Knowledge, State, Event and Time in System Modeling

Spectrums of Modeling Languages and Transformations

Jin Song Dong Roger Duke Jun Sun Hai Wang

A comprehesive textbook
for both undergradute and postgradute study
on knowledge, software and system modeling
NUS and UQ

2008

Acknowledgement

We would like to thank Yuzhang Feng, Ping Hao, Fang Yuan Li, Brendan Mahony, Shengchao Qin, Jing Sun, and other colleagues for their contribution and collaboration on various research topics covered in this book.

Contents

List of Figures

List of Tables

Summary

People gain knowledge by searching and reading the Web. In order to let softare program understand and process information on the Web, knowledge representation on the Web needs to be semantically formal. The current development of the Web ontology languages, RDF, OWL and SWRL, is reminiscent of the early development of specification languages in software engineering communities. Indeed, from the expressiveness point of view, Web ontology languages are subsets of Alloy, UML/OCL, VDM, Z and Object-Z. One can futher predict that the modeling languages for capturing the behaviours of the Semantic Web Services and Agents can be drawn from the rich collections of software dynamic modeling techniques, i.e., Message/Live Sequence Charts, Automata, Process Algebra and Integrated Design Methods.

From software engineering point of view, before any software system can be developed, the application domain must be well understood and precisely documented. Ontology is one effective tool for domain engineering – the beginning of software life cycle. This book will present Modeling Languages Spectrums that includes some key representative modeling languages ranging from simple static Web Ontology modeling techniques to expressive dynamic integrated modeling techniques. Comparisons and transformations between those languages will be discussed. Furthermore, based on transformation approaches, the latest research results on verification and synthesis among various modeling techniques will be demonstrated.

Chapter 1

Overview

1.1 Goal and Objectives

The purpose of this book is to incrementally introduce various modeling techniques from Semantic Web and Software System Specification domains, and also to present the state of the art of the latest research and development of the various transformation approach for model visualisation, verification and synthesis.

1.2 Background and Observation

W3C (www.w3c.org) has successfully pushed XML in the main stream of software industries. Currently one of the main W3C's effort is to develop the Semantic Web [10] as the next generation of the Web in which data is given well-defined and machine-understandable semantics so that it can be processed by intelligent software services. Data are defined in terms of ontologies, which capture concepts and relationships. Ontology languages such as RDF [73] Schema provide basic vocabularies for describing Web resources. In a simplified view, knowledge is about making statements about entities in the form of a subject-predicate-object, called a triple in RDF terminology. For example, Paul-age-20 is statement about "Paul's age is 20". The more statements (triples) on a particular subject will provide more knowledge on the subjects. For example, Paul-study-Law and Paul-sonOf-Roger will give us (or program) more knowledge about Paul. To get knowledge well organized, conceptualisation and classification are necessary. For example, Paul and Roger are all from Person Class and the Person Class can be further subclassified into Student and Professor Classes. The notion of Class and SubClass are meta techiques for organizing knowledge and they are naturally the main language constructs in RDF Schema. RDF Schema can be regarded as a simple ontology language but is lacking of expressiveness. The OWL [24] languages extended RDf Schema with more richer language constructs. Recently

W3C has concentrated on developing Semantic Web Rule Languages (SWRL) [63] that further extends OWL with more expressive language constructs. The future research topics, i.e., Semantic Web Services and Agents, will certainly shift the W3C research focus from the static data modeling towards dynamic behaviour modeling. The development of the Web ontology languages, RDF, OWL and SWRL, is reminiscent of the early development of specification and design languages in software engineering communities. Knowledge update e.g. from Paul-age-20 to Paul-age-21 by a happyBirthday operation will eventually need techniques for specifying state transitions in the pre/post condition form. Indeed, from the expressiveness point of view, RDF, OWL and SWRL are subsets of Alloy [65], UML/OCL [123], VDM [67], Z [128] and Object-Z [38, 107]. In the Web context, happyBirthday operation can well be an Web service and perhaps may involve (orchestrate) possible other Web services, i.e., partyInvitation, restaurantResevation and taxiBooking. One can further predict that the modeling languages for capturing the behaviours of the Semantic Web Services and Agents can be drawn from the rich collections of formal behavioural modeling techniques, i.e., state machines (e.g. statechart[49], timed automata [2]), process algebra (e.g. CSP [59], Pi-calculus [89]) and (even possibly) integrated formalisms (e.g. Circus [127], TCOZ [80]).

1.3 Modeling Language Spectrum

Given the wide range of modeling techniques from the Semantic Web and Software Engineering communities, one main part of this book will present concise Modeling Languages Spectrums that includes key representative modeling languages ranging from simple static data Web ontology modeling techniques to expressive dynamic integrated modeling techniques. Comparisons between those languages will be discussed.

1.4 Software Reasoning Tools for Semantic Web

Semantic Web not only emerges from the Knowledge Representation and the Web Communities, but also brings the two communities closer together. The Software Engineering community can also play an important role in the Semantic Web development. Modeling and verification techniques can be useful at many stages during the design, maintenance and deployment of Semantic Web ontology. We believe Semantic Web will be a new research and application domain for software modeling techniques and tools. For example, the current Web Ontology reasoners such as FaCT [62] and RACER [48] have been developed to reason ontologies with a high degree of automation. However, complex ontology-related properties may not be expressible within the

current web ontology languages, consequently they may not be checkable by RACER and FaCT. Through transformation techniques, we demonstrate that the software engineering techniques and tools, i.e., Z/EVES [100] and Alloy Analyzer(AA) [66] can complement the ontology tools for checking Semantic Web documents [32, 33, 121].

1.5 Analysing Complex Models Through Transformations

Transformation techniques between different languages are useful and important not only for checking Web ontology through software modeling languages and tools, but also for checking and analysing complex software design models. For example, integrated formal modeling techniques are well suited for presenting more complete and coherent requirement models for complex systems. However, the challenge is how to analyze and check these models with tools support. We believe one effective approach is to project (transform) the integrated requirement models into multiple domains, then to use existing specialized tools in those domains to perform the checking and analyzing tasks. This book will also address some of the research issues in the transformation techniques between different modeling techniques [31, 30].

1.6 Synthesis

Jun writes a brief paragraph here

1.7 Possible Examples and Case Studies

- Family Relationships
- Extended Timed Buffer Systems
- Various Puzzles
- Light Control Systems
- Multi-lifts Control Systems

1.8 Chapter Outlines

The structure of the book is divided into 3 Parts. Part one *Ontology, Relational and State-based Formalisms* includes 5 chapters

1.8.1 Chapter 2 XML and RDF

Graph form of Ontology in UML Class Diagram

1.8.2 Chapter 3 OWL and SWRL

1.8.3 Chapter 4 Alloy

Semantics of OWL in Alloy

1.8.4 Chapter 5 Z

Semantics of OWL in Z

1.8.5 Chapter 6 Object-Z

Part two *Charts, Automata and Event-based Formalisms* includes 4 chapters.

1.8.6 Chapter 7 Message and Live Sequence Charts

1.8.7 Chapter 8 CSP and Timed CSP

1.8.8 Model Checking LSC in CSP/FDR

1.8.9 Synthesis from LSC

1.8.10 Synthesis from Object-Z with Temporal Logic Invariants

1.8.11 Chapter 9 Automata and Timed Automata

Timed Patterns

Model Checking Timed CSP in TA tools

Part three *Intergrated Formalisms and Synthesis* inculdes ? chapters.

1.8.12 Chapter 10 TCOZ

Smith and Derrick Approach

CSPOZ Approach

Circus

Projections to TA

1.8.13 Chapter 11 OZTA

1.8.14 Chapter 12 Insert Z into LSC and Synthesis

Part I: Web Ontology, Relational and State-based Modeling Techniques

Chapter 2

XML and RDF

The Semantic Web [10] has been recognized as a promising technology that exhibits huge commercial potential, and attracts significant attention from both industry and the research community. It proposes an evolution of the current Web from a web of documents to a distributed and decentralized, global knowledge-base. The realization of the Semantic Web has facilitated the markup and manipulation of complex taxonomic and logic relations between entities published on the Web. By semantically annotating the relevant aspects of declarative Web knowledge descriptions in a machine-readable format that can facilitate logical reasoning, such knowledge descriptions become interpretable based on their meanings, rather than simply on a symbolic representation. The advantage of this is that many of the tasks involved in using Web can be (semi-) automated, for example: the Web service discovery, selection, composition, mediation, execution, monitoring, etc.
In this chapter we will present an overview about the Semantic Web vision and one of the important technologies to realize this vision – RDF.

2.1 Semantic Web overview

As a huge information space, the Web should be useful not only for human-human communication, but also allows machines to participate and help. However, nowadays most information on the Web is designed for human consumption and the structure of the data is not evident for a robot browsing the Web. There are two distinct approaches to enable the machine to automatically manipulate the information in the Web. One approach that comes from artificial intelligence is machine learning.

The machine is trained to behave like a person. However this approach is domain-dependent and requires a huge training process. The *Semantic Web* (SW) [10] approach instead develops languages for expressing information in a machine processable form. The *World Wide Web Consortium* (W3C) gives the following definition for the Semantic Web:

> The Semantic Web is an extension of the current Web in which information is given a well-defined meaning, better enabling computers and people to work in cooperation.

The Semantic Web is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. With the SW, the machine can potentially do many complicate tasks which currently can only be performed manually. For example, user could directly send the following request to web agent – "Book me a holiday next weekend somewhere warm, not too far away, and where they speak Chinese or English". The Web agent will be able to 'understand' the request and perform it for the user.
SW extends the current World Wide Web by embedding knowledge in the form of semantic annotations within Web pages. The inclusion of content with a well-defined meaning has meant that documents and resources published on the web can be more easily accessible by computer programs, thus better enabling computers and people to work in cooperation.
HyperText Markup Language (HTML), the current Web data standard, is aimed at delivering information to the end user for human-consumption (e.g. display this document). XML [119] is aimed at delivering data to systems that can understand and interpret the information. XML is focused on the syntax (defined by the XML schema [120] or DTD) of a document and it provides essentially a mechanism to declare and use simple data structures. However there is no way for a program to actually understand the knowledge contained in the XML documents.
The *Resource Description Framework* (RDF)[73], one of the important technologies to enable the SW vision, is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. RDF Schema [**?**] provides the basic vocabulary to describe RDF vocabularies, and can be used to define properties and types of the web resources. In this respect, RDF Schema plays a similar role to XML Schema; XML Schema gives specific constraints on the structure of an XML document, while RDF Schema provides information about the interpretation of the RDF statements.
In the following sections, we will introduce these languages in more details.

2.2 HTML and XML

By now, HTML and XML are widely known in the WWW community and are the basis for a rapidly growing number of software development activities. In this section, we briefly discuss HTML and XML, in order to keep this book self-contained.
HTML is the predominant markup language for Web pages. It was designed to describe the structure of text-based information in a document – by adding markups to certain text as links, headings, paragraphs, lists, and so on. An HTML document can be viewed by using 'browsers', which 'display' the document based on its markups. There exists many different 'browsers', such as *Internet Explorer*, *Firefox* and *Safari*, offering different browsing abilities.
HTML is written in the form of tags, surrounded by angle brackets (<>). HTML uses tags (markups) to divide the text of a document into blocks called *elements*. These markups can be divided into two broad categories – those that define how the BODY of the document is to be displayed by browsers, and those that define information 'about' the document, such as the title or relationships to other documents. HTML can also include embedded scripting language code (such as JavaScript) that can affect the behavior of Web browsers and other HTML processors.
For example, the following is a simple HTML document which displays some basic information about this book.

```
<html>
  <head>
    <title>A  HTML DOCUMENT</title>
  </head>
  <body>
    <UL>
        <LI>Title: Spectrums of Modeling Languages and Transformations
        <LI>Author: Jin Song Dong
        <LI>Year: 2009
     </UL>
  </body>
</html>
```

Extensible Markup Language (XML) **??**, as a simple and flexible text format derived from SGML (ISO 8879), is a markup language much like HTML. XML was designed to meet the challenges of large-scale electronic publishing and exchange of a wide variety of data on the Web.
XML and HTML are both markup languages, but with different design goals. HTML is about displaying information and focuses on how data looks. By contrast, XML is about carrying information and focuses on how data is represented. XML is not a replacement for HTML.

XML documents are composed of markup and content. The most common form of markup is *elements*, delimited by angle brackets. Most elements are used to identify the nature of the content they surround. An element normally begins with a start-tag, <*element*>, and ends with an end-tag, </*element*>. For example, the following simple XML document represents some information about this book.

```
<?xml version="1.0"?>
<book>
    <title>Spectrums of Modeling Languages and Transformations</title>
    <author>Jin Song Dong</author>
    <year>2009</year>
</book>
```

There are two correctness levels for an XML document – the well-formness and validity. An XML document is *well-formed* if it conforms a set of syntax rules; e.g. if a start-tag (<>) appears without a corresponding end-tag (</ >), it is not well-formed. By definition, if a document is not well-formed, it is not XML. This means that there is no such thing as an XML document which is not well-formed, and XML processors are not required to do anything with such documents.
A well-formed document is *valid* only if it contains a proper document type declaration and if the document conforms the additional constraints of that declaration (such as the element sequence and nesting are valid, required attributes are provided, attribute values are of the correct type and etc.). An XML schema or DTD can be used to define the users' constrains. For example, the following XML schema describe a type of the XML book element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Book">
   <xs:complexType>
     <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="year" type="xs:gYear"/>
    </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

2.2.1 Lack Semantics in XML

The extensibility of XML allows users to create their own markup (e.g. < *Book* >) and it has been accepted as the standard for data interchange on the Web. However,

there are a few limitations that obstruct XML to be used for realizing the Semantic Web vision.
Firstly, XML documents do not embed enough semantics about the information they carry and computers can not understand the meaning of these information embedded between tags. From computational perspective, the following XML element contains the same amount of information as the one we showed in the previous section.

```
<book>
    <title>*(008*)2 $  &&^*&(33ou 3Uijkduw ae &*&(E!E </title>
    <author>^$* dfsdr dkslg</author>
    <year>dfsd</year>
</book>
```

The name of the markup seems to carry some semantics. From the element names, we may guest what information might lie "between the tags". In the book schema, it can be predicted that a book could have one title and one or more authors. Even this, from computational perspective, even tag names do not carry any extra semantics. For computers, it also makes no difference between the following XML element with the book element we showed in the previous section.

```
<?xml version="1.0"?>
<*&!@!>
    <&a@!>Spectrums of Modeling Languages and Transformations </&a@!>
    <wert*!>Jin Song Dong</wert*!r>
    <qwe>2009</qwe>
</*&!@!>
```

XML is a way to standardize data formats and many people treat it as just the next level of data above the character level, which has been standardized on such similarly unglamorous technologies as ASCII and Unicode. However, we can not underplay the importance of XML. XML, as a data-format standard, provides an infrastructure for many glamorous technologies, including many Semantic Web technologies.

2.3 Resource Description Framework (RDF)

Resource Description Framework (RDF) [73], as one of the key technologies under the auspices of the World Wide Web Consortium (W3C) to enable the Semantic Web vision, is designed on top of XML, where XML is used as a common syntax for the exchange and processing of metadata. RDF is a language for representing, exchanging and reusing knowledge information about in the Web. It is designed to overcome some of the obstructions XML has.

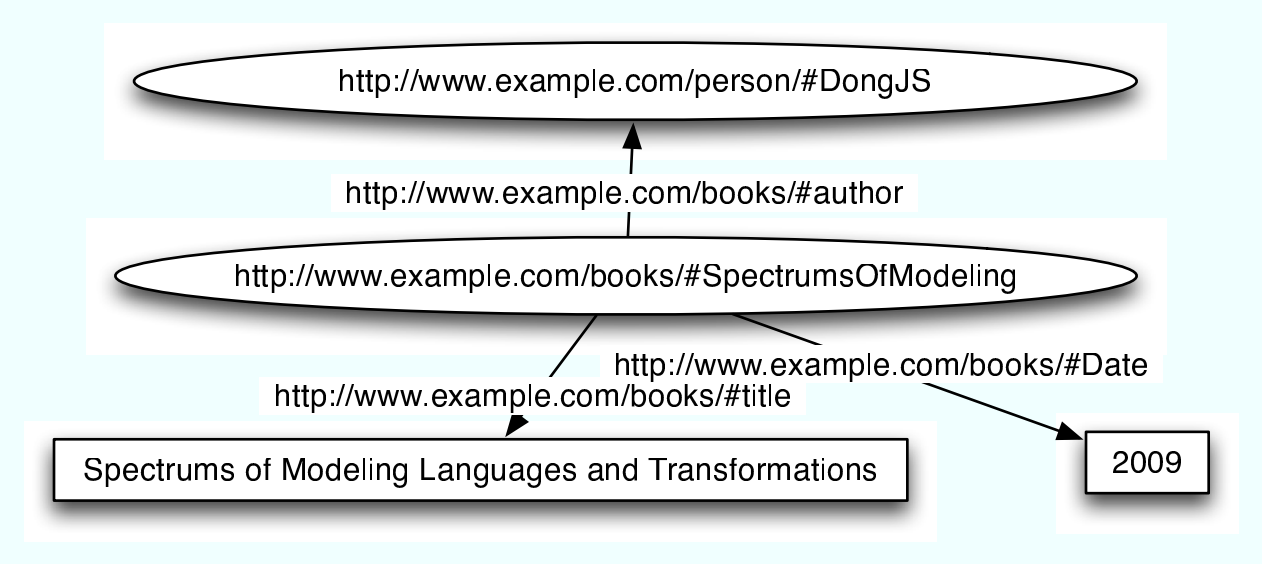


Figure 2.1: An RDF graph example

As being discussed before, XML exchanges data by defining *how* to represent it, i.e. by defining the document syntax such as the order of elements. Different users may have different ways to express data. RDF simplifies this by focusing on defining *what* data to be represented. It is no more than a way to express and process a series of simple assertions.

2.3.1 The RDF Data Model

RDF provides a simple model for describing resources in the World Wide Web. A resource can be anything that has identity. RDF identifies things using Web identifiers (called *Uniform Resource Identifiers* (URIs)), and describes resources by representing simple statements about resources in a unified way (in terms of simple properties and property values). The RDF data model is essentially a graph representation of knowledge, where nodes and arcs representing the resources, and their properties and values. The RDF graph (Figure 2.1) represents a group of statements as "there is a book identified by `http://www.example.com/books/#SpectrumsOfModeling`, whose title is 'Spectrums of Modeling Languages and Transformations', whose author is a person identified by `http://www.example.com/person/#DongJS`, and it is written in 2009". In this figure, ellipses represent Web resources, rectangles represent data values and arrows represent properties.
The RDF data model is based upon the idea of making statements about resources in

the form of *subject-predicate-object* expressions, where the subject denotes the thing the statement is about, and the predicate denotes the property or characteristic of the subject that the statement specifies and the object identifies the value of that property. In the above example, one statement could be rewritten in English as:

> The book `http://www.example.com/books/#SpectrumsOfModeling` has a *title* which is *Spectrums of Modeling Languages and Transformations*.

RDF's simplicity brings it more power. The information is represented as a set of statements. Unlike XML, users only need to focus on what information they want to represent, rather than how to represent it. The advantage of RDF is that it is readily extensible with schemas that are also machine-readable, bringing about an unprecedented level of automation. This facilitates to realize the Semantic Web vision, as to enable machine to perform tasks like resource discovery and data processing automatically.
RDF also has an XML-based syntax for exchanging these graphs on Web. The following shows a part of RDF model (Figure 2.1) in XML syntax. RDF processors can convert it to the RDF triples.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description
         rdf:about="http://www.example.com/books/#SpectrumsOfModeling">
            <http://www.example.com/books/#title>
             Spectrums of Modeling Languages and Transformations
            </http://www.example.com/books/#title>
   </rdf:Description>

   <rdf:Description
         rdf:about="http://www.example.com/books/#SpectrumsOfModeling">
            <http://www.example.com/books/#author
                 rdf:resource="http://www.example.com/person/#DongJS"/>
   </rdf:Description>
</rdf:RDF>
```

2.4 RDF Schema

RDF describes resources with statements of *subject*, *property*, and *value*. In addition, RDF also needs a way to define application-specific classes and properties, as different users might use different terms to represent a common knowledge in their statements. Application-specific classes and properties must be defined using extensions to RDF.

One such extension is *RDF Schema* (RDFS) [120]. RDF Schemas are used to declare vocabularies, the sets of semantics property-types defined by a particular community. RDF schemas also define the valid properties in a given RDF description, as well as any characteristics or restrictions of the property-type values themselves.

2.4.1 Class and Subclass

Resources may be divided into groups called classes (declared by *rdfs:Class*) and the members of a class are known as instances of the class. Classes are themselves resources. The *rdf:type* property may be used to state that a resource is an instance of a class. For example, the following RDF schema definition declares a class *book* and one of its instances *SpectrumsOfModeling*.

```
<?xml version="1.0"?>
<rdf:RDF  xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xml:base=  "http://www.example.com/books#">

     <rdfs:Class rdf:ID="Book"/>
     <rdf:Description rdf:ID="SpectrumsOfModeling">
          <rdf:type rdf:resource=#Book/>
     </rdf:Description>
</rdf:RDF>
```

In RDFS schema, classes can also be organized into superclass-subclass hierarchy by using *rdfs:subClassOf* construct. Subclasses are more specific ('are subsumed by') than their superclasses. Every instance of a subclass is also an instance of its superclasses. For example, we can model that 'Every *Book* is a Publication' as following.

```
     <rdfs:Class rdf:ID="Publication"/>
     <rdfs:Class rdf:ID ="Book">
          <rdfs:subClassOf rdf:resource="#Publication"/>
     </rdfs:Class>
```

This means that 'the members of the class *Publication* include all members of the class *Book*' and '*Book* is subsumed by *Publication*'. These statements have same meaning.

2.4.2 Property, Domain and Range

Similarly RDFS allows users to organize properties into hierarchy. The *rdfs:subPropertyOf* construct is used to state that one property is a subproperty of another. If a property

P is a subproperty of property Q, then all pairs of resources which are related by P are also related by Q.
Properties may have a *domain* and a *range* specified using *rdfs:domain* and rdfs:range. They are used to restrict the set of resources that may have a given property (the property's domain) and the set of valid values for a property (its range). For example, the property *author* would probably link individuals belonging to the class *Publication* to individuals belonging to the class *Person*.

```
<rdf:Property rdf:about="http://www.example.com/books/#author">
        <rdfs:range
               rdf:resource="http://www.example.com/books/#Publication"/>
        <rdfs:domain
               rdf:resource="http://www.example.com/books/#Person"/>
 </rdf:Property>
```

2.5 Conclusion

In this chapter we present an overview about the Semantic Web vision and two important Semantic Web technologies – RDF and RDFS. In next chapter, we will discuss another important concept for Semantic Web – Web ontologies.

Chapter 3

OWL and SWRL

With the arrival of the Semantic Web and the widespread focus on information sharing in sciences, "ontologies" have come to the center of attention. "Semantics and Metadata with everything" - E-Research, Web Services [**?**] , Woundersrkflows [**?**], Digital Libraries [**?**], and resources such as the Gene Ontology[1] and other Open Biomedical Ontologies (OBO)[2], the National Cancer Institute Thesaurus [**?**] and the SNOMED terminology for medical applications[3]. *Web Ontology Language* (OWL) [24] is latest standard in ontology languages and is endorsed by the World Wide Web Consortium. OWL (more precisely OWL DL and OWL Lite) semantics are based on Description Logics, which are a small decidable fragment of First Order Logic. SWRL [63], also known as ORL, extends OWL by introducing rule axioms to add more expressive power to the Semantic Web, particularly with respect to what can be said about properties.

OWL and SWRL, together with RDFS we introduced in the previous chapter, are toward one end of the Modeling Languages Spectrums. They emphasis on providing a precise representation of a large scale knowledge base with automatic reasoning service by sacrificing certain expressive power.

In this chapter, we will briefly introduce the concept of *ontologies* and the important ontology languages for Semantic Web – OWL and SWRL.

[1] `http://www.geneontology.orgs`
[2] `http://www.obofoundry.org`
[3] `http://www.ihtsdo.org`

3.1 Introduction

The Semantic Web vision is designed as the future of the Web, in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. Ontologies are considered as one of the pillars of the Semantic We. In *computer science* and *information science*, an **ontology** is an explicit formal representation of a set of concepts within a domain and also the relationships between those concepts. RDF Schema we introduced in the previous chapter is an ontology language and allows us to define very simple datatyping models for RDF.
OWL [24] is the latest standard in ontology languages, which was developed by members of the World Wide Web Consortium (W3C) [4] and Description Logic community. It is designed for the Semantic Web vision. OWL takes RDF Schema a step further, by giving more in depth properties and classes and allows one to be even more expressive than with RDF Schema. More complicated inference can be derived using OWL.
Semantic Web Rule Language (SWRL) [63] is a proposal for a Semantic Web rules-language. It extends OWL even further by combining of the OWL DL and OWL Lite sublanguages of the OWL with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language.

3.2 OWL Ontologies

Like many other existing Ontology formalisms [**?**, **?**] and RDFS, an OWL ontology consists of classes, properties and individuals. *Classes* are interpreted as sets of objects that represent the individuals in the domain of discourse. *Properties* are binary relations that link individuals and are represented as sets of ordered pairs that are subsets the cross product of the set of objects. In addition, OWL also provides new facilities. It has a richer set of operators - e.g. intersection, union and negation. It is based on a different logical model which makes it possible for concepts to be 'defined' as well as described. Complex concepts can therefore be built up in denitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner which can check whether or not all of the statements and denitions in the ontology are mutually consistent and can also recognize which concepts t under which definitions. We will discuss more details about these features of OWL in this section.

[4] `http://www.w3.org`

3.2.1 The Three Sublanguages of OWL

OWL ontologies may be categorized into three species or sub-languages: *OWL-Lite*, *OWL-DL* and *OWL-Full* with increasing expressiveness. *OWL-Lite* is the least expressive sub-language and supports the classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. *OWL-Full* is the most expressive sub-language. For example, in *OWL-Full* a class can be treated simultaneously as a collection of individuals and also as an individual in its own right. However, it is theoretically impossible to develop an automatic reasoning software which could support complete reasoning for every feature of *OWL-Full*. The expressiveness of *OWL-DL* falls between that of *OWL-Lite* and *OWL-Full*. *OWL-DL* may be considered as an extension of *OWL-Lite* and *OWL-Full* may be considered as an extension of *OWL-DL*. *OWL-DL* provides users the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). All OWL language constructs can be used in *OWL-DL* under certain restrictions, such as the disjointness between classes and individuals.
Essentially, every legal *OWL-Lite* ontology is a legal *OWL-DL* ontology and every legal *OWL-DL* ontology is a legal *OWL-Full* ontology. Specific communities and users can choose the most suitable specie of OWL for their usage in terms of needs of expressiveness and reasoning services.
This book only focuses on *OWL-DL*, the most widely used OWL species. unless explicitly stated otherwise, when we say OWL, it refers to OWL-DL.

3.2.2 Basic Elements of OWL Ontology

Same as RDFS, an OWL ontology also consists of classes, properties and individuals. This section presents some of the essential language components related these elements.

Class

In OWL, *classes* (the construct *owl:Class*) are interpreted as sets of objects that represent the individuals in the domain of discourse. This interpretation is slightly different from users' common understanding of *classes* in object-oriented programming [**?**] or Frames ontology formalisms [**?**], where *classes* are used as blueprints or templates to create objects. For example, in OWL, the class *Animal* would contain all the individuals that are animals in our universe. Classes can be organized into *taxonomy* (superclass-subclass hierarchy) using *rdfs:subClassOf* construct. Subclasses are more specific ('are subsumed by') than their superclasses. Every instance of a subclass is also an instance of its superclasses. For example, we can say that the class *Male*

is a subclass of the class *Animal* (or *Animal* is a superclass of *Male*). This means that 'every *Male* is an *Animal*', 'the members of class *Animal* include all members of the class *Male*' and '*Male* is subsumed by *Animal*'. These statements have same meaning. The subclass relation is transitive. For example, if *Man* is a subclass of *Male* and *Male* is a subclass of *Animal*, then *Man* is a subclass of *Animal*.
OWL predefines the class *owl:Thing* to denote all the individual in the domain that we are interested in. Each user-defined class is implicitly a subclass of *owl:Thing*. Similarly the class *owl:Nothing* is defined to denote the empty class and each user-defined class is a superclass of *owl:Nothing*.
Unsatisfiable classes An OWL class could be deemed to be *unsatisfiable* if, because of its description, it cannot possibly have any instances. For example, the class *AnimalPlant* would be found to be unsatisfiable if it was asserted to be a subclass of the intersection class $Animal \sqcap Plant$, and *Animal* and *Plant* are disjoint from each other. The class *AnimalPlant* could never have any instances in *any* model and would therefore be flagged by a reasoner as being unsatisfiable. We will discuss more on the formal meaning of unsatisfiable classes and reasoners in later this chapter.

Individual

Individuals represent objects in our universe of things. An individual is minimally introduced by declaring it to be a member of a class.
An important difference between OWL and many other knowledge modeling formalisms, like UML, Frames, is that OWL does not use the *Unique Name Assumption* (UNA). UNA means that by default, different names refer to different things. Such an assumption is invalid in OWL: different names (URI references) could refer to the same thing and things referenced by different URIs are might assumed to be the same unless they could be proven to be different. There is no UNA in OWL because it has been designed for an open environment of the next generation of the World Wide Web, the Semantic Web. On the Web, which is a highly open and distributed environment, the UNA is very restrictive. The Semantic Web should allow definitions to be extended or specialised by any number of users. For example, the same person can be referred to in many different ways (i.e. with different URI references) depending on the referrer's focus or application (work-related, personal, image-related etc.). Unless an explicit statement is made that two URI references refer to the same or different individuals, OWL tools should assume either situation is possible.
For example, the following fragment of ontology models that two individuals, *Simba* and *Mufasa*, are the lions (suppose that *Lion* is a subclass of *Animal*) living in *Pride Lands*, which is an instance of the class *Place*. If with UNA, we can imply that *Simba* and *Mufasa* are two different loins and conclude that *Pride Lands* has at least two different lions. However, an OWL reasoner will assume that *Simba* and *Mufasa* could be the same instances unless we explicitly assert otherwise. Therefore, without an

explicit assertion that *Simba* and *Mufasa* are different (using the OWL *differentFrom* statement), we can conclude only that *Pride Lands* has at least one lion. We cannot infer that *Pride Lands* have at least two loins, because *Simba* and *Mufasa* could be two different names referring to the same lion.

```
<owl:Class rdf:ID="Lion">
    <rdfs:subClassOf rdf:resource="#Animal" />
</owl:Class>
<owl:Class rdf:ID="Place"/>
<owl:ObjectProperty rdf:ID="liveIn"/>
<Place rdf:ID="PrideLands"/>

<Lion rdf:ID="Simba">
    <liveIn rdf:resource="#PrideLands"/>
</Lion>
<Lion rdf:ID="Mufasa">
    <liveIn rdf:resource="#PrideLands"/>
</Lion>
```

In order to state that *Simba* and *Mufasa* are different individuals, we have to explicitly assert the following statement.

```
<Lion rdf:ID="Simba">
  <owl:differentFrom rdf:resource="#Mufasa"/>
</Lion >
```

Property

In OWL, *properties* are binary relations that link individuals or data values. There are three distinguished types of properties as:

object properties, which are relationships between two individuals (instances of two OWL classes),

datatype properties, which are relationships between instances of classes and RDF literals and XML Schema datatypes (such as Integer, String),

annotation properties, which can be used to add information (metadata - data about data) to classes, individuals and object/datatype properties. The annotations are mainly used to provide some human readable information about the OWL entities and that information are generally ignored by OWL reasoners.

For instance, in the previous example, the property *liveIn* links the individual *Simba* and *PrideLands*.
In OWL, just as classes, properties may also have sub properties, so that it is possible to form hierarchies of properties. For example, the property *hasFather* can be defined as a more specialized property of *hasParent* as follows.

```
<owl:ObjectProperty rdf:ID="hasFather">
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
</owl:ObjectProperty>
```

From the above definition, we can conclude that if *Simba* has the father as *Mufasa*, it also has Mufasa as a parent.
As we have introduced in the previous chapter, one can define *domain* and *range* axioms for a property.
Properties can be limited to have a unique value and we call these properties as *functiona*l properties. Properties can also be either *transitive* or *symmetric*. These different property characteristics will be explained more in Chapter 3.2.5.

3.2.3 Property Restrictions

OWL classes fall into two main categories – named classes, as we have shown previously, and anonymous (unnamed) classes. Anonymous (unnamed) classes are formed from logical descriptions. They contain the individuals that satisfy the logical description. Anonymous classes may be sub-divided into *restrictions* (owl:Restriction) and '*logical class expressions*'. *Restrictions* act along properties, describing sets of individuals in terms of the types of relationships that the individuals participate in. The *owl:onProperty* element indicates the restricted property. We focus on restrictions in this subsection and *logical class expressions* will be discussed in the next subsection.

Value Restrictions

Universal (allValuesFrom) restrictions An *allValuesFrom* restriction (also being called *universal* restriction) defines an anonymous class that contains the individuals that for a given property *only* have relationships along this property to individuals that are members of a specified class. The '*allValuesFrom*' restriction is sometimes paraphrased as '*only*' restriction, because to say that *all* values come from a given class is the same as saying that values may *only* come from that class. For example, 'the class of individuals that only have *eat* relationships to members of *plant* class' can be represented as:

```
<owl:Restriction>
```

```
    <owl:onProperty rdf:resource="#eat"/>
    <owl:allValuesFrom rdf:resource="#Plant"/>
</owl:Restriction>
```

Note that we can not deduce from an *allValuesFrom* restriction alone that there actually is at least one value for the property, i.e., 'only' does not imply 'some'. For example, the class *rose* defined as following can be considered as a subclass of the above restriction, because that rose is plant that can not eat anything.

```
<owl:Class rdf:about="#Rose">
  <rdfs:subClassOf>
     <owl:Restriction>
         <owl:onProperty rdf:resource="#eat"/>
         <owl:allValuesFrom rdf:resource="owl:Nothing"/>
      </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

There is nothing inconsistent about a restriction that includes *allValuesFrom owl:Nothing*. It just means that, for the property *eat*, no values are allowed. Therefore, the above universal (*allValuesFrom*) restriction can be 'trivially satisfied' – i.e. satisfied by the trivial case in which there is no value at all for the property *eat*.

