In this section, we show how to model concurrent and distributed object-oriented systems using UML. The basic UML element for modeling concurrent systems is the active object. An active object maintains its own thread of control and runs concurrently with other active objects.

As an example of a simple concurrent model, we will discuss the standard example of the dining philosophers. A class diagram for the philosophers and forks is given in Figure 5.2. The philosophers eat and think, as usual, but they must pick up two forks, left and right, before starting to eat. The Philosopher

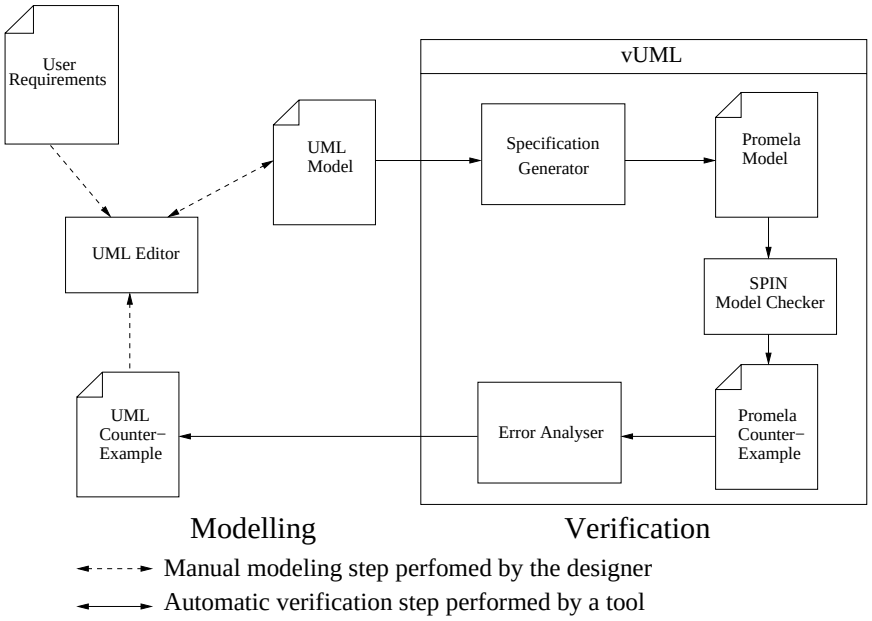



Figure 5.1: Using vUML

class is marked as active, so all the objects instantiated from this class will run concurrently.

In our example, we use a statechart diagram to describe the behavior of the instances of the fork as shown in Figure 5.3. We abstract the behavior of a fork object into two different states: Available, when the fork is over the table, and Taken, when the fork is in the hand of a philosopher. The fork reacts to two call events: get and release. Call events are synchronous, i.e. the caller is blocked until they are consumed by the receiver.

The behavior of a fork object is as follows: If a fork dispatches a get event while it is in the Available state, it switches to the Taken state. The fork will

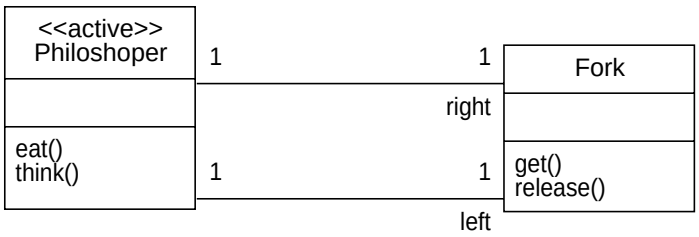



Figure 5.2: Class Diagram for the Dining Philosophers

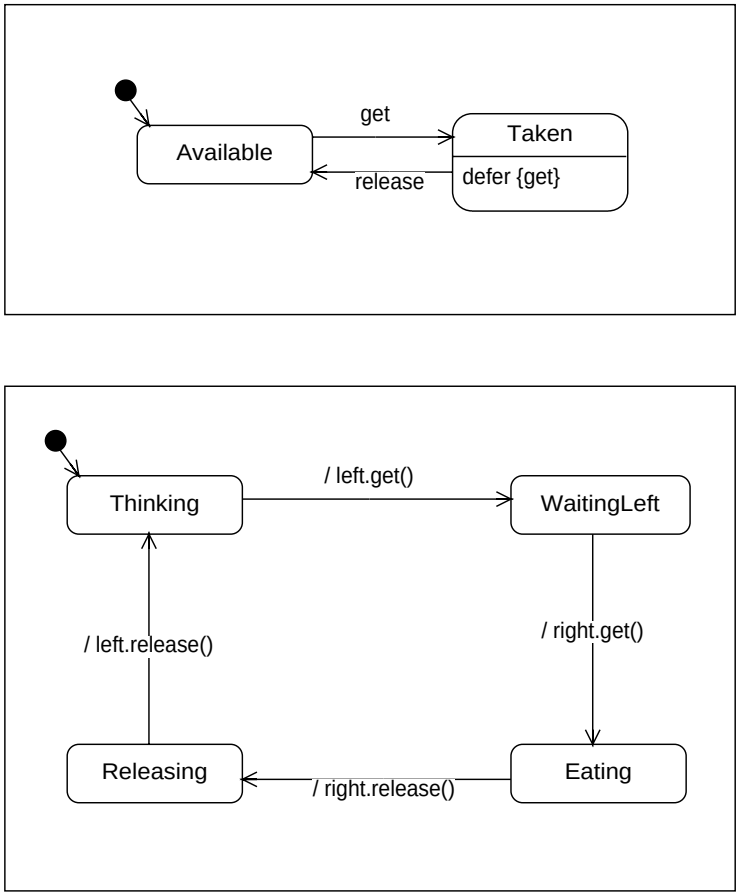



Figure 5.3: Behavior of the Forks (top) and Philosophers (bottom)

switch back to the Available state when it dispatches a release event. However, if a get event is received while the fork is taken, the event will be deferred. It will be placed in a special queue and it will be dispatched again when the fork becomes available. The diagram at the right of Figure 5.3 shows the behavior of the philosopher class. A philosopher is initially in the Thinking state. When it has finished thinking, a completion transition fires and the philosopher attempts to get its left fork. A completion transition is a transition without trigger and it can be fired when the activity associated with its source state is completed. Section 4.3 contains a more detailed description of call events, deferred events and completion transitions.

### 5.2.1 Describing Properties of a Model

The goal of the vUML tool is to be as automatic and transparent to the designer as possible. However, the designer must add some extra information into the design so the tool can extract the temporal properties to verify. The vUML tool uses two new stereotypes to define behavioral constraints to be satisfied by the model: the ≪progress≫ and ≪invalid≫ stereotypes. These stereotypes are not part of the UML standard, but they may help the designer to introduce safety and liveness requirements into the model without using a complex formula in a temporal logic.

The meaning of these stereotypes is as follows: If the state $s$ is mark as invalid it is interpreted as the linear temporal logic (LTL) [80] formula: $\Box\neg s$. The square or $\Box$ symbol is read as "always" and the previous formula can be read as "always not in $s$", i.e. "never in $s$". The ≪progress≫ stereotype is interpreted as $\Box(\Diamond s \wedge \Diamond\neg s)$. The diamond or $\Diamond$ is read as "eventually". The formula can be read as: always the object will eventually be in the state $s$ and eventually in a different state.

In our example, we can consider whether there is an error in the model if a philosopher tries to release a fork that is not taken. We can model this condition by adding a transition to an invalid state as shown in the leftmost diagram in Figure 5.4. We can introduce a liveness requirement, e.g. a philosopher should eat infinitely many times, by marking the Eating state as a ≪progress≫ state.

It is also possible to define a constraint over a class or over an association of two classes. A constraint over a class is usually called a class invariant and it restricts the behavior of all instances of the class. A constraint over an association restricts the combined behavior of the objects that are linked together.

Constraints are predicates over the attributes and states of the instances of the class. A constraint has the following syntax:

```
constraint = (constraint)
```

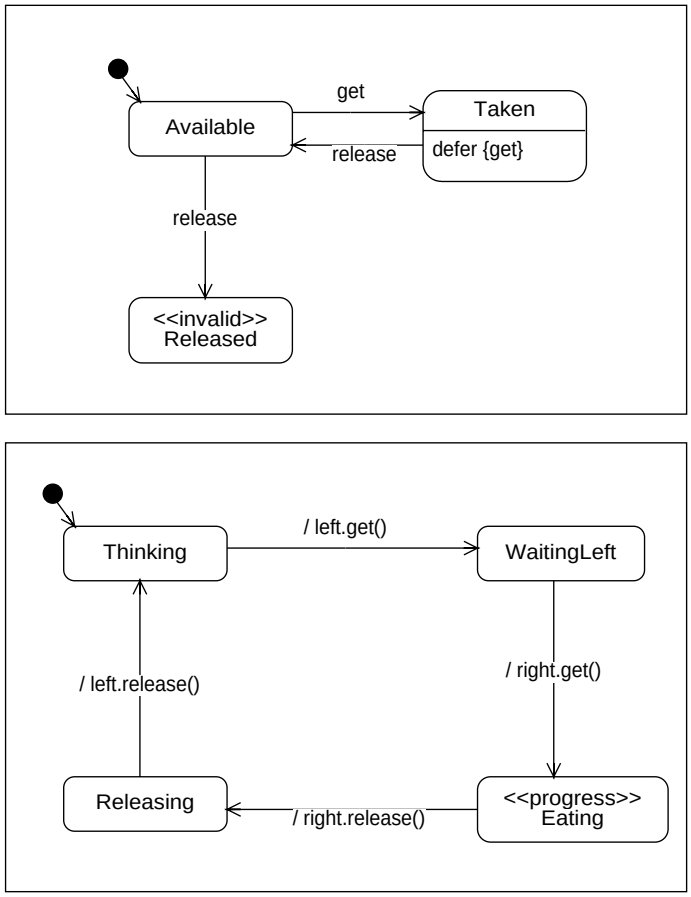



Figure 5.4: Temporal Requirements Embedded in the Statecharts

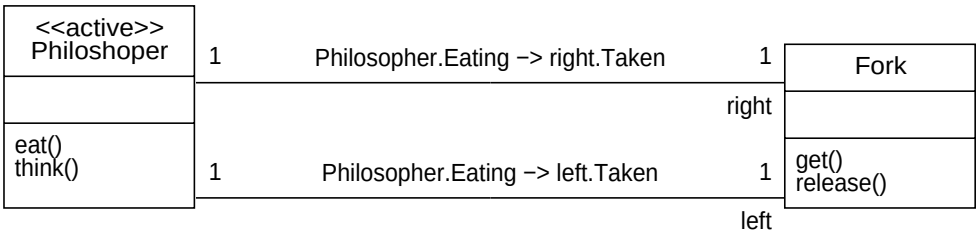



Figure 5.5: Class Diagram for the Dining Philosophers with Constraints

```
           | constraint && constraint
           | constraint || constraint
           | constraint -> constraint
           | ! constraint
           | role.state
           | expression
expression = (expression)
           | expression binaryopetator expression
           | unaryoperatior expression
           | role.attribute
           | number
bynaryoperator= + | - | * | / | % | < | > | <= | >= | ==
unaryoperator = - | !
```

A constraint of the form *role.state* is true when the state *state* is active in the object associated to the role *role*. An expression of the form *role.attribute* is evaluated as the current value of the attribute in the object associated to the role *role*. A constraint $c$ is interpreted as the LTL formula $\Box c$ during the verification.

In the example, we can constrain the philosopher and fork classes by requiring that whenever a philosopher is in the state Eating, its left and right fork must be Taken. These constraints are shown in Figure 5.5 over the associations between the classes. These implications are not trivially true since the Eating state is a progress state and it will eventually be active.

### 5.2.2 Collaboration of objects

A UML Statechart describes the behavior of all the objects created from a given class, but it does not itself create the objects. We use a collaboration diagram to create and link the objects in a model. Objects in a collaboration diagram interact by exchanging messages. A message is displayed over a link between its source and target object. We can use sequence numbers to order the messages chronologically.

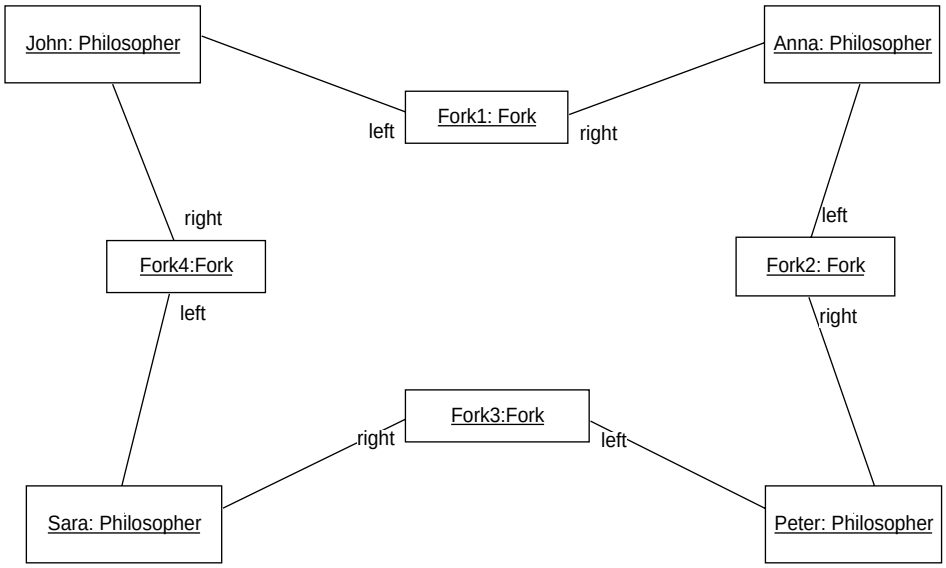



Figure 5.6: A Dinner for Four

Figure 5.6 shows a collaboration diagram where four philosophers are sharing four forks. All the objects are in their initial state, and no messages have been exchanged.

The objects of a collaboration diagram can reside in the same processor or in different nodes of a distributed system. This does not modify the interpretation of the collaboration diagram, since the semantics does not make any assumption concerning the speed of the links and the processors. However, it is not always easy to determine what messages will be exchanged and in which order when active objects appear in a collaboration diagram.

A collaboration diagram can show one or several behaviors of a system, but it does not exclude other possible behaviors, perhaps not expected by the designer. In our example, if the philosopher class is not designed carefully, and we were intentionally careless, the objects in the collaboration diagram can easily arrive at a deadlock, where none of the objects can progress any further. Figure 5.7 shows a collaboration diagram derived from Figure 5.6 where each philosopher holds the left fork and is waiting for the right fork to become available. Yet, according to the behavior described in the state machines, this will never happen, so the philosophers have arrived at a deadlock.

In the following section we describe how the vUML tool uses SPIN to capture these kinds of errors in a model.

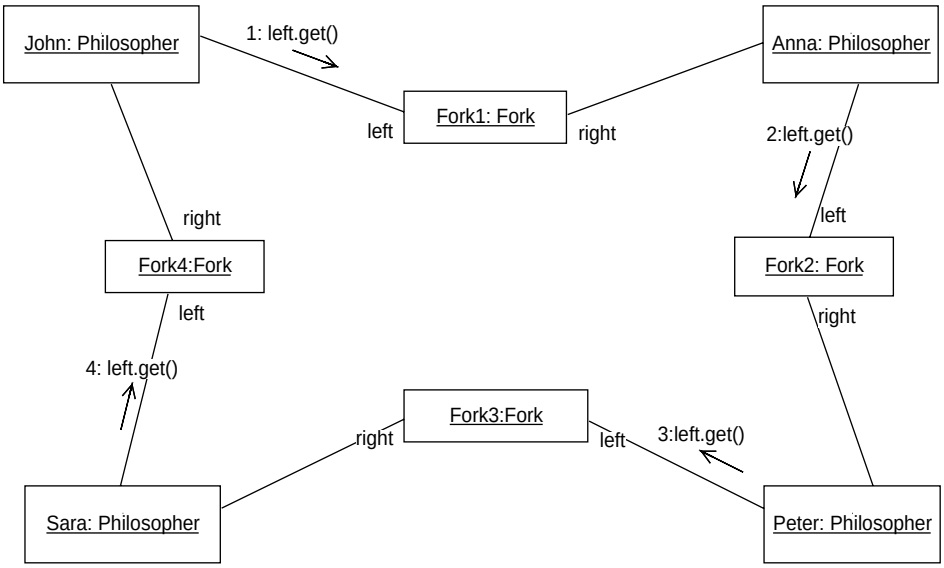



Figure 5.7: After Sara's Move, the Dinner is Suddenly Over

## 5.3 The vUML Tool

Basically vUML is a model generator and a front-end for the SPIN model checker. The tool is based on the semantic formalization presented in the previous chapter. However, vUML and SPIN impose some restrictions on the analyzable UML models.

In the vUML tool we use PROMELA as the language for actions, guards and constraints in the UML models. Activities are not supported, although the tool can generated completion events associated with *abstract* activities.

Each class may define a set of attributes although the type of the attributes is restricted to integer. This restriction is imposed by SPIN. The designer should use attributes sparingly since they increase the memory needed during the verification.