Existential (someValuesFrom) restriction A *someValuesFrom* restriction (also being called *existential restriction*) defines an anonymous class which contains the individuals that have *at least one* (some) relationship along a specified property to an individual that is a member of a certain class. For example, 'the set of individuals that eat *at least* some animals' can be represented as:

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#eat"/>
  <owl:someValuesFrom rdf:resource="#Animal"/>
</owl:Restriction>
```

Existential restrictions are probably the most commonly used type of restrictions in OWL ontologies. Note that one of the most common errors made by newcomers to OWL is to use universal (*allValuesFrom*) rather than existential (*someValuesFrom*) as the default qualifier. This error is pernicious because the results often appear to work initially, as an '*allValuesFrom* restriction can be 'trivially satisfied', with the problems only becoming evident later in the course of developing the ontology.

It is a good practice for ontology engineers or OWL software developers to regard existential *someValuesFrom* restrictions as the default restrictions from the beginning.

***HasValue* restriction** A *hasValue* restriction defines the set of individuals that have *at least one* relationship along a specified property to a specified individual. Hence,

an individual will be a member of such a class whenever at least one of its property values is equal to the hasValue resource.
The following example describes the class of individuals who live in the individual place referred to as PrideLands:

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#liveIn"/>
  <owl:hasValue rdf:resource="#PrideLands"/>
</owl:Restriction>
```

Restricted Cardinality Restrictions

In OWL, the *restricted cardinality restrictions* are used to describe the class of individuals that have different number of relationships with other individuals or datatype values. There are three different types of restricted cardinality restrictions that can be defined.

Minimum Cardinality Restriction (*owl:minCardinality*): It specifies that for a given property P, the minimum number (n) of P relationships that an individual must participate in. This restriction is another way of saying that the property is required to have *at least* n values for all instances of the class.

Maximum Cardinality Restriction (*owl:maxCardinality*): It specifies that for a given property P, the maximum number (n) of P relationships that an individual can participate in. This restriction is another way of saying that the property is required to have *at most* n values for all instances of the class.

Cardinality Restriction (*owl:cardinality*): It specifies that for a given property P, the exact number (n) of P relationships that an individual can participate in.

The following example describes a class of individuals that have at most two parents:

```
<owl:Restriction>
   <owl:onProperty rdf:resource="#hasParent" />
   <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">2
    </owl:maxCardinality>
</owl:Restriction>
```

3.2.4 Logical Class Expressions

OWL class extensions are sets consisting of the individuals that are members of the class. OWL also allows users to use the basic set operations to manipulate class extensions.

Intersection (*owl:intersectionOf*): The intersection of two classes M and N describes all individuals that belong to both the extension of M and N.

Union (*owl:unionOf*): The union of two classes M and N describes all individuals that belong to the extension of either M or N.

Complement (*owl:complementOf*): The complement of a class M describes all individuals that do not belong to the extension of M. It is literally the set difference between *owl:Thing* and M.

For example, the following class description describes a class for which the class extension contains both individuals from the class *Plant* and *Animal*.

```
<owl:Class>
  < owl:unionOf >
    <owl:Class rdf:about="#Plant"/>
    <owl:Class rdf:about="#Animal"/>
  </owl:unionOf >
</owl:Class>
```

3.2.5 Property Characteristics

Comparing with RDF-S, OWL allows users to enrich the meaning of properties through the use of property characteristics. The enrichment includes to put some global cardinality constraints on properties (e.g. *owl:FunctionalProperty* and *owl:InverseFunctionalP* and some logical characteristics of properties (e.g. *owl:TransitiveProperty* and *owl:SymmetricPropert*

Functional property

A property can be defined as functional (*owl:FunctionalProperty*) if for a given individual, there can be at most one individual that is related to the individual via this property. For example, the following axiom states that the *hasFather* property is functional. It means that an individual can only have *one* father. If we say that *Simba hasFather Mufasa* and also say that *Simba hasFather Scar*, then because that *hasFather* is a functional property, the fact that *Mufasa* and *Scar* must be the same individual would be inferred. If *Mufasa* and *Scar* were explicitly asserted to be different individuals (such as using *owl:differentFrom*) then the ontology would be inconsistent.

```
<owl:FunctionalProperty rdf:about="#hasFather" />
```

Inverse Functional Property

As the name suggested, if a property is declared to be inverse-functional (*owl:InverseFunctionalProperty*), then it means that its *inverse* property is *functional*. More precisely, if we say that a property P is inverse-functional, there cannot be two distinct individuals x and y which both relates to a same individual through P. For example, a property *isFatherOf*, whose inverse property is the functional property *hasFather*, can be declared as inverse functional. If we have that *Mufasa is the father of Simba* and also *Scar* is the father of Simba, then we can infer that Mufasa and Scar are the same individual.

Transitive Property

A property P can be specified as *transitive*, and this means for any individuals a, b and c, if a is related to b via property P and b is related to c via property P, then we can conclude that a is related to c via property P as well. For example, we can define a property *ancestor* as transitive. If the individual *Ahidi* is an ancestor that is *Mufasa*, and *Mufasa* is an ancestor that is *Simba*, then we can infer that *Ahidi* is an ancestor of *Simba*.

Symmetric Properties

A symmetric property is a property for which holds that if the individual x is related to y via property P, then y is also related to x via P. For example, we can define a property *isRelative* as symmetric. If the individual *Simba* is a relative of *Mufasa*, then we can infer that *Mufasa* is a relative of *Simba* as well.

3.2.6 Primitive and Defined Classes

In OWL, a class can have *necessary* conditions, *necessary and sufficient* conditions or both. Necessary conditions for a class mean that "if something is a member of this class then it is *necessary* to fulfill these conditions". All of the classes we have created so far have only the necessary conditions. By contrast, necessary and sufficient conditions mean that "if something fulfills these conditions then it *must* be a member of this class". In OWL, a class that *only* has necessary conditions is also known as a *primitive* class and a class that has necessary and sufficient conditions is known as a *defined* class. Being able to define both 'primitive' and 'defined' classes is one of the major novelties of OWL. However, understanding the difference between them is also one of the major stumbling blocks for newcomers.
Many potential OWL users are familiar with either frames systems or object-oriented programming and analysis and UML. In these formalisms, things can be described by necessary conditions but not as defined classes with sufficiency conditions. However,

in OWL, by using necessary and sufficient conditions, new concepts can be built up from existing concepts by fitting them together in definitions like blocks of Lego. The construct *owl:equivalentClass* is used to define a class. For example, the following ontology defines a subclass of *animal* – *carnivore*, which are animals that only eat animals.

```
<owl:Class rdf:about="Carnivore">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="#Animal"/>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#eat"/>
                        <owl:allValuesFrom rdf:resource="#Animal"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
```

If with only *necessary* conditions defined (i.e., using *rdfs:subClassOf*), we can only say that if something is a *carnivore*, it is *necessarily* an *animal* and it *necessarily* eats only animals. *Necessary and sufficient* conditions allow us to sufficiently determine if any (random) individual that satisfies them must be a member of a class. Suppose that we have a random individual. If we know that this individual is a member of class *animal* and also only eats animals and given the above definition of the class *Carnivore*, it is sufficient to determine that the individual is a member of the class *Carnivore*.

Because a class can be defined in terms of the set of conditions that are sufficient for an object to be identified as an instance of the class, we can infer that, based on its definition, one class is always a subclass of another. A DL reasoner can perform such inference automatically.

Suppose that we define a class *Shark* as *animals* which only eat fishes (elsewhere we defined fish as a kind of animals). We can automatically infer that *Shark* is a subclass of *Carnivore* because it satisfies all the sufficient conditions for the latter class.

'Defined' class is a very useful feature of OWL in practice. For example, we can use the reasoner to automatically compute a classification and this can be used as a mechanism for 'normalising' ontologies [**?**]. More details will be discussed in Chapter 3.2.9.

3.2.7 Open World Assumption

The biggest single hurdle to understanding OWL (and other Description Logics) is the use of *Open World Assumption* (OWA). Almost certainly, all systems that newcomers to OWL will have encountered previously use closed world reasoning with 'negation as failure' – i.e. if something cannot be found, it is assumed to be absent, e.g. databases, logic programming, constraint languages in frame systems, etc. By contrast, OWL uses open world assumption with negation as unsatisfiability - i.e. something is false only if it can be proved to contradict other information in the ontology. To better illustrate the concept, we attampt to define a class *Herbivore* as following.

```
<owl:Class rdf:about="Herbivore">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="#Animal"/>
                 <owl:complementOf>
                     <owl:Restriction>
                         <owl:onProperty rdf:resource="#eat"/>
                          <owl:someValuesFrom rdf:resource="#Animal"/>
                       </owl:Restriction>
                 </owl:complementOf>
               </owl:intersectionOf>
          </owl:Class>
      </owl:equivalentClass>
</owl:Class>
```

This definition means that a herbivore is *any* animal that, *amongst other things*, does *not* eat *any* animals. We can also define a class *Rabbit* as *animals* which, *amongst other things*, eat *Forbs* and *Grasses*. Suppose that *Forb* and Grass are subclasses of the OWL class *Plant* defined elsewhere. *Plant* and *Animal* are disjoint classes.

```
<owl:Class rdf:about="Rabbit">
       <rdfs:subClassOf rdf:resource="#Animal"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#eat"/>
                <owl:someValuesFrom rdf:resource="#Forb"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
```

```
            <owl:Restriction>
                <owl:onProperty rdf:resource="#eat"/>
                <owl:someValuesFrom rdf:resource="#Grass"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
```

Given the definitions so far, *Rabbit* does not classify as *Herbivore*. There is nothing in *Rabbit*'s definition that makes it contradictory to eat some animals.
For example we can define *FunnyRabbit* as a subclass of *Rabbit*. A FunnyRabbit is a kind of rabbit which also eat *some* lions. The class *FunnyRabbit* has nothing inconsistent.

```
<owl:Class rdf:about="FunnyRabbit">
       <rdfs:subClassOf rdf:resource="#Rabbit"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#eat"/>
                <owl:someValuesFrom rdf:resource="#Lion"/>
            </owl:Restriction>
        </rdfs:subClassOf>
 </owl:Class>
```

The above definition of *Rabbit* was inadequate because of the open world assumption adopted by OWL. This should be clear if we take a closer look at the paraphrases about the *Rabbit* class, which is defined as *animals* which, *amongst other things*, eat Forbs and Grasses. The phrase '*amongst other things*' is specifically added to capture the open world assumption implicit in all OWL expressions, where everything is permitted until it is prohibited. In order to capture the intuitive understanding of that rabbits only east grasses and forb (let's assume that is the case), we have to revise the definition of the class *Rabbit* as following.

```
<owl:Class rdf:about="Rabbit">
       <rdfs:subClassOf rdf:resource="#Animal"/>
       <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#eat"/>
                <owl:someValuesFrom rdf:resource="#Forb"/>
            </owl:Restriction>
```

```
    </rdfs:subClassOf>
    <rdfs:subClassOf>
         <owl:Restriction>
             <owl:onProperty rdf:resource="#eat"/>
             <owl:someValuesFrom rdf:resource="#Grass"/>
         </owl:Restriction>
    <rdfs:subClassOf>
         <owl:Restriction>
             <owl:onProperty rdf:resource="#eat"/>
             <owl:allValuesFrom>
                 <owl:Class>
                     <owl:unionOf rdf:parseType="Collection">
                         <rdf:Description rdf:about="#Forb"/>
                         <rdf:Description rdf:about="#Grass"/>
                     </owl:unionOf>
                 </owl:Class>
             </owl:allValuesFrom>
         </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

The final restriction is known as a 'closure restriction' or 'closure axiom' because it closes off the possibility of further additions for a given property. '*allValuesFrom*' can be paraphrased as '*only*', because to say that *all* values come from a given class is the same as saying that values may *only* come from that class. In our restriction, it means that rabbits can only eat grasses or forb.

3.2.8 Description Logics and OWL

From a formal point of view, OWL-DL can be seen to be equivalent to a very expressive Description Logic (DL) [**?**]. Description Logic is a small decidable fragment of first order logic (FOL) designed specially for knowledge representation. In this section, we present a brief introduction to DL.

Description Logic history

The **D**escription **L**ogic (DL) [**?**] is an important powerful class of logic-based knowledge representation languages. The DL is used to represent and to reason about terminological knowledge and it was evolved from two knowledge representation formalisms Frames and Semantic Networks. Frames developed by Minsky [**?**] are record-like data structures for representing stereotyped situations and objects. Attached to

each frame is all the information necessary for treating a situation, which may include information about how to use the frame, information about what one can expect to happen next and information about what to do if these expectations are not confirmed and etc. Semantic Network, develop after the work of Quillian ([**?**], is a graph-based representation formalism to capture the semantics of nature language. The common problem of both Frames and Semantic Networks is the lack of formal semantics. This may lead to the result that every system behaved differently from the others. In response to this problem, the researchers tried to develop knowledge representation languages equipped with a formal semantics to precisely capture its meaning independently of the underlying inference machine.

Knowledge representation in DL

A DL-system consists of two components. The first component, known as the knowledge base, provides a precise characterization of the type of the knowledge to be specified to the system. The second is the reasoning engine, which provides various inference services. The knowledge base in DL can further be divided into the *TBox*, which is concerned with classes and their subclasses, and the *ABox*, which is concerned with individuals.

TBox A TBox stores the conceptual knowledge of an application domain. It defines the intentional knowledge in the form of a terminology (reason for the term 'TBox'). The terminology consists of *concepts*, which denote sets of individuals, and roles, which denote binary relations between individuals. The DL systems can build atomic concepts and roles (concept and role names) and can also build complex descriptions of concepts and roles.

The different DL systems are distinguished by their description language used for building complex concepts and roles.

As an example of the formal representation of description logics, consider the $\mathcal{AL}$-language, introduced in [**?**], as the minimal such language that is of practical interest. It has the following syntax rule:

$$
\begin{aligned}
C, D \rightarrow\ & A \mid (\textit{atomic concept}) \\
& \top \mid (\textit{universal concept}) \\
& \bot \mid (\textit{bottom concept}) \\
& \neg A (\textit{atomic negation}) \\
& C \sqcap D (\textit{intersection}) \\
& \forall\, R.C (\textit{value restriction}) \\
& \exists\, R.\top (\textit{limited existential quantification})
\end{aligned}
$$

A concept in $\mathcal{AL}(C, D)$ must be in one of the forms listed in the right-hand side of arrow (A, T, , $\exists\, R.\top$). In $\mathcal{AL}$, negation can only be applied to atomic concepts, and

OWL Construct	DL Construct
intersectionOf	$C^1 \sqcap ... \sqcap C^n$
unionOf	$C^1 \sqcup ... \sqcup C^n$
complementOf	$\neg C$
toClass	$\forall P.C$
hasClass	$\exists P.C$
hasValue	$\exists P.\{x\}$
minCardinalityQ	$\leq nP.C$
maxCardinalityQ	$\geq nP.C$
cardinalityQ	$= nP.C$

Table 3.1: OWL and DL constructs

only the top concept is allowed in the scope of an existential quantification over a role. For example, $\neg \exists R.\top$ and $\exists R.A$ are not valid class concept in $\mathcal{AL}$. For example, suppose that *Animal* and *Female* are atomic concepts. The $Animal \sqcap Female$ and $Animal \sqcap \neg Female$ are $\mathcal{AL}$-concepts describing, those animals that are female and those that are not female.
The DL language $\mathcal{FL}^-$ is a sublanguage of $\mathcal{AL}$ by disallowing atomic negation. $\mathcal{FL}_0$ is a sublanguage of $\mathcal{FL}^-$ by disallowing existential quantification. The $\mathcal{AL}$ can be extended to $\mathcal{ALU}$, $\mathcal{AL}\varepsilon$, $\mathcal{ALN}$ and $\mathcal{ALC}$ if the union of concepts, full existential qualification, number restriction and negation of arbitrary concepts is allowed accordingly. Particularly important DL variant is $\mathcal{SHOIN}$(D), as it is the logic base for OWL (more precisely OWL DL). $\mathcal{SHOIN}$(D), is similar to the well known $\mathcal{SHOQ}$(D) [**?**] description logic, which augments the $\mathcal{ALC}$ with number restrictions, role hierarchies, transitively closed roles, qualified number restrictions, individuals and concrete datatypes, but is extended with inverse roles (I) and restricted to unqualified number restrictions (N). Allowing more concept constructs makes a DL language more expressive, but more difficult and complex to reason about. Table 3.1 summarize some of the OWL elements and their corresponding DL constructs in $\mathcal{SHOIN}$(D).

ABox

An *ABox* contains extensional knowledge about the domain of interest. It introduces the assertional knowledge (reason for the term 'ABox') (world description). Whereas TBoxes restrict the set of possible words, ABoxes allow us to describe a specific state of the world by introducing individuals (or instances) together with their properties. In the Abox, knowledge can be divided into a concept assertion, which states an individual is a member of concept (in the form C(a)), and a role assertion with a pair of individuals (in the form `R(a, b)`). When we say an ABox `A` is defined with respect to a Tbox, the concept description in `A` may contain defined names of TBox.
For example *Simba* and *PrideLands* are individuals. The *Lion(Simba)* and *Place(PrideLands)* describe that Simba is an instance of the class Lion and PrideLands is an instance of

the class Place. *liveIn(Simba, PrideLands)* describes that *Simba* and *PrideLands* are related by the role *liveIn*.

Formal Semantics

We use $\mathcal{AL}$-language as an example to show how the formal semantics can be defined for a DL Language. The formal semantics of DL also serve as the formal foundation for OWL. In order to define a formal semantics of $\mathcal{AL}$-concepts, we consider *interpretation* I first. An interpretation I is a pair $I = (\Delta^I, \bullet^I)$ consisting of a non-empty set Δ^I (called the domain) and an *interpretation* function $\bullet^I$) that assigns to every atomic concept A a set $A^I \subseteq \Delta^I$ to every atomic role R a binary relation $R^I \subseteq \Delta^I \times \Delta^I$. The interpretation function is extended to concept description by the following inductive definitions:

$$
\begin{array}{ll}
T^I & = \Delta^I \\
\bot^I & = \varnothing \\
(C \sqcap D)^I & = C^I \sqcap D^I \\
(\forall\, R.C)^I & = \{a \in \Delta^I \mid \forall\, b.(a,b) \in R^I \rightarrow b \in C^I\} \\
(\exists\, R.T)^I & = \{a \in \Delta^I \mid \exists\, b.(a,b) \in R^I\}
\end{array}
$$

For example, two concepts C, D are equivalent, and write $C \equiv D$, if $C^I = D^I$ for all interpretation I.
A knowledge representation system based on DLs is able to perform specific kinds of reasoning. Implicit knowledge can be made explicit through inferences. The standard kinds of reasoning tasks are presented as follows. Let $\mathcal{T}$ be a Tbox and $\mathcal{A}$ be a Abox respected with T.

Satisfiability A concept C is satisfiable with respect to $\mathcal{T}$ if there exists a model I of $\mathcal{T}$ such that C^I is nonempty. In this case we say also that I is a model of C.

Subsumption A concept C is subsumed by a concept D with respect to $\mathcal{T}$ if $C^I \subseteq D^I$ for every model I of $\mathcal{T}$. In this case we write $C \subseteq_{\mathcal{T}} D$ or $\mathcal{T} \models C \subseteq D$.

Equivalence Two concepts C and D are equivalent with respect to $\mathcal{T}$ if $C^I = D^I$ for every model I of $\mathcal{T}$. In this case we write $C \equiv_{\mathcal{T}} D$ or $\mathcal{T} \models C \equiv D$.

Disjointness Two concepts C and D are disjoint with respect to $\mathcal{T}$ if $C^I \cap D^I = \varnothing$ for every model I of $\mathcal{T}$.

Instantiation Instantiation is a reasoning task which tries to check if an individual is an instance of a class. An individual a is an instance of C if $a^I \in C^I$ for every model I of $\mathcal{A}$ respected with $\mathcal{T}$.

Description Logic and FOL

The basic DL is considered as a fragment of first-order logic. We use L^k to denote first order predicate logic over unary and binary predicates with at most k variables and we use C^k to denote first order predicate logic over unary and binary predicates with at most k variables and counting quantifiers $\exists^{\geqslant n}$,$\exists^{\leqslant n}$. The basic DL concepts can be translated into L^2 formulae or C^2 if the number restriction is allowed. L^2 and C^2 are known to be decidable and NExpTime-complete, so the basic DL is decidable and NExpTime-complete. Both L^2 and C^2 are far more expressive than basic DL.
Different DL languages can be extended from the basic DL language. Some of the extension can be as expressive as L^2 and some can be as expressive as L^3. For the latter case, the DL language becomes undecidable.
Besides increasing the number of variables in the predicates, a certain extension of the DL makes it go beyond first order logic, e.g. including transitive closure of roles. On the whole, the DL can be considered as a subset of FOL. The reason FOL is not directly used to represent knowledge without additional restrictions is that:

- the expressive power is too high for obtaining decidable and efficient inference problems;
- the inference power may be too low for expressing interesting, but still decidable theories.

3.2.9 OWL reasoner

One of the most important distinguishing features of logic based ontology languages like OWL from other ontology formalisms is that it has formal semantics, as we showed in Chapter 3.2.8. Staying within OWL-DL, allows us to build reasoners which can make automatic inferences over our knowledge base. Reasoners can be used at development time to help users to build and manage their ontologies more easily.
Many knowledge engineers believe that building large, reusable ontologies with rich multiple inheritance by manual effort unaided by formal checks is virtually impossible. This is doubly so in any collaborative environment where work from several authors must be reconciled and integrated. In general, the use of OWL reasoners includes but is not limited to the following aspects.

Diagnosis

An OWL ontology is an engineered artifact which may inevitably contain flaws which may include:

- Logical inconsistency. We have seen examples of classes that have been found inconsistent 3.2.2. Normally, this is because that the knowledge model has been

over constrained and contains contradicting facts. If we're lucky, these contradictions might be locally defined for that class, but often they are propagated over other inconsistent classes. Manually discovering the logic inconsistence is an impossible mission for a large and complex ontology.

- Unexpected inference. Since the inference in OWL is based on the assumption that the knowledge base in incomplete, there can often be unexpected inferences caused by missing information.

- Unexpected type coercion for individuals. Apart from its asserted type, we can infer if individuals have any additional inferred types.

Using OWL reasoners directly the first kind of flaws can be automatically and efficiently identified. With tool support and additional interaction from users, the causes of the last two flaws can normally be discovered as well.

Composition

As we mentioned before, OWL allows us to build more complex concepts out of simpler concepts – it is sometimes referred to as 'Conceptual Lego'. The fundamental advantage of using reasoners is that they allow composition, i.e. they allow new concepts to be defined systematically from existing concepts. This can result in a dramatic reduction in the number of facts that have to be maintained explicitly.

Normalisation and Managing Polyhierarchies

Managing and maintaining a complex polyhierarchy of concepts is hard. Changes often need to be made in several different places, and keeping everything in step is error prone and laborious. Using a reasoner, we have developed a mechanism for 'normalisation' of ontologies [**?**]. In normalised ontologies, even the most complex polyhierarchies are built out of simple non-overlapping trees. Concepts bridging the trees are created by definitions. The relationships amongst defined concepts are maintained by the reasoner. In our animal example, if we need to add a new kind of animal, we just need to describe what kinds of food it eats, without worrying if it is a kind of *Herbivore* or *Carnivore*. Reasoner will maintain the class structure automatically.

Existing reasoners

Currently, there exists a few OWL-DL reaoners which can perform all or some of the reasoning tasks effectively. A brief summary of those tools is given as follows.

FaCT++ FaCT++[5] is a free open-source C++-based reasoner for OWL-DL with qualifying cardinality restrictions. It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and incomplete support of ABoxes (retrieval).

KAON2 KAON2[6] is a free (free for non-commercial usage) Java reasoner for OWL extended with the DL-safe fragment of SWRL. It implements a resolution-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering).

Pellet Pellet[7] is a free open-source Java-based reasoner for OWL. It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering. It supports the OWL-API, the DIG-API, and Jena interface and comes with numerous other features.

RacerPro RacerPro[8] is a commercial (free trials and research licenses are available) lisp-based reasoner for SHIQ with simple datatypes (i.e., for OWL-DL with qualified number restrictions, but without nominals). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, nRQL query answering). It supports the OWL-API and the DIG-API and comes with numerous other features.

3.3 Semantic Web Rule Language (SWRL)

Although OWL includes a relatively rich set of class constructors, the language provided for expressing properties is much weaker. SWRL [63] intends to overcome the expressive restriction of OWL properties by extending OWL with some form of 'rule language'. SWRL is based on a combination of the OWL DL and OWL Lite sub-languages of the OWL Web Ontology Language with the Unary/Binary Catalog sub-languages of the Rule Markup Language. SWRL introduces a high-level abstract syntax for Horn-like rules in both the OWL DL and OWL Lite sub-languages of OWL. SWRL extends OWL by also allowing rule axioms, i.e., by adding the construct:

$$axiom ::= rule$$

[5] http://owl.man.ac.uk/factplusplus
[6] http://kaon2.semanticweb.org
[7] http://clarkparsia.com/pellet
[8] http://www.franz.com/agraph/racer

A rule axiom consists of an antecedent and a consequent, each of which consists of a set of atoms which could be class membership (*C(x)*), property membership (*P(x,y)*) or individual in/equalities(*differentFrom(x,y)/sameAs(x, y)*). Informally, a rule means that if the antecedent holds (is "true"), then the consequent must also hold. A simple example of the rules could be used to express the knowledge that "if $?x1$ is a child of $?x2$ and $?x2$ is a brother of $?x3$, then $?x3$ is an uncle of $?x1$". Informally, this rule could be written as:

$$hasChild(?x2, ?x1) \wedge hasBrother(?x2, ?x3) \Rightarrow hasUncle(?x1, ?x3)$$

Unfortunately, the extension of SWRL is too expressive to be decidable [**?**]. However, several decidable sub-languages of SWRL have been identied in the literature.

3.4 Conclusion

In this chapter, we introduced two important Semantic Web ontology languages – OWL and SWRL. They are toward one end of the modeling languages spectrums with less expressive power, but having efficient reasoning support for large knowledge bases.

Chapter 4

Alloy

Alloy Overview

Alloy (developed at MIT by D. Jackson's group) is a structural modelling language based on first-order logic (a subset of Z) and specifications organised in a tree of *modules*

Signature: A signature (`sig`) paragraph introduces a basic type and a collection of relation (called field) in it along with the types of the fields and constraints on their value. A signature may inherit fields and constraints from another signature.

Function: A function (`fun`) captures behaviour constraints. It is a parameterised formula that can be "applied" elsewhere,

Fact: Fact (`fact`) constrains the relations and objects. A `fact` is a formula that takes no arguments and need not to be invoked explicitly; it is always true.

Assertion: An assertion (`assert`) specifies an intended property. It is a formula whose correctness needs to be checked, assuming the facts in the model.

Alloy Analyser (AA)

- Constraint solver with automated simulation & checking
- Transforms a problem into a (usually huge) boolean formula
- A *scope* (finite bound) must be given

Alloy Basics

`x` (a scalar), `{x}` (a singleton set containing a scalar), `(x)` (a tuple) and `{(x)}` (a relation) are all treated as the same as `{(x)}`. The relational composition (or join) and product:

```
{(X1, .., Xm, S)}.{(S, Y1,.., Yn)} = {(X1, .., Xm, Y1, .., Yn)}
{(X1, .., Xm, S)} -> {(S, Y1,.., Yn)} = {(X1, .., Xm, S, S, Y1, .., Yn)}
```

Alloy Expression Examples

```
children = ~parents
ancestors = ^parents
descendants = ~ancestors
Man = Person - Woman
mother = parents & (Person->Woman)
father = parents & (Person->Man)
siblings = parents.~parents - iden [Person]
cousins = grandparents.~grandparents - siblings - iden [Person]
```

Alloy Logical Operators

```
!F // negation: not F
F && G // conjunction: F and G
F || G // disjunction: F or G
F => G // implication: F implies G; same as !F || G
F <=> G // biimplication: F when G; same as F =>G && G => F
F => G,H // if F then G else H; same as F => G && !F => H
```

Quantifiers

```
all x: e | F
some x: e | F
no x: e | F
sole x: e | F
one x: e | F
one x:e, y:f | F
all disj x,y: e | F
```

Examples

```
// no polygamy
all p: Person | sole p.spouse
// a married person is his or her spouse's spouse
all p: Person | some p.spouse => p.spouse.spouse = p
// no incest
no p: Person | some (p.spouse.parents & p.parents)
// a person's siblings are those persons with the same parents
all p: Person | p.siblings = {q: Person | q.parents = p.parents} - p
// everybody has one mother
all p: Person | one p.parents & Woman
// somebody is everybodys ancestor
some x: Person | all p: Person | x in p.*parent
```

Module, Sig, Fact, Fun and Assert (example)

Simon (1979 song) said "One Man's Ceiling Is Another Man's Floor". Does it follow that "One Man's Floor Is Another Man's Ceiling"?

```
module CeilingsAndFloors
sig Platform {}
sig Man {ceiling, floor: Platform}
fact {all m: Man | some n: Man | Above (n,m)}
fun Above (m, n: Man) {m.floor = n.ceiling}
assert BelowToo {all m: Man | some n: Man | Above (m,n)}
run Above for 2
check BelowToo for 2
```

Man1 has no living space, is this the problem?

```
pred Geometry (){no m: Man | m.floor = m.ceiling}
assert BelowToo' {Geometry() => all m: Man | some n: Man | Above (m,n)}
check BelowToo' for 2 expect 0

//but, still have problem with an increased scope
check BelowToo for 3 expect 1
```

So, is sharing the problem?
Daniel Jacksons solution no sharing (see his 2001-1004 example version):

```
pred NoSharing() {no disj m,n: Man | m.floor = n.floor || m.ceiling = n.ceiling}
assert BelowToo'' {NoSharing() => all m: Man | some n: Man | Above (m,n)}
check BelowToo'' for 6 expect 0
check BelowToo'' for 10 expect 0
```

but, this is too restrictive and unreasonable
So, is sharing really the problem? What about proper sharing //Jin Songs correction (2005) to Daniels solution:

```
//need a living space
fact {no m: Man | m.ceiling = m.floor}
//proper sharing
fact {all disj m,n: Man|m.floor = n.floor iff m.ceiling = n.ceiling}

assert BelowToo' {all m: Man | some n: Man | Above (m,n)}
check BelowToo' for 6 expect 0
```

This is not the end of the story?, the question is: is the artist realistic?

```
sig building {
 abv: Man -> Man }
{
all m: Man | Above(m, m.abv)
}

pred showbuilding(b: building){some b.abv}

run showbuilding for 2 but 1 building

run showbuilding for 3 but 1 building

//acyclic property??
```

```
sig building {
 abv: Man -> Man }
{
all m: Man | Above(m, m.abv)
no m: Man | m in m.^abv  // acyclic property
}

pred showbuilding(b: building){some b.abv}

run showbuilding for 3 but 1 building
```

The 'acyclic' constraint will show that the artist might not be logical with the current phyical reality but could be abstract/imaginary.
Trace pattern: module util/ordering[elem]
Creates a single linear ordering over the atoms in elem. It also constrains all the atoms to exist that are permitted by the scope on elem. That is, if the scope on a signature S is 7, opening util/ordering[S] will force S to have 7 elements and create a linear ordering over those 7 elements.
The predicates and functions below provide access to properties of the linear ordering, such as which element is first in the ordering, or whether a given element precedes another.

```
module util/ordering[elem]

one sig Ord {
   first_, last_: elem,
   next_, prev_: elem -> lone elem
}{
  // constraints that actually define the total order
  prev_ = ~next_
  one first_
  one last_
  no first_.prev_
  no last_.next_
   // either elem has exactly one atom, which has no predecessor or successor...
  ((one elem && no elem.prev_ && no elem.next_) ||
    all e: elem | {
      // ...each element (except the first) has one predecessor, and...
      (e = first_ || one e.prev_)
      // ...each element (except the last) has one successor, and...
      (e = last_ || one e.next_)
      // ...there are no cycles
      (e !in e.^next_)})
  // all elements of elem are totally ordered
  elem in first_.*next_
}
// first and last
fun first (): elem { Ord.first_ }
fun last (): elem { Ord.last_ }

// return the predecessor of e, or empty set if e is the first element
fun prev (e: elem): lone elem { e.(Ord.prev_) }
```

```
// return the successor of e, or empty set of e is the last element
fun next (e: elem): lone elem { e.(Ord.next_) }

// return elements prior to e in the ordering
fun prevs (e: elem): set elem { e.^(Ord.prev_) }
// return elements following e in the ordering
fun nexts (e: elem): set elem { e.^(Ord.next_) }

// e1 is less than e2 in the ordering
pred lt (e1, e2: elem) { e1 in prevs (e2) }
// e1 is greater than e2 in the ordering
pred gt (e1, e2: elem) { e1 in nexts (e2) }
// e1 is less than or equal to e2 in the ordering
pred lte (e1, e2: elem) { e1=e2 || lt (e1,e2) }
// e1 is greater than or equal to e2 in the ordering
pred gte (e1, e2: elem) { e1=e2 || gt (e1,e2) }
// returns the larger of the two elements in the ordering
fun larger (e1, e2: elem): elem { if lt (e1,e2) then e2 else e1 }

// returns the smaller of the two elements in the ordering
fun smaller (e1, e2: elem): elem { if lt (e1,e2) then e1 else e2 }

// returns the largest element in es or the empty set if es is empty
fun max (es: set elem): lone elem { es - es.^(Ord.prev_) }

// returns the smallest element in es or the empty set if es is empty
fun min (es: set elem): lone elem { es - es.^(Ord.next_) }
```

Bridge crossing with one umbrella

A family with a father, a mother, a child and an old grandmother, are going to cross a bridge (a shelter at each end) from the east side to the west side. The bridge is very narrow that allows at most two person to pass at the same time. It is raining hard that they cannot move without an umbrella (they don't want to get wet). However, the whole family has only one umbrella. So some of them need to pass the umbrella back and forth. It takes 1, 2, 5, and 10 minutes for the father, the mother, the child and the grandmother to overpass the bridge respectively, and if two of them walk together, the duration depends on the one who takes longer time.
Father does all the work (total time 19) is the fastest way ?
The question is, can this be done less than 19, say 17 mins ?

```
module examples/puzzles/bridgecrossing

open util/ordering[State] as ord
abstract sig Person { time: Int }
one sig Father, Mother, Son, Grandma extends Person {}

fact crossing {int Father.time=1 &&  int Mother.time=2  && int Son.time=5 &&int Gra

sig State {
   east: set Person,
   west: set Person,
   time, u: Int
}
// In the initial state, all objects are on the east side.
fact initialState {
   let s0 = ord/first() |
     s0.east = Person && no s0.west && int s0.time = 0 && int s0.u=0
}

pred crossBridge (from, from', to, to': set Person, t, t', u, u': Int) {
 (some disj p1, p2: from {
    from' = from - p1 - p2
    to' = to + p1 + p2
    int p1.time > int p2.time =>  int t'= int t + int p1.time
    int p1.time < int p2.time =>  int t'= int t + int p2.time
    int u =0 => int u'=1
    int u=1 => int u'=0
 } )||
 (some p: from {
    from' = from - p
    to' = to + p
    int t'= int t + int p.time
    int u =0 => int u'=1
    int u=1 => int u'=0
 })
}
fact stateTransition {
  all s: State, s': ord/next(s) {
    s.west != Person =>
    (int s.u=0 =>
```

```
        crossBridge(s.east, s'.east, s.west, s'.west, s.time, s'.time, s.u, s'.u),
        crossBridge(s.west, s'.west, s.east, s'.east, s.time, s'.time,s.u, s'.u)),
     s'=s }
}
pred solvePuzzle () {
     ord/last().west = Person && int ord/last().time=<17
}

run solvePuzzle for 6 State, 9 Int, 6 int expect 1
```

Alloy Semantics for DAML+OIL

```
sig Resource {}


disj sig Class extends Resource
    {instances: set Resource}


disj sig Property extends Resource
    {sub_val: Resource -> Resource}


fun subClassOf(c1, c2: Class)
    {c2.instances in c1.instances}
fun disjointWith (c1, c2: Class)
    {no c1.instances & c2.instances}


fun subPropertyOf (p1, p2:Property)
    {p1.sub_val in p2.sub_val}
```

Alloy Semantics for DAML+OIL (continue)

```
fun toClass (p:Property, c1:Class, c2:Class)
{all a1, a2: Resource | a1 in c1.instances <=>
    a2 in a1.(p.sub_val) => a2 in c2.instances}
```