The tool supports signals, call events, deferred events and completion events. Events may have parameters, but the type of the parameters is again restricted to integer. When dispatching an event, the actual parameters are copied into local variables of the process, so parameters share the same properties as the attributes of an object. In order to reduce the size of the verification problem, the tool assumes that all the communication queues have a fixed size and the event dispatching policy is FIFO.

Time events are not supported. Time events are constraints on the implementation of the model. Since we do not have any information about the

performance of the final system, we can not estimate when a time event will occur. A simple timeout transition can be emulated by a completion transition. In this case, the abstract activity of the state acts as a timer. When the activity is completed the timeout transition fires.

### 5.3.1 Open Models

Some models need external stimuli to proceed with their task. External events can be generated by another class or actor, not present in the collaboration diagram. A model can also abstract from the activity carried out in a state. In this case, the model does not generate the completion event associated to the activity. Abstract states may be refined further into a composite state if the designer deems it necessary.

*Open models* that react to messages sent by external model elements, cannot be directly verified with a model checker. However, vUML can convert an open model into a closed one, where no external event instances are needed, if the external events do not carry parameters. vUML determines whether the source of an event is not defined within the model and creates an event generator for those events. Since the model does not contain any information about the generation of these events, the event generator is nondeterministic. vUML automatically discards events directed to a missing link.

As an example, the philosopher state machine from Fig. 5.3 has four completion transitions, but the activities associated with these completion transitions are not described. The model is open since it does not specify when the completion events are generated.

The event generator is *demonic*, it generates the events trying to reveal an error in the model. The two key characteristics of the event generator are: it eventually produces an event and it will not send an event to an object if the event will not trigger a transition. The first constraint ensures that the model will never deadlock because the event generator would have decided not to send an event to an object. The second constraint has been introduced to speed up the verification. As defined in the semantics, the state of an object does not change when an object receives an unexpected event. By not sending events that will not fire a transition we can reduce the verification problem and avoid a possible livelock situation where the event generator floods an object with unexpected events.

The event generator cannot handle external events carrying parameters. An event generator for those events could create an *state-space explosion* [98] during the verification. Models with this kind of event cannot be verified directly with vUML.

### 5.3.2 Structure of the PROMELA Model

In this section we explain the structure of the PROMELA models generated by the vUML tool. We assume that the reader is familiar with the PROMELA language. PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation. It includes basic data types and the input and output operations from Hoare's CSP language. The basic syntax of PROMELA is similar to the syntax of the C programming language.

vUML maps each UML class into a PROMELA process-type. An instance of a process-type is a process, so each UML object in the collaboration diagram is thus mapped into a PROMELA process. PROMELA processes run concurrently and are scheduled nondeterministically. We can achieve the semantics of collaboration diagrams described in section 4.4 by using the atomic statement that ensures that a given PROMELA block is executed atomically.

The PROMELA models generated by the vUML tools include many print statements describing the behavior of the model. Print statements are ignored by SPIN while performing a verification. However, they are interpreted during a simulation, i.e., while reconstructing a counter-example. In this way, the vUML can trace the activity of a model and generate counter-examples without knowing the internals of SPIN.

Each process has many variables to store the active configuration of the statechart, the attributes of the object and the internal variables used to calculate the RTC step. Since PROMELA is not an object-oriented language, we must store most of these variables in global arrays. The arrays are indexed by a unique *object id* integer that is passed as a parameter to each PROMELA process. It could be possible to store the state of each object as local variables of a process. However, the encapsulation mechanism of PROMELA would prevent us from implementing constraints over several objects.

We define for each orthogonal region or composite state a different variable for storing the state configuration. States are encoded in binary and the size of the state variables is a byte. This arrangement works best with statecharts with many states in few state containers (composite states or orthogonal regions). For statecharts with less than 8 states per container it would be better to use bit fields to store the state configuration.

In the current implementation, constraints are translated into PROMELA and share the syntax and semantics of PROMELA expressions. However, a constraint can refer to a state of the statechart or an attribute of the object by its UML name. In this case, the tool silently translates the UML name into the corresponding internal representation in the PROMELA model.

In our example of the dinning philosophers, we define an array of 4 bytes to store the active configuration of the four fork objects. The name of the state variables is created by the concatenation of the class name, the container name

and the suffix `st`. The actual PROMELA code is as follows:

```
#define MAX_INST_FORK 4
byte Fork_Top_st[MAX_INST_Fork];
```

We also define constants to enumerate the different events, transitions and state names in order to improve the readability of the generated code. In our example, we define these constants:

```
#define EVENT_get 1
#define EVENT_release 2
/* Constants for class Fork */
#define Fork_TRANS_tGet 1
#define Fork_TRANS_tRelease 2
#define Fork_STATE_Available 1
#define Fork_STATE_Taken 2
```

We define a variable for each orthogonal region to store the transition that is selected for firing. Although, we can fire many transitions in the same step we can only fire one transition per orthogonal region. Again, we enumerate the transitions and encode them in binary. We consider the top state to be a region. The selection variables use the string `fire` as a suffix:

```
byte Fork_Top_fire[MAX_INST_Fork]; /* selection of top region */
```

Each process also has a variable to store the current input event and its parameters and the attributes of the represented object. There are other variables used by the external event generator and the deferred event queue, but they are explained below.

The first parameter of a process-type is the object id used to index the arrays containing the state of the object. The other parameters are the channel used to receive input events and a channel used as a synchronization mechanism in call events.

```
proctype Fork(byte id ; chan input ; chan myCB  ) {
  xr input;        /* PROMELA optimization               */
  /* local variables                                     */
  byte event=0;    /* current event                      */
  chan cb=0;       /* parameters of current event        */
  bit end=9;       /* end of RTC                         */
  bit deferIt=0;   /* We must defer the current event    */
  deferredQueue(DEFERQSIZE); /* Queue for deferred events*/
```

The first action of each PROMELA process is to execute the initial transition. The initial transitions of all the objects are executed sequentially but in any arbitrary order. However, each process waits until all UML objects are initialized before proceeding with the run to completion step. In the fork object the initial transition enters the Available states.

```
atomic {
 /* zero transition selection variables */
 Fork_Top_fire[id] = NOFIRE;
 /* enter Top*/
   /* enter Available*/
   Fork_Top_st[id] = Fork_STATE_Available;
   printf("vUML: STATE Fork %d  Available \n",id);
 waitForOthers=waitForOthers-1;
}

/* Wait until all the objects have been constructed */
waitForOthers==0;
```

The proper run to completion step is divided into four parts: Obtaining the current event, selecting the transitions to fire, firing the transitions, and dealing with special cases such as deferred events and completion transitions.

The mechanism to obtain the current event varies depending upon whether the statechart reacts to external events or not. If the statechart does not react to any external event, then we obtain the current event just by reading it from the input queue. If the statechart contains transitions fired by external events then there is a set of flag variables indicating whether we can fire a transition by an event from the queue or by another external event. In PROMELA, we create a non-deterministic if statement like this:

```
if
 :: fromQueue  -> input?event, parameters ;
 :: generateE1 -> event=E1;
 :: generateE2 -> event=E2;
 ...
 :: generateEn -> event=En;
fi
```

Where `E1..En` are external events, `fromQueue` is a flag indicating that there is at least one transition that can be fired by an internal event, and the flag `generateEi` indicates that there is a least one transition that can be fired by the external event `Ei`. These flags must be updated accordingly after each step that fires a transition.

The selection mechanism is based on the definitions of the previous chapter. The vUML resolves the static priorities between transitions before generating the code. As an example, the following PROMELA code implements the selection of the two transitions of the Fork statechart

```
/* Selection transition tGet */
if
  :: ( event == EVENT_get &&
       Fork_Top_st[id] == Fork_STATE_Available) ->
        Fork_Top_fire[id] = Fork_TRANS_tGet
  :: else
fi;
/* Selection transition tRelease */
if
  :: ( event == EVENT_release &&
       Fork_Top_st[id] == Fork_STATE_Taken) ->
        Fork_Top_fire[id] = Fork_TRANS_tRelease
  :: else
fi;
```

Once transitions are selected, they can be fired sequentially. If the effect of a transition is to send a call event, then the process is blocked until the receiver consumes the event. The objects are synchronized using a rendez-vous channel that is passed as a parameter of the call event and copied into a variable named `cb`.

The following code implements the firing of the two transitions of the Fork. Each action includes two print statements to indicate that the call event has been dispatched by the fork object `id` and that the state machine changes to a new state.

```
/* Fire transition tGet */
if
  :: Fork_Top_fire[id] == Fork_TRANS_tGet ->
    printf("vUML: RC Fork %d  %d get \n",id,cb);
    cb!1; /* return call event */
    /* enter Taken*/
    Fork_Top_st[id] = Fork_STATE_Taken;
    printf("vUML: STATE Fork %d  Taken \n",id);
  :: else
fi;
/* Fire transition tRelease */
if
```

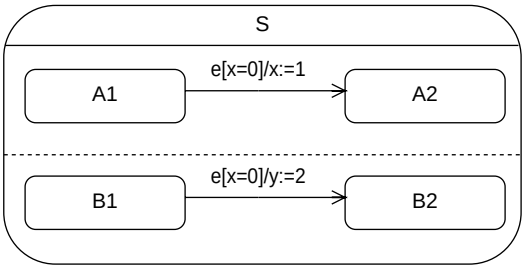



Figure 5.8: Example of Why it is Necessary to Separate Selection from Firing

```
    :: Fork_Top_fire[id] == Fork_TRANS_tRelease ->
      printf("vUML: RC Fork %d  %d release \n",id,cb);
      cb!1; /* return call event */
      /* enter Available*/
      Fork_Top_st[id] = Fork_STATE_Available;
      printf("vUML: STATE Fork %d  Available \n",id);
    :: else
  fi;
```

We separate the selection of transitions from their firing process since the actions associated with the transitions, entered and exited states could affect the guards of the enabled transitions. As an example, consider the statechart shown in Figure 5.8. If both states A1 and B1 are active, x equals 0 and event e is present then both transitions should be fired in the same step according to the UML semantics. However, the first transitions modifies the value of x so, although the guard of the second transition holds when the event is dispatched, it becomes false during the middle of the step. At the end of the step, states A2 and B2 should be active, x should be equal to 1 and y to 2, independently of the order of firing the transitions. There are two strategies to implement this behavior: One option is to split the step in two phases as we have shown before. An alternative is to perform the selection and firing in one step but using temporary variables to store the attributes modified by the actions. In the example, the transitions would not modify x but a temporary copy of x, named x'. At the end of the step all the temporary variables are copied into the actual attributes.

If we split the selection from the actual firing we need to use an extra variable to store the selection for each orthogonal region. If we use the temporary variable method then we need the double amount of space to store the attributes, plus a linear time increment to update the attributes. Both alternatives increase the size of the state vector of the PROMELA model and the optimal approach varies from model to model.

If we cannot fire a transition by the current event we should check whether we must defer it. In the PROMELA model, we know that we have not selected any transition if all the selection variables are equal to the constant `NOFIRE`. In this case we should check whether one of the current states defers the current event. If all these checks are positive, we place the deferred event into a PROMELA channel that is used as a list.

When we fire one or more transitions in the current step and the channel containing deferred events is not empty, we can put back its contents in the input channel of the process, so they will be processed in the next iteration of the RTC loop.

### 5.3.3 Verification with SPIN

After generating the PROMELA model, vUML invokes the SPIN model checker to perform the verification.

There are two mechanisms to introduce correctness requirements in SPIN: never claims, that must hold at all times during the verification, and process assertions, that are only checked when a process executes an assert statement. vUML does not implement the correctness requirements of the UML models as SPIN never claims but instead as process assertions. vUML requirements are temporal propositions over the states of the UML statecharts and not over the PROMELA model implementing the RTC step of a statechart. This distinction is important in constraints involving several objects. In this case, the constraint may not hold while one of the objects is executing its run-to-completion step, but it must always hold when the objects are in a stable state.

SPIN is used to detected the following error types in the UML model: deadlocks, reaching an invalid state, constraint violations, sending an event to a terminated object, queue overruns and livelocks.

**Deadlocks:** A deadlock is produced when it is not possible to dispatch any event in any object. This error can be detected automatically by the SPIN model checker in any PROMELA model.

**Reaching an invalid state:** The verification will fail if a state machine reaches an invalid state. We implement this condition in PROMELA using a false assertion as the entry action of the state.

**Constraint violations:** A constraint is a predicate over the attributes and active configurations of a set of objects. A constraint is implemented in PROMELA as an assertion that is evaluated after each run-to-completion step of any of the two objects involved.

**Sending an event to a terminated object:** It is an error to send events to a terminated object. We can catch this error by adding the following code:

```
end_CLASSNAME:
```

```
input ? _ ;
printf("vUML: RTERMINATED %d",id);
assert(0)
```

A PROMELA process jumps to this section of code when the top state ends. The first statement blocks the process until any event is received by the input queue of the object. In this case, the assert statement will be evaluated and the verification will fail.

**Input queue overrun:** Each object has a finite input queue. Sending an event to a full queue produces an error. Some models need bigger queue sizes than the default provided by the tool. The size of the queues can be defined as a parameter when invoking vUML. Nevertheless, before increasing the size of the queues, we should consider first whether this error is concealing a livelock. Queue overruns are detected automatically by SPIN in any PROMELA model.

**Defer event queue overrun:** Each object has a finite input queue to store deferred events. If the queue is full, the verification fails. This error occurs e.g. when a machine is blocked in a state where it defers events.

**Livelocks:** An object contains a livelock if none of the states marked with the ≪progress≫ stereotype are visited infinitely often during a normal execution of the model. A check for this error is implemented using the progress label in SPIN.

The tool can check for all these kinds of errors simultaneously, except for livelocks. When it is looking for livelocks, it does not report other kinds of errors.

### 5.3.4 Representation of Counter-Examples

The verification of a model fails if SPIN finds one of the previous errors in the model. In this case, SPIN generates an error trace showing how to reproduce the error. The error trace contains all the output generated by the print statements in the model. vUML analyzes all the lines that contain the prefix vUML: in order to reconstruct the activity of the statechart and represent the counter example as a UML sequence diagram. Although the tool uses a collaboration diagram as the starting point of the verification, it uses a sequence diagram for representing the counter-example since they explicitly show the time dimension. Thus, sequence diagrams are easier to understand than collaboration diagrams when a large number of messages are exchanged.

XSPIN, a graphical front-end for SPIN, also generates sequence diagrams from error traces. Unfortunately, SPIN sequence diagrams reflect the activity of the PROMELA model containing both the UML model and the run-to-completion step. Therefore, counter-examples generated by XSPIN are larger and more difficult to understand than the ones generated by vUML.

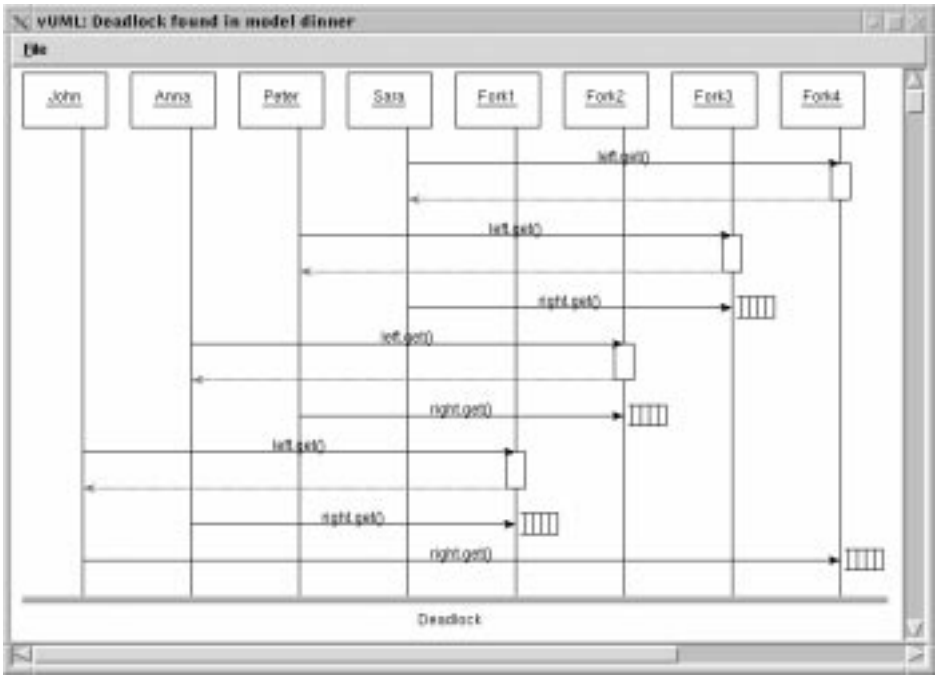



Figure 5.9: vUML Window Showing a Counter-Example

Figure 5.9 shows an example of an error found by vUML. The tool has been used to verify the collaboration diagram of the dining philosophers example and it has found the deadlock situation described before. The diagram shows how the requests for getting the right forks cannot be granted and are queued for late processing. At the end of the diagram, the four philosopher objects are blocked. In this example there are actually an infinite number of sequence diagrams that lead to this deadlock. The one shown in the figure is one of the shortest, generated by using a special command line when invoking the tool.