```
fun hasValue (p:Property, c:Class, r:Resource)
    {all a:Resource |
    a in c.instances => a.(p.sub_val) = r}
```

....

Consequence: Alloy Approach to Checking Web Ontology

- J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *Proceedings of 12th Internation Symposium on Formal Methods Europe: FM'03*, pages 796–813, Pisa, Italy, September 2003. LNCS, Springer-Verlag.

Chapter 5

Z

This chapter introduce the Z notation.

5.1 Z Basics

The Z notation[113] is a state-oriented formal specification language based on set theory and predicate calculus. It was developed by the Programming Research Group at Oxford University. Z is classified as a model-based specification language (like VDM).
A Z specification typically includes a number of state and operation schema definitions. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by values taken by variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the 'before' and 'after' states corresponding to one or more state schemas. A schema calculus facilitates the composition of complex specifications. Z has been widely adopted to specify a range of software systems (see [53]). The formal semantics of Z has also been developed[112, 14]. A number of textbooks on Z are also available, e.g. [113, 96, 130, 26]. Most constructs of Z use fairly standard mathematical notation; we shall assume the reader is familiar with the basics of Z (the Z glossary in [53] is provided in Appendix A).

Predicate Calculus

A predicate (proposition) is a statement that is either true or false.

- today is Monday
- $x + y = 9$ $P(x, y)$
- Logic operators:

- Not ($\neg$), e.g. $\neg\,(11 < 3)$ is true
- And ($\wedge$), e.g. $(11 > 3) \wedge (2 + 2 = 4)$
- Or ($\vee$), e.g. $P \vee (\neg\, P)$ (a **tautology**)
- Implies ($\Rightarrow$), e.g. $(11 < 3) \Rightarrow (2 + 2 = 5)$ is true
- Equivalence ($\Leftrightarrow$), e.g. $P \Leftrightarrow P$ (is a tautology)

Universal Quantifier ($\forall$)

Consider the predicate

> "all natural numbers are bigger than zero".

We can write this formally as

$$\forall \quad n \quad : \quad \mathbb{N} \quad \bullet \quad n > 0$$

More generally,

$\forall\, x : X \bullet P(x)$ abbreviates $P(a) \wedge P(b) \wedge P(c) \wedge \cdots$

Are the following predicates true of false?

$\forall\, n : \mathbb{N} \bullet n^2 > n$

$\forall\, n : \mathbb{N} \bullet (n^2 = n) \;\Rightarrow\; (n = 0 \;\vee\; n = 1)$

Existential Quantifier ($\exists$)

Consider the predicate

> "there is a natural number bigger than zero".

We can write this formally as

$$\exists \quad n \quad : \quad \mathbb{N} \quad \bullet \quad n > 0$$

More generally,

$\exists\, x : X \bullet P(x)$ abbreviates $P(a) \vee P(b) \vee P(c) \vee \cdots$

Are the following predicates true or false?

$$\exists\, x : \mathbb{N} \bullet x = x + 1$$

$$\forall\, x : \mathbb{N} \bullet (\exists\, y : \mathbb{N} \bullet y > x)$$

Sets

A set is a collection of elements (or members). e.g.

$$\{a, b, c\}, \quad \{3, 1, 16\}$$

- the elements are not ordered

  $\{a, b, c\}$ is the same set as $\{b, a, c\}$

- the elements are not repeated

  $\{a, a, b\}$ is the same set as $\{a, b\}$

Some Given Sets

$\mathbb{N} == \{0, 1, 2, \ldots\}$ natural numbers
$\mathbb{N}_1 == \{1, 2, 3, \ldots\}$
$\mathbb{Z} == \{0, 1, -1, 2, -2, \ldots\}$ integers
$\mathbb{R}$ real numbers
$\varnothing$ empty set: the set with no elements

Membership

$$x \in X$$

is a predicate which is

- true if x is in the set X, e.g $a \in \{a, b, c\}$ (T)
- false if x is not in the set X, e.g $d \in \{a, b, c\}$ (F)

Notice the difference between ':' and '$\in$':

$$\forall\, x : \mathbb{Z} \bullet x > 5 \;\Rightarrow\; x \in \mathbb{N}$$

$x : \mathbb{Z}$ declares a new variable x of type $\mathbb{Z}$
$x \in \mathbb{N}$ is a predicate which is true or false depending upon the value of the previously declared x

Set Expressions

$\{a, b, c, d\}$ (is a *finite* set)
$\mathbb{N}$ (is an *infinite* set)

We can express a set by listing its elements, but this is impractical if the set is large, and impossible if the set is infinite.

Instead, a set can be defined by giving a predicate which specifies precisely those elements in the set.
e.g. the set of all natural numbers less than 99 is:

$$\{ \quad n : \mathbb{N} \qquad | \qquad n < 99 \quad \}$$

In general, the set

$$\{x : X \mid P(x)\}$$

is the set of elements of X for which the predicate P is true.

Examples

the set of even integers is

$$\{z : \mathbb{Z} \mid \exists\, k : \mathbb{Z} \bullet z = 2k\}$$

the set of natural numbers which when divided by 7 leave a remainder of 4 is

$$\{n : \mathbb{N} \mid \exists\, m : \mathbb{N} \bullet n = 7m + 4\}$$

$\mathbb{N}$ is the set $\{z : \mathbb{Z} \mid z \geqslant 0\}$

$\mathbb{N}_1$ is the set $\{n : \mathbb{N} \mid n \geqslant 1\}$

if a, b are any natural numbers then $a \mathinner{..} b$ is defined as the set of all natural numbers between a and b inclusive, i.e.

$$a \mathinner{..} b \text{ is the set } \{n : \mathbb{N} \mid a \leqslant n \leqslant b\}$$

Subset ($\subseteq$) and Proper Subset ($\subset$)

If S and T are sets,

$S \subseteq T$ (S is a subset of T)
is a predicate equivalent to $\forall s :$
$S \bullet s \in T$

$S \subset T$ (S is a proper subset of T)
is a predicate equivalent to $S \subseteq$
$T \land S \neq T$

e.g. the following predicates are true

$\{0, 1, 2\} \subseteq \mathbb{N}$
$2 \mathinner{..} 3 \subseteq 1 \mathinner{..} 5$
$\{a, b\} \subseteq \{a, b, c\}$
$\varnothing \subseteq X$ for any set X
$\{x\} \subseteq X \Leftrightarrow x \in X$

Power Set ($\mathbb{P}$)

If X is a set,

$\mathbb{P} X$ (the power set of X)

is the set of all subsets of X.

$$A \in \mathbb{P} B \quad = \quad A \subseteq B$$

e.g. the following predicates are true

$\mathbb{P}\{a, b\} = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}$
$\mathbb{P}\,\varnothing = \{\varnothing\} \quad (\neq \varnothing)$
$1\,..\,5 \in \mathbb{P}\,\mathbb{N}$
$2\,..\,4 \in \mathbb{P}(1\,..\,5)$

If X has k elements, $\mathbb{P}\,X$ has 2^k elements.

Set Union ($\cup$)

Suppose $S, T : \mathbb{P}\,X$ (i.e. $S \subseteq X$ and $T \subseteq X$); then
$S \cup T$ (S union T)
is a set equal to
$\{x : X \mid x \in S \lor x \in T\}$

e.g. the following predicates are true

$\{a, b, c\} \cup \{b, g, h\} = \{a, b, c, g, h\}$
$(1\,..\,5) \cup (3\,..\,7) = 1\,..\,7$
$\mathbb{N}_1 \cup \{0\} = \mathbb{N}$
$A \cup \varnothing = A$ (for any set A)

Set Intersection ($\cap$)

Suppose $S, T : \mathbb{P}\,X$; then
$S \cap T$ (S intersection T)
is a set equal to
$\{x : X \mid x \in S \land x \in T\}$

e.g. the following predicates are true

$\{a, b, c\} \cap \{b, g, h\} = \{b\}$
$(1\,..\,5) \cap (3\,..\,7) = 3\,..\,5$
$\{a, b, c\} \cap \{d, g\} = \varnothing$ (the sets are *disjoint*)
$A \cap \varnothing = \varnothing$ (for any set A)

Set Difference ($-$)

Suppose $S, T : \mathbb{P}\, X$; then
$S - T$ (S subtract T)
is a set equal to
$\{x : X \mid x \in S \land x \notin T\}$

e.g. the following predicates are true

$\{a, b, c\} - \{b, g, h\} = \{a, c\}$
$(1 \,..\, 5) - (3 \,..\, 7) = 1 \,..\, 2$
$\mathbb{N}_1 = \mathbb{N} - \{0\}$
$A - \varnothing = A$ (for any set A)

Cartesian Product ($\times$)

If A and B are sets,

$A \times B$ (A cross B)

is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$.

e.g. the following predicates are true

$\{a, b\} \times \{a, c\} = \{(a, a), (a, c), (b, a), (b, c)\}$
$(5, -1) \in \mathbb{N} \times \mathbb{Z}$
$(5, -1) \notin \mathbb{N} \times \mathbb{N}$
$6 \notin \mathbb{N} \times \mathbb{N}$
$A \times \varnothing = \varnothing$ (for any set A)

$\mathbb{R} \times \mathbb{R}$ is the Cartesian plane

Cardinality

If X is any finite set,

$\#X$

is a natural number denoting the cardinality of (i.e. the number of elements in) X.

e.g.

$$\#\{a, b, c\} = 3$$

$$\#\varnothing = 0$$

$$\#\,\mathbb{P}\,A = 2^{\#A} \qquad \text{(for any finite set } A\text{)}$$

Types

Z is strongly typed: every expression is given a type.
Any set can be used as a type.
The following are equivalent within set comprehension

$$(x, y) : A \times B$$
$$x : A;\ y : B$$
$$x, y : A \qquad \text{(when } B = A\text{)}$$

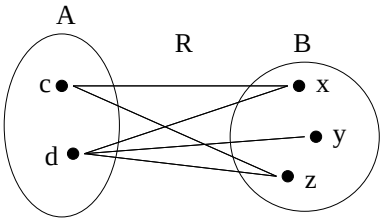
Notice that

$$\forall\, S : \mathbb{P}\, A \bullet \ldots \quad \text{not} \quad \forall\, S \subseteq A \bullet \ldots$$

$$\forall\, S : \mathbb{P}\, A \bullet (\forall\, y : S \bullet \ldots) \quad \text{not} \quad \forall\, S : \mathbb{P}\, A;\ y : S \bullet \ldots$$

Relations

A relation R from A to B, denoted by
$$R : A \leftrightarrow B,$$
is a subset of $A \times B$.



R is the set $\{(c, x), (c, z), (d, x), (d, y), (d, z)\}$

Notation: the predicates

$$(c, z) \in R \quad \text{and} \quad c \mapsto z \in R \quad \text{and} \quad c\, \underline{R}\, z$$

are equivalent.

$\mathrm{dom}\, R$ is the set $\{a : A \mid \exists b : B \bullet a \underline{R} b\}$
$\mathrm{ran}\, R$ is the set $\{b : B \mid \exists a : A \bullet a \underline{R} b\}$

Examples

$$\_ \leqslant \_ : \mathbb{N} \leftrightarrow \mathbb{N}$$

$$\forall x, y : \mathbb{N} \bullet \quad x \leqslant y \Leftrightarrow \exists k : \mathbb{N} \bullet x + k = y$$

i.e. the relation $\leqslant$ is the infinite subset

$$\{(0,0), (0,1), (1,1), (0,2), (1,2), (2,2), \ldots\}$$

of ordered pairs in $\mathbb{N} \times \mathbb{N}$.

$$\mathit{divides} : \mathbb{N}_1 \leftrightarrow \mathbb{N}$$

$$\forall x : \mathbb{N}_1;\ y : \mathbb{N} \bullet \quad x \underline{\mathit{divides}}\, y \Leftrightarrow \exists k : \mathbb{N} \bullet x\, k = y$$

$3 \underline{\mathit{divides}}\, 6$ but $\neg\, (3 \underline{\mathit{divides}}\, 7)$

Domain and Range Restriction

Suppose $R : A \leftrightarrow B$ and $S \subseteq A$ and $T \subseteq B$; then

$S \lhd R$ is the set $\{(a, b) : R \mid a \in S\}$
$R \rhd T$ is the set $\{(a, b) : R \mid b \in T\}$

Notice that both are true:

$S \lhd R \in A \leftrightarrow B$ and $R \rhd T \in A \leftrightarrow B$

e.g. if

$\mathit{has\_sibling} : \mathit{People} \leftrightarrow \mathit{People}$ then

$female \lhd has\_sibling$ is the relation is_sister_of
$has\_sibling \rhd female$ is the relation has_sister

Domain and Range Subtraction

Suppose $R : A \leftrightarrow B$ and $S \subseteq A$ and $T \subseteq B$; then

$S \lhd\!\!\lhd R$ is the set $\{(a, b) : R \mid a \notin S\}$
$R \rhd\!\!\rhd T$ is the set $\{(a, b) : R \mid b \notin T\}$

The following predicates are true

$S \lhd\!\!\lhd R = (A - S) \lhd R$
$R \rhd\!\!\rhd T = R \rhd (B - T)$
$S \lhd\!\!\lhd R \in A \leftrightarrow B$
$R \rhd\!\!\rhd T \in A \leftrightarrow B$

$female \lhd\!\!\lhd has\_sibling$ is the relation $is\_brother\_of$
$has\_sibling \rhd\!\!\rhd female$ is the relation $has\_brother$

Relational Image

Suppose $R : A \leftrightarrow B$ and $S \subseteq A$

$R(\!| S |\!) = \{b : B \mid \exists a : S \bullet a \mathrel{\underline{R}} b\}$

$R(\!| S |\!) \subseteq B$

$divides(\!| \{8, 9\} |\!)$
$= \{x : \mathbb{N} \mid \exists k : \mathbb{N} \bullet x = 8k \lor x = 9k\}$
$= \{$numbers divided by 8 or 9$\}$

$\leqslant (\!| \{7, 3, 21\} |\!) = \{x : \mathbb{N} \mid x \geqslant 3\}$

$has_sibling(\!| male |\!) = \{$people who have a brother$\}$

Inverse

Suppose $R : A \leftrightarrow B$

$$R^{-1} = \{(b, a) : B \times A \mid a \underline{R} b\}$$

$$R^{-1} \in B \leftrightarrow A$$

$$has\_sibling^{-1} = has\_sibling$$
$$divides^{-1} = has\_divisor$$

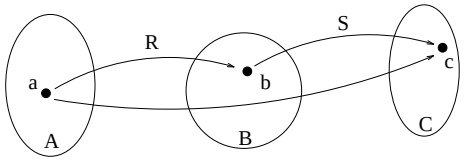
$$succ : \mathbb{N} \leftrightarrow \mathbb{N}$$
$$\forall x, y : \mathbb{N} \bullet$$
$$x \underline{succ} y \Leftrightarrow x + 1 = y$$

$$succ^{-1} = pred$$

Relational Composition

Suppose $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$

$$R \fatsemi S = \{(a, c) : A \times C \mid \exists b : B \bullet a \underline{R} b \land b \underline{S} c\}$$



$R \fatsemi S \in A \leftrightarrow C$

e.g.

$$is\_parent\_of \fatsemi is\_parent\_of = is\_grandparent\_of$$

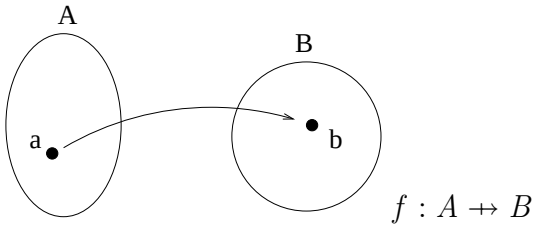
$$R^0 = \mathrm{id}[A], \quad R^1 = R, \quad R^2 = R \fatsemi R, \quad R^3 = R \fatsemi R \fatsemi R, \ldots$$

Functions

A (partial) function f from a set A to a set B, denoted by

$$f : A \pfun B,$$

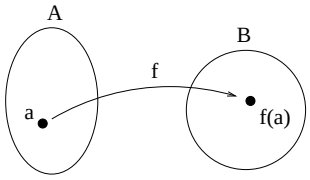
is a subset f of $A \times B$ with the property that for each $a \in A$ there is at most one $b \in B$ with $(a, b) \in f$.



$\mathrm{dom}\, f$ is the set $\{a : A \mid \exists\, b : B \bullet (a, b) \in f\}$
$\mathrm{ran}\, f$ is the set $\{b : B \mid \exists\, a : A \bullet (a, b) \in f\}$

Function Application

Suppose $f : A \pfun B$ and $a \in \mathrm{dom}\, f$; then $f(a)$ denotes the unique *image* in B that a is mapped to by f.



The predicates

$$(a, b) \in f \quad \text{and} \quad f(a) = b$$

are equivalent.

Total Functions

A function $f : A \pfun B$ is a *total* function, denoted

$$f : A \fun B,$$

if and only if $\mathrm{dom}\, f$ is the set A.

Specifying Functions

(1) Using a Look-up Table

If a function $f : A \pfun B$ is finite (and not too large) we can specify the function explicitly by listing all the pairs (a, b) in the subset of $A \times B$ where $f(a) = b$.

e.g.

$address : PassportNo \fun Address$

PassportNo	Address
A001017	77 Sunset Strip
...	...
...	...
G707165	19 Mail Street
...	...

(2) Declaring Axioms

A function can be specified by giving a predicate determining which pairs (a, b) are in the function.

(a)

$$\begin{array}{|l} double : \nat \fun \nat \\ \hline \forall\, n : \nat \bullet double(n) = 2n \end{array}$$

(b)

$$\begin{array}{|l} halve : \nat \pfun \nat \\ \hline \dom halve = \{n : \nat \mid \exists\, m : \nat \bullet 2m = n\} \\ \forall\, n : \dom halve \bullet 2 * halve(n) = n \end{array}$$

(c)

$$\begin{array}{|l} root : \nat \pfun \nat \\ \hline \dom root = \{n : \nat \mid \exists\, m : \nat \bullet m^2 = n\} \\ \forall\, n : \dom root \bullet (root(n))^2 = n \end{array}$$

(d)

$$\begin{array}{|l} + : \nat \times \nat \fun \nat \\ \hline \forall (n, m) : \nat \times \nat \bullet +(n, m) = n + m \end{array}$$

(e) Let *People* be the set of all living people.

$$\begin{array}{|l} birth : People \to \mathbb{N} \\ \hline \forall\, p : People \bullet birth(p) \text{ is the year of } p\text{'s birth} \end{array}$$

(3) Using Recursion

This is a variant on the previous declarative specification; a function is defined recursively in terms of itself.

e.g.

$$\begin{array}{|l} fact : \mathbb{N}_1 \to \mathbb{N} \\ \hline fact(1) = 1 \\ \forall\, n : \mathbb{N}_1 - \{1\} \bullet fact(n) = n * fact(n-1) \end{array}$$

so

$$\begin{array}{l} fact(1) = 1 \\ fact(2) = 2 * fact(1) = 2 * 1 = 2 \\ fact(3) = 3 * fact(2) = 3 * 2 = 6 \\ fact(4) = 4 * fact(3) = 4 * 6 = 24 \end{array}$$

and so on....

Function Overriding

Suppose $f, g : A \nrightarrow B$; then

$$f \oplus g \quad \text{is the function} \quad (\operatorname{dom} g \lhd f) \cup g$$

i.e. the following predicates are true

$$\begin{array}{l} \operatorname{dom} f \oplus g = \operatorname{dom} f \cup \operatorname{dom} g \\ \forall\, a : \operatorname{dom} g \bullet (f \oplus g)(a) = g(a) \\ \forall\, a : \operatorname{dom} f - \operatorname{dom} g \bullet (f \oplus g)(a) = f(a) \\ f \oplus g \in A \nrightarrow B \end{array}$$

e.g.

$$\{a \mapsto x, b \mapsto y, c \mapsto x\} \oplus \{a \mapsto y\} \;=\; \{a \mapsto y, b \mapsto y, c \mapsto x\}$$

$$double \oplus root \;=\; \{(0,0),(1,1),(2,4),(3,6),(4,2),\cdots\}$$

Sequences

A sequence s of elements from a set A, denoted

$$s : \operatorname{seq} A,$$

is a function $s : \mathbb{N} \pfun A$ where $\operatorname{dom} s = 1\,..\,n$ for some natural number n. For example,

$\langle b, a, c, b\rangle$ denotes the sequence (function)$\{1 \mapsto b, 2 \mapsto a, 3 \mapsto c, 4 \mapsto b\}$

The empty sequence is denoted by $\langle\ \rangle$.

The set of all sequences of elements from A is denoted $\operatorname{seq} A$ and is defined to be

$$\operatorname{seq} A == \{s : \mathbb{N} \pfun A \mid \exists\, n : \mathbb{N} \bullet \operatorname{dom} s = 1\,..\,n\}$$

We define $\operatorname{seq}_1 A$ to be the set of all non-empty sequences, i.e.

$$\operatorname{seq}_1 A == \operatorname{seq} A - \{\langle\ \rangle\}$$

Notice that: $\langle a, b, a\rangle \neq \langle a, a, b\rangle \neq \langle a, b\rangle$

Special Functions for Sequences

Concatenation

$$\langle a, b\rangle \frown \langle b, a, c\rangle = \langle a, b, b, a, c\rangle$$

Head

$$\begin{array}{|l} head : \operatorname{seq}_1 A \to A \\ \hline \forall\, s : \operatorname{seq}_1 A \bullet head(s) = s(1) \end{array}$$

$$\operatorname{head}\langle c, b, b\rangle = c$$

Tail

$$\begin{array}{|l} tail : \operatorname{seq}_1 A \to \operatorname{seq} A \\ \hline \forall\, s : \operatorname{seq}_1 A \bullet \langle head(s)\rangle \frown tail(s) = s \end{array}$$

$tail\langle c, b, b\rangle = \langle b, b\rangle$

5.2 Z Schemas

Z Case Study: A Message Buffer

- A number of messages are transmitted from one location to another.
- Because of other traffic on the line each message for transmission is placed in a buffer which outputs the message when the line is free.
- This buffer may contain several messages at any time, but there is a fixed upper limit on the number of messages the buffer may contain.
- The buffer operates on a first in/first out (FIFO) principle.

Formal Specification

The State Schema

$[MSG]$ (The exact nature of these messages is not important)

is the set of all possible messages that could ever be transmitted.

$max : \mathbb{N}$ (The actual value of max is not important)

is the constant maximum number of messages that can be held in the buffer at any one time.

Buffer

$items : \operatorname{seq} MSG$	declaration
$\#items \leqslant max$	predicate

e.g. suppose $MSG = \{m_1, m_2, m_3\}$ and $max = 4$

Then $items = \langle m_1, m_2\rangle$ is an instance, but $items = \langle m_3, m_1, m_1, m_2, m_2\rangle$ is not

- a schema specifies a relationship between variable values; *Buffer* is a *state* schema
- a state schema specifies a 'snapshot' of a system

- variables are declared and typed in the top part of the schema
- a predicate (axiom) restraining the possible values of the declared variables is given in the bottom part of the schema
- an instance of a schema is an assignment of values to variables consistent with their type declaration and satisfying the predicate

Operation Schema

The state schema *Buffer* gives a static view of the system. To specify how the system can change we need to specify *operation* schema.
An operation can be thought of as taking an instance of the state schema and producing a new instance.
To specify such an operation we express as a predicate the relationship between the instance of the state before the operation and the instance after the operation.
We adopt the convention that the value of state variables before the operation are denoted by unprimed identifiers, while values after the operation are denoted by primed identifiers.
For the message buffer there are two operations:

Join (a new message is added to the buffer)

Leave (a message leaves the buffer)

The Join Operation

Join

$$items, items' : \operatorname{seq} MSG$$
$$msg? : MSG$$

$$\#items \leqslant max$$
$$\#items' \leqslant max$$
$$\#items < max$$
$$items' = items \frown \langle msg? \rangle$$

- *items* denotes the sequence of messages in the buffer before the operation
- $items'$ denotes the sequence of messages in the buffer after the operation
- the decoration ? denotes an *input*

- there is an implicit $\wedge$ between each line
- the first two lines of the predicate indicate that we have a valid instance of the state schema *Buffer* both before and after the operation
- the third line of the predicate is a pre-condition for the operation: it indicates that for the *Join* operation to be possible the buffer must not already be completely full
- the last line of the predicate specifies the relationship between the buffer contents before and after the operation: the input message is appended to the sequence of messages already in the buffer. e.g. suppose

  $$MSG = \{m_1, m_2, m_3\} \quad \text{and} \quad max = 4 \quad \text{and}$$
  $$items = \langle m_1, m_2, m_1 \rangle \quad \text{and} \quad msg? = m_3;$$

  then after the operation

  $$items = \langle m_1, m_2, m_1, m_3 \rangle$$

Schema Inclusion

Because we always have a 'before' and 'after' instance of the state schema for any operation we make the following syntactic simplification: define

ΔBuffer

$items, items' : \text{seq}\, MSG$

$\#items \leqslant max$
$\#items' \leqslant max$

and we can now write *Join* by including this schema:

Join

$\Delta Buffer$
$msg? : MSG$

$\#items < max$
$items' = items \frown \langle msg? \rangle$

In general, including a schema in the declaration part of another schema means that the included schema has its declaration added to the new schema, and its predicate conjoined to the predicate of the new schema.
e.g. if

$$\begin{array}{|l} A \\ \hline x : T_1 \\ y : T_2 \\ \hline P(x, y) \\ \hline \end{array} \qquad \begin{array}{|l} S \\ \hline A \\ z : T_3 \\ \hline Q(x, y, z) \\ \hline \end{array}$$

then S expands to the schema

$$\begin{array}{|l} S \\ \hline x : T_1 \\ y : T_2 \\ z : T_3 \\ \hline P(x, y) \land Q(x, y, z) \\ \hline \end{array}$$

The Leave Operation

$$\begin{array}{|l} Leave \\ \hline \Delta Buffer \\ msg! : MSG \\ \hline items \neq \varnothing \\ items = \langle msg! \rangle \frown items' \\ \hline \end{array}$$

- the decoration ! denotes an *output*

- the first line of the predicate is a pre-condition for the operation: it indicates that for the *Leave* operation to be possible the buffer must not be empty

- the last line of the predicate specifies the relationship between the buffer contents before and after the operation: the output message is taken from the head of the sequence of messages in the buffer, leaving just the tail of the sequence in the buffer.

The Initial State

To complete the specification of the message buffer we need to specify the initial state of the buffer:

$$\begin{array}{|l} \textit{Buffer}_{\text{INIT}} \\ \textit{Buffer} \\ \hline items = \langle\,\rangle \end{array}$$

i.e.

$$\begin{array}{|l} \textit{Buffer}_{\text{INIT}} \\ items : \operatorname{seq} MSG \\ \hline \#items \leqslant max \\ items = \langle\,\rangle \end{array}$$

We have specified the message buffer in terms of what an observer of the buffer can expect to see. Initially the buffer would be empty, and then the operations of *Join* and *Leave* can occur whenever they are enabled (i.e. when their pre-conditions are satisfied). Operations are assumed to be atomic (i.e. occur instantaneously). At all times an observer would notice that the state schema for the buffer is satisfied.

Consider the buffer is taking an input of sequence of messages rather than a single message, the following scenanios can be allowed.

(a) an operation *titanic* whereby a sequence of messages is appended to the queue except those messages for which there is no room are discarded (the queue is like a life-boat on the Titanic: people queue to get on, but once the boat is full all the remaining people are left behind);

(b) an operation *penguin* whereby, like the operation *titanic*, a sequence of messages is input to the queue, but this time the messages on the end of the sequence are accepted while those at the front are discarded if there is no room (the messages are acting like penguins, pushing out the messages already in the queue once the queue is full).

These two scenarios can be modelled as the following operations.

$$\begin{array}{|l} \textit{titanic} \\ \Delta \textit{Buffer} \\ s? : \operatorname{seq} MSG \\ \hline items' = (1 \mathinner{..} max) \lhd (items \frown s?) \end{array}$$

$$\begin{array}{|l} \textit{penguin} \\ \Delta \textit{Buffer} \\ s? : \operatorname{seq} MSG \\ s? : \operatorname{seq} MSG \\ \hline \exists\, s : \operatorname{seq} MSG \bullet \\ \quad s \frown items' = items \frown s? \\ \quad s \neq \langle\,\rangle \;\Rightarrow\; \#items' = max \end{array}$$

Extending Specifications

Example: A Slow Buffer

$$delay : \mathbb{N}$$

SlowBuffer

$$Buffer$$
$$idle : \mathbb{N}$$

i.e.

SlowBuffer

$$items : \operatorname{seq} MSG;\ idle : \mathbb{N}$$

$$\#items \leqslant max$$

SlowBuffer$_{\text{INIT}}$

$$SlowBuffer$$
$$Buffer_{\text{INIT}}$$

$$idle = 0$$

i.e.

SlowBuffer$_{\text{INIT}}$

$$items : \operatorname{seq} MSG$$
$$idle : \mathbb{N}$$

$$\#items \leqslant max \land items = \langle\,\rangle \land idle = 0$$

Merging Schemas

A

$$x : T_1$$
$$y : T_2$$

$$P(x, y)$$

B

$$y : T_2$$
$$z : T_3$$

$$Q(y, z)$$

C

$$A$$
$$B$$

where

C

$$x : T_1$$
$$y : T_2$$
$$z : T_3$$

$$P(x, y) \land Q(y, z)$$

- type compatibility is needed to merge schemas

Slow Operations

SlowJoin

$\Delta SlowBuffer$
$Join$

$idle \geqslant delay$
$idle' = 0$

i.e.

SlowJoin

$items, items' : \text{seq}\, MSG$
$idle, idle' : \mathbb{N}$
$msg? : MSG$

$\#items \leqslant max \land \#items' \leqslant max$
$\#items < max$
$items' = items \frown \langle msg? \rangle$
$idle \geqslant delay \land idle' = 0$

SlowLeave

$\Delta SlowBuffer$
$Leave$

$idle \geqslant delay \land idle' = 0$

Tick

$\Delta SlowBuffer$

$idle' = idle + 1 \land items' = items$

Exercise:
give the expanded form of the operation schemas *SlowLeave* and *Tick*.

Reasoning About the Specification

Can we verify that the message buffer as specified has the FIFO property, i.e. messages leave the buffer in the same order as they arrive?
To do this we introduce *auxiliary* variables which do not alter the functionality of the specification but aid in the analysis.
In this case we introduce auxiliary sequences *inhist* and *outhist* to record the history of the flow of messages into and out of the buffer.
The new system obtained from the buffer by adding these auxiliary variables can be specified by including the original schemas into new schemas which contain the extra information about auxiliary variables.

RecordedBuffer

$$Buffer$$
$$inhist : \text{seq}\, MSG$$
$$outhist : \text{seq}\, MSG$$

RecordedBuffer$_{\text{INIT}}$

$$RecordedBuffer$$
$$Buffer_{\text{INIT}}$$

$$inhist = \langle\,\rangle$$
$$outhist = \langle\,\rangle$$

RecordedJoin

$$\Delta RecordedBuffer$$
$$Join$$

$$inhist' = inhist \frown \langle msg? \rangle$$
$$outhist' = outhist$$

RecordedLeave

$$\Delta RecordedBuffer$$
$$Leave$$

$$inhist' = inhist$$
$$outhist' = outhist \frown \langle msg! \rangle$$

e.g. the schema *RecordedJoin* with the included schemas expanded becomes:

RecordedJoin

$$items, items' : \text{seq}\, MSG$$
$$inhist, inhist' : \text{seq}\, MSG$$
$$outhist, outhist' : \text{seq}\, MSG$$
$$msg? : MSG$$

$$\#items \leqslant max$$
$$\#items' \leqslant max$$
$$\#items < max$$
$$items' = items \frown \langle msg? \rangle$$
$$inhist' = inhist \frown \langle msg? \rangle$$
$$outhist' = outhist$$

How can we use the auxiliary variables to prove that the buffer satisfies the FIFO property ?

Theorem

$$\forall\, RecordedBuffer \bullet inhist = outhist \frown items$$

Proof:
Use structural induction.
Initially $inhist = outhist = items = \langle\,\rangle$,
so the predicate is true.

Suppose the predicate is true, and *RecordedJoin* occurs.
After the operation

$$inhist' = inhist \frown \langle msg? \rangle \wedge outhist' = outhist \wedge items' = items \frown \langle msg? \rangle$$

Hence: $inhist'$
$$= inhist \frown \langle msg? \rangle = (outhist \frown items) \frown \langle msg? \rangle$$
$$= outhist \frown (items \frown \langle msg? \rangle) = outhist' \frown items'$$

and the predicate remains true. A similar argument shows that the operation *RecordedLeave* also preserves the predicate. □

Conjunction

$$SlowRecordedBuffer \mathrel{\widehat=} SlowBuffer \wedge RecordedBuffer$$

is equivalent to merging the schemas:

SlowRecordedBuffer
SlowBuffer
RecordedBuffer

Also

$$SlowRecordedBuffer_{\text{INIT}} \mathrel{\widehat=} SlowBuffer_{\text{INIT}} \wedge RecordedBuffer_{\text{INIT}}$$

$$SlowRecordedJoin \mathrel{\widehat=} SlowJoin \wedge RecordedJoin$$

If A and B are schemas:

- the declaration of $A \wedge B$ is the union of the declarations of A and B;
- the predicate of $A \wedge B$ is the conjunction of the predicates of A and B.

Disjunction

$$Flag ::= ok \mid error$$

JoinOK

$Join$
$flag! : Flag$

$flag! = ok$

JoinError

$\Xi Buffer$
$flag! : Flag$

$\#items = max \land flag! = error$

$CompleteJoin \mathrel{\widehat=} JoinOK \lor JoinError$

CompleteJoin

$\Delta Buffer$
$msg? : MSG;\ flag! : Flag$

$\#items < max \land items' = items \frown \langle msg? \rangle \land flag! = ok$
$\lor$
$\#items = max \land items' = items \land flag! = error$

If A and B are schemas:

- the declaration of $A \lor B$ is the union of the declarations of A and B;
- the predicate of $A \lor B$ is the disjunction of the predicates of A and B.

In general

A

$x : T_1$
$y : T_2$

$P(x, y)$

B

$y : T_2$
$z : T_3$

$Q(y, z)$

conjunction

A ∧ B

$x : T_1;\ y : T_2;\ z : T_3$

$P(x, y) \land Q(y, z)$

disjunction

A ∨ B

$x : T_1;\ y : T_2;\ z : T_3$

$P(x, y) \lor Q(y, z)$

Composition

$Join \fatsemi Leave$

is an (atomic) operation with the effect of a *Join* followed by a *Leave*. Defining $JoinLeave \mathrel{\widehat=} Join \fatsemi Leave$ gives

JoinLeave

$$\Delta Buffer$$
$$msg?, msg! : MSG$$

$$\#items < max$$
$$\exists\, items'' : \operatorname{seq} MSG \bullet items'' = items \frown \langle msg? \rangle \land items'' = \langle msg! \rangle \frown items'$$

- the pre-state of *Join* is the pre-state of $Join \fatsemi Leave$
- the post-state of *Join* is identified with the pre-state of *Leave* and hidden within $Join \fatsemi Leave$
- the consequent post-state of *Leave* is the post-state of $Join \fatsemi Leave$

Composition in general

A

$$x : T_1$$
$$y : T_2$$

$$P(x, y)$$

AOP$_1$

$$\Delta A$$
$$t_3? : T_3$$
$$t_4! : T_4$$

$$Q_1(x, x', y, y', t_3?, t_4!)$$

AOP$_2$

$$\Delta A$$
$$t_5? : T_5$$
$$t_6! : T_6$$

$$Q_2(x, x', y, y', t_5?, t_6!)$$

AOP$_1 \fatsemi$ *AOP*$_2$

$$\Delta A$$
$$t_3? : T_3;\ t_4! : T_4;\ t_5? : T_5;\ t_6! : T_6$$

$$\exists\, x'' : T_1;\ y'' : T_2 \bullet$$
$$\quad Q_1(x, x'', y, y'', t_3?, t_4!) \land Q_2(x'', x', y'', y', t_5?, t_6!)$$

Piping

Duplicate

$$msg? : MSG;\ duplicate! : \operatorname{seq} MSG$$

$$duplicate! = \langle msg?, msg? \rangle$$

$$LeaveDuplicated \mathrel{\widehat=} Leave \gg Duplicate$$

LeaveDuplicated

$$\Delta Buffer$$
$$duplicate! : \operatorname{seq} MSG$$

$$items \neq \langle\ \rangle$$
$$\exists\, m : MSG \bullet items = \langle m \rangle^\frown items' \land duplicate! = \langle m, m \rangle$$

- the output variables of *Leave* and the input variables of *Duplicate* with identical bases (i.e. ignoring the decorations '?' and '!' respectively) have their values identified and hidden in $Leave \gg Duplicate$.

Piping in general

A

$$x : T_1;\ y : T_2$$

$$P(x, y)$$

D

$$v : T_3;\ w : T_4$$

$$Q(v, w)$$

AOP

$$\Delta A$$
$$t_5? : T_5$$
$$t_6! : T_6$$

$$RA(x, x', y, y', t_5?, t_6!)$$

DOP

$$\Delta D$$
$$t_6? : T_6$$
$$t_7! : T_7$$

$$RD(v, v', w, w', t_6?, t_7!)$$

$AOP \gg DOP$

ΔA
ΔD
$t_5? : T_5$
$t_7! : T_7$

$\exists\, t : T_6 \bullet RA(x, x', y, y', t_5?, t) \land RD(v, v', w, w', t, t_7!)$

Non-determinism

G

$a, b, c : \mathbb{N}$

$a \leqslant b \land a = c$

$GOP1$

ΔG

$a' = 3$

- b' can take any value $\geqslant 3$ regardless of the value of b
- $c' = 3$ because the state invariant gives $a' = c'$
- if the value of b is to remain unchanged, $b = b'$ must be added

$GOP2$

ΔG
$out! : \mathbb{N}$

$out! = a + b + c$

- the values of a', b', c' are undetermined except that $a' \leqslant b'$ and $a' = c'$
- compare with using ΞG

Renaming

A

$x : T_1$
$y : T_2$

$P(x, y)$

$R \mathrel{\widehat=} A[z/y]$ expands to

R

$x : T_1$
$z : T_2$

$P(x, z)$

e.g $TwoJoins \mathrel{\widehat=} Join[msg_1?/msg?] \fatsemi Join[msg_2?/msg?]$

TwoJoins

$\Delta Buffer$
$msg_1?, msg_2? : MSG$

$\#items < max - 1 \land items' = items \frown \langle msg_1? \rangle \frown \langle msg_2? \rangle$

(compare this with $Join \fatsemi Join$)

Schemas as Types

Instantiation

- a schema determines a type
- a variable of type schema (an instance) can be declared
- variables within an instance are referenced using the 'dot' notation

e.g.

TwoBuffers

$a, b : Buffer$

$a.items = b.items$

expands to give

TwoBuffers

$a, b : Buffer$

$\#a.items \leqslant max$
$\#b.items \leqslant max$
$a.items = b.items$

In general, if

A

$x : T_1$
$y : T_2$

$P(x, y)$

then

I

$a : A$

expands to give

I

$a : A$

$P(a.x, a.y)$

$H1$

A

$x : T_3$

has a type clash unless $T_1 = T_3$

$H2$

$a : A$

$x : T_3$

has no type clash

Global Definitions (constants)

$$square : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall n : \mathbb{N} \bullet square(n) = n^2$$

$$temp : \mathbb{N}$$

$$temp < 451$$

$$max : \mathbb{N}$$

Schemas can be used as types in global definition:

$$length : Buffer \rightarrow \mathbb{N}$$

$$\forall b : Buffer \bullet \\ length(b) = \#b.items$$

Generic Typing

In schemas:

$Buffer\ [T]$

$items : \text{seq}\ T$

$\#items \leqslant max$

$Join\ [T]$

$\Delta Buffer[T]$

$t? : T$

$\#items < max \land items' = items \frown \langle t? \rangle$

In global definitions:

$$
\begin{array}{|l}
\hline
[T] \\
\hline
\text{head} : \text{seq}_1\ T \to T \\
\hline
\forall\, s : \text{seq}_1\ T \bullet \text{head}\, s = s(1) \\
\hline
\end{array}
$$

Alternative Syntax

$$
\begin{array}{|l}
\hline
A \\
\hline
x : T_1 \\
y : T_2 \\
\hline
P(x, y) \\
\hline
\end{array}
$$

can be written as

$$A \mathrel{\widehat=} [x : T_1;\ y : T_2 \mid P(x, y)]$$

e.g.

$$Buffer \mathrel{\widehat=} [items : \text{seq}\, MSG \mid \#items \leqslant max]$$

$$SlowBuffer_{\text{INIT}} \mathrel{\widehat=} [SlowBuffer;\ Buffer_{\text{INIT}} \mid idle = 0]$$

$$\Delta Buffer \mathrel{\widehat=} [Buffer;\ Buffer'] \qquad (= Buffer \land Buffer')$$

Z Case Study: Alternating-Bit Protocol

$$Tag == \{0, 1\}$$

$$[MSG]$$

$$TagMsg == Tag \times MSG$$

Trans

$$buf : \text{seq } TagMsg$$
$$tag : Tag$$
$$\#buf \leqslant 1$$
$$\forall t : Tag;\ m : MSG \bullet buf = \langle (t, m) \rangle \Rightarrow tag = t$$

Rec

$$exptag : Tag$$

MsgChan

$$msgchan : \text{seq } TagMsg$$

AckChan

$$ackchan : \text{seq } Tag$$

$$State \mathrel{\widehat=} Trans \land Rec \land MsgChan \land AckChan$$

- *buf* contains any tagged message that has been transmitted but not yet acknowledged
- *tag* is the tag of the last tagged message to be transmitted
- *exptag* is the tag of the next message expected by the receiver
- *msgchan* is the sequence of tagged messages on route to the receiver
- *ackchan* is the sequence of tags of messages acknowledged by the receiver on route to the transmitter

$State_{\text{INIT}}$

$$State$$
$$buf = \langle\ \rangle$$
$$tag = 0$$
$$msgchan = \langle\ \rangle$$
$$exptag = 1$$
$$ackchan = \langle\ \rangle$$

Operations

If *buf* is empty, a message can be accepted from the environment, tagged and this tagged message transmitted.

TransMsg

$$\Delta Trans$$
$$\Delta MsgChan$$
$$m? : MSG$$
$$buf = \langle\,\rangle$$
$$tag' = 1 - tag$$
$$buf' = \langle (tag', m?) \rangle$$
$$msgchan' = msgchan \frown buf'$$

If *buf* is not empty, its contents can be retransmitted.

Retrans

$$\Xi Trans$$
$$\Delta MsgChan$$
$$buf \neq \langle\,\rangle$$
$$msgchan' = msgchan \frown buf$$

If *msgchan* is not empty, the tagged message at its head can be accepted by the receiver.

If its tag is the expected tag, the message is output to the environment.

RecMsg

$$\Delta MsgChan$$
$$\Delta Rec$$
$$m! : MSG$$
$$msgchan \neq \langle\,\rangle$$
$$(exptag, m!) = \text{head}\, msgchan$$
$$msgchan' = \text{tail}\, msgchan$$
$$exptag' = 1 - exptag$$

If the tag of the tagged message at the head of *msgchan* is not the expected tag, the message is rejected.

RejMsg

$$\Delta MsgChan$$
$$\Xi Rec$$
$$msgchan \neq \langle\,\rangle$$
$$\nexists m : MSG \bullet (exptag, m) = \text{head}\, msgchan$$
$$msgchan' = \text{tail}\, msgchan$$

The tag of the last message output to the environment can be transmitted back as an acknowledgement.

TransAck

ΞRec
$\Delta Ackchan$

$ackchan' = ackchan \frown \langle 1 - exptag \rangle$

When an acknowledgement is received by the transmitter, *buf* is emptied if the acknowledgement equals *tag*; otherwise it is rejected.

RecAck

$\Delta AckChan$
$\Delta Trans$

$ackchan \neq \langle\,\rangle$
$ackchan' = \text{tail}\, ackchan$
$tag = \text{head}\, ackchan \;\Rightarrow\; buf' = \langle\,\rangle$
$tag \neq \text{head}\, ackchan \;\Rightarrow\; buf' = buf$
$tag' = tag$

At any time the tagged message at the head of *msgchan* can be lost.

LoseMsg

$\Delta MsgChan$

$msgchan \neq \langle\,\rangle$
$msgchan' = \text{tail}\, msgchan$

At any time the acknowledgement at the head of *ackchan* can be lost.

LoseAck

$\Delta AckChan$

$ackchan \neq \langle\,\rangle$
$ackchan' = \text{tail}\, ackchan$

Behavioural Modelling in Z

- move from an initial state to successor states by a sequence of enabled operations
- no inbuilt constraints on the selection of enabled operations
- history (trace) constraints must be explicitly introduced with history variables

- non-determinism in both operation specification and in the selection of enabled operations

5.3 Free Types in Z

The Z free type construct[113] is a way of combining two defined sets into a new set. As Z is based on typed set theory, it is invalid to write:

$$Value == \mathbb{B} \cup \mathbb{Z} \cup \mathbb{C}$$

where $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{C}$ denote booleans, integers and characters respectively. In Z, a more elaborate approach (free type definition) is used to model the above situation:

$$Value ::= bool\langle\langle \mathbb{B} \rangle\rangle \mid int\langle\langle \mathbb{Z} \rangle\rangle \mid char\langle\langle \mathbb{C} \rangle\rangle$$

In the above free type definition, the *bool*, *int* and *char* constructors are injections (total one-to-one functions) from the sets $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{C}$ to *Value*.
A consequence of using free types is that disjoint sets can be labeled by the names of constructors and grouped together. For instance, an element, '3', of $\mathbb{Z}$ (integers) corresponds to an element, '$int(3)$', of *Value*; an element of *Value*, say '*val*' which is in the range of *int*, corresponds to an element '$int^{\sim}(val)$' of $\mathbb{Z}$ (where the super-script $^{\sim}$ denotes function inverse).
Free type definitions can be recursive and therefore recursive structures can be represented by the free type definitions. For example, a simple linked-list can be specified in Z by using the free type definition as

$$LinkedList ::= nil \mid node\langle\langle \mathbb{Z} \times LinkedList \rangle\rangle$$

This introduces a new type *LinkedList*, a constant *nil* of type *LinkedList* and an injective constructor function *node*, that, given an integer and a linked-list, returns a linked-list. For example, elements '*nil*', '$node(3, nil)$', '$node(1, node(3, nil))$' etc are members of *LinkedList*.
A free type definition can be transferred into other more primitive Z constructs, namely given types and a set of constructor functions (axiomatic descriptions). For instance, the free type *LinkedList* can be transferred into the following definitions:

$$[LinkedList]$$

$$
\begin{array}{|l}
nil : LinkedList \\
node : (\mathbb{Z} \times LinkedList) \rightarrowtail LinkedList \\
\hline
\textbf{disjoint} \langle \{nil\}, \operatorname{ran} node \rangle \\
\forall\, W : \mathbb{P}\, LinkedList \bullet \{nil\} \cup node(\!|\, \mathbb{Z} \times W \,|\!) \subseteq W \Rightarrow LinkedList \subseteq W
\end{array}
$$

In the above axiomatic definition, the first predicate constrains the range of *node* so that it does not contain the constant *nil*. The second predicate constrains the set *Linklist* to be closed under the constructor *node*. As the domain of *node*, $\mathbb{Z} \times$ *LinkedList*, is finitary[1], the two predicates implies

$$\langle \{nil\}, \operatorname{ran} node \rangle \ \textbf{partition}\ LinkedList$$

For a detailed discussion of the theoretical issues, such as the issue of consistency of free types, see [6, 102].
The free type construct is used in Chapter 4 to define the abstract syntax of simple expressions in Z. In Chapter 5, the free type construct is compared to the class-union construct.

[1]Examples of finitary constructions include finite sets $\mathbb{F}\, S$, Cartesian products $S \times T$, finite functions $S \nrightarrow T$, etc and any composition of finitary constructions.

Chapter 6

Object-Z and Advanced Object Modeling Techniques

6.1 Object-Z Basics

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. The Object-Z specification language has been developed at the University of Queensland. There are other object-oriented extensions to Z and VDM, such as MooZ[86], OOZE[1], Z++[70], Fresco[125] etc. Among them, Object-Z is regarded by some as the most mature[71]. The semantics of Object-Z have also been developed in [35, 36, 45, 46, 103, 105] [1].

6.1.1 Class

The essential extension to Z in Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.
Syntactically, a class definition is a named box, optionally with generic parameters. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas.

[1] The semantics foundations for object identity[37] has been developed in [45, 46], where the work is related to the Z base standard project [14]. Axiomatic semantics of Object-Z has been developed in [105] by extending the logic $\mathcal{W}$[129].

ClassName[*generic parameters*]

inherited classes

state schema

initial state schema

operation schemas

A generic collection example

Consider the following specification of the generic class $Collection[T]$ which denotes a collection of elements of T. The class contains operations to add elements to, and delete elements from, the collection.

Collection$[T]$

$max : \mathbb{N}$
$elemts : \mathbb{P}\ T$

$\#elemts \leq max$

INIT

$elemts = \varnothing$

Add

$\Delta(elemts)$
$x? : T$

$x? \notin elemts$
$elemts' = elemts \cup \{x?\}$

Delete

$\Delta(elemts)$
$x! : T$

$x! \in elemts$
$elemts' = elemts \setminus \{x!\}$

The state schema is nameless and contains declarations (the attributes) above the short dividing line and a predicate (class invariant) below the line. In this example, it has one attribute *elemts* denoting a set of elements of the generic type T. The class invariant stipulates that the size of the set cannot exceed the number *max*.
The initial state schema is distinguished by the keyword INIT. The state schema is implicitly included in the initial state schema. In this example, an initialised collection contains no elements of T (i.e. *elemts* is the empty set).
The remaining two schemas are operation schemas. Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value (further discussion on Δ-lists can be found in Chapter 3). If an attribute does not appear in the Δ-list of any operation of a class, then the attribute is a constant. In the class $Collection[T]$, the attribute *max* is a constant.

Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Hence the class invariant holds at all times: in each possible initial state and before and after each operation.
In this example, operation *Add* adds a given input $x?$ to the existing set provided the set has not already reached its maximum size (an identifier ending in '?' denotes an input). Operation *Delete* outputs a value $e!$ defined as one element of *elemts* and reduces *elemts* by deleting $e!$ from the original set (an identifier ending in '!' denotes an output).

6.1.2 Inheritance

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.
Essentially, all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialisation schemas of inherited classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.
Inheritance in Object-Z can be used to define a new class by extending an existing class. For instance, the generic class $DAG[T]$ denoting a directed acyclic graph can be defined by inheriting $Collection[T]$.

DAG[*T*]

Collection[*T*]

$$linkto : T \leftrightarrow T$$

$$\operatorname{dom} linkto \cup \operatorname{ran} linkto \subseteq elemts$$
$$\nexists e : elemts \bullet (e, e) \in linkto^+$$

Delete

$$\Delta(linkto)$$

$$linkto' = \{x!\} \lhd linkto \rhd \{x!\}$$

AddLink

$$\Delta(linkto)$$
$$from?, to? : T$$

$$\{from?, to?\} \subseteq elemts$$
$$linkto' = linkto \ \cup \{(from?, to?)\}$$

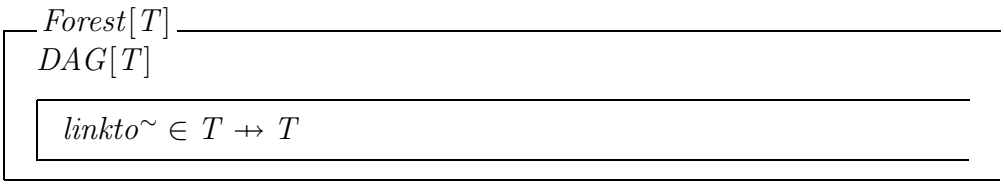
DeleteLink

$$\Delta(linkto)$$
$$from?, to? : T$$

$$(from?, to?) \in linkto$$
$$linkto' = linkto \setminus \{(from?, to?)\}$$

The class $DAG[T]$ inherits the state variable *elemts* and the operation *Add* from $Collection[T]$. It also includes explicitly the state variable *linkto* denoting the links between members of *elemts* and the extra operations *AddLink* and *DeleteLink*. The operation *Delete* for $DAG[T]$ is defined as the conjunction of the operation *Delete*

inherited from $Collection[T]$ and the operation *Delete* declared explicitly in $DAG[T]$. Inheritance in Object-Z can be also used to add constraints to the state schema of a derived class. For instance, a generic *Forest* class can be defined by inheriting the *DAG* class.



The additional state invariant $linkto^{\sim} \in T \nrightarrow T$ of the class $Forest[T]$ ensures that no member of *elemts* is linked by more than one member (parent) of *elemts*.

6.1.3 The Use of Generic Classes

The Object-Z classes presented so far are all generic classes. A generic class can be instantiated by substituting a specific type for its generic type parameter to define a specific class. For instance, a forest of integers can be modelled as:

$$IntegerForest \mathrel{\widehat=} Forest[\mathbb{Z}]$$

Generic classes are particularly useful for class reuse.

6.1.4 Instantiation

If A is the name of a class, the identifier A semantically also denotes the set of identities of possible objects of class A. (In particular cases, it will be clear from the context whether an identifier is being used as a name of a class or to denote the set of identities of objects of that class.) Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* of A, it is with the understanding that we are in fact referring to an object whose identity is in A. Because object identities uniquely identify objects, no ambiguity arises.
Objects may have object references as attributes, i.e. conceptually, an object may have constituent objects. Such references may either be individually named or occur in aggregates. For example, the declaration $c : C$ declares c to be a reference to an object of the class described by C. A declaration $c, d : C$ need not mean that c and d reference distinct objects. If the intention is that they do so at all times, then the predicate $c \neq d$ would be included in the class invariant.
The term $c.att$ denotes the value of attribute *att* of the object referenced by c, and $c.Op$ denotes the evolution of the object according to the definition of *Op* in the class C.

Example: Two Forests

Suppose we want to model an aggregate of two forests (f_1 and f_2) with operations to add an element into a given forest f_1 or f_2, to delete an element from f_1 or f_2 (nondeterministically selected), to transfer an element from f_1 to f_2 and to relocate all the elements of f_1 into f_2 one by one. (One aim of this example is to demonstrate the use of different kinds of operators in Object-Z.)

TwoForests[T]

$f_1, f_2 : Forest[T]$

$f_1.elemts \cap f_2.elemts = \varnothing$

INIT

$f_1.\text{INIT} \land f_2.\text{INIT}$

SelectForest

$f? : Forest[T]$

$(f? = f_1) \lor (f? = f_2)$

$AddOne \mathrel{\widehat=} SelectForest \bullet f?.Add$

$DeleteOne \mathrel{\widehat=} f_1.Delete \Box f_2.Delete$

$Transfer \mathrel{\widehat=} f_1.Delete \parallel f_2.Add$

$RelocateAll \mathrel{\widehat=} Transfer \fatsemi RelocateAll$
$\Box$
$[f_1.elemts = \langle\,\rangle \lor \#f_2.elemts = f_2.max]$

The declaration $f_1, f_2 : Forest[T]$ models an aggregate of two forest objects.
The evolution of a constituent forest object f_1 or f_2 in the aggregate is affected by defining a selection environment such as *SelectForest* that selects a forest (f_1 or f_2); the operation *AddOne* is then applied to the chosen object. (The notation $schema_1 \bullet schema_2$ means that variables declared in the signature of $schema_1$ are accessible when interpreting $schema_2$.)
The choice operator '$\Box$' used in the definition of *DeleteOne* indicates non-deterministic choice of one component operation from those component operations with satisfied preconditions.
The parallel operator '$\parallel$' used in the definition of *Transfer* achieves inter-object communication: the operator conjoins constraints and equates variables with the same name and also equates and hides any input variable to one of the components of $\parallel$ with any output from the other component that has the same basename (i.e. the inputs and outputs are denoted by the same identifier apart from ? and ! decorations)

The sequential composition operator '$\fatsemi$' used in the definition of *RelocateAll* is similar to the sequential composition operator defined in the Z notation; it behaves like forward relational composition. The operation *RelocateAll* is recursively defined. The recursion is a relational composition chain of *Transfer* operations terminated by the operation

$$\left[f_1.elemts = \langle\,\rangle \lor \#f_2.elemts = 100 \right].$$

The operator '$\Box$' in the operation *RelocateAll* behaves deterministically as exactly one precondition of the two branch operations is true.
The formal semantics of the Object-Z operators, such as '$\land$', '$\|$', '$\Box$' and '; ' (including the recursive operations) are formally defined in [104, 105].

6.1.5 Polymorphism

In Object-Z, the notation $a : \downarrow A$ means that a is an identity of an object in class A or in any subclasses (by inheritance) of A. For instance, declaration $x : \downarrow Collection[T]$ introduces x as a reference to an object of class $Collection[T]$ or any (direct or indirect) derivative of $Collection[T]$, such as $DAG[T]$ and $Forest[T]$.
Another mechanism for polymorphism in Object-Z is the *class union* construct [29], which explicitly lists a collection of classes.
An overview of Z and Object-Z has been presented in this chapter. The following chapters present the main contributions of this thesis.

6.2 The Role of Secondary Attributes

When specifying an object, its state is captured by the values of its attributes and its potential behaviour is determined by the specification of its operation. In general, there are many ways to model an object. For example, the shape of a triangle can be modelled as an object with three attributes representing either (the lengths of) the three sides of the triangle, or two sides and (the size of) their included angle, or two angles and the side between the two angles. In general, it is debatable which attributes are more important. However, given a specific system, there is normally an implied preference and an explicit indication of this preference can add clarity to the specification. In large and complex systems, clarity becomes particularly important. Rumbaugh[99] suggests that the contents of an object should be separated into two parts — *base* (primary) information and *derived* (secondary) information for clarity representation. For instance, if the sides of a triangle are given preference, then attributes a, b, c of *Triangle* become the base attributes and other attributes become derived. The notion of secondary attributes has also been realised in [37], where

a concept similar to Rumbaugh's derived attribute, namely *dependent variable*, is introduced into the Object-Z. Although Rumbaugh's introduction of the concept of derived information is informal, it provides a good starting point for discussion of the notion of secondary attributes in Object-Z. This section clarifies the roles and explores the implications of secondary attributes in formal modelling.

6.2.1 Secondary Attributes and Delta Lists

In this section, we use the specification of a simple hotel management system to illustrate the semantic relationship between secondary attributes and Δ-lists.

A Hotel Management System

Consider a simple hotel management system. Suppose the system stores information on rooms and their occupancy (total number of rooms in the hotel is 100). A client may *CheckIn* to an unoccupied room or *CheckOut* of an occupied room. Vacant rooms can be listed at any time (*ListVacancies*).
Let [*Room*] represent the type of a room in a hotel. This simple system can be modelled in Object-Z as:

HotelManagementSystem

$$rooms : \mathbb{P}\, Room$$
$$occupied : \mathbb{P}\, Room$$
$$\Delta$$
$$vacant : \mathbb{P}\, Room$$
$$occupied \subseteq rooms$$
$$vacant = rooms \setminus occupied$$

INIT

$$\#rooms = 100$$
$$occupied = \varnothing$$

CheckIn

$$\Delta(occupied)$$
$$x? : Room$$
$$x? \in vacant$$
$$occupied' = occupied \cup \{x?\}$$

CheckOut

$$\Delta(occupied)$$
$$x? : Room$$
$$x? \in occupied$$
$$occupied' = occupied \setminus \{x?\}$$

ListVacancies

$$rooms! : \mathbb{P}\, Room$$
$$rooms! = vacant$$

The primary attributes are *rooms* and *occupied* while the only secondary attribute is *vacant*. The dependency of the secondary attribute *vacant* on primary attributes *rooms* and *occupied* is specified in the second conjunct of the class invariant. In Object-Z, the declaration of primary and secondary attributes are syntactically separated by the Δ symbol, which indicates that secondary attributes are implicitly included in the Δ-list of every operation. The understanding of the Δ-list of an operation in Object-Z is that attributes which are so listed are subject to change. Therefore secondary attributes are always subject to change whenever any operation is invoked. Primary attributes not included in the Δ-list of an operation are implicitly unchanged on application of the operation. For instance, if operation *CheckIn* is applied, the predicate $rooms' = rooms$ is implicitly included, whereas the other attributes, *occupied* and *vacant*, are subject to change. The changes are specified by the predicate of *CheckIn* and the class invariant. Note that even though the secondary attribute *vacant* is implicitly included in the Δ-list of operation *ListVacancies*, it remains unchanged because both attributes, *rooms* and *occupied*, being unlisted are unchanged and *vacant* is functionally determined by them.
Notice that the *HotelManagementSystem* class can be defined by using the generic class $Collection[T]$ (appeared in Chapter 2) with appropriate renaming, i.e.

HotelManagementSystem

$Collection[Room][occupied/elemts, CheckIn/Add, CheckOut/Delete]$

$rooms : \mathbb{P}\, Room$
Δ
$vacant : \mathbb{P}\, Room$

$occupied \subseteq rooms$
$vacant = rooms \setminus occupied$

INIT

$max = 100$
$\#rooms = max$

ListVacancies

$rooms! : \mathbb{P}\, Room$

$rooms! = vacant$

The *HotelManagementSystem* example has illustrated the underlying semantics of a secondary attribute in terms of the Δ-list. In the following sections, the different roles of secondary attributes in formal system modelling are demonstrated.

6.2.2 Adding Clarity in Specification

In this section, we use the specification of a complex number variable to illustrate how secondary attributes can simplify and clarify a specification.

A Complex Variable

Consider a complex variable with two component attributes, a real part and an imaginary part, and suppose the variable can be changed by the following few operations (Figure 6.1):

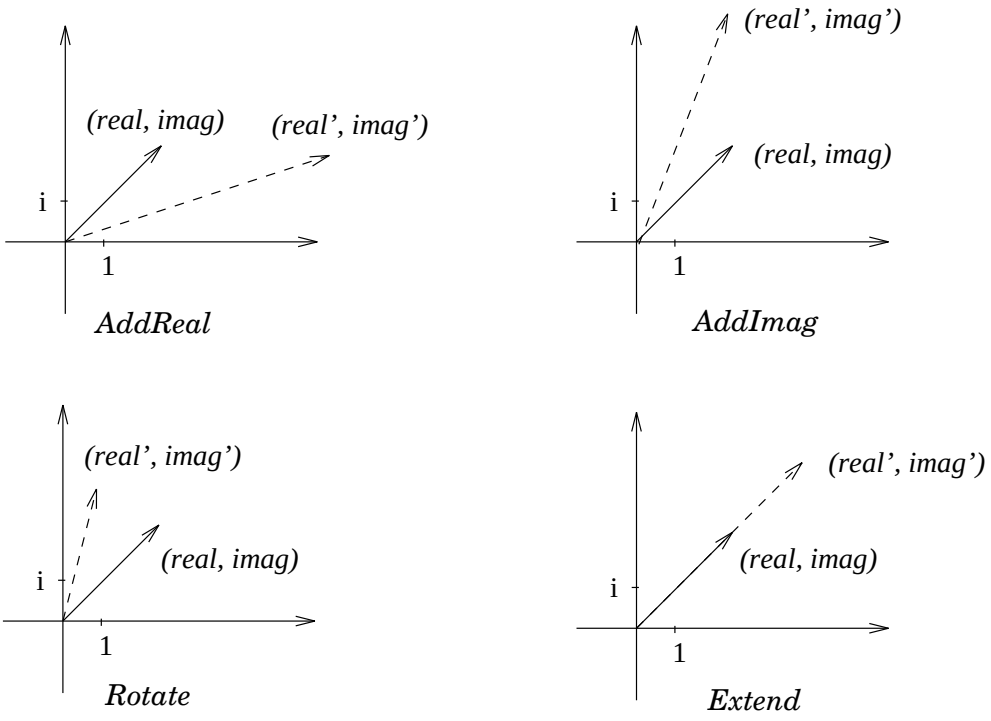


Figure 6.1: Operations to manipulate a complex number variable.

(1) add a given real number to its real part,

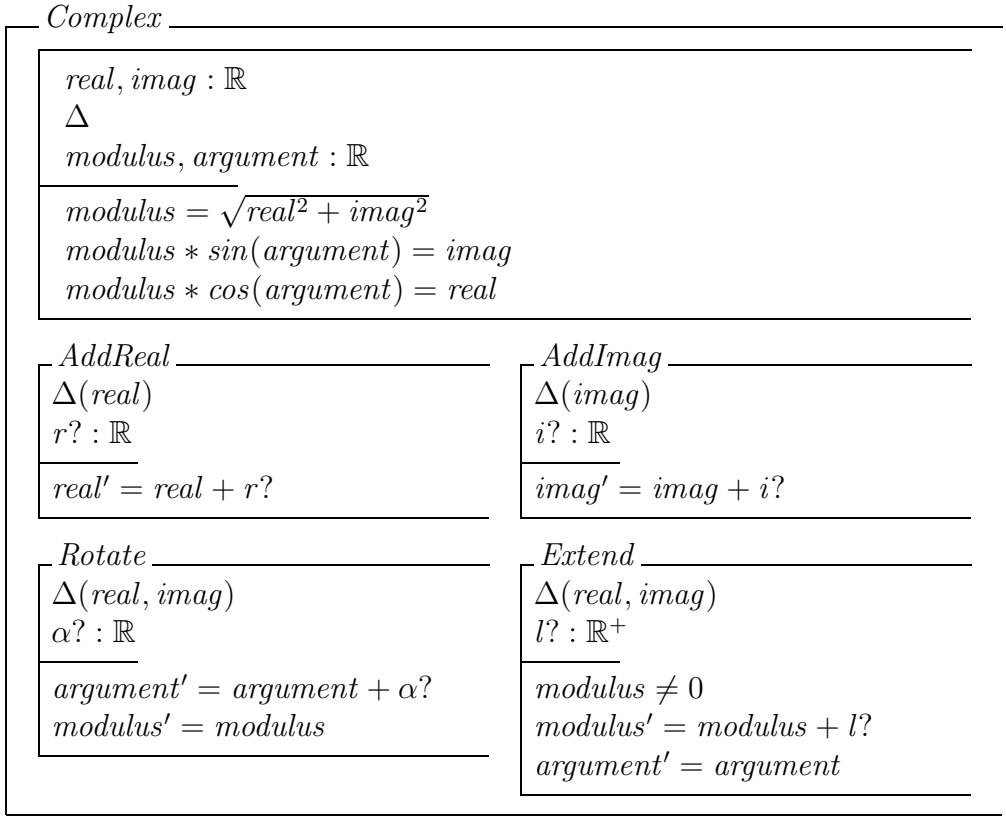
(2) add a given real number to its imaginary part,

(3) rotate by a given angle and

(4) extend the modulus by a given positive real number.

When modelling a complex variable as an object, if only two attributes, *real* and *imag*, are considered, then operations *Rotate* and *Extend* (particularly *Rotate*) are difficult and cumbersome to construct. However, if additional secondary attributes, such as the modulus and the argument, are introduced, then operations *Rotate* and

Extend can be easily defined. A suitable model of a complex variable in Object-Z is the class *Complex*.



The dependency of the secondary attributes (*modulus*, *argument*) on primary attributes (*real*, *imag*) is specified in the class invariant.
The benefit of introducing the secondary attributes *modulus* and *argument* in *Complex* is that operations *Rotate* and *Extend* are easily defined by expressing change in terms of them; moreover, in this case the required changes to the primary attributes *real* and *imag* are unambiguously deducible. Another interesting point demonstrated by this example is that, in a particular situation, the value of a secondary attribute need not be uniquely determined. For instance, when $modulus = 0$, the value of the secondary attribute *argument* is arbitrary and even if $modulus \neq 0$, *argument* is only determined modulo 2π.

6.2.3 Dependency on Environment: Sharing

In this section, we consider a system in which a secondary attribute plays an important role in capturing a notion of object sharing; also, the value of the secondary attribute

is not determined by its objects' primary attributes but by the environment.

A Game Sharing Group

Consider a situation in which a group of personal computer (PC) users share their computer games through a communication device. Each member (player) of the group owns a PC and a exclusive set of computer games stored in the PC. Individual players can add a new game to or delete an existing game from their PC. Any game of any player can be accessed (played) by all players of the group. Games can also be transferred between players. See Figure 6.2 for an illustration. The following is an Object-Z specification of this system.
Firstly, let skeletal class *Game* represent a computer game.

Game

Play

...

...

A player of the group can be modelled as:

Player

$my\_games : \mathbb{P}\, Game$
Δ
$other\_games : \mathbb{P}\, Game$

AddGame

$\Delta(my\_games)$
$g? : Game$

$g? \notin my\_games$
$my\_games' = my\_games \cup \{g?\}$

DelGame

$\Delta(my\_games)$
$g? : Game$

$g? \in my\_games$
$my\_games' = my\_games \setminus \{g?\}$

Choose

$g? : Game$

$g? \in (my\_games \cup other\_games)$

$Play \mathrel{\widehat=} Choose \bullet g?.Play$

The primary attribute *my_games* denotes the local games. The secondary attribute *other_games* denotes all other public games, i.e. games not local to a player but which can be accessed by the player. However such an intention cannot be expressed

as an invariant of the class *Player* because, for any object $p : Player$, the value of $p.other\_games$ depends on the games which are owned by players other than p in the group. Therefore the value of $p.other\_games$ is dependent on the environment of p. The precise meaning of this attribute is defined by the state invariant of the system class below.

GameSharingGroup

$$players : \mathbb{P}\, Player$$
$$\Delta$$
$$games : \mathbb{P}\, Game$$

$$\forall\, p_1, p_2 : players \bullet p_1 \neq p_2 \Rightarrow p_1.my\_games \cap p_2.my\_games = \varnothing$$
$$games = \bigcup \{p : players \mid p.my\_games\}$$
$$\forall\, p : players \bullet p.other\_games = games \setminus p.my\_games$$

Select

$$p? : Player$$

$$p? \in players$$

SelectTwo

$$p_1?, p_2? : Player$$

$$p_1? \neq p_2?$$
$$\{p_1?, p_2?\} \subseteq players$$

$$Play \mathrel{\widehat=} Select \bullet p?.Play$$
$$NewGame \mathrel{\widehat=} Select \bullet p?.AddGame$$
$$DiscardGame \mathrel{\widehat=} Select \bullet p?.DelGame$$
$$Transfer \mathrel{\widehat=} SelectTwo \bullet p_1?.DelGame \wedge p_2?.AddGame$$

The system consists of a set of players. The first predicate of the class invariant ensures that each player's *my_games* is a distinct set of games. The other predicates of the class invariant ensures that the value of *other_games* of a player in a group depends on those games which are locally contained by any other player of the group. Incidentally, the definition is facilitated by the introduction of the secondary attribute *games*. This example demonstrates that, given an object $p : Player$ in the group, the value of $p.other\_games$ is determined by the existence of some game objects in the environment and those game objects are not even referred to by the primary attribute $p.my\_games$. Therefore the value $p.other\_games$ is subject to change even without applying an operation to p. To illustrate this in detail, let's consider an instance of *GameSharingGroup* (Figure 6.2).

Suppose the system performs an operation *DiscardGame* ($Select \bullet p?.DelGame$) or *NewGame* ($Select \bullet p?.AddGame$); if *Player*2 is selected (i.e. $p? = Player2$) and a

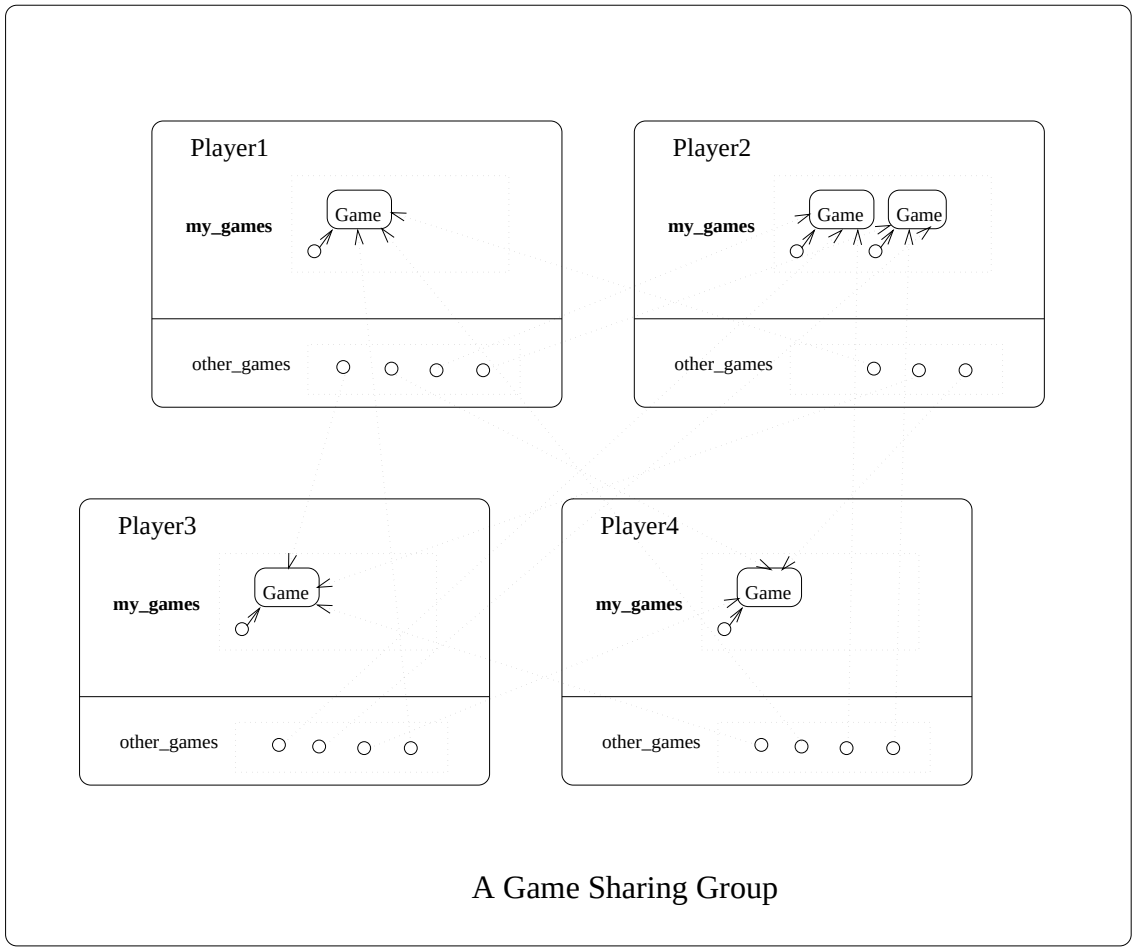


Figure 6.2: A group of PC users sharing games.

game is deleted from (added to) $Player2.my\_games$, then the game is deleted from (added to) all $Player1.other\_games$, $Player3.other\_games$ and $Player4.other\_games$. Although every game is accessible by all the players of the group, the existence of local games (in my_games) is controlled solely by the player; i.e. for any player p, $p.my\_games$ can be changed only by performing $p.AddGame$ or $p.DiscardGame$ whereas p has no control of the existence of $p.other\_games$. In the definition of $Player$, this difference is captured precisely by distinguishing my_games and $other\_games$ as primary and secondary respectively.

In the next section, we investigate the role of secondary attributes in modelling recursive structures.

In this section, the implications of applying secondary attributes in formal specifica-

tion have been explored. Two examples analyse and demonstrate 2 different usages of secondary attributes in formal modelling, namely:

- improving the clarity and simplicity of object-oriented system specifications in general, and
- capturing a notion of object sharing.

We have seen in this section how the notion of secondary attributes enables invariant relationships between objects to be clearly specified. This is particularly valuable when designing large or complex computer systems.
Secondary attributes also facilitate the development of the notations for object containment (next section) in Object-Z specifications.

6.3 Object Containment

In object-oriented systems, references between objects are maintained so as to facilitate inter-object communication[13, 43, 87]. For example, consider a banking system consisting of account objects, customer objects and bank objects. In such a system, a customer object will have attributes whose values reference account objects. These references enable customers to operate their accounts (e.g. to make deposits, withdrawals, etc.). If a bank permits shared accounts, several customers may even reference the same account. In addition, an account object may well have an attribute whose value is a reference to a customer object, so that access to the account can be authorised. Furthermore, a bank object will have attributes whose values reference account objects, so that the bank can operate the accounts for the purpose of adding charges or changing credit limits.
The association between objects determined by the object references in a system will generally result in a complex structure whose design and specification is a crucial part of the development and implementation of the system. An aim of our work is to look at ways of capturing formally object reference structures that occur frequently in object-oriented systems.
As an example, suppose that in a banking system no account is shared between banks. In this case it would follow that any account referenced by one bank is distinct from any account referenced by a different bank. When giving a formal specification of this banking system, such an important structural property of the object references should be clearly captured by the specification, ideally as a global system invariant.
As another example, consider a system consisting of car objects and wheel objects. Each car will have attributes that reference its wheel objects. If wheels are not shared between cars, distinct car objects will reference distinct wheel objects.
In this chapter we formally investigate the general nature of the object references implicit in the last two examples. The properties of such object references are captured

within a formal framework and incorporated into the Object-Z specification language as predicate rules (resembling in flavour the axioms of combinatorial geometric structures such as projective planes).

The example above of the car and its wheels suggests a notion of geographical location (a car physically contains its wheels) and for this reason we refer to the resulting object-references as *containment*, i.e. a car object is said to (directly) *contain* the wheel objects it references. As the banking system illustrates, however, the structure of object containment also arises in situations where the objects have no relevant geographical location: it would be inappropriate to think of an account object as being physically contained within a bank object, but nevertheless, because distinct banks reference distinct accounts, a bank object is said to (directly) contain the account objects it references. In the section this notion of object containment is precisely characterised by its (geometric) properties.

In general, some attributes of an object will reference contained objects, and other attributes will not. For example, in addition to the references to its contained wheel objects, a car object may well reference a person object corresponding to the owner of the car. We would not expect the person reference to denote object containment as a person may well own several cars. That is, in general an object 'has a' set of references to other objects, only some of which may be 'contained' references in the sense defined in this section.

Various notions of object association, such as aggregation and composition, have been discussed in the literature[17, 18, 27, 58, 74, 91, 92]. Our notion of containment has features in common with them, but is not identical to any. A notion of containment is also defined by Kilov and Ross[68] in extended Object-Z and used (as a hierarchical subordination) in a way different from the treatment in this section.

In this section, the properties implied by object containment are modelled by a constraint relationship between objects. Two examples are presented in Section 7.1 to demonstrate that this containment relationship can be captured explicitly in Object-Z by predicates in the state invariant of a class. However, when a system is large and complex, capturing the properties of the containment relationship explicitly in this way is cumbersome. Therefore, in Section 7.2 an extension of the Object-Z notation is introduced to capture directly the geometric notion of object containment.

The notion of object containment is also partially supported by some object-oriented programming languages. Object containment in object-oriented programming languages such as Eiffel[88] (*expanded* class type) and C++[114] (object value type) is discussed and compared to our notion in Section 7.3.

In many object-oriented systems, object containment is closely related to object access: an object can access any object it contains. However, we view object containment as a purely geometric notion, quite distinct from the issue of object access. An example in Section 7.4 illustrates this. Chapter 8 further discusses the distinction

between the notions of object containment and exclusive object control.
Objects may overlap or share contained objects, e.g. two rooms may share a wall in a building. An object may also contain only part of another object, e.g. a street may pass though and hence be partially contained by several suburbs. In Section 7.5 this generalised view of object containment is illustrated and formally captured in Object-Z.
Finally, the geometry of object containment can be applied to simplify the specification of abstract recursive structures, such as trees and directed acyclic graphs. Examples are presented in Section 7.6.

6.3.1 Capturing the Properties of Containment

In this section the properties of object containment are stated precisely. The notion of object containment suggests a forest-like geometric relationship between contained objects. As a motivating example, in Figure 6.3 u, v, w, x, y and z denote objects where u directly contains w and x, and similarly w directly contains y and z, but u and v are not related by containment, and neither are w and x. In this case object u *indirectly* contains y and z.

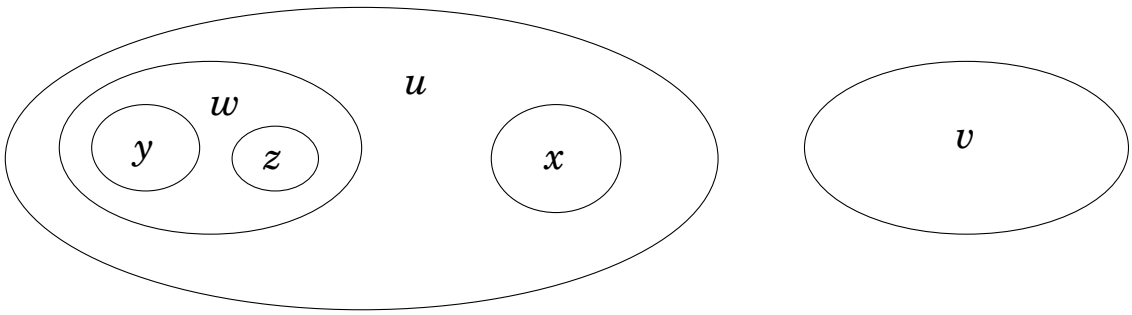


Figure 6.3: The geometry of object containment

Figure 6.3 is based on an idea of geographical object location. However, object containment can arise in systems where the objects are not related geographically. Nevertheless, the figure suggests two geometric ideas that characterise the general notion of object containment[2]:

(1) an object cannot directly or indirectly contain itself; and

[2]In our work, the term 'geometry' is used in a combinatorial sense. A geometric structure is defined by the rules that determine whether or not one object 'contains' another. Like projective geometries, no geographical notion of physical location is implied by the term.

(2) an object cannot be directly contained within two distinct objects.

To capture these ideas formally, let

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation whereby

$$ob_1 \; \underline{dcon} \; ob_2$$

if and only if object ob_1 has a reference to a $\underline{d}$irectly $\underline{con}$tained object ob_2. Put another way, $dcon$ is the set of all those ordered pairs (ob_1, ob_2) of (identities of) objects in the system where ob_1 directly contains ob_2.
The first condition above requires that

$$\nexists ob : \mathbb{O} \bullet ob \; \underline{dcon^+} \; ob$$

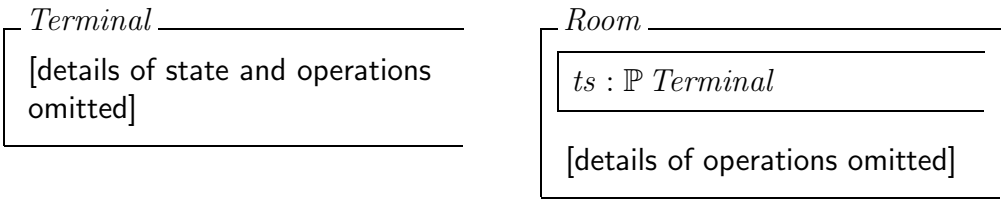
where $dcon^+$ is the transitive closure of $dcon$. The second condition requires that

$$dcon^\sim \in \mathbb{O} \nrightarrow \mathbb{O}$$

(i.e. the inverse of the relation $dcon$) is a partial function.
In any system, the relation $dcon$ is not static; it will change dynamically if object relocation is permitted, i.e. if there are operations in the system that affect the containment geometry. This is discussed further in Section 7.2.4.
When specifying an object-oriented system using Object-Z, the above two properties of object containment can be captured explicitly by class invariants, as is illustrated in the following two examples.

Example: Terminal Location

Consider the situation where a campus consists of a set of buildings, with each building containing a set of rooms and each room containing a (possibly empty) set of terminals. A specification in Object-Z would be



Building

$rs : \mathbb{P}\, Room$

$\forall\, r_1, r_2 : rs \bullet$
$\quad r_1 \neq r_2 \Rightarrow$
$\quad\quad r_1.ts \cap r_2.ts = \varnothing$

[details of operations omitted]

Campus

$bs : \mathbb{P}\, Building$

$\forall\, b_1, b_2 : bs \bullet$
$\quad b_1 \neq b_2 \Rightarrow$
$\quad\quad b_1.rs \cap b_2.rs = \varnothing$
$\quad\quad \forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet$
$\quad\quad\quad r_1.ts \cap r_2.ts = \varnothing$

[details of operations omitted]

The class invariant of the *Building* class captures the idea that no terminal can be in two distinct rooms in a building. Similarly, the class invariant of the *Campus* class captures the idea that no room can be in two distinct buildings of the campus. Furthermore, despite the fact that the predicate of the *Building* class states that no terminal can be in two distinct rooms, as this applies only to the rooms of a given building it says nothing about rooms in distinct buildings. Hence the predicate

$$\forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet r_1.ts \cap r_2.ts = \varnothing$$

needs to be conjoined to the predicate of the *Campus* class.
Clearly, capturing the properties of object containment explicitly in this way is cumbersome, particular if the system is large and complex. We would like to be able to give a global invariant that captures directly the condition that distinct rooms anywhere contain distinct terminals, and distinct buildings anywhere contain distinct rooms. The condition that distinct rooms contain distinct terminals, for example, is not an internal invariant of the *Room* class, but rather an invariant of any system containing room objects; nevertheless, it would be convenient to be able to attach such global conditions directly to the *Room* class. A way of doing this is given in Section 7.2.

Example: Russian Dolls

Object containment is sometimes an important property of recursive structures. For example, consider the situation of Russian dolls (see Figure 6.4). Each doll is either solid or hollow, and each hollow doll contains another doll that is itself solid or hollow. The fundamental property of this set of recursively embedded dolls is that no doll directly or indirectly contains itself.
An object-oriented specification in Object-Z would be

$$Doll \mathrel{\widehat=} Solid \cup Hollow$$

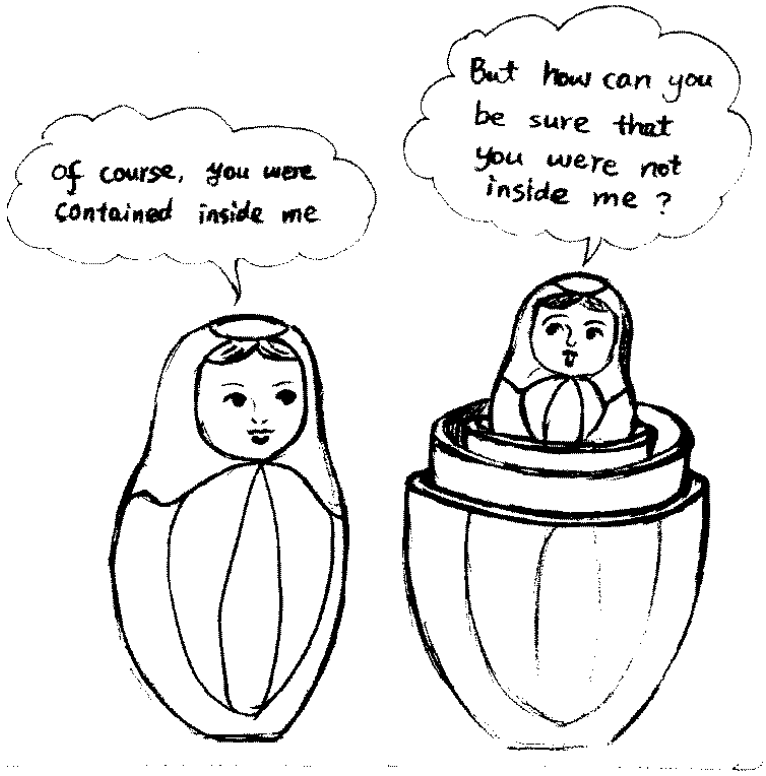


Figure 6.4: Russian Dolls.

Solid

$$dolls : \mathbb{P}\, Doll$$
$$dolls = \varnothing$$

Hollow

$$inside : Doll$$
$$dolls : \mathbb{P}\, Doll$$
$$dolls = \{inside\} \cup inside.dolls$$
$$self \notin dolls$$

The attribute *inside* identifies the doll directly contained within a hollow doll; the attribute *dolls* denotes the set of all dolls directly or indirectly contained within a doll. By specifying *Doll* to be the union of the classes *Solid* and *Hollow*, the doll identified by *inside* is either solid or hollow.
The predicate $self \notin dolls$ of the class *Hollow* ensures that a doll does not directly or indirectly contain itself.
This specification takes an object-oriented view, modelling each doll as an object with a unique identity. It could be argued that it is more complex than a functional

recursive specification using the free type. However, when object containment is captured by global predicates in Section 7.2, the object-oriented specification mimics in style the definition using free types.

6.3.2 Capturing the Geometry of Containment

In the examples in the last section, the properties of object containment were formally captured in Object-Z by writing explicit class invariants. There are several consequences of that approach:

- The invariants that result can be complex, particularly as object containment is often a significant aspect of a system's design.
- An appropriate invariant needs to be placed in each relevant class. As the invariant is capturing the same concept of object containment in each case, the specification can become repetitious.
- Only the *properties* that follow from object containment are being captured by the invariant. The geometric notion as to which objects are actually contained within a given object is not explicitly stated.

In this section, specific notation is introduced into Object-Z to capture directly the geometric relationship of object containment. This notation enables the specifier to state explicitly, as part of the class specification, which objects will be directly contained within an object of that class.
To be precise, let each class have an implicitly declared attribute

$$dcontain : \mathbb{P}\,\mathbb{O}$$

where the value of *dcontain* is the set of directly contained objects. Then in any system the relation

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

introduced in Section 7.1 is determined by

$$\forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ ob_1 \mathrel{\underline{dcon}} ob_2 \Leftrightarrow ob_2 \in ob_1.dcontain.$$

The properties of the relation *dcon* (as stated in Section 7.1) imply invariant conditions on the system that need not be stated explicitly. In terms of the attribute

dcontain these conditions are:

$$\begin{array}{l} \nexists s : \operatorname{seq} \mathbb{O} \bullet \\ \quad \#s > 1 \\ \quad \forall\, i : 1 \,..\, \#s - 1 \bullet s(i+1) \in s(i).dcontain \\ \quad s(1) = s(\#s) \end{array} \qquad [\mathsf{dc1}]$$

(i.e. no object directly or indirectly contains itself)

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \quad ob_1 \neq ob_2 \Rightarrow ob_1.dcontain \cap ob_2.dcontain = \varnothing \end{array} \qquad [\mathsf{dc2}]$$

(i.e. no object is directly contained in two distinct objects). The two predicates dc1 and dc2 are global invariants of any Object-Z specification.

Examples Revisited

With this notation the specification of both the terminal-location and Russian-dolls systems is significantly simplified.

Terminal Location

Terminal

[details of state and operations omitted]

Room

$ts : \mathbb{P}\, Terminal$

$dcontain = ts$

[details of operations omitted]

Building

$rs : \mathbb{P}\, Room$

$dcontain = rs$

[details of operations omitted]

Campus

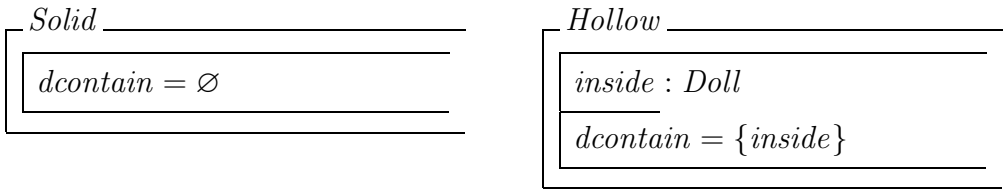
$bs : \mathbb{P}\, Building$

$dcontain = bs$

[details of operations omitted]

Russian Dolls

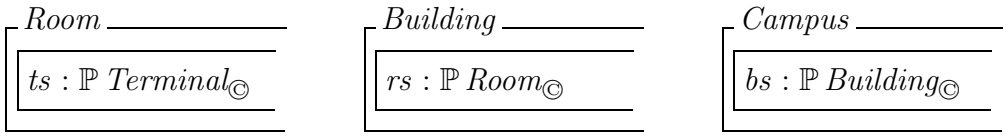
$$Doll \mathrel{\widehat=} Solid \cup Hollow$$



In both these examples, there are implicit class invariants that follow directly from the properties of *dcontain* stated above. Indeed, from these properties the explicit class invariants given in Sections 7.1.1 and 7.1.2 can be deduced.
The convention is adopted that no mention of *dcontain* need be made if there are no contained objects. Therefore the predicate $dcontain = \varnothing$ can be omitted from the class *Solid*.

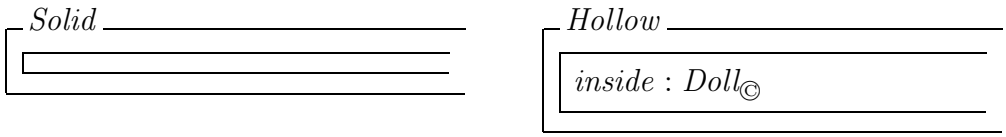
A Notational Simplification

If the role of an attribute is to always identify directly contained objects, this can be indicated when the attribute is declared by appending a subscript '$_{\copyright}$' to the appropriate type. This removes the necessity to write an explicit predicate involving *dcontain*. For example, adopting this syntactic convention, the relevant classes in the specification of the terminal location system becomes



while the specification of the Russian-dolls system becomes

$$Doll \mathrel{\widehat=} Solid \cup Hollow$$

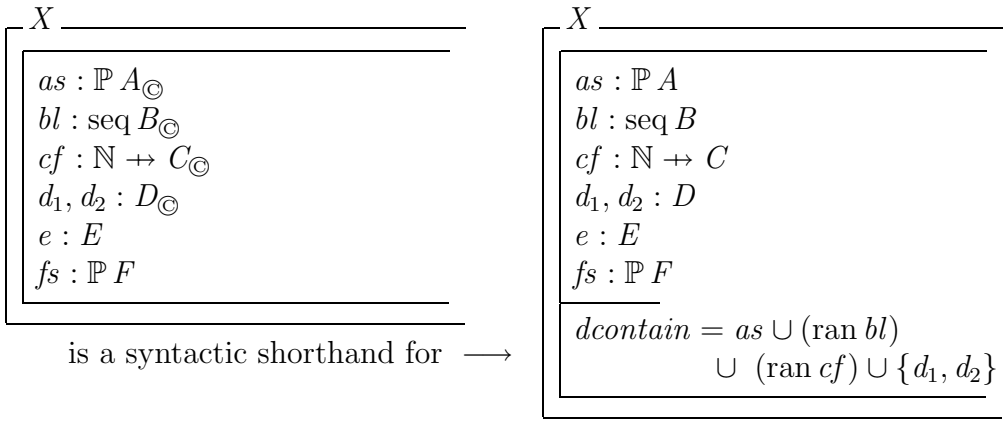


The subscript '$_{\copyright}$' is appended to the type of the attribute rather than the attribute itself because the attribute may identify a complex data structure rather than an object reference. For example, the declaration

$$ts : \mathbb{P}\ Terminal_{\copyright}$$

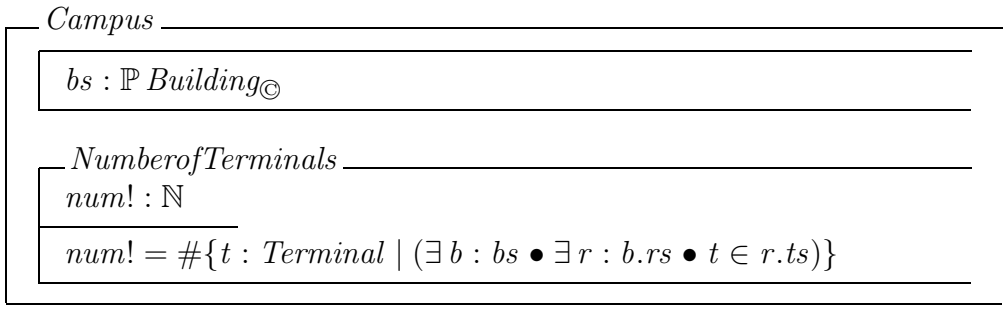
in the *Room* class declares *ts* to be a set, not an object reference. From its type, *ts* is a set of references to objects of class *Terminal*; the $_{\copyright}$ implies these references are to

contained objects. Type declarations involving $\copyright$ can be converted into declarations that directly use the attribute *dcontain* instead; for instance:



Indirectly Contained Objects

Although the introduction of the attribute *dcontain* is sufficient to capture the above properties of containment, it is often useful to explicitly identify all those objects that a given object directly or indirectly contains. For example, considering the terminal-location system, suppose an operation exists to output the number of terminals which are contained in the campus. The *Campus* class would then be



The predicate of the operation *NumberofTerminals* can be captured more easily if each class has an implicitly declared attribute

$$contain : \mathbb{P}\,\mathbb{O},$$

where the value of *contain* is the set of directly or indirectly contained objects, i.e. in terms of the relation *dcon* introduced in Section 6.3.1,

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1 \in o_2.contain \Leftrightarrow o_2 \; \underline{dcon^+} \; o_1.$$

To be precise, every class has an implicit class invariant

$$\begin{aligned} contain = \{ ob : \mathbb{O} \mid \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet \\ s(1) \in dcontain \\ s(\#s) = ob \\ \forall\, i : 1 \,..\, \#s - 1 \bullet s(i+1) \in s(i).dcontain \}. \end{aligned}$$

The predicate of the operation *NumberofTerminals* can now be written as

$$num! = \#(contain \cap Terminal).$$

Notice that in terms of the attribute *contain*, the global invariant dc1 becomes

$$\forall\, ob : \mathbb{O} \bullet ob \notin ob.contain.$$

Object Relocation

The geometry of object containment of a system is dynamic because a contained object can relocate from one container to another in the system. For example, considering again the terminal-location system, suppose operations exist to add a terminal to a room, to remove a terminal from a room and to transfer a terminal from one room to another inside a building. The appropriate part of the system specification would then become

Room

$ts : \mathbb{P}\, Terminal_{\copyright}$

Add

$\Delta(ts)$
$t? : Terminal$

$t? \notin ts \land ts' = ts \cup \{t?\}$

Remove

$\Delta(ts)$
$t? : Terminal$

$t? \in ts \land ts' = ts - \{t?\}$

Building

$rs : \mathbb{P}\, Room_{\copyright}$

SelectRoom

$r? : Room$

$r? \in rs$

Select2Rooms

$r_1?, r_2? : Room$

$r_1? \neq r_2? \land \{r_1?, r_2?\} \subseteq rs$

$Add \mathrel{\widehat=} SelectRoom \bullet r?.Add$

$Remove \mathrel{\widehat=} SelectRoom \bullet r?.Remove$

$Transfer \mathrel{\widehat=} Select2Rooms \bullet r_1?.Remove \parallel r_2?.Add$

It is to be understood that the Δ-list of the operations *Add* and *Remove* in the *Room* class implicitly includes the secondary attributes *dcontain* and *contain* as these values are subject to change whenever the value of the state variable *ts* changes.
Notice that the implicit conditions implied by the geometry of object containment will be maintained at all times[3]. For instance, it will be the case, even though it has not been stated explicitly within the *Add* operation schema that the new terminal added to the room is not already in any other room on the campus.
In some cases, a contained object may be a fixed component of its containing object, i.e. object relocation may not be possible. For example, a room is usually a fixed component of a building. This is implicitly captured by having no operation that can change the attribute *rs* in the class *Building*.

6.3.3 Containment in Programming Languages

Some object-oriented programming languages support a view of object containment. In Eiffel, for instance, if the type of an attribute is an *expanded* class, the value of the attribute will be an actual object rather than an object reference. In effect, an attribute of *expanded* class type denotes a contained object. A consequence of this is that although the internal values of a contained object can be updated directly, relocation is not possible: the contained object is treated as a fixed component. Furthermore, both value semantics and reference semantics for objects need to be defined. This contrasts with the approach adopted in this section where reference semantics for objects is uniformly maintained. Not only does this simplify the Object-Z semantics, but it permits the relocation of contained objects.

[3]In Z and Object-Z the state invariant must hold before and after each operation.

To illustrate this distinction, consider a car that contains four wheel objects and references an owner object. In Eiffel this would be