We use two new icons for showing in a sequence diagram when an event is deferred and dispatched again. These icons are not defined in the UML standard. Although deferring an event is an internal operation of the state machine, it is necessary to know when it happens in order to understand some sequence diagrams. Figure 5.10 shows the new icons and their relative position in the sequence diagram.

## 5.4 Example: The Production Cell

In this section we describe a larger example of how to model and verify a concurrent system with vUML. The studied system is the Production Cell [66],

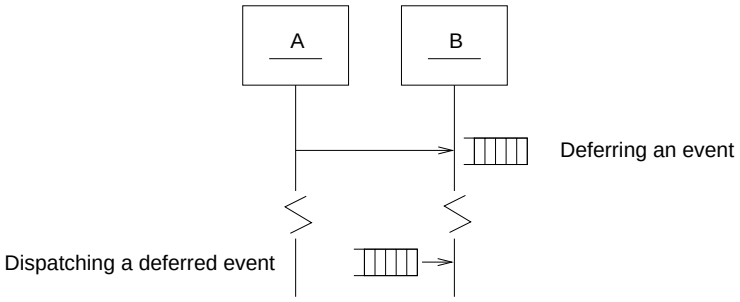



Figure 5.10: Icons Showing When an Event is Deferred and Dispatched Again

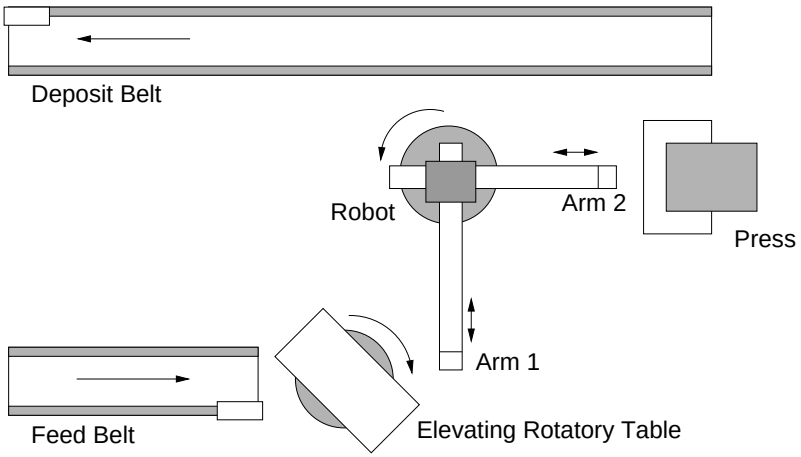



Figure 5.11: Top View of the Production Cell

a standard example for evaluating methodologies for designing embedded systems.

A top view of the production cell is given in Figure 5.11. The model includes several machines that must be coordinated in order to forge metal blanks. The machines should avoid dropping the blanks to the floor or colliding against each other. There is a conveyor belt that feeds the blanks into a rotary table. The table rotates and lifts the blanks so they can be picked up by a robot. The robot has two arms, one for feeding the press with blanks and another to extract the forged blanks from the press. The robot can use an arm to place a forged blank into the deposit belt while the other arm carries a new blank to the press.

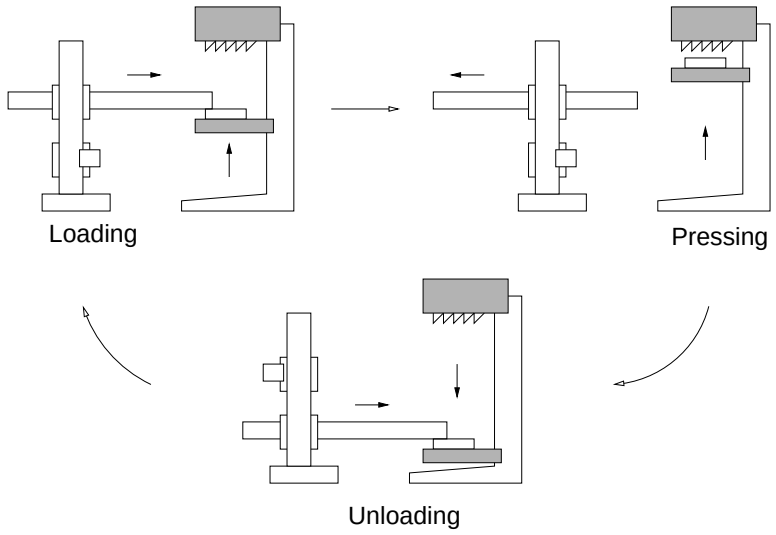



Figure 5.12: Robot and Press

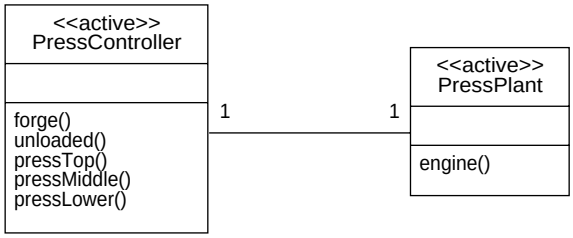



Figure 5.13: Class Diagram of the Press

### 5.4.1 Modeling the Press

We decided to model first the central element of the system: the press. The task of the press is to forge metal blanks. The mobile part of the press is initially in the middle position until the robot places a blank into the press. After the press is loaded, the press moves upwards until the blank is forged and then it moves downwards to the unloading position. The robot gets the forged piece and the cycle starts again. Figure 5.12 represents these steps graphically.

We decided to split the model of the press in two parts: the plant and the controller. These elements are represented using two classes: PressPlant and PressController. Figure 5.13 is the class diagram of the Press. A factory with many similar presses can be modeled as many object instances of these classes. The instances of the PressPlant and PressController are active objects.

Figure 5.14 shows the statechart diagram for the controller of the press. The

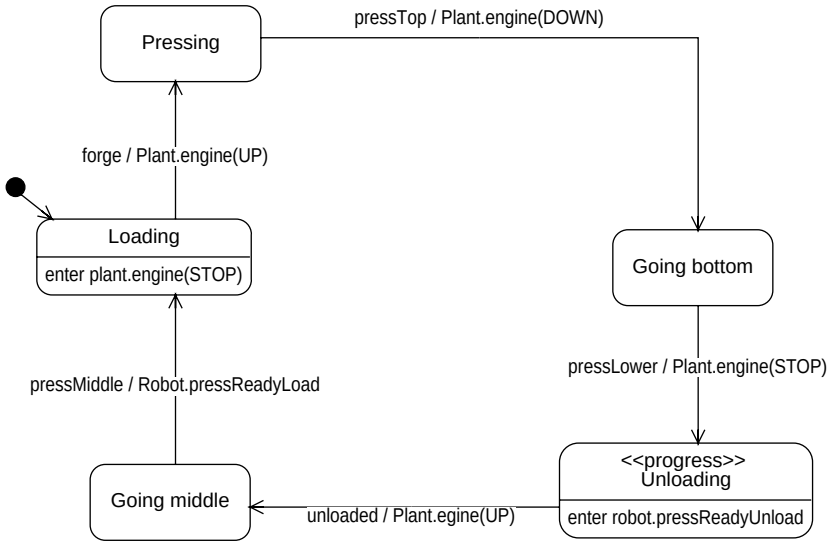



Figure 5.14: Controller of the Press

first action performed by the controller is to enter the Loading state. There is an entry action associated to this state. Each time that the controller enters this state, the action is executed. In this case, it informs the robot that the press is ready to be loaded. When the robot places a new blank into the press, it sends the forge event to the press controller. The forge event is the trigger for a transition from the Loading state to the Pressing state. When the press dispatches the forge event, it leaves the Loading state and it enters the Pressing state. The transition contains an action that commands the plant to start the engine to lift the press. The Pressing state is exited when the mobile part of the press arrives at the top. Then the controller switches to the GoingBottom state and it commands the engines to go down. When the mobile part is at the bottom of the press, the controller enters the Unloading state. It stops the engine and it signals the robot to indicate that the press is ready to be unloaded. After taking the forged blank, the robot sends an Unloaded event so the press can move to the loading position.

### Adding User Requirements to the Model

The Unloading state is marked with the ≪progress≫ stereotype. Every time that the machine has processed one piece, the controller enters this state. This stereotype indicates the fact that the object has made a unit of work. If the press is fed with an unlimited amount of blanks, the controller should enter and exit the Unloading state an infinite amount of times.

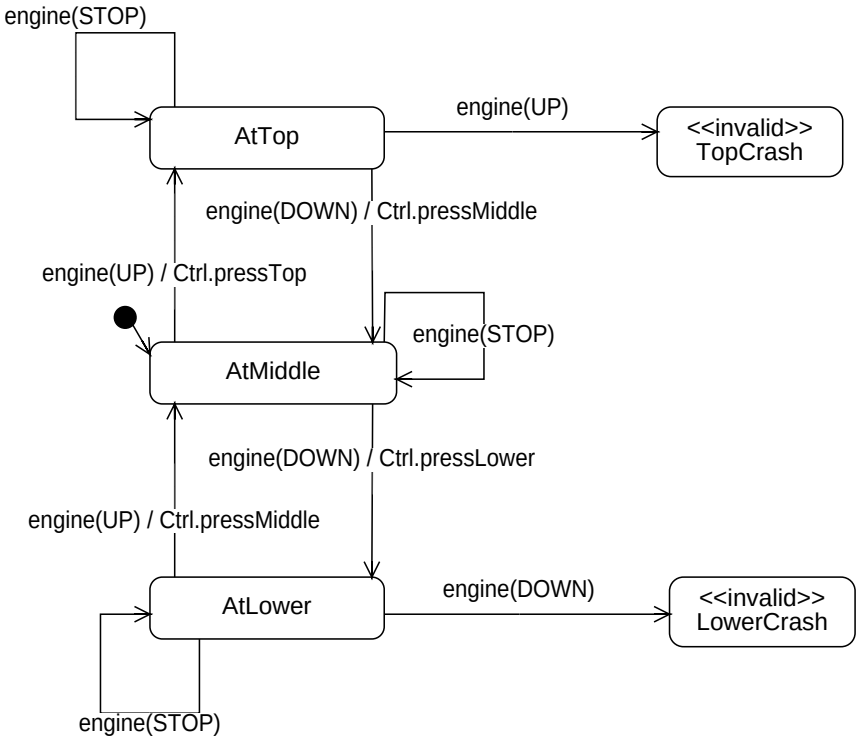



Figure 5.15: Model for the Plant of the Press

Figure 5.15 shows a model for the plant of the press. If the plant receives the event Up, the mobile part will start moving and, *eventually*, it will enter the AtTop state.

If the mobile part is in the top position and the engine receives the Up event, the press can break itself by moving the plate past its physical limit. This is modeled with the TopCrash state, that is marked with the ≪invalid≫ stereotype. This stereotype is used to capture a safety requirement: An invalid state should never be entered. The complete requirement for the press class is $(\Box\neg TopCrash) \wedge (\Box\neg LowerCrash) \wedge (\Box(\Diamond Unloading \wedge \Diamond\neg Unloading))$.

We must also ensure that the press is always in the right position during the load and unload phases to avoid a collision with the robot arms. Since this property involves two objects, the press and the robot, we must state it as a constraint over their link. We formulate this requirement using the constraint $\Box((Ctrl.Loading \Rightarrow Plant.AtMiddle) \wedge (Ctrl.Unloading \Rightarrow Plant.AtLower))$.

Figure 5.16 is a simple collaboration diagram showing an instance of the press controller and the plant. It shows the collaboration in its initial state where the objects have not exchanged any messages yet. We can use the vUML

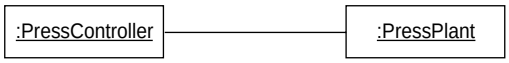



Figure 5.16: Press Collaboration Diagram

tool to automatically generate all possible evolutions of the collaboration diagram and to check whether our model holds the safety and liveness requirements that we have introduced to it.

Both the PressController and the PressPlant are active objects. This means that each one has its own thread of control and evolves independently and asynchronously from the other objects in the collaboration and there can be delays in the communication and dispatching of an event. For example, if the PressPlant is in the AtMiddle state and dispatches the engineUP event, the state machine will switch to the AtTop state and send the pressTop event instantaneously. However, there can exist an arbitrary delay during the communication of the event. This delay represents inter-object communication and synchronization. Once the event is in the queue of the PressController, there can be another arbitrary delay until the event is dispatched. This delay can be produced because there are other events in the queue or simply because the receiver is not scheduled for execution. If this asynchronous behavior is not acceptable, the designer should use synchronous call events for inter-object communication.

It is also important to note that the collaboration diagram does not include an object representing the robot. The press controller sends the pressReadyLoad and pressReadyUnload through a link named robot, and it expects the forge and unload events from any of its links. However, we have not modeled the robot yet. The model is *open*. The vUML tool detects this condition and creates an event generator for the forge and Unloaded events. This allows us to verify the composition of the press controller and the plant while abstracting from the robot.

### 5.4.2 Modeling the robot

The most complex object in the production cell is the robot. The robot controller must coordinate the execution of its activity with the other objects of the model. As we can observe in Fig. 5.12, the controller of the robot and the controller of the press must synchronize their operation during the loading and unloading phases in order to avoid a collision. This is a normal design requirement in models containing active objects that evolve independently.

We synchronize the activity of the two objects by using call events and deferred events. The press sends the pressReadyUnload event to the robot to indicate that, effectively, the press is ready to be unloaded. If the robot cannot

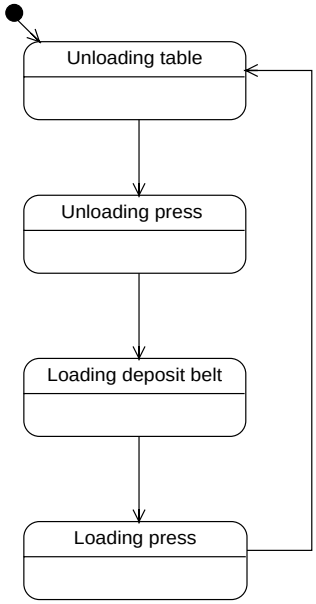



Figure 5.17: Overall Behavior of the Robot Controller

process the request of the press, e.g. because it is taking a piece from the table, the event will be deferred.

Actually, the robot must also coordinate its activity with the table and the deposit conveyor belt. Figure 5.17 shows a statechart diagram describing the behavior of the controller of the robot. It is divided into four main composite states, corresponding to the four main tasks of the robot, as described in [66]. Each composite state is refined into several substates.

The controller of the robot starts in the Unloading Table state. Immediately after entering this state, it enters the Moving to 1 state, where the robot rotates until Arm 1 points to the table. If the activity inside the composite state reaches the final state the completion transition fires and the object leaves the composite state. In this case, the robot will leave the Unloading Table state and enter Unloading Press.