```
class Car feature
w1, w2, w3, w4 :expanded class Wheel;
owner: Person;
...
end - - class Car
```

while in Object-Z it would be

Car

$w1, w2, w3, w4 : Wheel_{©}$
$owner : Person$

$\#\{w1, w2, w3, w4\} = 4$

...

The *Car* class invariant states that the four wheels are distinct.
Pictorially, the Eiffel and Object-Z models are given in Figure 6.5, where a dotted or solid arrow denotes an object reference, with the solid arrow denoting a reference corresponding to containment. In Object-Z, operations can be specified to allow a wheel of a car to be switched to a different position or even replaced. However, in Eiffel the use of the *expanded* class type means that such operations are not possible. The value semantics of the Eiffel *expanded* class type also ensures that no aliasing[4] is possible for any object of such a class. If the source (right hand side) of an assignment is an object of *expanded* class type then the value of the source object is copied to the target; a reference to the source object is not copied. A consequence is that, in Eiffel, an object is the unique client of any object of expanded class type it contains; in effect, a client has exclusive control of its contained objects. This provides for aliasing protection, although with this approach the notions of control and containment are not distinguished; this issue is discussed further in Section 7.4.
C++ has a notion of object containment similar to that of Eiffel except it makes object containment the default class type and containment does not imply unique control; i.e. C++ does not allow the relocation of a contained object from one container to another, but it does allow a contained object to be referenced by objects other than its containing object. In C++, the car example would be