### 5.4.3 Verification

We do not show the statechart representing the behavior of the table and the conveyor belt, since they are conceptually similar to the ones representing the press or the robot. The complete model is still an open model because the object modeling the feed belt expects the reception from the environment of the event newBlank. This event indicates the arrival of a new metal blank to the production cell. There is a safety requirement that states that it should be only

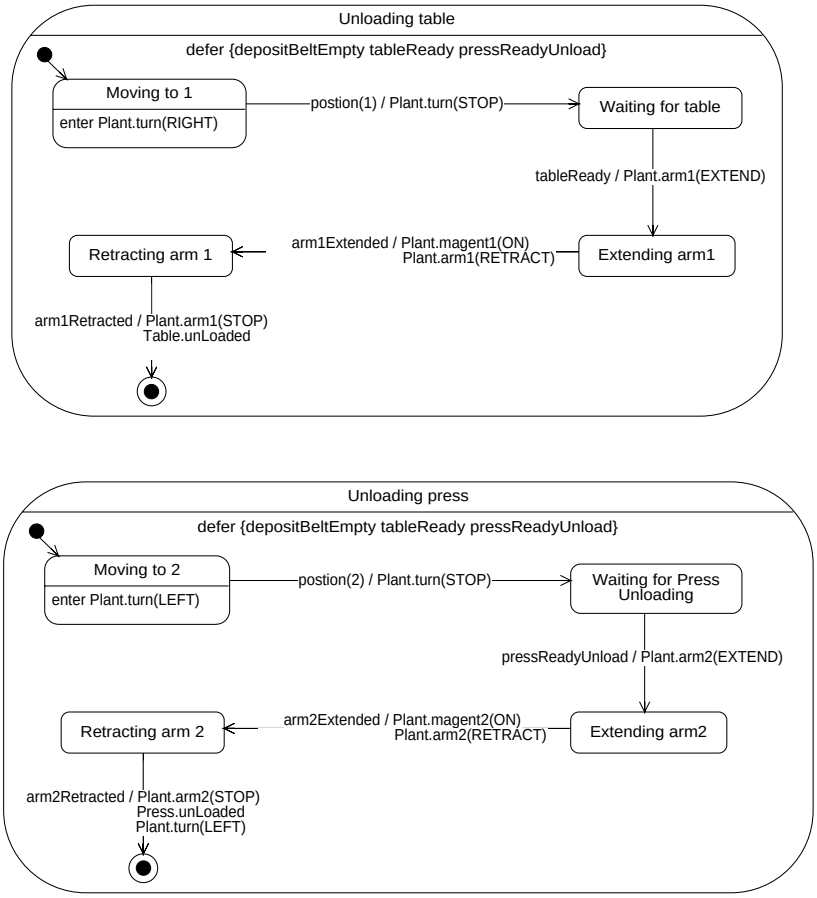



Figure 5.18: Detailed Behavior of the Robot Controller

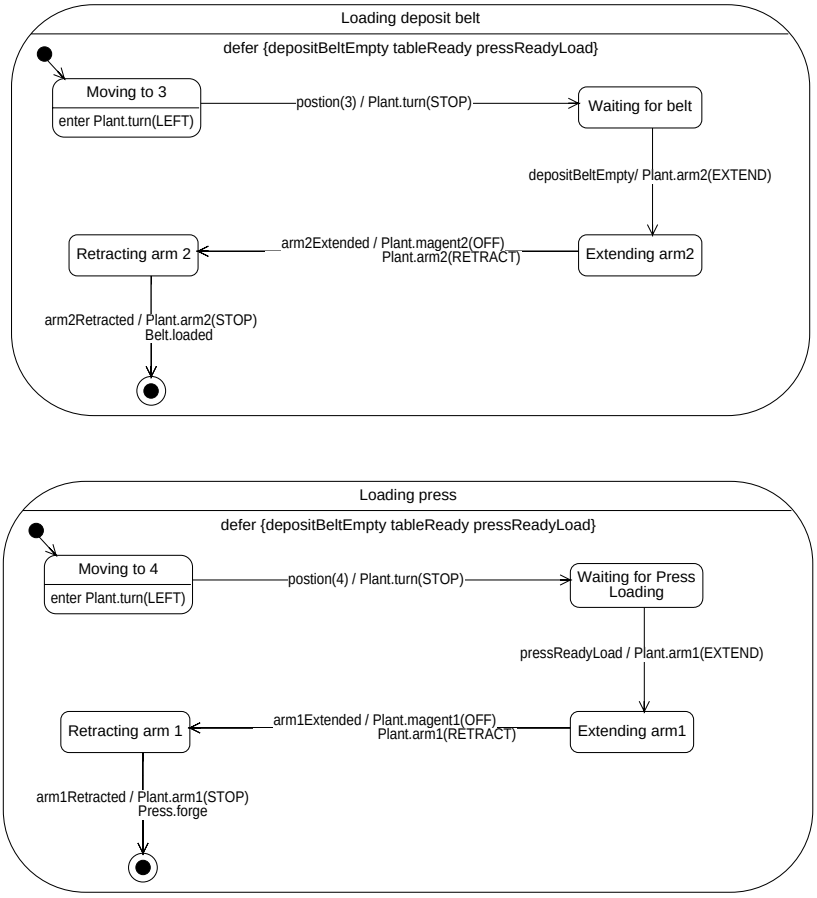



Figure 5.19: Detailed Behavior of the Robot Controller (cont.)

| Model | States | Memory | Time |
|---|---|---|---|
| Press Safety | 57 | 2.5 Mb | 7 s |
| Press Livelocks | 238 | 2.5 Mb | 8 s |
| Complete Safety | 52060 | 51.9 Mb | 37s |

Table 5.1: Performance of the vUML Tool

one metal blank on a conveyor belt at maximum. When we verify the model using vUML, the event generator will generate newBlank events only when the feed belt will accept them, that is, when it is empty.

We verify the complete model looking for safety properties. Table 5.1 shows the memory and time used during the verification. The measures have been collected using a Pentium II computer running at 233Mhz with 64Mb of RAM, with partial order reductions and optimized code generation. At the moment, we are not able to verify the complete model for livelocks because of state space explosion [98].

## 5.5 Conclusions and Related Work

In this chapter we have presented a methodology to verify collaborations of statecharts and a prototype of a tool supporting the methodology. The tool is based on the formalization of UML statecharts presented in the previous chapter. We have also discussed a case study to show the viability of our approach

The need for a tool like vUML should be clear. In Section 4.3 we have seen that the semantics of UML state machines are non-trivial. Therefore, manual conversion of UML models into a PROMELA specification or other state-based formalisms, like Petri nets, is a difficult, cumbersome and error prone task. Moreover it also requires a deep knowledge of UML and the model checker. With vUML, the designer does not need to know how to prepare the input model or how to interpret the counter-examples produced by the model checker.

### 5.5.1 Evaluation of the tool

Four of the most important aspects to consider when evaluating a verification tool are the notation used to express the models, how the tool fits into the design process, the different correctness requirements that can be verified and the computational resources needed to perform the verification.

**Notation:** vUML uses UML as the input language for the models. The behavior of the objects in the model is described graphically using UML state-

chart diagrams. Since the UML standard is becoming more and more accepted, we expect that designers will be more familiar with UML than with other formalisms like PROMELA, Petri nets or communicating automata. vUML still uses the PROMELA language for describing guards and actions since there is no standard syntax for these elements in UML.

**Design process:** UML models often describe software under construction. The vUML tool is able to verify open models where the environment is not completely specified or the objects contain abstract states. This feature allows the designer to use the vUML tool from the beginning of the design process. We can thus use vUML in a iterative and incremental design process, where the model is constructed and verified continuously until it is complete and correct. The verification process is automatic and it can be repeated as many times as needed with little or no effort from the designer. vUML can also be used in large projects. It is easy to split a large model into several different diagrams and verify the model in parts, assuming that the environment always sends the expected events. This is possible thanks to the object-oriented semantics of UML state machines and the vUML option to close automatically a model.

**Verification:** The designer can include constraints to define safety properties of the model. vUML checks the model for violations of the constraints and for six other different kinds of errors, including deadlocks and livelocks. vUML does not directly support linear temporal logic formulas. SPIN supports LTL properties, but how to express a LTL formula in a UML diagram is an open problem. In order to keep the model checking process as transparent as possible to the designer, we decided not to verify LTL formulas in this version of the tool. In the future we would like to use OCL constraints in the vUML model. However, this requires a complete formalization of the OCL language and its translation to PROMELA.

**Performance:** The verification of models containing some UML features like call events, deferred events or completion transitions increases the complexity of the specification and, therefore, the resources needed for the verification. This problem is introduced by the semantics of UML rather than by vUML. SPIN is an excellent model checker but its underlaying process model is completely different than the UML statecharts model. The run-to-completion step of each UML class is coded in PROMELA. Therefore, SPIN does not only verify the model but also the whole run-to-completion step. Also, the PROMELA model uses many auxiliary variables that increase the size of the state vector.

The vUML tool is still under development. Currently the tool uses a proprietary input format for UML models but we plan to support the more standard XMI format in the future. However, the most urgent development point is the reduction of the state space. We want to study the generation of more efficient PROMELA models. Also, we are interested in how to improve the representation of counter-examples in UML. We are exploring ways to show both the

internal and external activity of an object in the same diagram and methods to automatically group counter-examples that lead to the same error. Furthermore we are looking at the possibility of filtering out events that are present in the counter-example but do not contribute to the error.

### 5.5.2 Related work

We can find many interesting contributions to the Production Cell case study in [66]. We think that the UML notation is in many cases more intuitive and scales better than other graphical formalisms without a notion of hierarchy, like Petri nets. A UML design is partitioned into classes and the behavior of each class can be represented as a hierarchy of states or sub-behaviors. We think this helps the designer to create and understand large designs. Compared to the work presented by Sekerinski in chapter 6 of [90], the UML approach is completely graphical and the verification of the model is automatic. Actually, we can combine both approaches: first model the system with UML and verify it with vUML and then translate the statecharts into a B specification that can be refined into an executable program.

There are other tools similar to vUML. MOCES by Mikk [75, 73] is a tool for verifying statecharts and it also uses the SPIN model checker. MOCES is based on the STATEMATE semantics of statecharts and, therefore, cannot be used to verify UML statecharts. Furthermore, MOCES verifies a single statechart, while vUML verifies a collaboration of objects. It is not possible to compare the performance of MOCES and vUML directly because the results will reflect not only the differences between the code generated by the tools but also the differences between UML and Harel's statecharts.

# Chapter 6

# Real-Time and Hybrid Behavior in Statecharts

> 'No, the answer is an *orange* and two lemons.'
>
> '*Lemons?*'
>
> 'If I have three lemons and three oranges and I lose two oranges and a lemon what do I have left?'
>
> 'Huh?'
>
> 'OK, so you think that time flows *that* way, do you?. Interesting.'
>
> — Douglas Adams, Mostly Harmless

## 6.1 Introduction

Hybrid systems [100] model continuous processes controlled by discrete computations. They are useful to study systems where a computer supervises and controls a physical phenomenon from the real world, like the movement of an elevator or a chemical plant. These processes evolve in real time and their state is often best represented using continuous mathematical equations rather than discrete data types.

Although UML is a general purpose modeling language, there are no facilities for modeling real-time and hybrid systems in *standard* UML 1.3. However, it is possible to extend the existing UML diagrams for this purpose. Many tool vendors have proposed their own extensions to the language and there is an ongoing effort to create a standard profile for modeling real-time systems in UML [94].

In this chapter, we show an extension to UML statecharts that can be used to model real-time and hybrid phenomena. We think that the key missing

element in UML is a well-defined model of time for the behavioral diagrams, like sequence, collaboration, activity and statechart diagrams. We also present a new and very general formalism that can be used to analyze real-time and continuous behavior in statecharts. We call this formalism continuous action systems and it based on the previous work on discrete action systems.

Action systems [7, 9] have been used successfully to model discrete systems, i.e. systems that use a discrete control upon a discrete state space. Their original purpose was to model concurrent and distributed systems. In the next sections, we show that the action system model can be easily adapted to model hybrid systems including hybrid statecharts. An important advantage of this adaptation is that standard proof techniques, developed for ordinary action systems, can be reused to reason about hybrid statecharts.

Our model for hybrid systems is based on a new approach to describe the state of a system. Essentially, the attributes will range over functions over time, rather than just over values. This allows an attribute to capture not only its present value, but also the whole history of values that the attribute has had, as well as the default future values that the attribute will receive. Updating an attribute is restricted so that only its future behavior can be changed, not its past behavior. Continuous action systems are inspired by, but differ from, the extension of action systems to hybrid systems described in [86].

Proofs about action system properties are based on the refinement calculus [12]. Action systems are intended for stepwise development, where the correctness of refinement steps is verified within the refinement calculus. Therefore, we get an implicit notion of refinement also for continuous action systems. Even though the refinement of hybrid systems is out of the scope of this thesis, the approach we adopt for hybrid systems fits well into the refinement calculus framework and it can be used in systems where correctness is a central concern.

We proceed as follows. We introduce the notion of hybrid statechart in Section 6.2. The action system model is briefly described in Section 6.3. Continuous action systems are described in Section 6.4. Their semantics is defined by explaining how to translate them into ordinary action systems. Section 6.5 shows how to represent hybrid statecharts as continuous action systems. In Section 6.6 we show how to proof safety properties. Conclusions and comparisons to related work are presented in Section 6.7.

## 6.2 Hybrid Statecharts

In this section we introduce the notion of hybrid statechart as a diagram for modeling hybrid behavior. We define a *hybrid statechart* as a statechart containing continuous attributes. A *continuous attribute* is an attribute of a classifier that stores a function of time. It represents a continuous behavior that de-

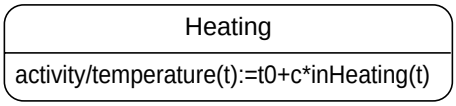
pends on time. For example, we can model the balance of a saving account as a function of time. Although the law defining the balance is constant, its value changes every day (or month). If the function defining a continuous attribute is a constant function, we obtain the usual behavior for an attribute.

Continuous attributes are the most significant element of a hybrid statechart They can be evaluated in the guard of a transition and updated as the effect of a transition or in any other action related to a statechart, for example when entering or exiting a state.

There are two important uses of continuous attributes: to define activities and as clocks. We can use a continuous attribute as a *clock* to measure the passage of time. A clock can be reset as the effect of a transition or in the entry or exit actions of a state. Clocks can be evaluated in the guards of transitions to model real-time behaviors, e.g, a guard [c(t)=1] will hold one time unit after the clock c has been reset to 0.

As a convenience for the designer, we assume that for each state s of the statechart there is a clock named inS that measures how long state s has been active. If the state is not active then the value of its associated clock is undefined.

Continuous attributes can be also associated with activities. The notion of activity is a standard concept from the UML definition. An *activity* is performed *while* an object is in a given state. It starts when the object enters the state and it stops when the machine leaves the state. A *continuous activity* is an activity defined by a tuple of continuous attributes. As an example, the following state has a continuous activity to model a variation of the temperature attribute.




This activity states that the continuous attribute temperature will be set to an initial value t0 and then it will increase linearly at rate c while the state Heating is active. The temperature is a function of time: for each instant t we obtain a different value for the attribute.

We can model real-time and hybrid behavior by combining clocks and continuous activities in a statechart. Figure 6.1 is an example of such diagram. Initially, the state Up is active, the clock c is reset and the attribute x will follow the value of the clock inUp. The machine will be in this state until the instant when the guard of the transition becomes true, that is when x(t)=1. At that instant, a change event will fire the transition and the machine will switch to the state Down and x will start decreasing according to the function 1-inDown(t).

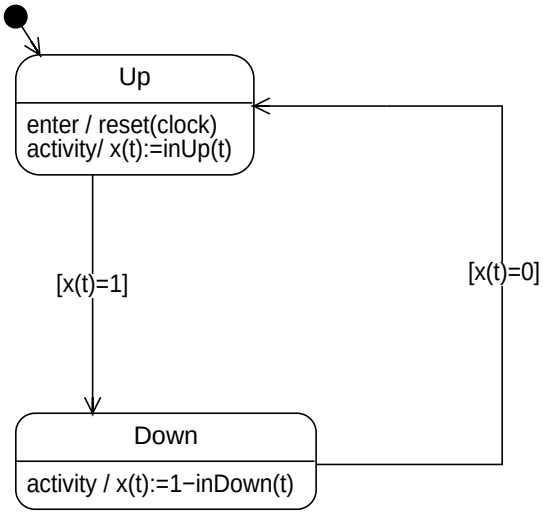



Figure 6.1: Hybrid Statechart Saw

We can represent the values of the attributes in a time diagram. Fig. 6.2, shows each attribute on the same time domain together with the points in time where a transition is fired. Since there are no final states on the statechart, the evolution of the attributes continues forever.

In the following sections we present the continuous action system formalism that we will use to explain the behavior of hybrid statecharts in a precise way.

## 6.3 Action Systems

We start by giving a brief overview of the action systems formalism. An action system is essentially a collection of attributes updated by a discrete control mechanism.

We define a finite set *Attr* of *attribute names*. An attribute $x$ in *Attr* of type $\Gamma$ is really a pair of two functions, the *value function* $valx : \Sigma \rightarrow \Gamma$ and the *update function* $setx : \Gamma \rightarrow \Sigma \rightarrow \Sigma$. The function $valx$ returns the value of the attribute $x$ in a given state, while the function $setx$ returns a new state where $x$ has a specific value, while the values of all other attributes are unchanged. This attribute system is explained with more detail in Section 3.3.

An *action system* consists of a finite set of attributes and a finite set of *actions* that act upon the attributes. This set of actions models the *control*

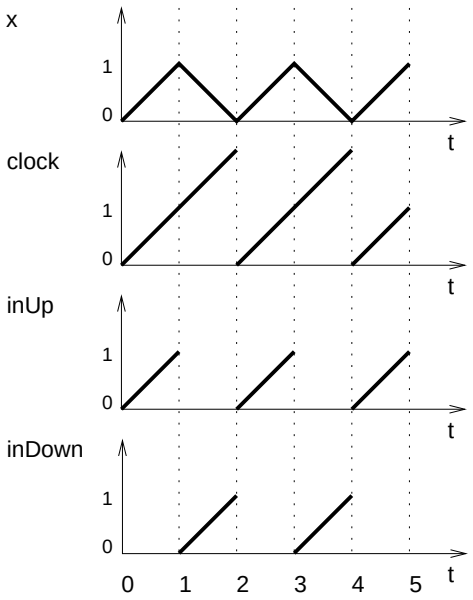



Figure 6.2: Evolution of the Hybrid Statechart Saw

*mechanism* over the system. An action system $\mathcal{A}$ has the following form:

$$\mathcal{A} \quad \triangleq \quad |[\mathsf{var}\ x \bullet S_0\ ;\ \mathsf{do}\ A_1 \Box\ \ldots \Box\ A_m\ \ \mathsf{od}\ ]| : y \tag{6.1}$$

Here $x = x_1, \ldots, x_n$ are the *local attributes* of the system, $S_0$ is a statement that initializes the attributes, while $A_i = g_i \rightarrow S_i$, $i = 1, \ldots, m$, are the *actions* of the system. Here $g_i$ is known as the *guard* of the action $A_i$ and $S_i$ as the *body* of the action. The attributes $y = y_1, \ldots, y_k$ are defined in the environment of the action system. We say that they are *imported attributes*. Attributes in $x$ may be *exported*, in the sense that they can be read, or written, or both read and written by environment actions. In this case, we decorate these with $-$, $+$ or $*$, respectively.