```
class Car {
Wheel w1,w2,w3,w4;
```

[4]Aliasing occurs when an object can be accessed in more than one way, see [23, 61, 87].

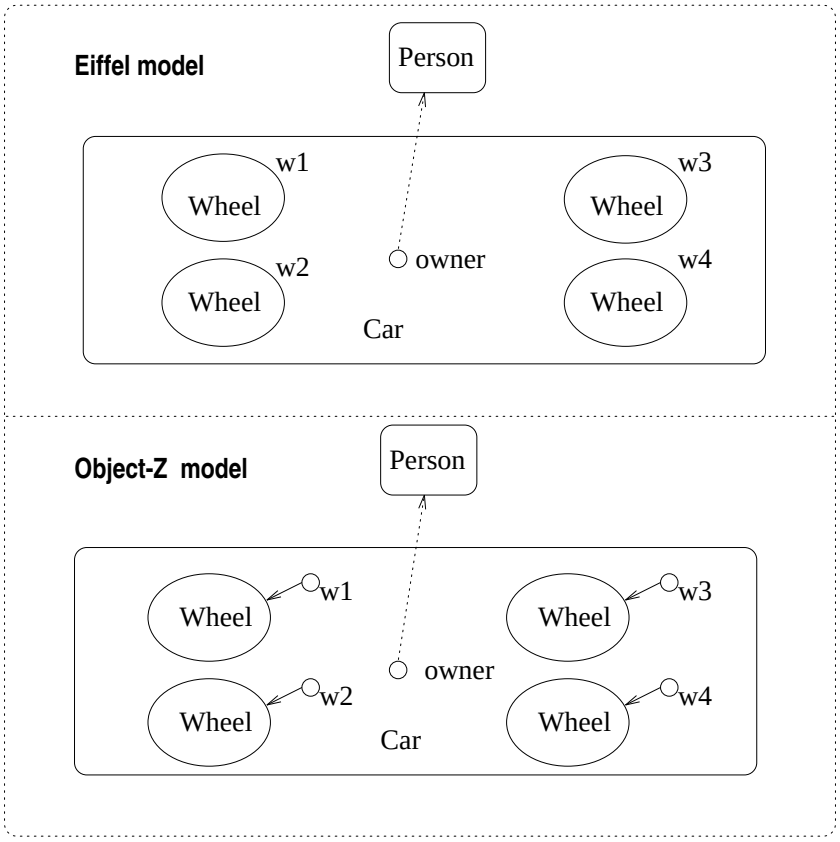


Figure 6.5: Containment in Eiffel and Object-Z

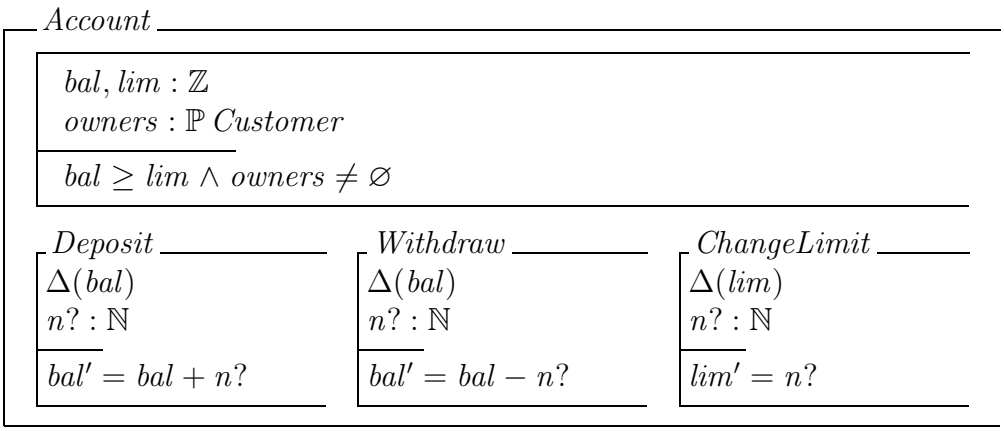
```
Person *owner;
...
}
```

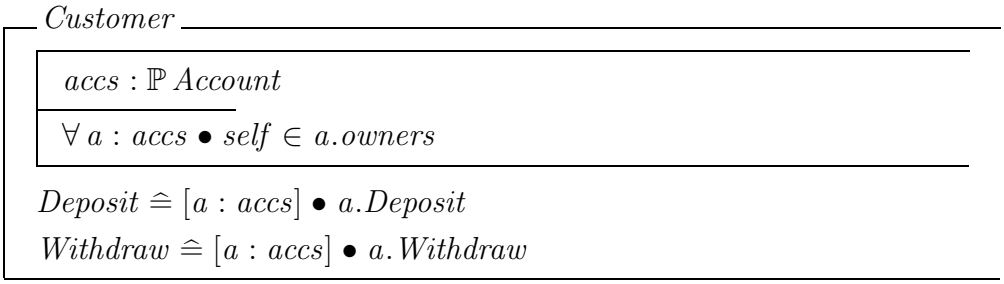
6.3.4 Containment and Access

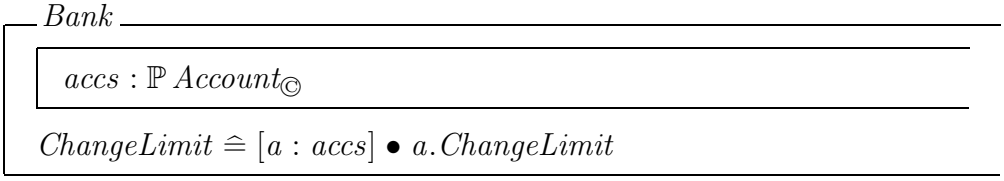
In object-oriented systems it is sometimes the case that a contained object can be accessed by only the containing object. However, the notions of containment and access are in general quite distinct and should not be confused in system modelling: containment is concerned with the relative geometry of objects; access is concerned with the right of one object to send a message to another object. Often within object-oriented systems the case arises when one object contained within another can be sent messages by a third object elsewhere in the system. This is illustrated in the following example.

Example: A Banking System

A banking system consists of account objects, customer objects and bank objects. An account has a balance, a limit below which the balance must not fall and a set of owners who can operate the account by making deposits or withdrawals. Each account is contained within a bank (i.e. an account is referenced by a unique bank) which is able to change the limit of the account. We shall suppose that an account may be shared between customers. In Object-Z this becomes







A banking system consists of a set of banks and a set of customers.

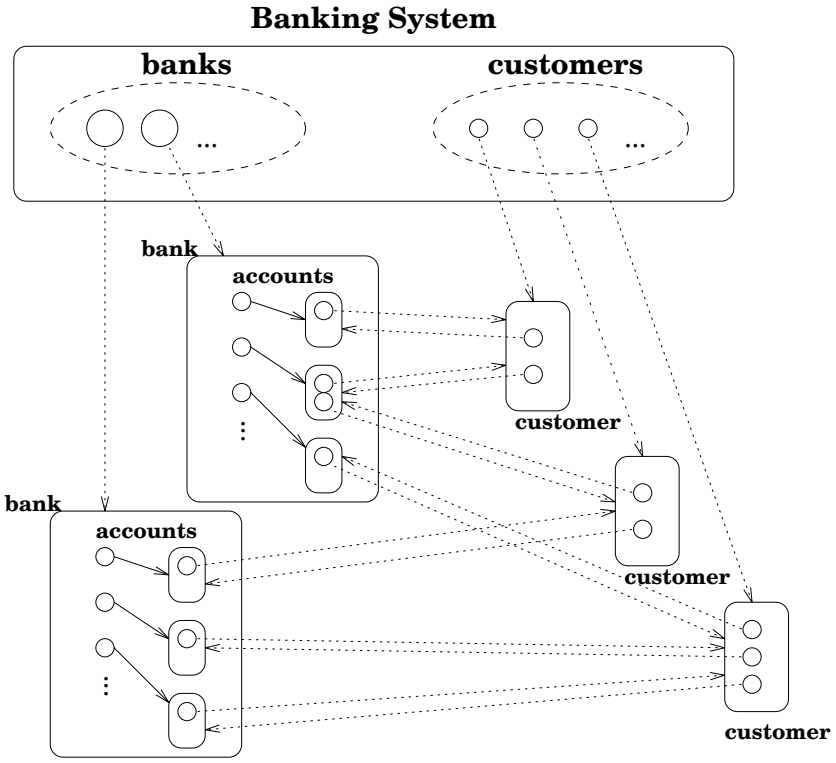


Figure 6.6: A banking system

BankingSystem

$$banks : \mathbb{P}\, Bank$$
$$customers : \mathbb{P}\, Customer$$

$$\bigcup\{b : banks \bullet b.accs\} = \bigcup\{c : customers \bullet c.accs\}$$

$$Deposit \mathrel{\widehat=} [c : customers] \bullet c.Deposit$$
$$Withdraw \mathrel{\widehat=} [c : customers] \bullet c.Withdraw$$
$$ChangeLimit \mathrel{\widehat=} [b : banks] \bullet b.ChangeLimit$$

The state invariant of this class ensures that the accounts contained in *banks* are precisely those accounts accessed by *customers*.
Because accounts are contained in banks, the set of accounts is partitioned between the banks, i.e. each account is uniquely associated with a bank. However, customers access the balance of accounts owned by them, while the banks access only the limit

of the accounts they contain. The object-reference structure of this banking system is illustrated in Figure 6.6 (where a dotted or solid arrow denotes an object reference, with the solid arrow denoting a reference corresponding to containment).
In Chapter 8, I further discuss the distinction between the notions of object containment and exclusive object control.

6.3.5 Modelling Shared Containment

The geometric notion of object containment considered so far has been that of *unique* containment, i.e. an object cannot be directly contained within two distinct objects. However, this is not the only containment geometry found within real systems. In general, objects may overlap and share contained objects, e.g. two rooms may share a wall in a building. Furthermore, in some systems an object may contain only part of another object, e.g. a street may be located in several suburbs and hence is partially contained by each of the suburbs that share it. That is, property (2) holds for unique containment, but not for a more general notion of shared containment.
The geometric complexities that arise with this notion of shared containment are illustrated in Figure 6.7. In that figure, object s is directly, but only partially, contained and shared by the three objects q, r and t. On the other hand, t is directly and uniquely contained by r. Also, s is indirectly (via t) partially contained by r. The graph on the right hand side of Figure 6.7 captures the geometric relationships between objects q, r, s and t, where a dashed (not dotted) arrow denotes direct shared containment and a solid arrow denotes direct unique containment.
Figure 6.7 suggests three ideas implicit in the notion of shared and unique containment:

(1) an object cannot directly or indirectly contain itself, regardless of whether the containment is shared or unique;

(2) an object cannot be directly uniquely contained within two distinct objects (as in Section 7.1); and

(3) for any object, its set of directly contained but sharable objects is disjoint from its set of directly uniquely contained objects.

To capture these ideas formally, let

$$sdcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation of direct but sharable containment, i.e.

$$ob_1 \ \underline{sdcon} \ ob_2$$

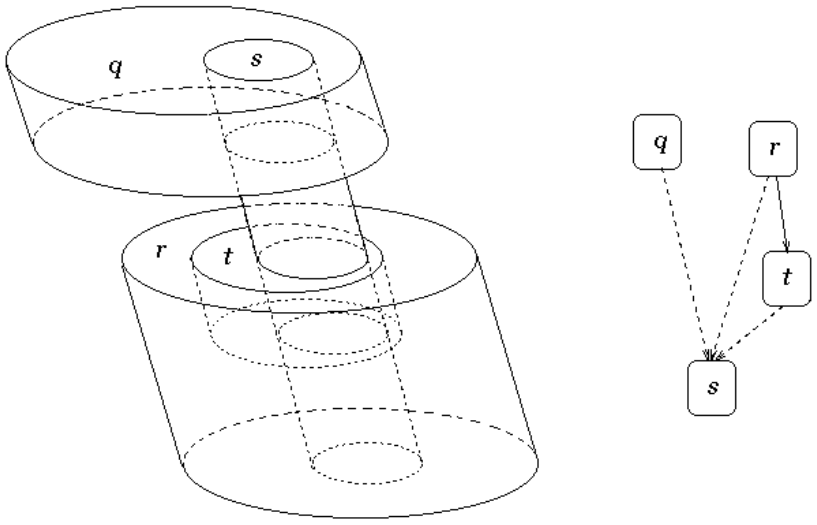


Figure 6.7: The geometry of shared containment

if and only if object ob_1 directly contains but may share object ob_2. Let DC (direct containment) denote the relation

$$DC : \mathbb{O} \leftrightarrow \mathbb{O}$$

defined to be the union of the two relations $dcon$ (as defined in Section 7.1) and $sdcon$, i.e.

$$DC = dcon \cup sdcon.$$

The three conditions above respectively require that

$$\nexists ob : \mathbb{O} \bullet ob \ \underline{DC^+} \ ob,$$
$$dcon^\sim \in \mathbb{O} \nrightarrow \mathbb{O},$$
$$dcon \cap sdcon = \varnothing.$$

The notion of shared containment can be incorporated into Object-Z in much the same way that unique containment was modelled in Section 7.2.2. Briefly, let every class have two implicit attributes

$$sdcontain, scontain : \mathbb{P}\,\mathbb{O}$$

where *sdcontain* denotes the set of directly contained but sharable objects, while *scontain* denotes the directly and indirectly contained but sharable objects. Each class has an implicit invariant

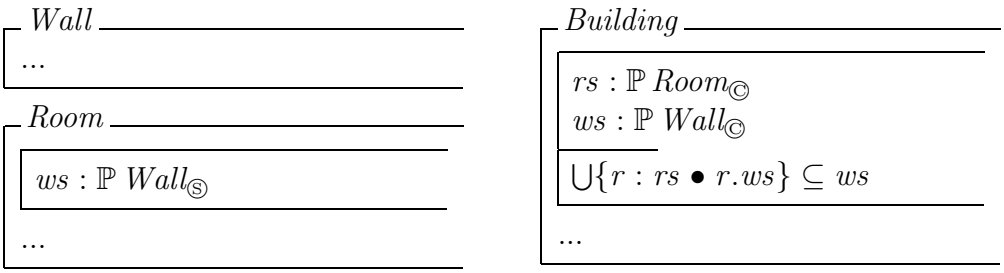
$$
\begin{aligned}
scontain = \{ & ob : \mathbb{O} \mid \\
& ob \notin contain \\
& \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet \\
& \qquad s(1) \in (dcontain \cup sdcontain) \\
& \qquad s(\#s) = ob \\
& \qquad \forall\, i : 1 \,..\, \#s - 1 \bullet s(i+1) \in (s(i).dcontain \cup s(i).sdcontain)\}
\end{aligned}
$$

In terms of the attributes *dcontain*, *contain*, *sdcontain* and *scontain*, the above conditions on the relations *dcon* and *sdcon* imply the following predicates are implicit invariants of any system:

$$\nexists ob : \mathbb{O} \bullet ob \in (ob.contain \cup ob.scontain),$$
$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1 \neq o_2 \Rightarrow o_1.dcontain \cap o_2.dcontain = \varnothing,$$
$$\forall\, ob : \mathbb{O} \bullet ob.dcontain \cap ob.sdcontain = \varnothing.$$

Example: Walls, Rooms and Buildings

Consider once again a campus consisting of buildings and rooms where each room in a building has a set of walls each of which may be shared with some other room. Furthermore, suppose that in the campus the buildings are physically separated so that no wall is shared between different buildings. An Object-Z specification of the campus would include the classes



where the notation '℗', like the notation '©' introduced in Section 7.2.2, is a syntactic simplification identifying the directly contained but sharable objects. Without this simplification the state schema of the *Room* class would have been

$$ws : \mathbb{P}\ Wall$$
$$sdcontain = ws$$

This specification captures explicitly the geometric view that a building uniquely contains both its rooms and its walls, even although these walls may be shared between the rooms. Notice that this specification does not demand that each wall be shared between rooms; rather, it simply indicates that each wall is possibly shared (i.e. is sharable).

Example: Buildings, Streets and Suburbs

It is possible for an object within a system to both uniquely contain some objects and sharably contain others. For instance, consider a town consisting of suburbs, streets and buildings. An Object-Z specification of the town would include the classes

Building
...

Street
...

Suburb
$bs : \mathbb{P}\ Building_{©}$
$ss : \mathbb{P}\ Street_{Ⓢ}$
...

This specification captures explicitly the geometric view that buildings are uniquely contained within suburbs, whereas streets may be contained but shared between suburbs.

Other Containment Geometries

Although the geometric notion of unique containment, and to a lesser extent that of shared containment, captures, in our experience, the geometry most commonly occurring in real systems, other geometries of object containment are possible. For example, consider the situation illustrated in Figure 6.8 where object m partially contains object n while at the same time n partially contains m.
Although it would be possible to introduce specific notation to formally capture such geometries in Object-Z, as such structures only occur quite rarely in practice it is adequate to capture the properties implied by such geometries explicitly as class invariants (as in Section 6.3.1) when the need arises. An example is given in Section 6.3.6, when modelling the abstract structure of a circular doubly-linked list.

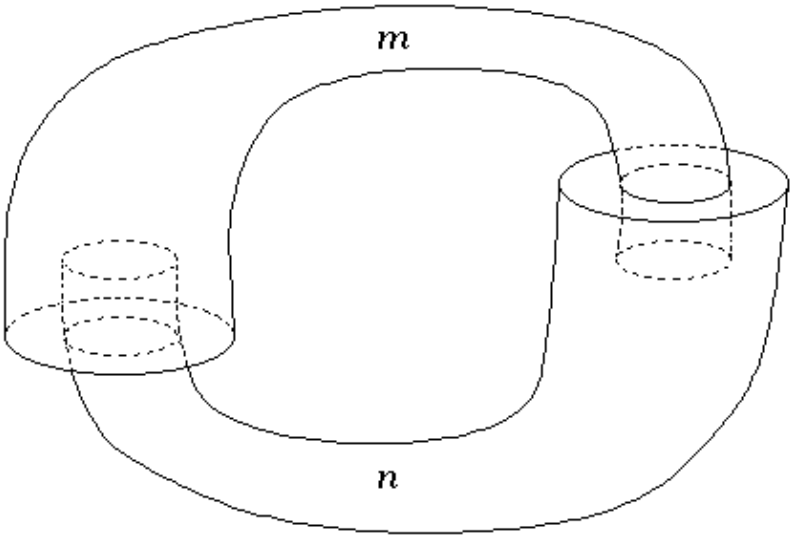


Figure 6.8: The geometry of circular partial containment

6.3.6 Containment in Abstract Structures

The object references that exist in object-oriented models of abstract recursive structures such as trees or directed acyclic graphs (DAGs) often satisfy the combinatorial properties of object containment, and hence the geometry of object containment can be applied directly when constructing object-oriented models for such structures. As an illustration, consider the following (partial) Object-Z specifications of a tree node and a DAG node where T is a generic type.

TreeNode[T]

$$value : T$$
$$subnodes : \mathbb{P}\, TreeNode[T]_{\copyright}$$

DAGnode[T]

$$value : T$$
$$subnodes : \mathbb{P}\, DAGnode[T]_{\textcircled{S}}$$

Each node in the tree (or the DAG) has associated with it a value of type T and a (possibly empty) set of subnodes.

The use of object containment in the above examples guarantees a tree (or a DAG) structure without the need to state explicitly as invariants the properties of such a structure.

The notions of unique and shared containment are particularly suitable for modelling acyclic abstract structures, but inadequate for cyclic structures. For instance, an

Object-Z specification of a node in a circular doubly-linked list would include

$$
\begin{array}{|l}
\hline
CDLLnode[T] \\
\hline
value : T \\
next, prev : CDLLnode[T] \\
\hline
next.prev = self = prev.next \\
\hline
\end{array}
$$

where the doubly-linked property is captured by an explicit invariant. As an example, consider two instances of the $CDLLnode[T]$ class linked together, i.e. where each one is the *next* (and *prev*) of the other. The (circular) object references between the two instances (nodes) can be viewed as an abstract realisation of the geometry illustrated in Figure 6.8.

6.3.7 Conclusion

In this section, the notion of object containment was captured within a formal framework, first by predicates incorporating the properties of containment within class invariants, and then by extending the Object-Z notation to capture the geometry of object containment directly. The advantage of this extension is that the properties of containment follow implicitly and do not need to be stated explicitly by invariants. Within this formal framework

- reference semantics for objects is sufficient: the introduction of value semantics (e.g. the *expanded* class type of Eiffel) is not needed to capture containment;
- there is a clear distinction between object containment and object access: it is possible to model systems where messages are sent to a contained object by objects other than the containing object; and
- relocation of contained objects is possible without compromising the underlying geometry: the structure of containment is maintained implicitly.

The notion of object containment is particularly useful not only explicitly to capture geometric ideas of object location, but also to specify abstract acyclic structures.

6.4 Synthesis from Object-Z with Temporal Logic Invariants

Part two *Charts, Automata and Event-based Formalisms* includes 4 chapters.

Chapter 7

Message and Live Sequence Chart

An introduction on the notion of sequence diagrams. Sequence diagrams are a natural means of capturing system requirements from a global perspective. They can be used in the early stage of system development manifesting use cases, stating visually what are the possible interactions between system components and thus identifying the input-output relations. They may also be used in later stage for testing purpose.
There have been a variety of system engineering methods that based on the notion of sequence diagrams. The classic notion of sequence diagram is called Message Sequence Chart (MSC), which is used to identify example system interactions. The recent development is the notion of Live Sequence Chart, which specifies not only possible system interactions, but also mandatory ones.

7.1 Message Sequence Charts

The notion of sequence diagrams (also called *trace diagrams* in some works) has long been used in the system development cycle. The language MSC, however, is standardized by the International Telecommunication Union (ITU) in 1996. MSC provides a means for visualization of the interaction of system components. The core of MSC is called the Basic Message Sequence Chart (BMSC), which concerns communications and actions only.
Figure 7.1 is an example of a BMSC. Each vertical line represents an active component (Z.120 terminology, an instance) in the system. The frame (Z.120 terminology, parallel frame) represents the environment. Instances can interact with other instances by sending messages, e.g., *message*1, *message*2, *message*3. A message originated from the frame represents an input from the unspecified environment, e.g., *input*. Similarly, a message targeting the frame is an output to the environment, e.g., *output*. The square labeled with *action* is a local action performed by instance *inst*2.
The timing information is captured by the following two rules and their transitive

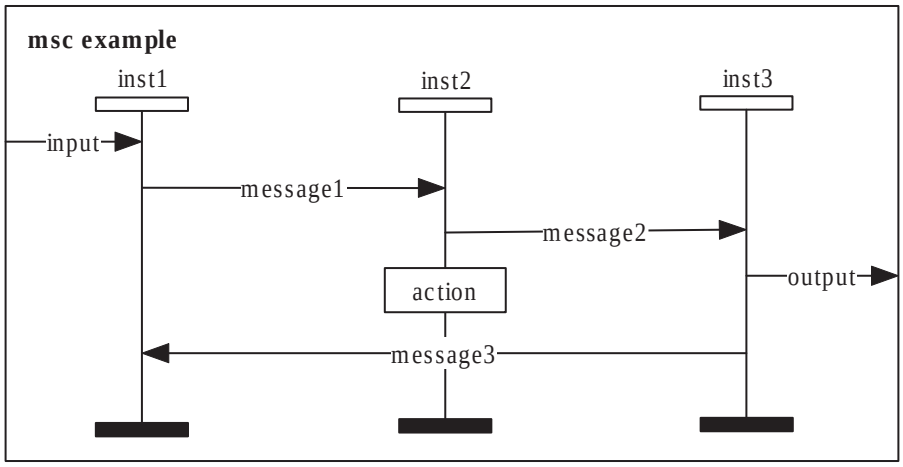


Figure 7.1: Basic Message Sequence Chart

closure: for each message passing, the message output event precedes the corresponding message input event and for each vertical line representing an instance, the time progresses from top to bottom. The two rules and the transitive closure define a partial event ordering relation, which captures the semantics of BMSC [85].
Then, additional basic concepts like process creation, termination, time handling, incomplete message events and conditions are added.
Figure 7.2 shows an MSC with an inline reference expressions. The rounded rectangle represents an inline reference expression, where *MSC-A* and *MSC-B* are MSCs and the key word **alt** denotes a (delayed) choice. Along the vertical line for instance *inst3*, there is a timeout event, which is drawn as a sandglass.
Later, more complicated constructs are introduced. They are inline expressions, MSC reference expressions and High-level Message Sequence Chart (HMSC), which enrich MSC with intricate possibilities of describing complex systems. High-level MSC can be constructed incrementally by referencing an MSC using its name (or equivalently using inline expressions). MSC can be combined vertically, horizontally or alternatively.
The chart in Figure 7.3 is a simple example of an HMSC. The triangle at the top represents the starting point. The one at the bottom represents the ending point. Each rounded rectangle abstracts an MSC. The semantics of the HMSC is captured by the process expression $(A \circ C)^{\circledast} \circ (A \circ B)$, where $\circ$ denotes sequential composition and $\circledast$ denotes infinite iteration.
Various constructors for composing MSCs are: *alt* for choices, *seq* for sequential composition, *par* for parallel composition, *opt* for optional, *exc* for exception and

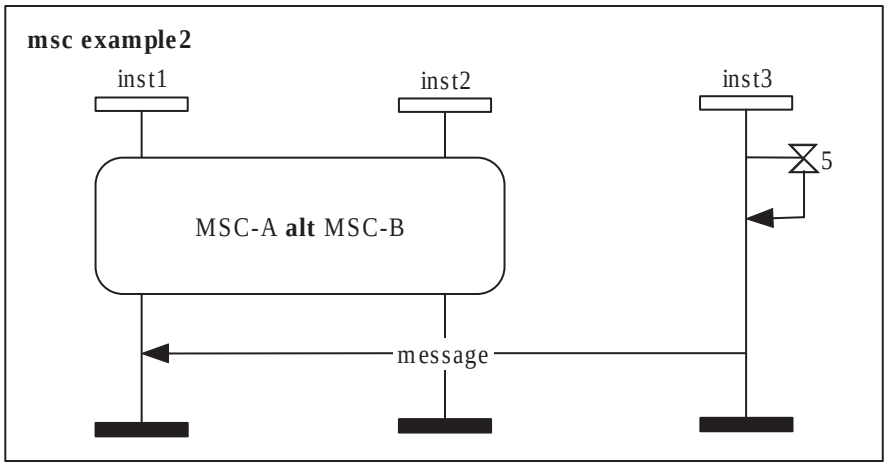


Figure 7.2: MSC with inline referencing

loop for iteration. Precise semantics are developed for these key words. An MSC reference expression is defined as the following:

$$
\begin{array}{lll}
MRE ::= & ref\langle\langle NAME\rangle\rangle \mid \epsilon \mid \delta & \text{– primitives} \\
\mid & (\_ \mp \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– delayed choice} \\
\mid & (\_ \parallel \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– delayed parallel} \\
\mid & (\_ \circ \_)\langle\langle MRE \times MRE\rangle\rangle & \text{– sequential composition} \\
\mid & (\_)^{\circledast}\langle\langle MRE\rangle\rangle \mid (\_)^{\infty}\langle\langle MRE\rangle\rangle & \text{– iteration}
\end{array}
$$

An HMSC can reference other HMSCs or BMSCs by their names. The two most primitive constructs are δ and ϵ. The former does nothing at all and the latter terminates immediately. The structural operator *delayed choice* is written as $\mp$. Graphically, it is a sub-chart marked with the keyword *alt*. The *delayed parallel* is written as $\parallel$. The notion of sequential composition in MSC is referred as *weak sequential composition*, denoted as $\circ$. Given sequential composition of two MSCs (say m_1 and m_2), interactions over shared instances in m_2 is delayed until interactions in m_1 completes. However, the execution of actions over instances not in m_1 from m_2 is allowed before m_1 has the option to terminate. The iteration operator $\circledast$ and ∞ is defined as an infinite series of sequential compositions.

A number of semantic models have been developed for MSC. Examples are the operational semantics based on process algebra [7, 64], Petri nets [56], as well as the informal semantics in [64]. In [64], semantics of various constructs of MSC are defined by sets of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set

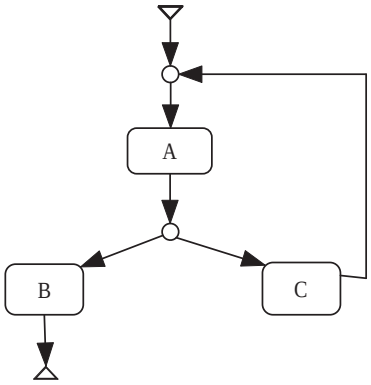


Figure 7.3: High-level Message Sequence Chart

of premises and C is the conclusion. Each individual premise and conclusion are of the form $s \xrightarrow{a} s'$ or $s \downarrow$ for arbitrary $s, s' \in MRE$ and $a \in A$, where A denotes all events represented by atomic actions in MSC, i.e., message input, message output, local action and timer events. For instance, no deduction rule is associated with δ because it does nothing. The only rule associated with ε is $\varepsilon \downarrow$, meaning termination. The semantics of $\mp$ is captured by the following rules:

$$\frac{x \downarrow}{x \mp y \downarrow}\,[\,DC1\,] \qquad \frac{y \downarrow}{x \mp y \downarrow}\,[\,DC2\,] \qquad \frac{x \xrightarrow{a} x', y \overset{a}{\not\rightarrow}}{x \mp y \xrightarrow{a} x'}\,[\,DC3\,]$$

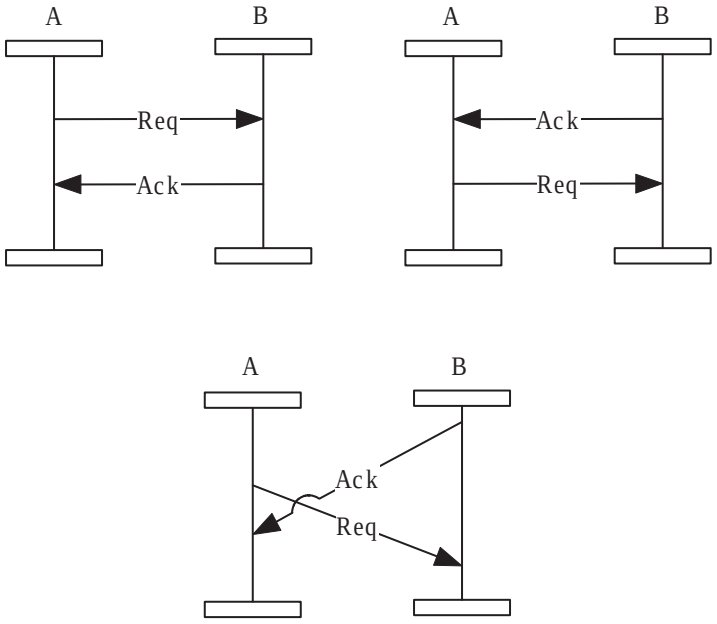
$$\frac{x \overset{a}{\not\rightarrow}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}\,[\,DC4\,] \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}\,[\,DC5\,]$$

The rules $DC1$ and $DC2$ express that the delayed choice of the two processes has the option to terminate if and only if at least one of the alternatives has this option. $DC3$ and $DC4$ express that the delayed choice will behave as one of the options given that some initial event of this option takes place. $DC5$ captures the idea that in case both of the alternatives are enabled, the choice is delayed. The rest of the constructs are similarly defined [64].

7.1.1 Implied Scenarios

The notion of implied scenarios is first discussed by R. Alur *el at* in [3]. It reviews the gap between inter-object specifications and intra-object models. Intuitively, implied

scenarios are additional behaviors that may be present in every distributed object system which is consistent with the specified scenarios, i.e., the set of MSCs.
The following charts show a typical example of an implied scenario (inspired by the example in [3]):



The first two MSCs are the specification, where the two events *Req*, *Ack* may occur in either order. The third is an implied scenario, where the reception of both events is delayed. In the third scenario, as far as the object A or B can tell, both of them are executing a specified scenario, i.e., A executes accordingly as the first and B executes accordingly as the second.
The existence of implied scenarios can be explained using the CSP notions. Let $\{M_j\}$ where $1 \leq j \leq n$ be the set of MSCs. Let M_j^i where $1 \leq i \leq m$ be the process capturing the behavior of instance i in the chart M_j. An implementation of the specification shall therefore exhibit exactly the following behaviors:

$$(M_1^1 \parallel M_1^2 \parallel \cdots \parallel M_1^m) \,\Box\, (M_2^1 \parallel M_2^2 \parallel \cdots \parallel M_2^m) \,\Box\, \cdots$$
$$\Box\, (M_n^1 \parallel M_n^2 \parallel \cdots \parallel M_n^m)$$

The distributed object system inferred from a set of MSCs should be composed of finite state processes modeling each of the objects appeared in the scenarios. Each object should exhibit as sequences of events at least all scenarios projected to the time line of that component. Formally, the behavior of an object shall at least exhibit the behaviors captured by the following expression: $M_1^i \,\Box\, M_2^i \,\Box\, \cdots \,\Box\, M_n^i$. Because an MSC model describes only examples of system behaviors, it is natural that the

composition of the components exhibits more behaviors than those explicitly captured in the scenarios. The existence of implied scenarios may be explained using CSP algebraic laws as the following:

$$\begin{aligned} &traces((M_1^1 \parallel M_1^2 \parallel \cdots \parallel M_1^m) \\ &\qquad \Box\ (M_2^1 \parallel M_2^2 \parallel \cdots \parallel M_2^m) \\ &\qquad \Box \cdots \Box\ (M_n^1 \parallel M_n^2 \parallel \cdots \parallel M_n^m)) \subseteq \\ &\quad traces((M_1^1 \Box M_2^1 \Box \cdots \Box M_n^1) \\ &\qquad\qquad \parallel (M_1^2 \Box M_2^2 \Box \cdots \Box M_n^2) \\ &\qquad\qquad \parallel \cdots \parallel (M_1^m \Box M_2^m \Box \cdots \Box M_n^m)) \end{aligned}$$

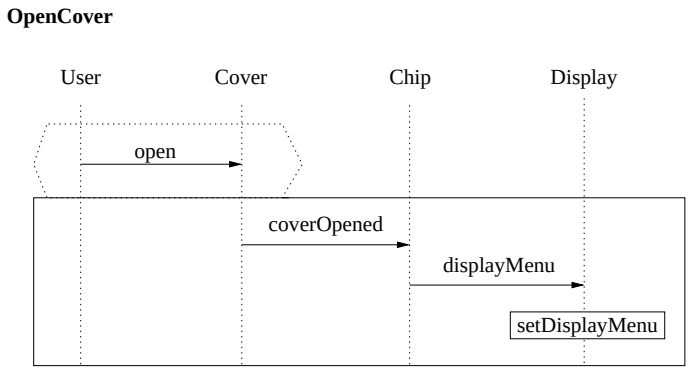
The occurrence of additional scenarios is because the scenario-based model describes allowed system behaviors from a global, system-wide perspective, whereas in the distributed object processes each agent acts locally based on local information.

7.2 Live Sequence Charts

MSC [64] is widely used to describe scenarios that capture communication between processes or objects. It is used in the early stages of system development. It has found its way into many methodologies [64, 118]. However, MSC (both BMSC and HMSC) suffers from the rather weak partial-order semantics that makes it incapable of capturing many kinds of behavioral requirements. Moreover, MSC only captures example runs of the system and thus it is not suitable to specify complete system behaviors. The notion of Live Sequence Chars (LSC) [19] is introduced by Damm and Harel to overcome the shortcomings of MSC by adding liveness. LSC extends MSC with constructs to distinguish scenarios that must happen from scenarios that may happen, conditions that must be fulfilled from conditions that may be fulfilled, etc. Together with the notion of symbolic objects and various high-level operators like bounded loop, if-then-else, LSC may well be used to specify complicated inter-object system requirements. A software package named *Play-Engine* has been developed by Damm and Harel to interactively "play-in" and "play-out" scenarios [51].

There are two kinds of charts in LSC. Existential charts are mainly used to describe possible scenarios of a system in the early stage of system development, i.e., the same role played by MSC except that existential charts are scoped. In later stages, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. A universal chart is typically preceded with a pre-chart, which serves as the activation condition of the main chart. Whenever a communication sequence matches the pre-chart, the system must proceed as specified by the main chart. A system run may activate a universal chart more

than once and some of the activations might overlap [84].
The following is a universal chart as part of the mobile phone specification:



This scenario *OpenCover* illustrates the interaction between the objects when the *user* opens the *cover*. Once the cover is opened by the user, the main chart is activated. The *chip* is notified that the *cover* is opened. It then requests the *display* to display the menu. Lastly, the *display* carries out a local action *setDisplayMenu* to initialize the menu screen.
Contrasted with MSC, which captures only examples of system behaviors, an LSC universal chart specifies mandatory behaviors. In other words, a universal chart constrains all behaviors of the system. Therefore, implied scenarios do not apply in the setting of LSCs.
Let $\mathcal{C}$ be the set of all possible charts. Let $\mathcal{E}$ be the set of existential charts. Let $\mathcal{U}$ be the set of universal charts. In this work, we assume that an LSC specification, denoted as $\mathcal{S}$, consists of a set of universal charts and existential charts. Throughout the section, c, e, u are used to denote a chart, an existential chart, a universal chart respectively. Let $\mathcal{B} \subset \mathcal{C}$ be the set of basic charts, i.e., Basic-MSC [64]. Let Σ be the set of all possible events. Σ is partitioned into two groups, communication messages M, e.g., *coverOpened* in Example **??** and local actions A, e.g., *setDisplayMenu*. A communication event $m : M$ is followed by '?' if it is an input event or '!' if it is an output event. A local action $a : A$ may be an assignment or a (local or external) function call. Each chart c is associated with a set of visible events, $\Sigma_c \subset \Sigma$. Only events visible to a chart are constrained by the chart. A chart typically consists of multiple instances (for instance, *User*, *Cover*, *Chip* and *Display*), which are represented as vertical lines graphically. Let $instances : \mathcal{C} \rightarrow \mathbb{P}\, Instances$ be the function returning the set of instances appearing in the chart. Along with each line, there are a finite number of locations. A location carries the temperature annotation for progress within an instance. Intuitively, locations can be thought as the joint points of instance lines and message lines. In the following, we use i to denote an instance, l_i to denote a location on instance i, l_i^0 to denote the first location on instance i and l_i^{max} to denote the very last location on instance i. We write the next location of l_i^k

along instance i in the same chart as l_i^{k+1}.

A location may be labeled as either cold or hot. A hot location means that a system run reaching this location has to move beyond. A system run may stay put at a cold location forever. Similarly, messages and conditions are also labeled. A hot message must be received, whereas a cold one may get lost. A hot condition must be met, whereas violation of a cold condition terminates the chart. A location is labeled with a finite number of events (more than one if it is a co-region) and at most one condition. Let *Location*, *Condition* be the set of all possible locations and conditions respectively. Function $label : Location \to \mathbb{P}\,\Sigma$ labels a location with a finite number of messages and local actions. Function $cond : Location \to Condition$ labels a location with a condition. If there is no condition associated with the location, it returns *true*. Function $eval : Condition \to \mathbb{B}$ evaluates a condition against the current valuation of the variables. Function $temp : Condition \to Cold \mid Hot$ tells the temperature of a condition. Function $temp : Location \to Cold \mid Hot$ tells the temperature of a location.

The universal charts in Figure 7.4 and the one in Example **??** constitute a self-containing set of scenarios, which specify a mobile phone specification. This example is partially inspired by the phone system specification presented in [52]. The system consists of six participating objects, a *user*, the *cover*, the *display*, the *speaker*, the *chip* and the environment where the incoming calls are from. Figure 7.4 illustrates scenarios of the system besides *OpenCover*, i.e., the user closes the cover, an incoming call arrives and the user picks up the phone and talks. All vertical lines in the charts are dotted, which means that all locations along the lines are cold and, therefore, the system may pause at any point of execution forever. This is possible because unexpected events like the battery runs out or the system breaks down may occur at any time. The set of visible events for each chart are exactly those appearing in the diagram except the scenario *Talk*. The message *close* from the user to the cover is forbidden in the scenario *Talk*, i.e., in order to carry out the scenario successfully, the user should not close the cover before the scenario completes.

LSC also supports advanced MSC features like co-region, hierarchy, etc. Moreover, symbolic instances and messages are used to group scenarios effectively. For a detailed introduction on a complete list of features of LSC, refer to [51]. LSC is far more expressive than MSC, which makes it capable of expressing complicated scenario-based requirements. However, we remark that the ability to specify hot and cold messages, i.e., whether a message is required to be received or may get lost, is redundant because of the facility for describing hot and cold locations. Essentially, the temperature of the locations takes precedence over the temperature of messages, so whether or not the message is received is determined entirely by the temperature of the message input. This questionable feature of LSC is recognized by Harel and Marelly who list the possible cases and conclude that the temperature of messages has no seman-

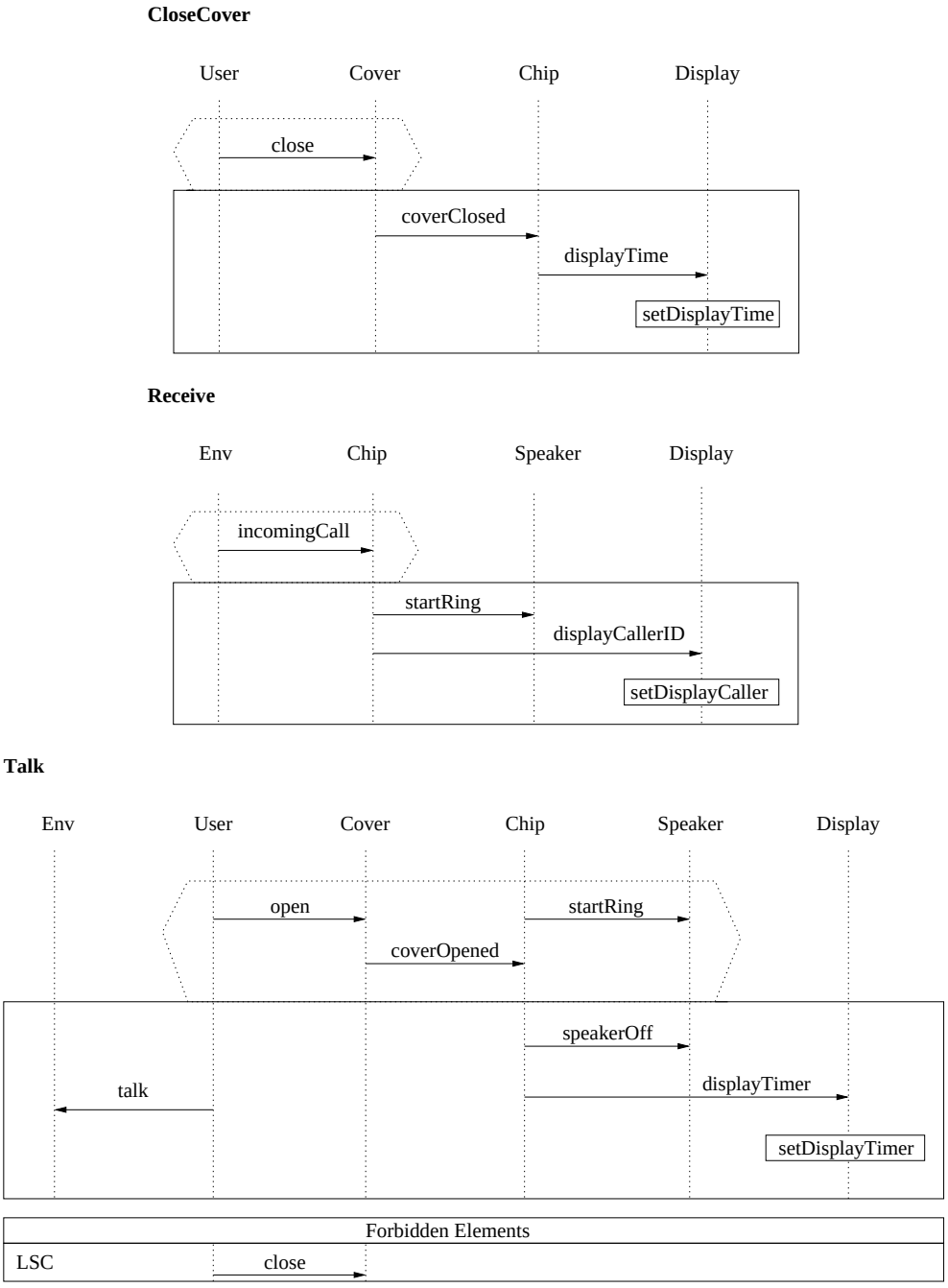


Figure 7.4: Mobile phone system scenarios

tic meaning [51]. Thus, in the following discussion, the temperature of messages is discarded.

7.2.1 Semantics of Live Sequence Charts

The semantics of LSC is briefly discussed in [19] using skeleton automata and program-like pseudo-codes. Only basic charts and pre-charts are covered. This section is devoted to a trace-based denotational semantics of LSC, which conforms to the original semantics in [19]. Our semantics completes theirs by defining precisely the traces of a set of universal charts and existential charts.

For simplicity, in this section we assume that no co-region is allowed and all messages are synchronized. There is nothing interesting about co-region except that it complicates the discussion. Asynchronous message passing is supported by explicitly modeling the behavior of the buffers, e.g., First In First Out (FIFO). A consequence of this assumption is that a message loss is captured as an infinitely long delay of the forwarding by the buffer instead of a *lost message* symbol. A hidden assumption is that the size of the communication buffers is finite.

The semantics of a basic chart b is defined to consist of all runs compatible with the partial order induced by b and its annotations. We define an automaton interpretation of b completing the skeleton automata in [19] and then define the languages of b based on the automaton. A chart induces a partial order over the events.

Definition 7.2.1 *The partial order is the smallest binary relation* $\ll$: *Location* $\leftrightarrow$ *Location satisfying the following axioms and closed under transitivity and reflexivity.*

$$\begin{array}{ll}
\forall\, l_i^k : \mathit{Location} \bullet l_i^k \ll l_i^{k+1} & \text{– vertical} \\
\forall\, l_1, l_2 : \mathit{Location} \mid l_1 \neq l_2 \bullet & \\
\quad \exists\, m : M \bullet m! \in \mathit{label}(l_1) \land m? \in \mathit{label}(l_2) \Rightarrow l_1 \ll l_2 & \text{– horizontal} \\
\quad \exists\, m : M \bullet m! \in \mathit{label}(l_1) \land m? \in \mathit{label}(l_2) \Rightarrow l_2 \ll l_1 & \text{– synchronous}
\end{array}$$

The first axiom states that along each vertical line time progresses from top to bottom. The second axiom states that message output event must precede the corresponding message input event. The third handles synchronous message passing. An LSC chart is well-formed if the relation $\ll$ is acyclic (except trivial cyclic relation between locations connected by synchronous message passing). In the rest of the thesis, we assume that all charts are well-formed. We define function *preset* to return the set of locations that precede a given location in the relation $\ll$.

$$\begin{array}{|l}
\mathit{preset} : \mathit{Location} \to \mathbb{P}\, \mathit{Location} \\
\hline
\forall\, l : \mathit{Location} \bullet \mathit{preset}(l) = \{x : \mathit{Location} \mid x \ll l \land \neg(l \ll x)\}
\end{array}$$

One of the basic concepts used for defining the semantics of LSC is the notion of a *cut*. A *cut* through the chart represents the progress each instance has made in the scenario. Let *cut* be the function which returns the set of all possible *cut*s of a chart. A *cut* is a set of locations, one for each instance, satisfying the following condition:

$$
\begin{array}{|l}
cut : \mathcal{C} \rightarrow \mathbb{P}\, Location \\
\hline
\forall\, c : \mathcal{C} \bullet \forall\, x : cut(c) \bullet \#(x) = \#instances(c) \\
\qquad \wedge\, \forall\, l : x \bullet \nexists\, l' : x \bullet l' \in preset(l)
\end{array}
$$

Intuitively, it means no location in a *cut* is preceded with another. We are now ready to define the automaton which accepts exactly the language of a basic chart.

Definition 7.2.2 *The automaton associated with basic chart b is defined as $A_b \mathrel{\widehat{=}} (S_b, S_b^0, F_b, \Sigma_b \cup \{\tau\}, T_b)$. S_b is the state space. $S_b \mathrel{\widehat{=}} \{Aborted, Terminated, Completed\} \cup Active$ where $Active \mathrel{\widehat{=}} cut(b)$. $S_b^0 \mathrel{\widehat{=}} \bigcup_i \{l_i^0\}$ is the initial state. F_b is the set of final (accepting) states.*

$$
\begin{aligned}
F_b \mathrel{\widehat{=}} & \{Aborted, Terminated, Completed\} \\
& \cup \{s : cut(b) \mid \forall\, l : s \bullet temp(l) = Cold\}
\end{aligned}
$$

Σ_b is the set of events appearing in the chart. The special event τ denotes temporal progress along a vertical line. $T_b : S_b \times \Sigma_b \cup \{\tau\} \rightarrow S_b$ is the least transition relation satisfying the following:

$$
\begin{array}{ll}
D1 & \forall\, x : \Sigma_b \bullet (Terminated, x, Terminated) \in T_b \\
D2 & \forall\, x : \Sigma_b \bullet (Completed, x, Completed) \in T_b \\
D3 & (\bigcup_i \{l_i^{max}\}, \tau, Completed) \in T_b \\
D4 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \exists\, a : A \bullet \\
 & \qquad (a \in label(l_i^k) \wedge eval(cond(l_i^k)) = true \wedge l_i^k \neq l_i^{max}) \\
 & \qquad\quad \Rightarrow (s, a, (s \setminus \{l_i^k\}) \cup \{l_i^{k+1}\}) \in T_b \\
D5 & s \in cut(b) \wedge \exists\, l_i^k, l_j^p : s \bullet \exists\, m : M \bullet \\
 & \qquad (m! \in lable(l_i^k) \wedge m? \in label(l_j^p) \wedge l_i^k \neq l_i^{max} \wedge l_j^p \neq l_j^{max} \\
 & \qquad \wedge\, eval(cond(l_i^k)) = true \wedge eval(cond(l_j^p)) = true) \\
 & \qquad\quad \Rightarrow (s, m, (s \setminus \{l_i^k, l_j^p\}) \cup \{l_i^{k+1}, l_j^{p+1}\}) \in T_b \\
D6 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \\
 & \qquad eval(cond(l_i^k)) = false \wedge temp(cond(l_i^k)) = Cold \\
 & \qquad\quad \Rightarrow (s, \tau, Terminated) \in T_b \\
D7 & s \in cut(b) \wedge \exists\, l_i^k : s \bullet \\
 & \qquad eval(cond(l_i^k)) = false \wedge temp(cond(l_i^k)) = Hot \\
 & \qquad\quad \Rightarrow (s, \tau, Aborted) \in T_b
\end{array}
$$

The chart is *completed* if all instances have reached the very last location. It is

terminated if a cold condition is violated, and *aborted* if a hot condition is violated. Otherwise, we say that the chart is *active*, i.e., there exists a *cut* through every instance in the chart. Initially, the chart is active and all instances are at their first location. A state is accepting if and only if either it is completed or terminated or aborted, or it is an active state where all instances are at a cold location. *D1* and *D2* state that all behaviors are allowed after a chart is terminated or completed. *D3* states that a chart is terminated only after all instances have reached their last locations. *D4* and *D5* state that a local action or a message passing may occur only if the system can reach a new *cut* after engaging in the communication event or local action. Whenever a cold condition is evaluated to false, the chart terminates (*D6*). If the condition is labeled *hot*, the chart aborts so that no further behavior is allowed (*D7*). No compositional operator offered by LSC is discussed in this definition. For a chart with hierarchy, we can flatten the sub-charts by adding transitions connecting the initial and *Terminated* state of the sub-chart to states in the automaton of the upper-level chart. For instance, a conditional branch can be flattened by connecting the last state of the upper-level chart to the initial states of both branches. As the flattening is a standard process, we omit the detail in this definition. Moreover, we adopt an interleaving semantics, e.g., no priority is associated with conditions, etc.
A trace of the automaton A_b is a sequence of events $\langle e_1, \cdots, e_k, \cdots e_n \rangle$, where there exists a run s_1, e_1, s_2, $\cdots$, s_k, e_k, s_{k+1}, $\cdots$, s_n, e_n, s_{n+1} such that $s_1 = S_b^0$ and $s_{n+1} \in F_b$ and for all $1 \leq k \leq n$ such that $(s_k, e_k, s_{k+1}) \in T_b$. The language of the automaton A_b, denoted as $\mathcal{L}(A_b)$, contains all traces of the automaton A_b.

Definition 7.2.3 *The language of a basic chart b, denoted as $\mathcal{L}_\beta(b)$, is the language of the automaton $\mathcal{L}(A_b)$. The executions of b which complete the whole chart, denoted as $\mathcal{F}_\beta(b)$, contain the traces of A_b which reach the state Completed once and once only.*

The semantics of existential charts is different from that of basic charts because existential charts, as universal charts, are scoped. Events invisible to the chart may occur freely between any two successive events in an execution of the chart. Given a set of events $\Sigma_i \subseteq \Sigma$, a trace filter, denoted as $tr \restriction \Sigma_i$, satisfies the following conditions:

$$\langle\rangle \restriction \Sigma_i \mathrel{\widehat{=}} \langle\rangle$$
$$(i \frown tr') \restriction \Sigma_i \mathrel{\widehat{=}} i \frown (tr' \restriction \Sigma_i), \text{ where } i \in \Sigma_i$$
$$(j \frown tr') \restriction \Sigma_i \mathrel{\widehat{=}} tr' \restriction \Sigma_i, \text{ where } j \notin \Sigma_i$$

In the following definition, forbidden events are properly handled, i.e., they are prevented from occurring until the chart completes.

Definition 7.2.4 *Let Σ_e be the set of events visible to an existential chart e. The language of e, denoted as $\mathcal{L}_\epsilon(e)$, is defined as: $\mathcal{L}_\epsilon(e) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid tr \restriction \Sigma_e \in \mathcal{L}_\beta(e)\}$.*

The executions of e which travel through the whole chart, denoted as $\mathcal{F}_\epsilon(e)$, is defined as:

$$\mathcal{F}_\epsilon(e) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid tr \in \mathcal{L}_\epsilon(e) \wedge tr \upharpoonright \Sigma_e \in \mathcal{F}_\beta(e)\}$$

A trace tr is a fragment of trace tr', denoted as $tr \ in \ tr'$, if and only if tr is a sub-sequence of tr'.

$$\begin{array}{|l} \_ \ in \ \_ : \Sigma^* \leftrightarrow \Sigma^* \\ \hline \forall\, tr, tr' : \Sigma^* \bullet tr \ \underline{in} \ tr' \Leftrightarrow \exists\, tr_1, tr_2 : \Sigma^* \bullet tr_1 \frown tr \frown tr_2 = tr' \end{array}$$

A universal chart is typically preceded with a pre-chart. Whenever an execution completes the pre-chart, the execution must proceed as specified by the main chart.

Definition 7.2.5 *Let $p, m : \mathcal{B}$ be the pre-chart and main-chart of a universal chart u. The language of u is $\mathcal{L}_\mu(u)$ satisfying the following:*

$$\mathcal{L}_\mu(u) \mathrel{\widehat{=}} \{tr : \Sigma^* \mid \nexists\, tr_1, tr_2 : \Sigma^* \bullet tr_1 \frown tr_2 \ in \ tr \wedge tr_1 \in \mathcal{F}_\epsilon(p) \wedge tr_2 \notin \mathcal{L}_\epsilon(m)\}$$

Intuitively, a trace violates a universal chart if and only if it completes the pre-chart but fails to conform to the main chart. By associating different sets of visible events to the pre-chart and main chart, various kinds of forbidden events [51] can be handled properly.
An LSC specification consists of a set of universal charts and existential charts, i.e., $\mathcal{S} \subset \{x : \mathcal{C} \mid x \in \mathcal{U} \vee x \in \mathcal{E}\}$. An implementation satisfies an LSC specification if and only if it always exhibits behaviors allowed by the universal charts and it is capable of exhibiting the behaviors captured by the existential charts.

Definition 7.2.6 *An implementation $\mathcal{I}$, whose executions are denoted as traces($\mathcal{I}$), satisfies an LSC specification $\mathcal{S}$, denoted as $\mathcal{I} \vDash \mathcal{S}$, if and only if:*

$$(traces(\mathcal{I}) \subseteq \bigcap_{u \in \mathcal{S}} \mathcal{L}_\mu(u)) \wedge (\forall\, e \in \mathcal{S} \bullet \mathcal{F}_\epsilon(e) \cap traces(\mathcal{I}) \neq \varnothing)$$

Chapter 8

CSP and Timed CSP

CSP/Timed CSP

- Hoare's CSP (Communicating Sequential Processes) an *event* based notation primarily aimed at de scribing the sequencing of behaviour within a process and the synchronisation of behaviour (or *communication*) between processes.
- Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronisation.

- S. Schneider. Concurrent and Real-time Systems: The CSP Approach, Wiley, 1999.
- A.W. Roscoe. The Theory and Practice of Concurrency. Prentice-Hall, 1997.
- J. Davies, Specification and Proof in Real-Time CSP, Cambridge University Press, 1993.
- C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.

Specifying a Process

A process is determined (specified) by what it can do;
i.e. a process is defined by its behaviour.
The perceived behaviour of a process will depend upon the observer.
We shall be mainly concerned with specifying the interaction between a system and its environment
(i.e. external (visible) behaviour).

Events

A process engages in *events*; each event is an atomic action. e.g. the events for a vending machine are

coin—insert a coin

choc—extract a chocolate

The set of events that a process can possibly engage in is the *alphabet* of the process e.g. the alphabet of the vending machine is

$\{coin, choc\}$

Traces

A *trace* is a finite sequence of events.
A (deterministic) process is specified by the set of traces denoting its possible behaviour. e.g. the traces of the vending machine:

$\langle\ \rangle$
$\langle coin \rangle$
$\langle coin, choc \rangle$
$\langle coin, choc, coin \rangle$
...

Any execution of the process will be one of these sequences. If $s \frown t$ is a trace of a process,
then so also is s;
i.e. the set of traces is prefix closed.

Basic Process Notation

- lower case identifiers denote events;
  x, y, z are variables denoting events

- upper case identifiers denote processes;
  X, Y are variables denoting processes

- A, B, C denote sets of events

- if P is a process,
  αP denotes the alphabet of P

- if P is a process,
  $traces(P)$ denotes the set of traces of P

Trace Notation

- if A is a set of events,
  $\mathrm{seq}\, A$ denotes the set of all finite sequences
  of events from A.

- if $s, t : \mathrm{seq}\, A$,
  $s \frown t$ is the *concatenation* of s with t
  e.g.

$$\langle b, a, b\rangle \frown \langle b, c, a\rangle = \langle b, a, b, b, c, a\rangle$$

-

$$\leqslant : \mathrm{seq}\, A \leftrightarrow \mathrm{seq}\, A$$

$$s \leqslant t \;\Leftrightarrow\; \exists\, u : \mathrm{seq}\, A \bullet s \frown u = t$$

- $s^n = s \frown s \frown s \frown \cdots \frown s$
  i.e. s concatenated with itself n times

Examples

(1) $STOP_A$ is the process with alphabet A that can do nothing.

$$traces(STOP_A) = \{\langle\,\rangle\}$$

(2) $CLOCK$ is the process with $\alpha CLOCK = \{tick\}$ which can 'tick' at any time.

$$traces(CLOCK) = tick^*$$

(3) *VM* is the process with $\alpha\, VM = \{coin, choc\}$ which repeatedly supplies a chocolate after a coin is inserted.

$$traces(\, VM) = \{s : \text{seq}\{coin, choc\} \mid \exists\, n : \mathbb{N} \bullet s \leqslant \langle coin, choc\rangle^n\}$$

(4) *WALK* is a one-dimensional random walk process with $\alpha\, WALK = \{left, right\}$.

$$traces(\, WALK) = (left \cup right)^*$$

(5) *LIFE* is the process with $\alpha LIFE = \{beat\}$ which can stop (die) at any time.

$$traces(LIFE) = beat^*$$

Prefix

A process which may participate in event a then act according to process description P is written

$$a\,@t \rightarrow P(t).$$

The event a is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially. The event a is sometimes referred to as the *guard* of the process. The (optional) timing parameter t records the time, relative to the start of the process, at which the event a occurs and allows the subsequent behaviour P to depend on its value. For examples:

$$VMU = coin \rightarrow STOP$$

$$SHORTLIFE = (beat \rightarrow (beat \rightarrow STOP)) = beat \rightarrow beat \rightarrow STOP$$

$$VMS = coin \rightarrow choc \rightarrow STOP$$

Understanding Timed Prefix

Let P be a process which has two free time variables t_1 and t_2. A possible execution of the prefix:

$$a\,@t_1 \rightarrow b\,@t_2 \rightarrow P$$

$\downarrow$ 3 (time passed)

$a\,@t_1 \to b\,@t_2 \to P[(t_1+3)/t_1]$

$\downarrow a$ (event occur)