An action $A$ of the form '$g \rightarrow S$' is a guarded statement that is executed in the loop $\mathsf{do}\ \ldots\ \mathsf{od}$ only when $g$ is enabled, i.e., when $g$ evaluates to $\mathsf{T}$. The actions are executed in the loop as long as there are enabled actions. Hence, the loop is thus just Dijkstra's guarded iteration statement.

The guard of an action is just an ordinary boolean expression. The body is a statement, defined by the following simple syntax:

$$S \quad ::= \quad \mathsf{abort} \mid \mathsf{skip} \mid x := e \mid \{x := x' | R\} \mid \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid S_1\ ;\ S_2$$

Here $x$ is a list of attributes, $e$ is a corresponding list of expressions, $x'$ is a list of (logical) variables standing for unknown values, and $R$ is a relation specified

in terms of $x$ and $x'$. Intuitively '$\mathsf{skip}$' is the stuttering action, '$x :\!= e$' is a multiple assignment, '$\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$' is the conditional composition of two statements, and '$S_1\ ;\ S_2$' is the sequential composition of two statements. Given a relation $R$ and a list of attributes $x$, we denote by $\{x :\!= x' | R\}$ the *non-deterministic assignment* of some value $x' \in R.x$ to $x$ (the effect is the same as $\mathsf{abort}$, if $R.x = \emptyset$). The action '$\mathsf{abort}$' always fails and is used to model disallowed behaviors.

The semantics of the actions language has been defined in terms of weakest preconditions in a similar way as for the language of contracts presented on page 77. In this case, it is a two-agent game between the user and the system and we do not need to denote for which agent are we calculating the $wp$ transformer. Thus, for any predicate $q$, we define

$$
\begin{array}{lcl}
wp.\mathsf{abort}.q & = & \mathsf{F} \\
wp.\mathsf{skip}.q & = & q \\
wp.(x :\!= e).q & = & q[x := e] \\
wp.\{x :\!= x' | R\}.q & = & \forall x' \in R.x \cdot q[x := x'] \\
wp.(S_1\ ;\ S_2).q & = & wp.S_1.(wp.S_2.q) \\
wp.\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}.q & = & \mathsf{if}\ g\ \mathsf{then}\ wp.S_1.q\ \mathsf{else}\ wp.S_2.q\ \mathsf{fi}
\end{array}
$$

Where $q[x := e]$ stands for the substitution of all free occurrences of the attribute $x$ in the predicate $q$.

The execution of an action system is as follows. The initialization $S_0$ will set the attributes to some specific values, using a sequence of possibly non-deterministic assignments. After the initialization, actions may become enabled, chosen and executed. The chosen actions will change the values of the attributes in a way that is determined by the action body. Two or more actions can be enabled at the same time, in which case one of them is chosen for execution, in a demonically nondeterministic way. The computation terminates when no action is enabled. Termination of an action system means the termination of the control over the system, which means fixing the final values of the attributes forever. Actions systems model parallel execution by interleaving atomic actions in a demonically nondeterministic fashion.

An action system is not usually regarded in isolation, but as a part of a more complex system. The rest of the system (the *environment*) communicates with the action system using shared (imported and exported) attributes, referred to as the *global* attributes. We can model other means of communication as well using the action systems framework [10].

A large action system is constructed from smaller ones using parallel composition. We define the parallel composition of two action systems

$$
\begin{array}{lcl}
\mathcal{A} & = & |[\mathsf{var}\ x : T \bullet S_0\ ;\ \mathsf{do}\ g_1 \rightarrow S_1 \Box\ \ldots \Box\ g_m \rightarrow S_m\ \ \mathsf{od}\ ]| : y \\
\mathcal{A}' & = & |[\mathsf{var}\ x' : T' \bullet S'_0\ ;\ \mathsf{do}\ g'_1 \rightarrow S'_1 \Box\ \ldots \Box\ g'_n \rightarrow S'_n\ \ \mathsf{od}\ ]| : y'
\end{array}
$$

as the action system $\mathcal{A} \parallel \mathcal{A}'$, defined by

$$\begin{array}{lll} \mathcal{A} \parallel \mathcal{A}' & \triangleq & |[\mathsf{var}\ x : T, x' : T' \bullet S_0 \,;\, S_0' \,; \\ & & \ \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m \Box\ g_1' \to S_1' \Box\ \ldots \Box\ g_n' \to S_n' \\ & & \ \mathsf{od}]| : (y \cup y') - (z \cup z)' \end{array}$$

We assume here that the attributes $x$ and $x'$ are disjointed. The initializations $S_0$ and $S_0'$ may only refer to their respective attributes $x$ and $x'$, so the order in which the initializations are executed does not matter.

The imported attributes are the union of the attributes imported by the component action systems without the exported attributes $z$ and $z'$ of the two systems. The actions are also the union of the actions of the two component systems.

In the next section, we extend the action system formalism by adding a notion of time and we show how to model attributes that are functions of time. These extensions to the formalism define a new model for hybrid systems.

## 6.4 Continuous Action Systems

A *continuous action system* consists of a finite set of time-dependent attributes that can range over discrete or continuous value domains, together with a finite set of actions that act upon the attributes. It is of the form

$$\mathcal{C} \quad \triangleq \quad |(\mathsf{var}\ x : \mathsf{Real}_+ \to T \bullet S_0 \,;\ \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m\ \ \mathsf{od}\ )| : y \quad (6.2)$$

Intuitively, the execution of a continuous action system proceeds as follows. There is an implicit attribute *now*, that shows the present time. Initially $now = 0$. The initialization $S_0$ assigns initial time functions to the attributes $x_1, \ldots, x_n$. These time functions describe the default future behavior of the attributes, whose values thus may change with the progress of time. The system will then start evolving according to these functions, with time (as measured by *now*) moving forward continuously. The guards of the actions may refer the value of *now*, as may expressions in the action bodies and the initialization statements.

As soon as one of the conditions $g_1, \ldots, g_m$ becomes true, the system chooses one of the *enabled* actions, say $g_i \to S_i$, for execution. The choice is nondeterministic if there is more than one such action. The body $S_i$ of the action is then executed. Execution is atomic and instantaneous. It will usually change some attributes by changing their future behavior. Attributes that are not changed will behave as before. After the changes stipulated by $S_i$ have been made, the

```
Saw ≙ |(  var x, clock, inUp, inDown : Real+ → Real ; Up, Down : Real+ → Bool
           • inUp :− (λt · t) ; Up :− (λt · T) ; Down :− (λt · F) ; x :− inUp;
          do x.now = 0 ∧ Down.now →
              Down :− (λt · F);
              clock :− (λt · t − now);
              inUp :− (λt · t − now);
              inDown :− (λt · 0);
              Up :− (λt · T);
              x :− inUP
          □ x.now = 1 ∧ Up.now →
              Up :− (λt · F);
              inUp :− (λt · 0);
              inDown :− (λt · t − now);
              Down :− (λt · T);
              x :− (λt · 1 − inDown.t)
          od
       )|
```

Figure 6.3: Continuous Action System $Saw$

system will evolve to the next time instance when one of the actions is enabled, and the process is repeated. The next time instance when an action is enabled may well be the same as the previous, i.e., time need not progress between the execution of two enabled actions. This is usually the case when the system is doing some (discrete, logical) computation to determine how to proceed next. Such a computation does not take any time. It is possible that after a certain time instance, none of the actions will be enabled anymore. This just means that, after that time instance, the system will continue to evolve forever according to the functions last assigned to the attributes.

As an example of a continuous action system consider the system in Fig. 6.3. We write $x :- e$ for an assignment rather than $x := e$, to emphasize that only the future behavior of the attribute $x$ is changed to the function $e$. The past behavior is unchanged.

This system has the same behavior as the statechart in Fig. 6.1. The initialization statement corresponds to the initial transition of the statechart, while the two actions correspond to the two transitions between the Up and Down states. We explain how to translate a statechart into an action system in Section 6.5.

One of the main advantages of this model for hybrid computation is that both discrete and continuous behavior can be described in the same way. In particular, if the attributes are only assigned constant functions, then we obtain a discrete computation.

### 6.4.1 Semantics of Continuous Action Systems

Let $\mathcal{C}$ be the continuous action system in (6.2). We explain the meaning of $\mathcal{C}$ by translating it into an ordinary action system. Its semantic interpretation is given by the following action system $\bar{\mathcal{C}}$ with an explicit model of time:

$$\begin{array}{lcll} \bar{\mathcal{C}} & \triangleq & |[ & \mathsf{var} \quad now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to T \\ & & & \bullet \quad now := 0 \,;\, S_0 \,;\, N; \\ & & & \mathsf{do} \quad g_1 \to S_1 \,;\, N \Box \ \ldots \Box \ g_m \to S_m \,;\, N \ \ \mathsf{od} \\ & & ]| : & y \end{array} \tag{6.3}$$

Here the attribute *now* is declared, initialized, and updated explicitly. It models the time moments that are of interest for the system, i.e. the starting time and the succeeding moments when some action is enabled. The value of *now* is updated by the statement $N$,

$$N \quad \triangleq \quad now := \mathsf{next}.gg.now$$

Here $gg = g_1 \vee ... \vee g_m$ is the disjunction of all guards of the actions and $\mathsf{next}$ is defined by

$$\mathsf{next}.gg.t \quad \triangleq \quad \begin{cases} min\{t' \geq t \ \mid \ gg.t'\}, & \text{if } \exists\ t' \geq t \text{ such that } gg.t' \\ t, & \text{otherwise.} \end{cases} \tag{6.4}$$

The function $\mathsf{next}$ models the moments of time when at least one action is enabled. Only at these moments can the future behavior of attributes be modified. If no action will be ever enabled, then the second branch of the definition will be followed, and the attribute *now* will denote the moment of time when the last discrete action was executed. In this case the discrete control terminates, i.e. the attributes will evolve forever according to the functions assigned to them last.

We assume that the minimum in the definition of $\mathsf{next}$ always exists when at least one guard is enabled in the present or future. Continuous action systems that do not satisfy this requirement are considered ill-defined. The minimum in the definition of $\mathsf{next}$ cannot be calculated when there are strict inequalities in time-dependent guards. E.g., the system (6.5) below is ill-defined.

$$\begin{array}{lcll} Bad & \triangleq & |( & \mathsf{var}\ x : \mathsf{Real}_+ \to \mathsf{Real} \\ & & & \quad \bullet \ x :- (\lambda t \cdot t); \\ & & & \quad \mathsf{do}\ now > 1 \to x :- (\lambda t \cdot 0) \ \ \mathsf{od} \\ & & )| & \end{array} \tag{6.5}$$

The *future update* $x :- e$ is defined by

$$x :- e \ \triangleq \ x := x/now/e$$

where

$$x/t_0/e \triangleq (\lambda t \cdot \mathsf{if}\ t < t_0\ \mathsf{then}\ x.t\ \mathsf{else}\ e.t\ \ \mathsf{fi})$$

For instance, consider the future update $clock :- (\lambda t \cdot t - now)$ from Figure 6.3). We have that

$$\begin{array}{ll} & clock :- (\lambda t \cdot t - now) \\ = & clock\ \ := clock/now/(\lambda t \cdot t - now) \\ = & clock\ \ := (\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ clock.t\ \mathsf{else}\ \ (\lambda t \cdot t - now).t\ \ \mathsf{fi}) \\ = & clock\ \ := (\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ clock.t\ \mathsf{else}\ \ t - now\ \ \mathsf{fi}) \end{array}$$

Thus, only the future behavior of *clock* is changed in this assignment. It is important to note that all attributes of a continuous action system are functions of time, except for *now*. For example, the statement $x :- (\lambda t \cdot t)$ updates the default future of $x$ with an increasing function, while $x :- (\lambda t \cdot now)$ updates it with a constant function.

This explanation of a continuous action system means that it essentially is a collection of time functions $x_1, \ldots, x_n$ over the non-negative reals, defined in a stepwise manner. The steps form a sequence of intervals $I_0, I_1, I_2, \ldots$, where each interval $I_k$ is either a left closed interval of the form $[t_i \ldots t_{i+1})$ or a closed interval of the form $[t_i, t_i]$, i.e. a point. The action system determines a family of functions $x_1, \ldots, x_n$ which are stepwise defined over this sequence of intervals and points. The extremes of these intervals correspond to the control points of the system where a digital discrete action is performed. In the example (Fig. 6.3), the sequence of intervals is

$$[0], [0 \ldots 1), [1 \ldots 2), [2 \ldots 3), \ldots$$

As such, the continuous action system is best understood as the limit of a sequence of approximations of the time functions $x_1, \ldots, x_n$, defined over successively longer and longer intervals $[0 \ldots t_i)$, where $i = 0, 1, 2, \ldots$. Looking at the example (6.3) in this way, its sequence of initial segments is

$$[0], [0 \ldots 1), [0 \ldots 2), [0 \ldots 3), \ldots$$

and the approximations that are defined are successively

$$
\begin{aligned}
x.t &= 0, \quad 0 \leq t \\
x.t &= t, \quad 0 \leq t \\
x.t &= \begin{cases} t, & 0 \leq t < 1, \\ 1 - t, & 1 \leq t \end{cases} \\
x.t &= \begin{cases} t, & 0 \leq t < 1, \\ 1 - t, & 1 \leq t < 2, \\ t - 2, & 2 \leq t \end{cases} \\
&\vdots
\end{aligned}
$$

Each attribute has a defined history of its past, i.e. on the interval [0,*now*), its present value in the point [*now*], and a default future. The execution of an action can modify the present value of a attribute and its default future, but not its past. It is important to note that such a definition does not necessarily determine a single function for each attribute $x_i$. Because of the non-deterministic choices involved, there might be a collection of such function tuples that are allowed by the continuous action system, and we do not know which one of these will actually be the one the system follows. Thus, the system behavior may only be determined up to a certain tolerance, and any system behavior that is within these limits is possible.

Another important observation regards the possibility of *Zeno behavior*. Our definition does not guarantee that the sequence of generated intervals will cover all the non-negative reals. They might only cover an initial segment of these. In this case, there is a limit point of time that the action system reaches when the number of iterations reaches infinity. These systems are well-defined but the simple explication of the behavior of the hybrid system is then not sufficient. However, in that case, we assume that the system is restarted at the limit point, and repeat the process again. This is meaningful if all attribute values converge upon a well-defined value in the limit. This restart can be carried out as many times as needed. Thus, a continuous action system may have multiple limit points in its execution. The standard action system semantics does not allow multiple limit points, so this is a point where the semantics really has to be extended. However, for simplicity, we will assume in the sequel that there is no Zeno-behavior, so a single limit point is sufficient. The absence of Zeno behavior means that the action system will define the values of the attributes for the whole domain of $\mathsf{Real}_+$.

A simple way of reaching a limit point is when a control computation (where time does not advance) does not terminate. This means that the continuous behavior of the system is stuck at the last time instance reached. Nontermination of the control computation is most certainly undesired and unintended.

This means that we have to prove that control computations where time does not advance always terminate.

### 6.4.2 Systems Using Differential Equations

The behavior of a dynamic system is often described using a system of differential equations. We can allow for this kind of definition by introducing the shorthand

$$\dot{x} :- f \triangleq \{x :- y \mid y.now = x.now \wedge \dot{y} = f.y, y \geq now\}$$

This will assign to $x$ a time function that satisfies the given differential equation and which is such that the function $x$ is continuous at *now*

As an example, if $f = (\lambda t \cdot c)$, where $c$ is a constant value, then we have that

$$\dot{x} :- (\lambda t \cdot c) \quad \triangleq \quad x :- (\lambda t \cdot x.now + c * (t - now))$$

The reader can find a complete model using differential equations, including proofs of safety properties in [5].

### 6.4.3 Real-Time Systems

We can use clocks or timers to measure the passage of time and to correlate the execution of an action with the time. A *clock attribute* is an attribute that measures the time elapsed since it was set to zero. Assume that $c$ is an attribute of type Real. We then use the following definition for resetting the clock $c$:

$$reset(c) \triangleq c :- (\lambda t \cdot t - now)$$

Since a clock is just a regular attribute, we can define as many clocks as we need and reset them independently. It is also possible to do arithmetic operations with clocks, e.g., to use time constraints as guards, or to refer to past values of an attribute, i.e. $x.(now - 1) = 1$ will hold one time unit after $x.now = 1$ holds.

### 6.4.4 Composition of Continuous Action Systems

In order to model complex systems, where several different subsystems or components evolve concurrently, we need to formally define the composition of continuous action systems. Actions systems communicate by means of shared variables. In some cases, we may also need to hide certain attributes of a component when describing more complex systems, but we ignore this aspect here for brevity.

We define the parallel composition of two continuous systems by using essentially the same method that we use for composing ordinary action systems [9]. Thus, if we have two continuous action systems

$$\begin{aligned} \mathcal{C} &= |(\mathsf{var}\ x : \mathsf{Real}_+ \to T \bullet S_0\ ;\ \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m\ \ \mathsf{od}\ )| : y \\ \mathcal{C}' &= |(\mathsf{var}\ x' : \mathsf{Real}_+ \to T' \bullet S'_0\ ;\ \mathsf{do}\ g'_1 \to S'_1 \Box\ \ldots \Box\ g'_n \to S'_n\ \ \mathsf{od}\ )| : y' \end{aligned}$$

then their parallel composition is the continuous action system $\mathcal{C} \parallel \mathcal{C}'$, defined by

$$\begin{aligned} \mathcal{C} \parallel \mathcal{C}' \triangleq |(\ & \mathsf{var}\ x : \mathsf{Real}_+ \to T, x' : \mathsf{Real}_+ \to T; \\ & \bullet\ S_0\ ;\ S'_0; \\ & \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m \Box\ g'_1 \to S'_1 \Box\ \ldots \Box\ g'_n \to S'_n\ \ \mathsf{od} \\ & )| : (y \cup y') - (z \cup z') \end{aligned}$$

We assume here that the attributes $x$ and $x'$ are disjoint. We need to combine the continuous action systems before we translate them into discrete action systems, because the local attribute *now* appears in both $\bar{\mathcal{C}}$ and $\bar{\mathcal{C}}'$. By combining the continuous action systems first, we ensure that the combined system uses a single *now* variable, which is checked by actions from both components.

Parallel composition essentially combines the attributes of the two component systems and, therefore, their continuous evolution. Because the actions in the parallel composition are the combined actions of the two systems, discrete changes will usually occur more frequently. Note that an action in one system may depend on an attribute in the other system, which may be again modified by actions of the former system. This means that the behavior of a system in a parallel composition is usually different from the behavior of the system when it is alone.

## 6.5 From Statecharts to Continuous Action Systems

In this section we show how to translate hybrid statecharts into continuous action systems. This translation has two objectives: to give a formal and unambiguous semantics to hybrid statecharts and to allow the mechanical analysis of safety properties of hybrid statecharts.

This translation is based on the definitions presented in Chapter 4. However, in order to simplify our exposition, we only present the translation of the basic model elements of the statechart notation. We reuse the definitions of the structure of statecharts from Chapter 4 but we need to update the dynamics of statecharts to include the notion of continuous attributes. As mentioned before, the main components defining the dynamics of a statechart are the event queue, the dispatching mechanism and the run-to-completion step. In this chapter, we will only model the run-to-completion step as a continuous action system. The

main reason is that the event queue and the dispatching mechanism are discrete components that would not benefit from this approach.

We assume that the language used to define the guards and actions of the statechart is the same language used for the action systems. The only difference is that, while in statecharts we use a more program-like notation for functions, e.g. temp(t)=t0+c*t, in action systems we use the lambda calculus notation, e.g. $temp = (\lambda t \cdot t_0 + c * t)$.

There are three steps in the translation of a hybrid statechart into a continuous action system. First, we must encode the attributes and states of the statechart into attributes of the action system. Second, we must define the initialization statement of the action system. The initialization statement models the initial transition of the statechart. Finally, we must translate each transition of the statechart into a corresponding action. After the translation, we obtain a continuous action system that describes the behavior of the hybrid statechart and can be analyzed mechanically.

### Encoding Attributes, Events and States

We can map the concepts of clocks and continuous attributes of a statechart directly onto the attribute model of a continuous action system. For each attribute of type $T$ of the statechart, we define an attribute of the action system with the same name and type Real+ $\rightarrow T$. We must use the future update statement to assign new values to these attributes.

We need a special attribute to keep the event dispatched to the machine. This is an imported attribute and it is defined by the dispatching mechanism. Its type is Real+ $\rightarrow E$, where $E$ is the set of possible events of the machine. Since events are dispatched one by one, we are only interested in the current event, i.e. *event.now*.

We also need one or more special attributes to keep the active state configuration of the statechart. In this section, we abstract the actual representation of the state configuration by defining three functions to manipulate it: *active*, *enter*, and *leave*. The *active* function evaluates to true if a given state is active, the *enter* function adds a given state to the active configuration and the *leave* removes a given state.

There are two main strategies for encoding the states of a statechart. One option is to use a single attribute that stores which state configuration is active. However, if the statechart contains hierarchical states or orthogonal regions the definition of the *active*, *enter* and *leave* functions are not trivial. The main advantage of this approach is that the representation of the state is more compact. This reduces the memory needed for a tool that represents the state of the model explicitly, like a model checker or an animation tool. A possible definition for the state handling functions when there are no hierarchical states

is as follows:

$$\begin{array}{rcl} active & \stackrel{\wedge}{=} & state.now = s \\ enter & \stackrel{\wedge}{=} & state :- (\lambda t \cdot s) \\ leave & \stackrel{\wedge}{=} & state :- (\lambda t \cdot none) \end{array}$$

Where *none* is a constant used to indicate that there is no state active in the statechart represented by the variable *state*. This only occurs during a transition and it is not observable.

Another approach is to use a boolean attribute for each state. In this way, for each state $\mathsf{s}$ in the statechart there is an attribute $s : \mathsf{Real}_+ \rightarrow \mathsf{Bool}$. In this case the operations with states are trivial: We say that the state $\mathsf{s}$ is active if the expression $s.now$ is true. Entering a state means setting the attribute to true, while leaving the state means setting the attribute to false.

$$\begin{array}{rcl} active' & \stackrel{\wedge}{=} & s \cdot s.now \\ enter' & \stackrel{\wedge}{=} & s :- (\lambda t \cdot \mathsf{T}) \\ leave' & \stackrel{\wedge}{=} & s :- (\lambda t \cdot \mathsf{F}) \end{array}$$

This approach is suitable for a theorem proving tool since reasoning about states is reduced to reason about simple boolean expressions.

### 6.5.1 Transitions

We translate each transition of the statechart into a different action. Assuming a transition $t = (s1, s2, e, g_t, a_t)$, we translate it into an action $g_a \rightarrow s_a$. The guard of the action $g_a$ is the conjunction of all the conditions needed to fire the transition:

- All source states are active, i.e. $\bigwedge_{s \in symb.s_1} active.s$
- The current event matches the trigger, i.e. $event.now = e$. If the event is a change event, then we must check that the change condition holds.
- The guard of the transition holds, i.e. $g_t.now$ holds.
- There is no conflict with a transition with higher static priority. If there is a transition $t'$ translated into an action ${g'}_a \rightarrow {s'}_a$ such that $t' \succ t$, we should only fire $t$ if the guard of the action association with $t'$ does not hold, i.e. $\neg {g'}_a.now$

The body of the action $s_a$ must perform the steps described in lines 13-29 of the RTC algorithm (see Fig. 4.8). We can stop an activity defined as continuous attributes by assigning a constant function with the current value of the activity or a non-deterministic function. We must also reset the clocks associated to the entering states.

**Initialization Statement**

The initialization statement represents the initial transitions of the statecharts. It defines the behavior of the machine from the time instant 0 until the first action is fired. It should carry out the following steps:

- Add the initial states to the active state configuration.
- Execute entry actions of the initial states.
- Start activities of the initial states.

The translation from a hybrid statechart to a continuous action system is an elaborate process since we must resolve the priorities between transitions statically. However, the translation can be performed automatically by a tool if the guards and actions of the statechart are given in the language of the action systems and the user provides a suitable set of functions to encode the state configuration.

The following section presents an example of how to translate a hybrid statechart into a continuous action system.

### 6.5.2 Examples

The Saw statechart in Figure. 6.1 can be translated into the action system in Fig. 6.3 using the previous template. The two transitions of the Saw statechart are triggered by a change event, so there is no need to add an attribute to keep the current event.

Another interesting example is the press of the Production Cell. In the previous chapter, we presented a *discrete* model of the press (Fig. 5.12). In this model, we identified three stable positions for the plate of the press: when the plate is at the bottom, at the middle and at the top of its travel. However, we did not model the real movement of the plate: we switched from one state to the other instantaneously. In this chapter we want to model the actual movement of the plate and we want to prove that the controller can not force the movement of the plate out of its bounds.

The press works as follows. Initially the lower part is in the middle position. The robot feeds a metal blank into the press and sends the forge event to the

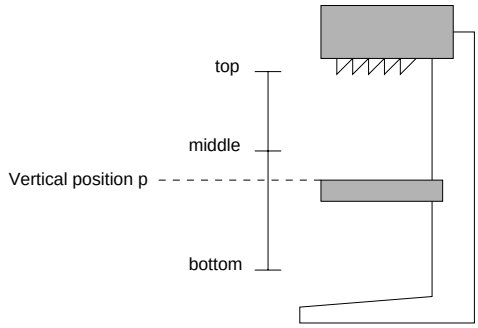



Figure 6.4: Vertical Position of the Press

press. Then the lower part of the press is raised until the top position and the blank is forged. The press will then move down until the bottom position and the forged blank is retrieved by the robot. When the press is unloaded (signaled by the unloaded event), the lower part is raised to the middle position, ready for being loaded again.

Figure 6.5 is a statechart describing the behavior of the press using continuous attributes. In the previous chapter, we used two different objects to model the controller and the plant of the press. Now, we can design a single hybrid statechart describing both the behavior of the continuous movement of the plate and its discrete control. This hybrid statechart is similar to the statechart of the discrete controller with the addition of the continuous attribute p to model the vertical position of the plate.

We can translate the hybrid statechart of the press into the continuous action system shown in Figure 6.6 by using the previous pattern. Please note that some of the transitions are fired by dispatching external events (forge and unloaded), while others are fired by change events.

Figure 6.7 shows a refinement of the Press action system where we have replaced the abstract functions for handling states with a more concrete version. We have decided to use a single attribute named *state* to keep the active configuration of the statechart. Since there are no hierarchical or orthogonal states, we can set and reset a state from the configuration just by assigning a constant representing each state to the *state* attribute.

We have also removed consecutive assignments to the *state* variable from the body of an action. The refinement rule that we use states that two assignments $x :- e_1 \,;\, x :- e_2$ can be reduced to a single assignment $x :- e_2$ if $x$ does not occur free in $e_2$. Although we do not discuss the refinement of action systems in this thesis, the reader can find a complete discussion of the topic in [12].

In the following section we show how to prove the safety properties of con-

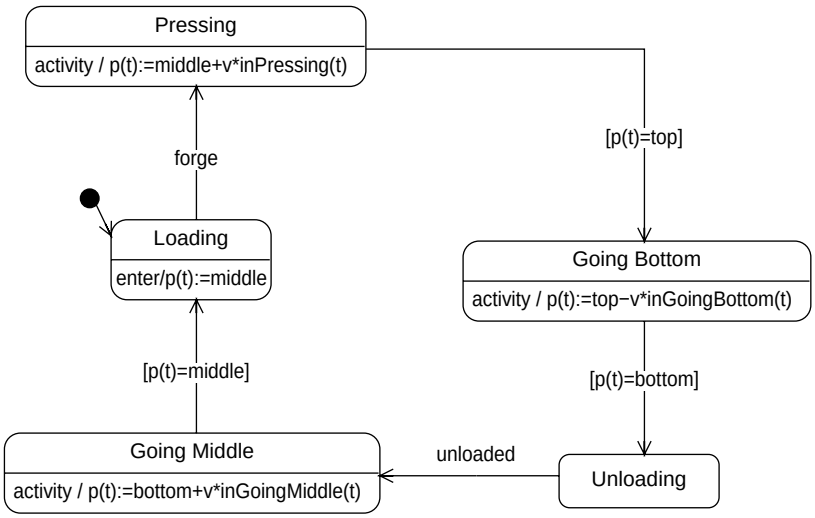



Figure 6.5: Hybrid Statechart for the Press

$$
\begin{array}{ll}
\mathcal{P}ress \triangleq |( & \mathsf{var}\ p : \mathsf{Real}_+ \to \mathsf{Real}; \\
& \quad inPressing, inLoading, inGoingBottom : \mathsf{Real}_+ \to \mathsf{Real}; \\
& \quad inUnloading, inGoingMiddle : \mathsf{Real}_+ \to \mathsf{Real}; \\
& \bullet\ reset(inLoading)\,;p :- \ middle\,;enter(loading); \\
& \mathsf{do}\ active(loading) \wedge event.now = forge \to \\
& \quad leave(loading)\,;reset(inPressing); \\
& \quad p :- (\lambda t \cdot middle + v * inPressing.t)\,;enter(pressing) \\
& \Box\ active(pressing) \wedge p.now = top \to \\
& \quad leave(pressing) : reset(inGoingBottom); \\
& \quad p :- (\lambda t \cdot top - v * inGoingBottom.t)\,;enter(goingBottom) \\
& \Box\ active(goingBottom) \wedge p.now = bottom \to \\
& \quad leave(goingBottom)\,;reset(inUnloading); \\
& \quad p :- (\lambda t \cdot bottom)\,;enter(unloading) \\
& \Box\ active(unloading) \wedge \neg unloaded.now \to \\
& \quad leave(unloading)\,;reset(inGoingMiddle); \\
& \quad p :- (\lambda t \cdot bottom + v * c.t)\,;enter(goingMiddle) \\
& \Box\ active(goingMiddle) \wedge p.now = middle \to \\
& \quad leave(goingMiddle)\,;reset(inLoading); \\
& \quad p :- \ middle\,;enter(loading) \\
& \mathsf{od} \\
)| & \quad : event
\end{array}
$$

Figure 6.6: First Version of the Press as a Continuous Action System.

$$
\begin{array}{l}
\mathcal{P}ress' \triangleq |( \quad \mathsf{var}\ p : \mathsf{Real}_+ \to \mathsf{Real}; \\
\qquad\qquad inPressing, inLoading, inGoingBottom : \mathsf{Real}_+ \to \mathsf{Real}; \\
\qquad\qquad inUnloading, inGoingMiddle : \mathsf{Real}_+ \to \mathsf{Real}; \\
\qquad\qquad state : \mathsf{Real}_+ \to \{loading, pressing, goingBottom, \\
\qquad\qquad\qquad\qquad unloading, goingMiddle\} \\
\qquad\quad \bullet\ reset(inLoading)\,;\, p :- middle\,;\, state :- loading; \\
\qquad\quad \mathsf{do}\ state.now = loading \wedge event.now = forge \to \\
\qquad\qquad reset(inPressing); \\
\qquad\qquad p :- (\lambda t \cdot middle + v * inPressing.t); \\
\qquad\qquad state :- pressing \\
\qquad\quad \Box\ state.now = pressing \wedge p.now = top \to \\
\qquad\qquad reset(inGoingBottom); \\
\qquad\qquad p :- (\lambda t \cdot top - v * inGoingBottom.t); \\
\qquad\qquad state :- goingBottom \\
\qquad\quad \Box\ state.now = goingBottom \wedge p.now = bottom \to \\
\qquad\qquad reset(inUnloading); \\
\qquad\qquad p :- (\lambda t \cdot bottom); \\
\qquad\qquad state :- unloading \\
\qquad\quad \Box\ state.now = unloading \wedge \neg unloaded.now \to \\
\qquad\qquad reset(inGoingMiddle); \\
\qquad\qquad p :- (\lambda t \cdot bottom + v * c.t); \\
\qquad\qquad state :- goingMiddle \\
\qquad\quad \Box\ state.now = goingMiddle \wedge p.now = middle \to \\
\qquad\qquad reset(inLoading); \\
\qquad\qquad p :- middle; \\
\qquad\qquad state :- loading \\
\qquad\quad \mathsf{od} \\
)| \qquad : event
\end{array}
$$

Figure 6.7: Final Version of the Press as a Continuous Action System.

tinuous action systems and their corresponding representation as hybrid statecharts.

## 6.6 Safety Properties

We prove a property of a continuous action system by proving that the property holds for the corresponding discrete action system. Hence, there is no special proof theory for continuous action systems, but the standard proof theory for action systems suffices (with the exception that we may need to consider multiple limit points, as was mentioned earlier). In this chapter, we will concentrate on safety properties since in many cases these are the kinds of properties that we want to establish initially for hybrid systems.

A common characterization for a safety property is that nothing 'bad' happens during the lifetime of the system. To put this another way, a safety property is a 'good' property $G$ that always holds, i.e.,

$$(\forall t \geq 0 \cdot G.t)$$

We can establish this property for the action system $\mathcal{C}$ in (6.2), by proving that a property

$$I \quad \stackrel{\wedge}{=} \quad (\forall t \mid 0 \leq t < now \cdot G'.t)$$

is an invariant of the corresponding discrete action system $\bar{\mathcal{C}}$, where $(\forall t \geq 0 \cdot G'.t \implies G.t)$. This implies the safety property, provided that the system does not have a Zeno behavior (i.e., *now* will go to infinity in the system). This means that the safety property $G$ will hold when the system is started in an initial state that satisfies $P$, if, and only if, the following conditions are satisfied for $\bar{\mathcal{C}}$:

$$\begin{aligned} (\forall t \geq 0 \cdot G'.t &\implies G.t) \\ P &\implies wp(now := 0\,;\, S_0\,;\, N, I) \\ I \wedge g_i &\implies wp(S_i\,;\, N, I), \quad i = 1, \ldots, m \end{aligned}$$

### 6.6.1 Proving the Safety of the Press

Consider the example of the press. We want to study two safety properties. First, we want to prove that the movable plate of the press does not pass the limits of the machine. Formally this is expressed by $(\forall t \geq 0 \cdot bottom \leq p.t \leq top)$, where $p$ is the vertical position of the plate. In the proofs, we assume that $bottom \leq middle \leq top$. Second, we also want to prove that $p$ is a continuous function on $\mathsf{Real}_+$. We need to choose an invariant $I$ that allows us to establish the safety property

$$(\forall t \geq 0 \cdot bottom \leq p.t \leq top) \wedge (p \textit{ continuous on } \mathsf{Real}_+)$$

using the above proof rule.

For the first conjunct of the safety property, an invariant of the form $(\forall t \mid 0 \leq t \leq now \cdot bottom \leq p.t \leq top)$ would be sufficient. However, to prove the global continuity property, we need a stronger invariant, which also ensures that the press remains in the correct position during the loading and unloading states. The following invariant $I$ is sufficient for establishing the required safety property:

$$\begin{aligned} I \quad \stackrel{\wedge}{=} \quad & (p \textit{ continuous on } [0, now] \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot bottom \leq p.t \leq top) \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot state.t = loading \Rightarrow p.t = middle) \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot state.t = unloading \Rightarrow p.t = bottom)) \end{aligned}$$

The proof must establish that the invariant is satisfied by the initialization from the moment 0 until the first moment an action is enabled and during the time elapsed between the execution of two actions.

We show below how to prove that the initialization statement establishes this invariant. The main work is in calculating the weakest precondition for the invariant in a way that avoids the formula becoming too large. We do this by applying simplifications at each intermediate state, whenever it is possible.

$$
\begin{array}{l}
wp(now := 0\,;\, reset(inLoading)\,;\, p :- (\lambda t \cdot middle)\,;\, state :- loading\,;\, N, I) \\
\equiv \{ \text{ substitute } next.gg.now \text{ for now in } I\} \\
\quad wp(now := 0\,;\, reset(inLoading)\,;\, p :- (\lambda t \cdot middle)\,;\, state :- loading, \\
\qquad (\forall now' \mid now' = min\{t' \geq now \mid\ (state.t' = loading \wedge forge.t') \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (state.t' = pressing \wedge p.t' = top) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (state.t' = goingBottom \wedge p.t' = bottom) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (state.t' = unloading \wedge unloaded.t') \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (state.t' = goingMiddle \wedge p.t' = middle) \\
\qquad\qquad\qquad\qquad \}\cdot \\
\qquad p \text{ continuous on } [0, now'] \wedge \\
\qquad (\forall t \mid 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge \\
\qquad (\forall t \mid 0 \leq t \leq now' \cdot state.t = loading \Rightarrow p.t = middle) \wedge \\
\qquad (\forall t \mid 0 \leq t \leq now' \cdot state.t = unloading \Rightarrow p.t = bottom)) \\
\equiv \{wp \text{ rule for sequential composition, splitting ranges, predicate calculus}\} \\
\quad wp(now := 0\,;\, reset(inLoading)\,;\, p :- (\lambda t \cdot middle), \\
\qquad (\forall now' \mid now' = min\{t' \geq now \mid\ (loading = loading \wedge forge.t') \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (loading = pressing \wedge p.t' = top) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (loading = goingBottom \wedge p.t' = bottom) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (loading = unloading \wedge unloaded.t') \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (loading = goingMiddle \wedge p.t' = middle) \\
\qquad\qquad\qquad\qquad \}\cdot \\
\qquad p \text{ continuous on } [0, now'] \wedge \\
\qquad (\forall t \mid 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge \\
\qquad (\forall t \mid 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge \\
\qquad (\forall t \mid now \leq t \leq now' \cdot loading = loading \Rightarrow p.t = middle) \wedge \\
\qquad (\forall t \mid 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom) \wedge \\
\qquad (\forall t \mid now \leq t \leq now' \cdot loading = unloading \Rightarrow p.t = bottom))) \\
\equiv \{\text{predicate calculus}\} \\
\quad wp(now := 0\,;\, reset(inLoading)\,;\, p :- (\lambda t \cdot middle), \\
\qquad (\forall now' \mid now' = min\{t' \geq now \mid forge.t'\}\cdot \\
\qquad p \text{ continuous on } [0, now'] \wedge \\
\qquad (\forall t \mid 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge \\
\qquad (\forall t \mid 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge \\
\qquad (\forall t \mid now \leq t \leq now' \cdot p.t = middle) \wedge \\
\qquad (\forall t \mid 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom))) \\
\equiv \{wp \text{ rule for sequential composition, future update}\}
\end{array}
$$

$wp(now := 0\,;\, reset(inLoading),$
$\quad (\forall now' \mid now' = min\{t' \geq now | forge.t'\}\cdot$
$\quad p/now/middle\ continuous\ on\ [0, now'] \wedge$
$\quad (\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge$
$\quad (\forall t | now \leq t \leq now' \cdot bottom \leq middle \leq top) \wedge$
$\quad (\forall t | 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$\quad (\forall t | now \leq t \leq now' \cdot middle = middle) \wedge$
$\quad (0 \leq t < now \wedge state.t = unloading \Rightarrow p.t = bottom)))$
$\equiv$ {$bottom \leq middle \leq top$ is an assumption of the model, predicate calculus }

$wp(now := 0\,;\, reset(inLoading),$
$\quad (\forall now' \mid now' = min\{t' \geq now | forge.t'\}\cdot$
$\quad p/now/middle\ continuous\ on\ [0, now'] \wedge$
$\quad (\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge$
$\quad (\forall t | 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$\quad (\forall t | 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom)))$
$\equiv$ {definition of future update }

$wp(now := 0, wp(reset(inLoading),$
$\quad (\forall now' \mid now' = min\{t' \geq now | forge.t'\}\cdot$
$\quad (\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \mathsf{else}\ middle\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge$
$\quad (\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge$
$\quad (\forall t | 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$\quad (\forall t | 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom))))$
$\equiv$ {$c$ does not appear in the postcondition, $wp$ rule for assignment}

$(now' = min\{t' \geq 0 | forge.t'\}\cdot$
$\quad (\lambda t \cdot \mathsf{if}\ t < 0\ \mathsf{then}\ p.t\ \mathsf{else}\ middle\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge$
$\quad (\forall t | 0 \leq t < 0 \cdot bottom \leq p.t \leq top) \wedge$
$\quad (\forall t | 0 \leq t < 0 \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$\quad (\forall t | 0 \leq t < 0 \cdot state.t = unloading \Rightarrow p.t = bottom))$
$\equiv$ {assume $t < 0 \equiv \mathsf{F}$, predicate calculus}

$(\forall now' \mid now' = min\{t' \geq 0 | forge.t'\}\cdot$
$\quad (\lambda t \cdot middle)\ continuous\ on\ [0, now'])$
$\equiv$ {$(\lambda t \cdot middle)$ is a constant function, so it is continuous}
$\mathsf{T}$

The last step in this proof assumes that the minimum exists. That is, $event.now = forge$ will eventually be true. We also assume that the system is non-Zeno has been established by other means. The proof shows that the invariant is satisfied by the initialization from the moment 0 until the first moment when an action is enabled.

After proving the initialization of system, we should consider whether the system behaves correctly after the execution of each action. We compute the weakest precondition of each action while assuming that the invariant of the system and the guard of the action hold. The proof for the first action follows.

$p$ *continuous on* $[0, now] \wedge$
$(\forall t | 0 \leq t \leq now \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t \leq now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t \leq now \cdot state.t = unloading \Rightarrow p.t = bottom) \wedge$
$state.now = loading \wedge forge.now$
$\vdash wp(reset(inPressing)\,;p :- (\lambda t \cdot middle + v * inPressing.t);$
$state :- pressing\,;N, I)$
$\equiv$ {substitute $next.gg.now$ for now in I, $wp$ rule for sequential composition}

$wp(reset(inPressing)\,;p :- (\lambda t \cdot middle + v * inPressing.t),$
$wp(state :- pressing,$
$(\forall now' | now' = min\{t' \geq now |\ (state.t' = loading \wedge forge.t') \vee$
$(state.t' = pressing \wedge p.t' = top) \vee$
$(state.t' = goingBottom \wedge p.t' = bottom) \vee$
$(state.t' = unloading \wedge unloaded.t') \vee$
$(state.t' = goingMiddle \wedge p.t' = middle))$
$\}\cdot$
$p$ *continuous on* $[0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t \leq now' \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t \leq now' \cdot state.t = unloading \Rightarrow p.t = bottom)))$
$\equiv$ {$wp$ rule for sequential composition, splitting ranges, predicate calculus}

$wp(reset(inPressing)\,;p :- (\lambda t \cdot middle + v * inPressing.t),$
$(\forall now' | now' = min\{t' \geq now |\ (pressing = loading \wedge forge.t') \vee$
$(pressing = pressing \wedge p.t' = top) \vee$
$(pressing = goingBottom \wedge p.t' = bottom) \vee$
$(pressing = unloading \wedge unloaded.t') \vee$
$(pressing = goingMiddle \wedge p.t' = middle))$
$\}\cdot$
$p$ *continuous on* $[0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | now \leq t \leq now' \cdot pressing = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom) \wedge$
$(\forall t | now \leq t \leq now' \cdot pressing = unloading \Rightarrow p.t = bottom))$
$\equiv$ {predicate calculus, $wp$ rule for seq. composition}

$wp(reset(inPressing), wp(p :- (\lambda t \cdot middle + v * inPressing.t),$
$(\forall now' | now' = min\{t' \geq now | p.t' = top\})\cdot$
$p$ *continuous on* $[0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t < now \cdot state.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t < now \cdot state.t = unloading \Rightarrow p.t = bottom))$
$\equiv$ {invariant assumption}

$wp(reset(inPressing), wp(p :- (\lambda t \cdot middle + v * inPressing.t),$
$(\forall now' | now' = min\{t' \geq now | p.t' = top\}) \cdot$
$p \text{ continuous on } [0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top)))$
$\equiv$ {rule for future update}

$wp(reset(inPressing), (\forall now' | now' = min\{t' \geq now | middle + v *$
$inPressing.t' = top\}) \cdot$
$p/now/(\lambda t \cdot middle + v * inPressing.t) \text{ continuous on } [0, now'] \wedge$
$(\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | now \leq t < now' \cdot bottom \leq p.t \leq top)))$
$\equiv$ {invariant assumption}

$wp(reset(inPressing), (\forall now' | now' = min\{t' \geq now | middle + v *$
$inPressing.t' = top\}) \cdot$
$p/now/(\lambda t \cdot middle + v * inPressing.t) \text{ continuous on } [0, now'] \wedge$
$(\forall t | now \leq t < now' \cdot bottom \leq p.t \leq top)))$
$\equiv$ $\{reset(inPressing) = (inPressing :- (\lambda t \cdot t - now))\}$

$(\forall now' | now' = min\{t' \geq now | middle + v * (t' - now) = top\} \cdot$
$(\lambda t \cdot \text{if } t < now \text{ then } p.t \text{ else } middle + v * (t - now) \text{ fi}) \text{ continuous}$
$\text{on } [0, now'] \wedge (\forall t | now \leq t \leq now' \cdot bottom \leq middle + v * (t - now) \leq top))$
$\equiv$ {computing $min$ function}

- $min\{t' \geq now | middle + v * (t' - now) = top\}$
  $=$ {assumption $top > middle > 0$, $v > 0$}
  $now + \dfrac{top - middle}{v}$

$\cdot (\forall now' | now' = now + \dfrac{top - middle}{v}) \cdot$
$(\lambda t \cdot \text{if } t < now \text{ then } p.t \text{ else } middle + v * (t - now) \text{ fi}) \text{ continuous}$
$\text{on } [0, now'] \wedge (\forall t | now \leq t \leq now' \cdot bottom \leq middle + v * (t - now) \leq top))$
$\equiv$ {1-point rule}

$(\lambda t \cdot \text{if } t < now \text{ then } p.t \text{ else } middle + v * (t - now) \text{ fi})$
$continuous \text{ on } [0, now + \dfrac{top - middle}{v}] \wedge$
$(\forall t | now \leq t \leq now + \dfrac{top - middle}{v} \cdot bottom \leq middle + v * (t - now) \leq top)$
$\equiv$ {assumptions $v > 0, bottom < middle < top$, real analysis, properties of $\mathsf{T}$}

$(\lambda t \cdot \text{if } t < now \text{ then } p.t \text{ else } middle + v * (t - now) \text{ fi})$
$continuous \text{ on } [0, now + \dfrac{top - middle}{v}]$
$\equiv$ { $p$ cont. on $[0, now] \wedge p.now = middle$, $lim_{t \to now^-} p.t = middle = p.now$}
$\mathsf{T}$

We do not show the proofs for other actions since they follow a similar pattern to the proofs presented so far.

## 6.7 Conclusions and Related Work

The definition of UML 1.3 does not include facilities for modeling real-time and hybrid systems. At the time of writing this text, there is an open request for proposals for a profile for scheduling, performance and time for UML. The proposed submission [94] enumerates many new model elements but does not define precise semantics for these new elements. In this chapter, we have shown an simple extension of the language that allows us to model real-time and hybrid behaviors in statecharts. We think that it is not necessary to add many new model elements and icons to model these concepts in UML but to define the dynamics of the existing model elements including a precise model of time.

The use of formal methods and models to describe hybrid systems has attracted quite a lot of attention in recent years, with a number of different models and formalisms being proposed in the literature (see e.g., [2, 76, 20]).

The concept of a hybrid statechart was first introduced by Pnueli in [60]. Pnueli's model for hybrid statecharts is based on the statecharts presented by Harel and formalized as *phase transition systems*. However, all the activities represented in the statechart must be combined into a single activity of the phase transition system.

We have formalized hybrid statecharts by introducing the notion of continuous action systems. The attributes in continuous action systems are functions over time and they are updated in a way that only changes their present and future behavior. Essentially, this amounts to extending the notion of state with both a history and a default future, thus generalizing the classical action systems approach that only handles the present state.

This extension allows us to model systems that combine discrete control with continuous behavior, the later either defined by explicit functions of time or by differential equations. We have also shown how to prove safety properties of continuous action systems using the classical invariant method. Finally, we have shown that the continuous action system model provides a simple way of defining the parallel composition of hybrid systems, using communication by means of shared variables. The reader can find a more elaborated case study using continuous action systems in [5].

The idea of extending an existing formalism to model real-time systems by introducing a variable representing the time was presented Abadi and Lamport in [1]. We follow the same approach here, extending an existing formalism to handle hybrid systems instead of creating a new formalism specific for such systems. This provides a clear advantage since now we can apply all the previous results on action systems to study real-time and hybrid models.

Rönkkö and Ravn [85] have already proposed a model for combining action systems and continuous behavior, called *hybrid action systems*. In their model, the continuous evolution of a variable is modeled as a special kind of atomic

action. An atomic action cannot be interrupted and its bounds are specified in advance. This affects the parallel composition of systems, since different simultaneous actions must be combined into a sequence of atomic actions. In the worst case, the parallel composition of two systems with $n$ and $m$ actions leads to a system with $n * m$ actions. Also, there is no implicit notion of time in their approach, which is also not intended for modeling real-time systems. In our model, parallel composition of two such systems gives a continuous action system with $n + m$ actions.

These advantages still exist when comparing our formalism with the *hybrid automaton* [2]. The number of states in the parallel composition of two hybrid automata is also the product of the number of states of the original automata. Note that in the hybrid automata formalism, transitions are fired synchronously while in the action system formalism actions are selected and executed asynchronously. The continuous action system formalism allows for the explicit failure of the system (modeled with the abort statement) and also allows references to historical values of the attributes in guards and expressions. Compared to hybrid automata, our model also allows the attributes to be selectively updated: only those attributes that are changed need to be mentioned in an action.

The translation from statecharts to continuous actions systems is a trade-off between expressiveness and tool support. In the previous chapter we have seen that we can use a model checker to verify a collaboration of objects defined using statecharts. A model checker allows us to study the joint behavior of many objects easily, but there are practical limitations in the description of the attributes of the objects. In this chapter we have seen how to analyze statecharts with complex attributes including attributes ranging over infinite types. This allows us to model complex dynamic behaviors but prevents us from using an automatic model checker to verify the models. We can remedy this situation partially by using a theorem prover with support for higher order logic and a theory for real numbers, like Isabelle [79], to perform the proofs mechanically. Another solution could be to restrict the model of hybrid statecharts so that it can be analyzed with a tool based on the hybrid automata, like Hytech [45].

# Chapter 7

# Conclusions

In this thesis we have presented different methods of analyzing UML models based on the refinement calculus and model checking. UML models are created during the whole software development process but often we have several diagrams describing the design of our system with a fair level of detail long before we start implementing our software. We can use the methods presented here to analyze and assure the quality of the models and detect some design errors before the first line of code is written.

The topics of software quality, correctness and validation have been heavily studied by the computer science community and there exist methods that enable the creation of software that is correct by construction or to verify that it satisfies some correctness requirements. However, many of these methods require as a starting point a formal specification of the software. Unfortunately, not all software developers are familiar with formal specification languages and not all of them are willing to learn or use them. Software development teams are more than often constrained by deadlines and lack of qualified personnel and do not always see how useful a specification language could be since it is not used directly to build software.

This situation is, however, changing thanks to the advent and widespread usage of UML and the other mature graphical languages such as SDL. Currently, modeling languages are part of undergraduate courses in software engineering and many software developers are now constructing software models just as another task to perform during their everyday software development process. These software models are often abstract and precise enough to be comparable to software specifications.

The challenge is now to show how to transform a model expressed in a graphical modeling language into a software specification expressed in a formal language and explain the benefits of doing so. When we find a precise mapping between a subset of a modeling language and a formal specification language

we are in fact promoting all models written in that language to formal specifications. These specifications can then be analyzed and proven correct or wrong using all the mathematical and computer tools supporting the specification method.

This thesis tackles this challenge and shows how to transform use cases and statecharts into formal specifications and shows that this formalization allows us to analyze these models.

The main problem that we must face when formalizing a language such as UML is how to validate our work. A precise formalization of UML can be useless if our formal semantics differ considerably from the intuitive understanding that the UML community has of the language. Also, a formalization will not be used if it does not cover a feature of the language that is used often. Our approach to this problem is to create many non-trivial examples that can be used both to explain or work and to allow the UML designer to validate it. At the end, validation is in the hands of the final user, in this case, the UML community.

We have not tried to formalize all diagrams and model elements of the UML language into a single formal language. We focus on behavioral diagrams since we consider that this aspect of UML has not been studied well enough yet. We have chosen the two diagrams that we consider are key for describing the behavior of a UML model. Use case diagrams are behavioral diagrams that show the system as a black box, from the point of view of a system. Statechart diagrams consider each single object in our model as a white box and describe its internal behavior in full detail.

We formalize these diagrams using two different formal methods: the refinement calculus and model checking. The refinement calculus and its contract language are well suited for describing use cases and the behavior of actors since they incorporate the concept of free will into the formalism. Model checking tools allow us to analyze automatically statecharts diagrams. This is not a surprise since both UML statecharts and most model checking tools are founded in the concept of the finite state machine. However, model checking tools are limited by the size of the state space of the model. We have shown how to use the refinement calculus again to model and analyze real-time and hybrid behaviors in statecharts that are intractable for a model checking tool.

In each case, the purpose of the formalization is clear: to allow the analysis of UML models. We show how to use the weakest precondition predicate transformer in a use case model to prove whether an actor can achieve a goal by using the system. We give formal semantics to UML state machines to show how we can use a model checker tool to verify simple temporal properties in our models. Finally, we propose a model of time for statecharts since we can show how to prove safety properties of those models.

We hope that these analysis methods will contribute to improve the quality of UML models. However, UML is a large language and its study and formaliza-

tion goes beyond the effort of a single person or research team. This is another reason why we have chosen to work with different formalisms. We expect that in the long run there will be many different verification and validation methods and tools for many different subsets and dialects of UML based on different formalisms. The UML designer will be free to choose the method or methods that better suit his model and the problem that needs to be solved.

In this thesis we have refrained from extending UML and proposing new model elements except in two situations that we found unavoidable. We add new stereotypes to represent temporal properties in statecharts and we have proposed the concept of continuous actions as a model for real-time and hybrid actions for statecharts. The first addition allows us to express temporal properties in a simple way. The second addition covers one of the big deficiencies of UML: support for real-time and hybrid systems. In both cases we give a precise explanation of the new model elements using a rigorous formalism.

In the future we plan to continue with the same line of research. We want to continue moving modeling languages towards formal methods with the intention of allowing the average software developer to profit from the vast research results produced by the formal methods community. The next step is to provide as much automation and tool support as possible, so much of the knowledge needed to analyze the models is embedded in a software tool that can be installed in the workstation of any UML designer.

# Bibliography


[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[2] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Revn, and H. Rischel, editors, *Hybrid Systems I*, volume Lecture Notes in Computer Science 736, pages 209–229. Springer-Verlag, 1993.

[3] M. Awad, J. Kuusela, and J. Ziegler. *Octopus: Object-Oriented Technology for Real-Time Systems*. Prentice Hall, 1996.

[4] R. J. Back. Correctness preserving program refinements: Proof theory and applications. *Mathematical Centre Tracts*, vol. 131, Mathematical Centre, Amsterdam, 1980.

[5] R. J. Back and C. Cerschi. Modeling and verifying a temperature control system using hybrid action systems. In *Proc. of the 5th Int. Workshop in Formal Methods for Industrial Critical Systems*, 2000.

[6] R. J. Back, J. Grundy, and J. von Wright. Structured calculational proof. Technical Report 65, Turku Center for Computer Science, November 1996.

[7] R. J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd Symp. on Principles of Distributed Computing*, volume Lecture Notes in Computer Science 873, pages 131–142. ACM SIGACT-SIGOPS, 1983.

[8] R. J. Back, L. Petre, and I. Porres. Analysing UML use cases as contracts. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.

[9] R. J. Back and K. Sere. Stepwise refinement of parallel algorithms. In *Science of Computer Programming 13*, pages 133–180, 1991.

[10] R. J. Back and K. Sere. From action systems to modular systems. In *Formal Methods Europe (FME '94)*, volume Lecture Notes in Computer Science 873, pages 1–25. Springer-Verlag, 1994.

[11] R. J. Back and J. von Wright. Doing high school mathematics carefully. Technical Report 140, Turku Center for Computer Science, November 1997.

[12] R. J. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction*. Springer-Verlag, 1998.

[13] R. J. Back and J. von Wright. Contracts, games and refinement. Technical report, Turku Centre for Computer Science, November 1997.

[14] A. Beugnard, J. M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[15] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[16] G. Booch. Describing software design in Ada. *SIGPLAN Notices*, 16(9):42–47, September 1981.

[17] G. Booch. Object oriented design. *Ada Letters*, I(3):64–76, March-April 1982.

[18] G. Booch. Object oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[19] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[20] M.S. Branicky. General hybrid dynamical systems: modeling, analysis and control. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume Lecture Notes in Computer Science 1066, pages 186–200. Springer-Verlag, 1996.

[21] M. Christerson and P. Jonsson. Tutorial 1: Object-oriented software engineering. Tutorial Notes, Presented at OOPSLA'95, 1995.

[22] Tony Clark. Type checking UML static diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA,*

*October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*, pages 503–517. Springer, 1999.

[23] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[24] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, Prentice Hall, 1990.

[25] D. Coleman, S. Bodoff (Contributor), and P. Arnold (Contributor). *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1993.

[26] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining uml family members using prefaces. In IEEE, editor, *Procceedings of TOOLS 32*, 1999.

[27] O. J. Dahl and K. Nygaard. Simula begin. Technical report, Norsk Regnesentral, Oslo, 1967.

[28] C. H. Damm, K. M. Hansen, M. Thomsen, and M. Tyrsted. Creative object-oriented modelling: Support for intuition, flexibility, and collaboration in CASE tools. In Elisa Bertino, editor, *ECOOP-2000 – Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 27–43, 2000.

[29] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[30] B. P. Douglass. *Real-time UML: developing efficient objects for embedded system*. Addison Wesley, 1998.

[31] D. D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[32] J. Rumbaugh et.al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[33] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society's Symposium in Applied Mathematics*, volume 19, pages 19–31, 1967.

[34] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Object Technology Series, 1999.

[35] R. France, J.M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the semantics of UML type structures with Z. In H. Bowman and J. Derrick,

editors, *Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*. Chapman and Hall, London, 1997.

[36] A. Romanovsky G. Di Marzo Serugendo, N. Guelfi and A. F. Zorzo. Formal development and validation of java dependable distributed systems. In *Proceedings of ICECCS'99 - Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 1999.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[38] C. Gane and T. Sarson. *Structured Systems Analysis*. Prentice-Hall, 1979.

[39] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, pages 72–81, September 1994.

[40] H. Giese. Contract-based component system design. In *Proceedings of HICSS'33 - The 33-rd Hawai'i International Conference on Systems Sciences*. IEEE press, 2000.

[41] A. Goldberg and A. Kay. *Smalltalk 72 Instruction Manual*. Xerox PARC, 1976.

[42] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[43] D. Harel. On visual formalism. *Communications of ACM*, 31(5):514–530, 1988.

[44] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions of Software Engineering and Methodology*, 5(4) Oct 1996.

[45] T. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1):110–122, 1997.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[47] C. A. R. Hoare. The emperor's old clothes (1980 turing award lecture). *Communications of the ACM*, pages 75–83, 1981.

[48] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[49] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, May 1997.

[50] K. De Hondt, C. Lucas, and P. Steyaert. Reuse contracts as component interface description. In *Proceedings of WCOOP'97 - Second International Workshop on Component-Oriented Programming*, number 5 in TUCS General Publications, pages 43–49. Turku Centre for Computer Science, 1997.

[51] International Telecommunication Union. *Recommendation G.727 (12/90) - 5-, 4-, 3- and 2-bits sample embedded adaptative differential pulse code modulation (ADPCM)*, 1990.

[52] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, 1988.

[53] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1993.

[54] I. Jacobson. Object-oriented development in an industrial environment. In *Proceedings of OOPSLA'87, special issue of SIGPLAN Notices*, volume 22, pages 183–191, 1987.

[55] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[56] I. Jacobson, M. Ericsson, and A. Jacobson. *The Object Advantage: Business Process Reengineering With Object Technology*. Addison-Wesley, 1995.

[57] Soon-Kyeong K. and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 1999.

[58] Soon-Kyeong K. and D. Carrington. An integrated framework with UML and Object-Z for developing a precise specification. In *Proceedings of APSEC 2000*. IEEE Computer Society, 2000.

[59] S. Kent, A. Evans, and B. Rumpe (Eds.). UML semantics FAQ. Available at http://www.univ-pau.fr/OOPSLA99/samplewr99.pdf.

[60] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. In *Lectures on Embedded Systems*, volume Lecture Notes in Computer Science 1494, pages 4–73. Springer-Verlag, 1998.

[61] C. Kobryn. Modeling components and frameworks with UML. *Communications of ACM*, 43(10):31–38, October 2000.

[62] P. Kruchten. *Rational Unified Process*. Addison-Wesley, 1998.

[63] J. Lamping. Typing the specialization interface. In *Proceedings of OOPSLA'93*, pages 201–214. ACM SIGPLAN Notices, 1993.

[64] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, Boston, 1999. Kluwer Academic Publishers.

[65] F. Levi. *Verification of Temporal and Real-Time Properties of Statecharts*. PhD thesis, University of Pisa-Genoa-Udine, Pisa, Italy, 1997.

[66] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*, volume Lecture Notes in Computer Science 891. Springer-Verlag, 1995.

[67] J. Lilius, M. Ahlfors, M. Alanen, and H. Sara. Analyzer component framework. http://aiken.cs.abo.fi/acf/.

[68] J. Lilius and I. Porres. The semantics of UML state machines. Technical Report 273, Turku Centre for Computer Science, 1999.

[69] Hitachi Ltd. *Hitachi Single-Chip RISC Microcomputer SH7032 and SH7034 Hardware Manual*. Hitachi, 1996.

[70] Hitachi Ltd. *SH7000/7600 Serires Programming Manual*. Hitachi, 1996.

[71] T. De Marco. *Structured Analysis and Systems Specifications*. Yourdon Inc., 1978.

[72] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition edition, 1997.

[73] E. Mikk. MOCES: MOdel ChEcking Statecharts. http://www.informatik.uni-kiel.de/∼erm/MOCES.

[74] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for Statecharts. In *3rd Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196, Berlin, 1997. Springer.

[75] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing Statecharts in Promela/SPIN. In *Workshop on Industrial-Strength Formal Specifications Techniques (WIFT'98)*, Boca Raton, FL, USA, 1998. IEEE Computer Society Press.

[76] A. Nerode and W. Kohn. Models for hybrid systems: automata, topologies, controllability, observability. In R.L. Grossman, A. Nerode, A.P. Revn, and H. Rischel, editors, *Hybrid Systems I*, volume Lecture Notes in Computer Science 736, pages 317–356. Springer-Verlag, 1993.

[77] OMG. OMG Unified Language Specification. Version 1.3 , March 2000, available at http://www.omg.org.

[78] G. Övergaard and K. Palmkvist. A formal approach to use cases and their relationshops. In *UML'98 International Workshop - Beyond the Notation*, Lecture Notes in Computer Science, pages 309–317. Springer-Verlag, 1998.

[79] L. C. Paulson. The Isabelle reference manual. University of Cambridge, Computer Laboratory. http://www.in.tum.de/~isabelle.

[80] A. Pnueli. The temporal logic of programs. In *Proceedings 18th IEEE Symposium. Foundations of Computer Science*, pages pp. 46–57, 1977.

[81] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, Berlin, 1991. Springer-Verlag.

[82] I. M. Holland R. Helm and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, pages 169–180. ACM SIGPLAN Notices, 1990.

[83] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In A. Evans and S. Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*. Springer, 2000.

[84] J. Robbins and D. Redmiles D. Hilbert. Software architecture critics in Argo. In *The 1998 International Conference on Intelligent User Interfaces, San Francisco, CA, USA, January 6-9*, 1998.

[85] M. Rönkkö and A.P. Ravn. Action systems with continuous behaviour. In P. J. Antsaklis, W. Kohn, M. Lemmon, A. Nerode, and S. Sastry, editors, *Hybrid Systems V*, volume Lecture Notes in Computer Science 1567, pages 304–323. Springer-Verlag, 1999.

[86] M. Rönkkö and K. Sere. Refinement and continuous behaviour. In *Hybrid Systems: Computation and Control*, volume Lecture Notes in Computer Science 1569, pages 223–237. Springer-Verlag, 1999.

[87] D. Rosenberg and K. Scott. *Use Case Driven Object Modelling with UML.* Object Technology. Addison Wesley, 1999.

[88] H. Sara. A verification component for uml state machines. Master's thesis, Department of Computer Science. Åbo Akademi University, April 2000.

[89] E. Seidewitz and M. Stark. *Reliable Object-Oriented Software.* Sigs, 1995.

[90] E. Sekerinski and K. Sere, editors. *Program Development by Refinement.* Springer-Verlag, 1998.

[91] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, New York, 1994.

[92] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis - Modelling the Wolrd in Data.* Computer Series. Yourdon Press, Englewood Cliffs, NJ, 1988.

[93] J. Smith. *UML Formalization and Transformation.* PhD thesis, Northeastern University College of Engineering, USA, December 1998.

[94] Artisan Software. UML profile for scheduling, performance and time. Available at http://www.omg.org/uml.

[95] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[96] A. Uselton and S. Smolka. A compositional semantics for statecharts using labeled transition systems. In B. Jonsson and J. Parrow, editors, *CONCUR'94 (Concurrency theory)*, number 836 in Lecture Notes in Computer Science, Uppsala, Sweden, 1994. Springer-Verlag.

[97] A. Valmari. A stubborn attack on state explossion. In *Proc. of the 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 1990.

[98] A. Valmari. The state space explosion problem. In *Advances in Petri Nets.* Springer-Verlag, 1999.

[99] M. van den Berg, M. Verhoef, and M. Wigmans. Formal specication of an auctioning system using VDM++ and UML. In J. Fitzgerald and P. G. Larsen, editors, *VDM in Practice*, 1999.

[100] A. van der Schaft and H. Schumacher. *An Introduction to Hybrid Dynamical Systems*, volume 251 of *Lecture Notes in Control and Information Sciences*. Springer-Verlag, 2000.

[101] J. von Wright. Program refinement by theorem prover. In *6th Refinement Workshop, London*, Lecture Notes in Computer Science. Springer-Verlag, Januay 1994.

[102] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[103] W. Weck. Inheritance using contracts & object composition. In *Proceedings of WCOOP'97 - Second International Workshop on Component-Oriented Programming*, number 5 in TUCS General Publications, pages 105–112. Turku Centre for Computer Science, 1997.