$b\,@t_2 \to P[3/t_1]$

$\downarrow 4$ (time passed)

$b\,@t_2 \to P[3/t_1][(t_2+4)/t_2]$

$\downarrow b$ (event occur)

$P[3/t_1][4/t_2]$

Other CSP/Timed-CSP primitives:

- $P;\ Q$ (sequential composition)
- $P\,|[\,X\,]|\,Q$ (synchronous), $P\ |||\ Q$ (asynchronous)
- $a \to P \,\Box\, b \to Q$ (external choice), $a \to P \sqcap b \to Q$ (intern al choice)
- $P_1 \,\triangledown\, e \to P_2$ (interrupt process)
- Wait $t;\ P$ (delay), $a \to P \,\triangleright\{t\}\, Q$ (time-out)

Sequential Composition

- The second form of sequencing is process sequencing. A distinguished event ✓ is used to represent and detect process termination.

- The sequential composition of P and Q, written $P;\ Q$, acts as P until P terminates by communicating ✓ and then proceeds to act as Q.

- The termination signal is hidden from the process environment and therefore occurs as soon as enabled by P. The process which may only terminate is written Skip.

Parallel composition

The parallel composition of processes P and Q, synchronised on event set X, is written

$$P\,|[\,X\,]|\,Q.$$

No event from X may occur in $P \,|[\, X \,]|\, Q$ unless enabled jointly by both P and Q. When events from X do occur, they occur in both P and Q simultaneously and are referred to as *synchronisations*. Events not from X may occur in either P or Q separately but not jointly. For example, in the process described by

$$(a \rightarrow P) \,|[\, a \,]|\, (c \rightarrow a \rightarrow Q)$$

all a events must be synchronisations between the two processes.
In an asynchronous parallel combination

$$P \;|||\; Q$$

both components P and Q execute concurrently without any synchronisations.

Choice

Diversity of behaviour is introduced through two choice operators.

The external choice operator allows a process a choice of behaviour according to what events are requested by its environment. The process

$$(a \rightarrow P) \,\Box\, (b \rightarrow Q)$$

begins with both a and b enabled. The environment chooses which event actually occurs by requested one or the other first. Subsequent behaviour is determined by the event which actually occurred, P after a and Q after b respectively.
Internal choice represents variation in behaviour determined by the internal state of the process. The process

$$a \rightarrow P \sqcap b \rightarrow Q$$

may initially enable either a, or b, or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.

Channel

A channel is a collection of events of the form $c.n$: the prefix c is called the *channel name* and the collection of suffixes is called the *values* of the channel.
When an event $c.n$ occurs it is said that *the value n is communicated on channel c.* When the value of a communication on a channel is determined by the environment (external choice) it is called an *input* and when it is determined by the internal state

of the process (internal choice) it is called an *output*.
It is convenient to write $c?n : N \rightarrow P(n)$ to describe behaviour over a range of allowed inputs instead of the longer $\Box\, n : N \bullet c.n \rightarrow P(n)$. Similarly the notation $c!n : N \rightarrow P(n)$ is used instead of $\Box\!\!\!\!\!\!\Big|\, n : N \bullet c.n \rightarrow P(n)$ to represent a range of outputs.

e.g.

$$COPYBIT = in.0 \rightarrow out.0 \rightarrow COPYBIT \;\; \Box \;\; in.1 \rightarrow out.1 \rightarrow COPYBIT$$

$$\alpha COPYBIT = \{in.0, out.0, in.1, out.1\}$$

Interrupt

The interrupt process $P_1 \;\triangledown\; e \rightarrow P_2$ behaves as P_1 until the first occurrence of interrupt event e, then the control passes to P_2.

Recursion

Recursion is used to given finite representations of non-terminating processes. The process expression

$$\mu\, P \bullet a?n : \mathbb{N} \rightarrow b!f(n) \rightarrow P$$

describes a process which repeatedly inputs a natural on channel a, calculates some function f of the input, and then outputs the result on channel b.

Traces of Processes

$$\begin{aligned} &traces(a \rightarrow P) = \\ &\qquad \{t : \operatorname{seq} A \;\mid\; t = \langle\,\rangle \\ &\qquad\qquad\qquad \lor \\ &\qquad\qquad\qquad \operatorname{head} t = a \;\land\; \operatorname{tail} t \in traces(P)\} \end{aligned}$$

$$\begin{aligned} &traces(a \rightarrow P \;\mid\; b \rightarrow Q) = \\ &\qquad traces(a \rightarrow P) \cup traces(b \rightarrow Q) \end{aligned}$$

$$
\begin{aligned}
&traces(x : B \to P(x)) = \\
&\quad \{t : \mathrm{seq}\, A \mid t = \langle\,\rangle \\
&\qquad\qquad \lor \\
&\qquad\qquad \mathrm{head}\, t \in B \ \land \ \mathrm{tail}\, t \in traces(P(\mathrm{head}\, t))\} \\
&\qquad = \cup\{x : B \bullet traces(x \to P(x))\}
\end{aligned}
$$

Traces of Recursive Process

$$
\begin{aligned}
&traces(\mu\, X : A \bullet F(X)) = \\
&\qquad \cup n : \mathbb{N} \bullet traces(F^n(STOP_A))
\end{aligned}
$$

$$VM = \mu\, X : \{coin, choc\} \bullet (coin \to choc \to X)$$

so in this case

$$F(X) = coin \to choc \to X$$

and

$$
\begin{aligned}
&F^0(STOP_A) = STOP_A \\
&F(STOP_A) = coin \to choc \to STOP_A \\
&F^2(STOP_A) \\
&\qquad = F(F(STOP_A)) \\
&\qquad = coin \to choc \to (coin \to choc \to STOP_A) \\
&\qquad = coin \to choc \to coin \to choc \to STOP_A \\
&F^3(STOP_A) = \cdots \\
&\cdots
\end{aligned}
$$

Timeout

The timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline.

The process

$$(a \to P) \triangleright \{t\}\ Q$$

will try to perform $a \to P$, but will pass control to Q if the a event has not occurred by time t, as measured from the invocation of the process. For example,

$$MayPrint1 = (receive \rightarrow print \rightarrow STOP) \triangleright\{60\}\ shutdown \rightarrow STOP$$

$$MP1(t) = (receive \rightarrow print \rightarrow STOP) \triangleright\{60 - t\}\ shutdown \rightarrow STOP$$

Exercise: Transmitter

A transmitter which repeatedly send a given message x until it receives and acknowledgement. Assume that the transmitter is in an environment which is always ready to accept a *send* message, then it will send the message every 5 time units until an *ack* message is received. (hint using recursion together with timeout).

Solution

$$Transmit(x) = send!x \rightarrow ((ack \rightarrow STOP) \triangleright\{5\}\ Transmit(x))$$

Delay

A process which allows no communications for period t then terminates is written WAIT t. The process

$$\text{WAIT } t;\ P\ =\ STOP \triangleright\{t\}\ P$$
$$a \xrightarrow{t} P\ =\ a \rightarrow \text{WAIT } t;\ P\ =\ a \rightarrow (STOP \triangleright\{t\}\ P)$$

is used to represent P delayed by time t.

State parameters

In general, the behaviour of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values.

It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state.

The approach adopted by CSP is to allow a process definition to be parameterised by state variables. Thus a definition of the form

$$P_{n:N} \mathrel{\widehat{=}} Q(n)$$

represents a (possibly infinite) family of definitions, one for each possible value of n.

There is no inherent notion of process state in CSP, but rather these annotations are a convenient way to provide a finite representation of an infinite family of process descriptions.

Exercise: A generic timed-collection

The generic timed-collection denotes a collection of elements of type X with a time stamp. Operations are allowed to add elements to and delete elements from the collection. When deleting an element from the collection, the oldest element should be removed and output to the element should be removed and output to the environment. The collection has the following timing properties. Firstly, that it updates the internal state during a *add* or *delete* operation. Secondly, each element of the collection becomes *stale* if it is not passed on within t_o time units of being added to the collection. Stale elements should never be passed on, but are instead purged from the collection upon becoming stale.
The generic function ps (purge stale) can be defined as

$$[X]$$
$$ps : (\mathbb{T} \times \mathbb{F}(\mathbb{T} \times X)) \to \mathbb{F}(\mathbb{T} \times X)$$
$$\forall\, t : \mathbb{T};\ s : \mathbb{F}(\mathbb{T} \times X) \bullet ps(t, s) = \{(t_o, e) : s \mid t_o > t \bullet (t_o - t, e)\}$$

e.g. $ps(2, \{(1, a), (3, b), (7, c)\}) = \{(1, b), (5, c)\}$.

Solution

$$TimedCollection \mathrel{\widehat=} TC_{\varnothing}.$$

$$TC_{\varnothing} \mathrel{\widehat=} left?e : X \to TC_{\{(t_o,e)\}}$$

$$TC_{\{(t,a)\}\cup s} \mathrel{\widehat=}$$
$$(left?e : X\,@t_i \to TC_{ps(t_i,\{(t,a)\}\cup s)\cup\{(t_o,e)\}} \ \Box$$
$$right!a\,@t_i \to TC_{ps(t_i,s)}) \rhd\{t\}\ TC_{ps(t,s)}$$

where $(t, a) = find\_oldest(\{(t, a)\} \cup s)$.

$$
\begin{array}{|l}
\hline
[X] \\
\mathit{find\_oldest} : \mathbb{P}_1(\mathbb{T} \times X) \to (\mathbb{T} \times X) \\
\hline
\forall\, s : \mathbb{P}_1(\mathbb{T} \times X) \bullet \\
\quad \exists (t, e) : s \bullet t = \mathit{min}(\operatorname{dom} s) \\
\quad \mathit{find\_oldest}(s) = (t, e) \\
\hline
\end{array}
$$

Summary

For such an example Timed CSP is superior to Object-Z as a means of describing process control.
Timed CSP also handles the timing issues of delays and timeouts simply and elegantly. The allowed sequences of events are clearly and concisely determined by the CSP code, there is no need to calculate preconditions nor is any other form of deep reasoning required to understand the ways in which the timed-collection may evolve.
On the other hand, the syntactic treatment of internal state in the above is complex and unwielding, distracting strongly from the basically elegant treatment of the delay and timeout issues.
CSP still has no standard support for state modeling in the form of mathematical toolkits and libraries nor are there modular techniques for constructing and reasoning about complex internal state.

8.0.2 Model Checking LSC in CSP/FDR

8.0.3 Synthesis from LSC

Chapter 9

Timed Automata and Timed Patterns

This chapter is devoted to an introduction on the notion of automata and timed automata. The graph-based modelling technique, Timed Automata (TA), has powerful mechanisms for designing real-time models using multiple clocks and has well developed automatic tool support. One weakness of TA is the lack of high-level composable graphical patterns to support the systematic design for complex systems. We define a set of composable graphical patterns can be defined based on the semantics of the TCOZ constructs, so that those patterns can be reused in a generic way.

9.1 Timed Automata

The notion of Timed Automata is proposed by R. Alur in the early 90's [2, 77]. Since then, it has gained a lot of attention for modelling, simulating and verifying real-time systems. Timed Automata are finite state machines with clocks. It was introduced as a formal notation to model the behavior of real-time systems. Its definition provides a general way to annotate state-transition graphs with timing constraints using finitely many real-valued clock variables.

Definition 9.1.1 *A timed automaton A is a tuple* (S, Σ, C, I, T)*, where* S *is a finite set of states,* Σ *is a set of actions/events,* C *is a finite set of clocks,* I *is a mapping that labels each state* s *in* S *with some clock constraint* $\Phi(C)$*, and* T *is a subset of* $S \times S \times \Sigma \times 2^C \times \Phi(C)$*, is the set of transitions. A switch* $\langle s, s', a, \lambda, \delta \rangle$ *represents a transition from state* s *to state* s' *on input symbol* a*. The set* λ *gives the clocks to be reset with this transition, and* δ *is a clock constraint over* C *that specifies when the switch is enabled.* $\Phi(C)$*is a set of clock constraints which is defined by the following*

grammar:

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2$$

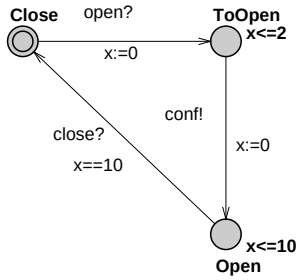


Figure 9.1: Car Door

A railcar door can be designed as in Figure 9.1. A whole railcar system will be demonstrated in a later section as a case study. The door has three states: *Open*, *Close* and *ToOpen* (we assume the door can be closed immediately, so there is no *ToClose* state). Through channels *open* and *close*, it responds to an open command from its car handler within 2 time units, and closes after the car handler send a close command. Passengers are allowed 10 time units for boarding. The state *Close* is the initial state, as designated by the double circle.

As usual for automata, an initial or set of initial states can also be specified, as well as terminal state(s). In the following, we assume that the special event τ, denoting an internal transition, is included in the events Σ. There are a number of tool support for verifying temporal logics with quantitative temporal operators over systems modelled using Timed Automata. The most successful one of them is UPPAAL [9]. UPPAAL is a tool for modelling, simulation and verification of real-time systems. It is developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. UPPAAL targets for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical. A system description shall consist of a set of instances of timed automata declared from the process templates, and of some global data, such as global clocks, variables, synchronization channels.

UPPAAL consists of three main parts: a system editor, a simulator and a model checker. The system editor provides a graphical interface for the tool. Its output is an XML representation of the timed automata. The simulator is a validation tool

which enables examination of possible dynamic executions of a system. Simulation during early design stages provides an inexpensive mean of fault detection prior to verification by the model checker which covers the exhaustive dynamic behavior of the system. The model checker checks invariant and bounded liveness properties by exploring the symbolic state space of a system, i.e., reachability analysis in terms of symbolic states represented by constraints. The model checking engine of UPPAALis designed to check a subset of Timed CTL [4] formula for networks of timed automata. The formulas contain no nested quantifiers and should be one of the following forms:

$\mathbf{A}[\,]\phi$	– invariantly ϕ
$\mathbf{E} <> \phi$	– possibly ϕ
$\mathbf{A} <> \phi$	– always eventually ϕ
$\mathbf{E}[\,]\phi$	– potentially always ϕ
$\phi \rightarrow \psi$	– shorthand for $\mathbf{A}[\,](\phi \Rightarrow \mathbf{A} <> \psi)$

where ϕ, ψ are local properties that can be checked locally on a state, i.e., boolean expressions over predicates on the states and integer variables, and clock constraints.

9.2 Timed Automata Patterns

High-level real-time system requirements often need to state the system timing constraints in terms of *deadline*, *timeout*, and *waituntil* commands which can be regarded as common timing constraint patterns. For example, "task A must complete within t time units" is a typical one (*deadline*). TCOZ was developed for modeling the high-level real-time system requirements; it has the composable language constructs that directly capture those common timing patterns. On the other hand, if TA is used to capture real-time requirements, then one often needs to manually cast those timing patterns into a set of clock variables with carefully calculated clock constraints, which is a process that is very much closer to design rather than specification. One interesting question is the following: can we build a set of TA patterns that correspond to the TCOZ timing constructs? If such a set of TA patterns can be formulated, then not only the transformation from TCOZ to TA can be readily achieved, but also TA can be more applicable for capturing high-level requirements if those TA patterns are utilized.

9.2.1 Z definition of Timed Automata

In this section, we define a set of composable TA patterns in the Z meta notation. First of all, we give the definition of TA in Z as follows,

$$[State, Event]$$
$$\mathbb{T} == \{x : \mathbb{R} \mid x \geq 0\}$$
$$Clock == \{x : \mathbb{R} \mid x \geq 0\}$$

There are two basic types, i.e., *State* and *Event*. $\mathbb{R}$ and *Clock* are two sets of positive real numbers.

$$\begin{aligned}\Phi ::= &(\_ \leq \_)\langle\langle Clock \times \mathbb{R}\rangle\rangle \mid (\_ \geq \_)\langle\langle Clock \times \mathbb{R}\rangle\rangle \mid \\ &(\_ < \_)\langle\langle Clock \times \mathbb{R}\rangle\rangle \mid (\_ > \_)\langle\langle Clock \times \mathbb{R}\rangle\rangle \mid \\ &(\_ \wedge \_)\langle\langle \Phi \times \Phi\rangle\rangle \mid true\end{aligned}$$

Φ defines the types of clock constraints, in which a *true* type is added here to represent the empty set of clock constraints. The following function is defined to extract the set of clocks from a clock constraint.

$$clk : \Phi \leftrightarrow \mathbb{P}\, Clock$$
$$\begin{aligned}&clk(true) = \varnothing \\ &\forall\, x : Clock;\ t : \mathbb{R} \bullet clk(x \leq t) = \{x\} \wedge clk(x \geq t) = \{x\} \\ &\qquad clk(x < t) = \{x\} \wedge clk(x > t) = \{x\} \\ &\forall\, \varphi_1, \varphi_2 : \Phi \bullet clk(\varphi_1 \wedge \varphi_2) = clk(\varphi_1) \cup clk(\varphi_2)\end{aligned}$$

A timed automaton $\mathcal{S}_{TA}$ is defined as a state binding (binding is a partial function from variables to values), in which S models states; i and e respectively represent the initial state and terminal state; I defines local invariants which give a clock constraint to each state; C is a set of clock variables; T models transitions; the second element of T, i.e., *Label*, models transition conditions, in which *Event* is an enabling event, $\mathbb{P}\, Clock$ gives a set of clocks, and Φ specifies a clock constraint.

$$Label \mathrel{\widehat=} Event \times \mathbb{P}\, Clock \times \Phi$$
$$Transition \mathrel{\widehat=} State \times Label \times State$$

$\mathcal{S}_{TA}$

$S : \mathbb{P}\, State;\quad i, e : State$
$\Sigma : \mathbb{P}\, Event$
$C : \mathbb{P}\, Clock$
$I : State \pfun \Phi$
$T : \mathbb{P}\, Transition$

$\{i, e\} \subseteq S \land \operatorname{dom} I = S$
$\forall \varphi : \Phi \bullet clk(\varphi) \subseteq C$
$\forall s, s' : State;\ l : Label \bullet (s, l, s') \in T \Rightarrow \{s, s'\} \subseteq S$
$\quad \land\ \pi_1(l) \in \Sigma \land \pi_2(l) \subseteq C$

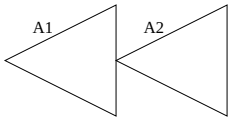
Without loss of generality, we define one initial/terminal state here instead of a set of initial/terminal states for a timed automaton. In case there are multiple initial/terminal states, we assume there is a unique initial/terminal state which is connected to all initial/termianl states through an internal transition τ. Note that the terminal state may be identical to the initial state, and in some cases it may be unreachable, and thus omitted from a concrete automaton.

9.2.2 TA patterns

In the following, a set of TA patterns are defined according to TCOZ constructs in Z together with their graphic presentations. In these graphical TA patterns, an automaton A is abstracted as a triangle, the left vertex of this triangle or a circle attached to the left vertex represents the initial state of A, and the right edge represents the terminal state of A. Those timed patterns together with their formal definitions can be readily understood. The timed composable patterns can be seen as a reusable high-level library that may facilitate a systematic engineering process when TA is used to design the timed systems. Furthermore, these patterns provide an interchange media for transforming TCOZ specifications into TA designs.

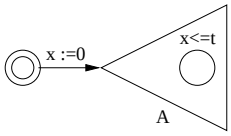
$seq : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \to \mathcal{S}_{TA}$

$\forall A_1, A_2 : \mathcal{S}_{TA} \bullet$
$\quad seq(A_1, A_2) = (\!|$
$\quad S \mathrel{\widehat=} A_1.S \cup A_2.S, i \mathrel{\widehat=} A_1.i, e \mathrel{\widehat=} A_2.e,$
$\quad \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma, C \mathrel{\widehat=} A_1.C \cup A_2.C, I \mathrel{\widehat=} A_1.I \cup A_2.I,$
$\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(A_1.e, (\tau, \varnothing, true), A_2.i)\}\ |\!)$



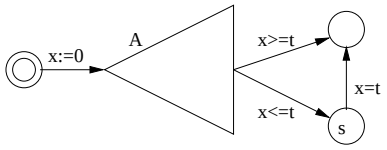
The *sequential composition* pattern: there are two timed automata A_1, A_2. By linking the terminal state of A_1 with the initial state of A_2, the resulting automaton passes control from A_1 to A_2 immediately when A_1 goes to its terminal state.

$$
\begin{array}{|l}
deadline : \mathcal{S}_{TA} \times \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A : \mathcal{S}_{TA};\ t : \mathbb{R};\ \exists\, x : Clock;\ i_0 : State \bullet \\
\quad deadline(A, t) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} A.e, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma, C \mathrel{\widehat=} A.C \cup \{x\}, \\
\quad I \mathrel{\widehat=} \{s : A.S \bullet (s, x <= t \land A.I(s))\} \cup \{i_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (\tau, \{x\}, true), A.i)\} \ |\!\rangle
\end{array}
$$



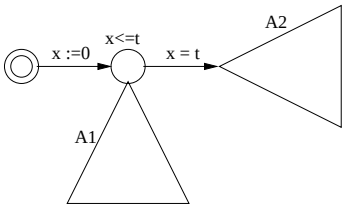
The *deadline* pattern: there is a single fresh clock x. When the system switches to the automaton A, the clock x gets reset to 0. The local invariant $x <= t$ covers each state of the timed automaton A and specifies the requirement that a switch must occur before t time units for every state of A. Thus the timing constraint expressed by this automaton is that A should terminate no later than after t time units.

$$
\begin{array}{|l}
waituntil : \mathcal{S}_{TA} \times \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A : \mathcal{S}_{TA};\ t : \mathbb{R};\ \exists\, x : Clock;\ i_0, e_0, s : State \bullet \\
\quad waituntil(A, t) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0, s, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma, C \mathrel{\widehat=} A.C \cup \{x\}, \\
\quad I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto true, s \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (\tau, \{x\}, true), A.i), \\
\qquad\qquad (A.e, (\tau, \varnothing, x >= t), e_0), (A.e, (\tau, \varnothing, x <= t), s), \\
\qquad\qquad (s, (\tau, \varnothing, x = t), e_0)\} \ |\!\rangle
\end{array}
$$



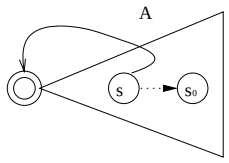
The *waituntil* timed pattern: the automaton is constrained to finish its process not earlier than after t time units. Two situations are captured here: if the process of A finishes earlier then t time units, then the automaton idles until t time units and if the process of A takes more than t time units, then the automaton terminates as A terminates.

$$
\begin{array}{|l}
timeout : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A_1, A_2 : \mathcal{S}_{TA};\ t : \mathbb{R};\ \exists\, x : Clock;\ i_0, e_0 : State;\ \varphi : \Phi \mid \\
\quad \varphi = I(A_1.i) \bullet timeout(A_1, A_2, t) = \langle\!| \\
\quad S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \\
\quad \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma, C \mathrel{\widehat=} A_1.C \cup A_2.C \cup \{x\}, \\
\quad I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\} \oplus \{A_1.i \mapsto x <= t \land \varphi\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \\
\qquad \{(i_0, (\tau, \{x\}, true), A_1.i), (A_1.i, (\tau, \varnothing, x = t), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \ |\!\rangle
\end{array}
$$



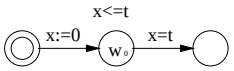
The *timeout* pattern: there are two timed automatons A_1 and A_2. If no transition has been triggered for t time units in timed automaton A_1, then A_1 will timeout and the control will be passed to A_2.

$$
\begin{array}{|l}
recursion : \mathcal{S}_{TA} \times State \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A : \mathcal{S}_{TA};\ s_0 : State \mid s_0 \in A.S \bullet \\
\quad recursion(A, s_0) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S, i \mathrel{\widehat=} A.i, e \mathrel{\widehat=} A.e, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma, C \mathrel{\widehat=} A.C, I \mathrel{\widehat=} A.I, \\
\quad T \mathrel{\widehat=} A.T \cup \{s : State, l : Label \mid (s, l, s_0) \in A.T \bullet (s, l, i)\} \\
\qquad \setminus \{s : State, l : Label \mid (s, l, s_0) \in A.T \bullet (s, l, s_0)\} \ |\!\rangle
\end{array}
$$



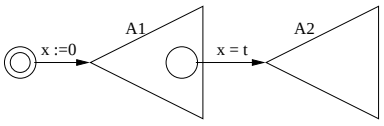
The *recursion* pattern: given a timed automaton A and a state s_0, which is a fixed point. The recursion is achieved by diverting all the transitions from pointing to s_0 to the initial state of A. The dotted arrow represents the transitions which are originally pointing to s_0. In the resultant automaton these transitions are replaced by transitions which points to the initial state of A.

$$
\begin{array}{|l}
wait : \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, t : \mathbb{R};\ \exists\, x : Clock;\ i_0, w_0, e_0 : State \bullet \\
\quad wait(t) = (\!|\ S \mathrel{\widehat=} \{i_0, w_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \\
\quad I \mathrel{\widehat=} \{i_0 \mapsto true, w_0 \mapsto x <= t, e_0 \mapsto true\}, \\
\quad \Sigma \mathrel{\widehat=} \varnothing, C \mathrel{\widehat=} \{x\}, \\
\quad T \mathrel{\widehat=} \{(i_0, (\tau, \{x\}, true), w_0), (w_0, (\tau, \varnothing, x = t), e_0)\}\ |\!)
\end{array}
$$



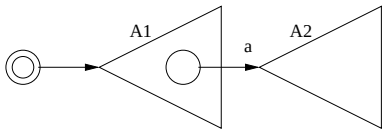
The *wait* pattern: Time idles at its second state for t time units then terminates.

$$
\begin{array}{|l}
tinterrupt : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A_1, A_2 : \mathcal{S}_{TA};\ t : \mathbb{R};\ \exists\, x : Clock;\ i_0, e_0 : State \bullet \\
\quad tinterrupt(A_1, A_2, t) = (\!|\ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma, \\
\quad C \mathrel{\widehat=} A_1.C \cup A_2.C \cup \{x\}, I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(i_0, (\tau, \{x\}, true), A_1.i)\} \\
\qquad \cup \{s : A_1.S \bullet (s, (\tau, \varnothing, x = t), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\}\ |\!)
\end{array}
$$



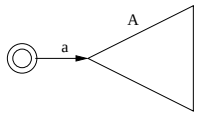
The *timed interrupted* pattern: it is composed of two automatons, A_1 and A_2, the control will be passed from A_1 to A_2 immediately if A_1 cannot finish its process within t time units, that is, when the value of clock x increases to t, there will be a transition to the initial state of A_2 from any state of A_1.

$$
\begin{array}{|l}
einterrupt : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times Event \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A_1, A_2 : \mathcal{S}_{TA};\ a : Event;\ \exists\, i_0, e_0 : State \bullet \\
\quad einterrupt(A_1, A_2, a) = (\!|\ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma \cup \{a\}, \\
\quad C \mathrel{\widehat=} A_1.C \cup A_2.C, I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(i_0, (\tau, \varnothing, true), A_1.i)\} \\
\qquad \cup \{s : A_1.S \bullet (s, (a, \varnothing, true), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\}\ |\!)
\end{array}
$$



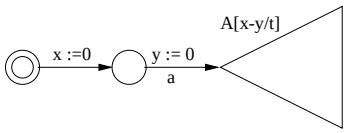
The *event interrupted* pattern: it is composed of two automatons, A_1 and A_2, the control will be passed from A_1 to A_2 immediately when event a happens.

$$
\begin{array}{|l}
eprefix : Event \times\ \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, a : Event;\ A : \mathcal{S}_{TA};\ \exists\, i_0 : State \bullet \\
\quad eprefix(a, A) = (\!|\ S \mathrel{\widehat=} A.S \cup \{i_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} A.e, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma \cup \{a\}, C \mathrel{\widehat=} A.C, I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (a, \varnothing, true), A.i)\}\ |\!)
\end{array}
$$



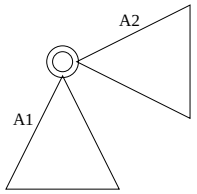
The *event prefix* pattern: if event a happens, then the control will be passed to A immediately.

$$
\begin{array}{|l}
\hline
tprefix : Event \times \mathbb{R} \times \ \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, a : Event;\ A : \mathcal{S}_{TA};\ t : \mathbb{R};\ \exists\, x, y : Clock; \\
\quad i_0, s_0 : State \mid t = x - y \bullet tprefix(a, t, A) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0, s_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} A.e, \Sigma \mathrel{\widehat=} A.\Sigma \cup \{a\}, \\
\quad I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto true, s_0 \mapsto true\}, C \mathrel{\widehat=} A.C \cup \{x, y\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (\tau, \{x\}, true), s_0)\} \cup \{(s_0, (a, \{y\}, true), A.i)\}\ |\!\rangle \\
\hline
\end{array}
$$



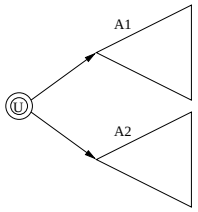
The *timed event prefix* pattern: there are two clocks, x is reset at the initial state, y is reset when event a happens. The time difference of x and y records the time point at which a happens and substitutes the variable t in the timed automaton enabled by event a for further use.

$$
\begin{array}{|l}
\hline
extchoice : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A_1, A_2 : \mathcal{S}_{TA};\ \exists\, i_0, e_0 : State \bullet \\
\quad extchoice(A_1, A_2) = \langle\!| \ \ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\} - \{A_1.i, A_2.i\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, C \mathrel{\widehat=} A_1.C \cup A_2.C, \\
\quad I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{(i_0, I(A_1.i) \wedge I(A_2.i))\} - \\
\qquad \{(A_1.i, I(A_1.i)), (A_2.i, I(A_2.i))\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \\
\qquad \cup \{l : Label, k : State \mid (A_1.i, l, k) \in A_1.T \\
\qquad\quad \vee (A_2.i, l, k) \in A_2.T \bullet (i_0, l, k)\} \\
\qquad \cup \{l : Label, k : State \mid (k, l, A_1.i) \in A_1.T \\
\qquad\quad \vee (k, l, A_2.i) \in A_2.T \bullet (k, l, i_0)\} \\
\qquad \setminus \{t : Transition \mid \pi_1(t) = A_1.i \vee \pi_1(t) = A_2.i \\
\qquad\quad \vee \pi_3(t) = A_1.i \vee \pi_3(t) = A_2.i \bullet t\}\ |\!\rangle \\
\hline
\end{array}
$$



The *external choice* pattern: timed automaton A_1 and A_2 share a common initial state and the environment has the choice to trigger one of them by different external events.

$$
\begin{array}{|l}
intchoice : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall A_1, A_2 : \mathcal{S}_{TA};\ \exists i_0, e_0 : State \bullet \\
\quad intchoice(A_1, A_2) = (\!| \ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma, \\
\quad C \mathrel{\widehat=} A_1.C \cup A_2.C, I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(i_0, (\tau, \varnothing, true), A_1.i), (i_0, (\tau, \varnothing, true), A_2.i), \\
\qquad (A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \ |\!)
\end{array}
$$



The *internal choice* pattern: there are two timed automata A_1 and A_2 and an urgent state as the initial state. The choice of which automaton to be triggered is decided by internal events of timed automata A_1 and A_2. An urgent state [72] is a state marked with 'U' where no delay is allowed.

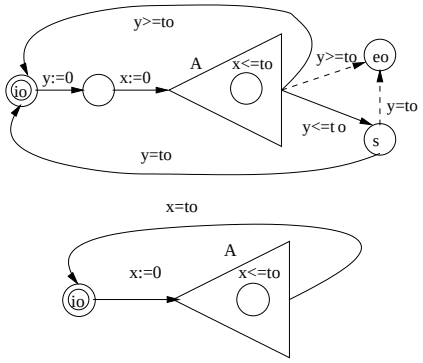
The patterns we presented are defined according to the basic TCOZ process constructs, i.e., TCSP constructs. Hence, these patterns are the most concise yet complete set of patterns which preserve the expressiveness of TCSP in describing dynamic real-time behaviors. A question may arise here is that: is there a small set of time patterns that can be used to construct the other patterns? In TCSP, the *wait* construct is the most essential one to represent timing constraints. *timeout*, *timedinterrupt* could be composed by *choice* and *wait*. Nevertheless, *wait* construct can also be constructed by *timeout*, *skip* and *stop* expressions. Hence it is subjective to define the smallest set of patterns. These issues have been discussed by Jim Davis in his Ph.D thesis [20]. However, we would like to give the definitions of all these patterns directly since they are the most common timing constraints and behaviors, thus can be regarded as the set of basic patterns.

9.2.3 Generating New Patterns

New patterns can be composed from the existing ones. For example, the new timing pattern *PeriodicRepeat*, which specifies "Task A is repeated every t_0 time units provided that A is guaranteed to terminate before t_0 time units", can be composed by three existing patterns - *deadline*, *waituntil* and *recursion*, as shown in Figure (a). Assuming A is the automaton which performs the task A, clocks x and y are generated

to respectively give the time constraints for the *deadline* and *waituntil* pattern. The terminal state of the automaton A_0 ($A_0 = waituntil(deadline(A, t_0), t_0)$), e_0, is the fix point for the recursion pattern, thus the transitions of A_0 which were originally pointing to e_0 are diverted and point to the initial state of A_0.

$$
\begin{array}{|l}
PeriodicRepeat : \mathcal{S}_{TA} \times \mathbb{R} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A, A_0 : \mathcal{S}_{TA};\ t_0 : \mathbb{R};\ e_0 : State \mid e_0 = A_0.e \\
\land\ A_0 = waituntil(deadline(A, t_0), t_0) \bullet \\
\quad PeriodicRepeat(A, t_0) = recursion(A_0, e_0) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A.\Sigma, \\
\quad C \mathrel{\widehat=} A.C \cup \{x\}, I \mathrel{\widehat=} \{s : A.S \bullet (s, x <= t_0 \land A.I(s))\} \\
\qquad \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} \{(i, (\tau, \{x\}, true), A.i)\} \cup \{A.e, (\tau, \varnothing, x = t_0), i)\} \cup\ A.T\ |\!\rangle
\end{array}
$$



The resultant automaton can be simplified as shown in Figure (b), in which any consecutive initial, terminal and intermediate states linked with a τ transition label are incorporated into one state to simplify the resultant automaton. To further reduce the state space of the resultant timed automaton, clock y is removed by reusing clock x. The two diverted transitions are merged into one transition since they can only be enabled when the value of clock x equals t_0.

Many complex real-time systems can be naturally modelled as collections of small processes lying in different layers of the systems, operating and interacting sequentially or concurrently. Our generic TA patterns provide a set of templates to decompose a complex real-time system into different layers and smaller components. There are certain guidelines for the engineers to use these generic timed patterns for systematic TA design:

- Decide the layers of the complex system. The abstracted triangle automaton can be seen as an outside layer, which will be substituted by its inside layer. The TA model of a system can finally be generated in a top-down way by stepwise refining its inside layers using appropriate patterns.

  - For most reactive systems, usually the outermost layer of such systems can be modelled by a recursion pattern.
  - For systems which run one time only, a sequential pattern can be used to model its outermost layer.

- Decompose the complex processes of one layer into smaller processes with simple behaviors. Those smaller processes mostly are composed together by sequential composition or alternation.
  - To describe a process which has different behaviors, usually the *external choice* or *internal choice* pattern should be applied.
  - To capture a process which has exceptions, usually the *timed interrupt* or *event interrupt* patterns should be chosen.
  - To specify requirement constraints, e.g., timing constraints, the *deadline* and *waituntil* patterns can be utilized.
  - Sequential behaviors can be captured by utilizing the sequential composition pattern.

Part three *Intergrated Formalisms and Synthesis* inculdes ? chapters.

Chapter 10

Timed Communicating Object Z

Timed Communicating Object Z (TCOZ) [83, 80] is essentially a blending of Object-Z [39] with Timed CSP [21], for the most part preserving them as proper sub-languages of the blended notation. The essence of this blending is the identification of Object-Z operation specification schemas with terminating CSP processes. Thus operation schemas and CSP processes occupy the same syntactic and semantic category, operation schema expressions may appear wherever processes may appear in CSP and CSP process definitions may appear wherever operation definitions may appear in Object-Z. The primary specification structuring device in TCOZ is the Object-Z class mechanism.
In this section we briefly consider various aspects of TCOZ. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [83]. The formal semantics of TCOZ is also documented [81, 97].

10.1 TCOZ Basics

In many ways, Object-Z and Timed CSP complement each other in their capabilities. Object-Z has strong data and algorithm modeling capabilities. The Z mathematical toolkit is extended with object oriented structuring techniques. Timed CSP has strong process control modeling capabilities. The multi-threading and synchronisation primitives of CSP are extended with timing primitives. Moreover, both formalisms are already strongly influenced by the other in their areas of weakness. Object-Z supports a number of primitives which have been inspired by CSP notions such as external choice and synchronisation. CSP practitioners tend to make use of notation inspired by the Z mathematical toolkit in the specification of processes with internal state. This is not surprising given their joint associations in the Programming Research Group, Oxford. Another important connection is the well-known duality between the state transition behavioural model and the event based behavioural model [54] which

makes it a simple matter to develop complementary semantics for the two languages. Given these factors it is natural to consider the possibility of blending the two notations into a more complete approach to modeling real-time and/or concurrent systems. Fischer [41] and Smith [106] have independently suggested CSP-style semantics for Object-Z classes in which operation calls become CSP events. Operation names take on the role of CSP channels, with input and output parameters being passed down the operation channel as values. This view fits nicely with the Object-Z interpretation of operations being atomic, but is not well suited to considering multi-threading and real-time. Restricting operations to atomic events collapses the spatial and temporal aspects of operations, everything happens at a single point and instantaneously. Identifying channel names with operation names creates unnecessary tensions between the data and process views of objects and considerably reduces the potential for reuse of operation definitions.

The approach taken in the TCOZ notation is to identify operation schemas with terminating CSP processes that perform only state update events; to identify (active) classes with non-terminating CSP processes; and to allow arbitrary (channel-based) communications interfaces between objects.

The syntactic implications of this approach is that the basic structure of a TCOZ document is the same as for Object-Z. A document consists of a sequence of definitions, including type and constant definitions in the usual Z style. TCOZ varies from Object-Z in the structure of class definitions, which may include CSP channel and processes definitions. In fact, all operation definitions in TCOZ are considered to define CSP processes. The CSP view of an operation schema is that it describes all the sequences of update events which change the system state as required by the schema predicate. The exact nature and granularity of these update events is left undetermined in TCOZ (at least at the syntactic level), but by allowing an operation to consist of a number of events, it becomes feasible to specify its temporal properties when describing the operation. Another advantage of this approach is that by combining simple operations with CSP operators it becomes possible to represent true multi-threaded computation even at the operation level. The Fischer/Smith approach of identifying operation names with CSP channels is not followed, channels are given an independent, first class role. This allows the communications and control topology of a network of objects to be designed orthogonally to their class structure. The CSP channel mechanism is the only (dynamic) way to pass information between objects as the state of objects is encapsulated by hiding all update events.

10.1.1 Defining operations

The operation schema is the basic tool for describing state change in TCOZ. In order to allow treatment of timing issues in schema definitions, a distinguished identifier δ is introduced to represent the duration of the state calculations performed by the

operation. When δ does not appear in the definition of an operation, the default interpretation is that there be no bound on the duration of the operation, although individual specification documents may choose to adopt a different convention.
Although the schema is the basic tool, the true power of TCOZ comes from the ability to make use of Timed CSP primitives is describing the process aspects of an operation's behaviour. All operation definitions in TCOZ are in fact Timed CSP process definitions, with operation schema being given the syntactic status of terminating Timed CSP processes.
As an example, consider the specification of the *Add* operation of the timed-collection example. The actual state-change allowed by the operation schema remains unchanged from the Timed Object-Z version, but the timing characteristics of the operation are expressed by the condition $\delta = t_a$, rather than $now' - now = t_a$.

$$
\begin{array}{|l}
\hline
Add_0 \\
\Delta(mems) \\
e? : X \\
t_i? : \mathbb{T} \\
\hline
\delta = t_a \land mems' = ps(t_i? + t_a, mems) \cup \{(t_o, e?)\} \\
\hline
\end{array}
$$

Since TCOZ operations are identified with terminating CSP processes, it is natural to allow their definition in terms CSP primitives, such as event sequencing, as well as through the schema calculus. The novelty of the full TCOZ version of *Add* lies in the adoption of CSP primitives in its definition. Item inputs are communicated to the *Add* operation along a channel *left*.

$$Add \mathrel{\widehat=} [e : X;\ t_i : \mathbb{T}] \bullet left?e\,@t_i \rightarrow Add_0$$

This definition of *Add* says that after the parameter e has been input on channel *left* at time t_i, the state-calculation Add_0 is performed. Several aspects of TCOZ name-space conventions are raised by this definition.
Firstly, observe that the parameters e and t_i occur in Add_0 with Z-style input decorations and in *Add* without them. In TCOZ the convention is adopted that the true name of all parameters is the undecorated version. The ? and ! decorations are used solely in operation schemas, to distinguish between inputs and outputs. This is analogous to the convention of using primed and unprimed versions to indicate the final and initial values of a state attribute. In fact, all parameters are treated in the same way as state attributes, with the exception that state attributes are available in every name environment in a class definition.
This leads to the second observation that may be made of the *Add* definition. The $[e : X;\ t_i : \mathbb{T}] \bullet \_$ construct is a local block definition in the state guard style

(state guards are explained further below). The other forms of local blocks are the intentionial forms of both internal and external choice, which use the usual Z-style schema-text conventions. For example, $\bigsqcap n : \mathbb{N} \mid n < 5 \bullet c!n \rightarrow P$ or $\Box\, n : \mathbb{N} \mid n < 5 \bullet c?n \rightarrow P$. The state guard serves as an alternate form of external choice, so that the *Add* process is equivalent to $\Box\, e : X;\ t_i : \mathbb{T} \bullet \mathit{left}?e@t_i \rightarrow \mathit{Add}_0$.
In TCOZ, the local name space may be changed either by a local block definition as above or else by the occurance of an operation schema. An operation schema removes all its input parameters from scope and replaces them with its output parameters. The output parameters then become available for use in subsequent communication events or as inputs to subsequent operation schemas.
In the case of the *Delete* operation, the communication of the deleting element must precede the updating of the collection state and in fact is the enabling event for the operation. Since the name convention is that outputs are only available to the right of a schema, this behaviour cannot be described using an output parameter. Instead, the update operation is described as a simple state update which removes the oldest item (and any others that become stale).

Delete_0

$$\Delta(\mathit{mems})$$
$$t_i? : \mathbb{T}$$

$$\mathit{mems} \neq \varnothing \land \delta = t_d \land \mathit{mems}' = \mathit{ps}(t_i + t_d, \mathit{mems} \setminus \{(t, \mathit{oldest})\})$$

The overall delete operation consists of this schema guarded by a communication on the *right* channel.

$$\mathit{Delete} \mathrel{\widehat=} [t_i : \mathbb{T} \mid \mathit{mems} \neq \varnothing] \bullet \mathit{right}!\mathit{oldest}@t_i \rightarrow \mathit{Delete}_0$$

The first part of the definition of *Delete* is a novel process control primitive known as a *state guard*.[1] The adoption of a state guard mechanism allows TCOZ to adopt a proper seperation between algorithm and process design issues. The sequencing of activities in an object is controlled explicitly through state guards rather than implicitly through the operation preconditions. In this way it becomes possible to reclaim the Z-style operation design and decomposition techniques abandoned by standard Object-Z.
Every process definition has (at least) an initial state which may be addressed using schema notation. This is the function of the first part of the expression defining

[1] Although if-then style commands appear in several dialects of CSP, for example CSP_M [98], we believe that TCOZ is unique in adopting the state guard as a seperate primitive in the style of Morgan's version of the guarded command language [90].

Delete. It is a schema-based method of restricting the action of the process to initial states for which the collection is non-empty. For other states this process will *deadlock* or *block*, which is to say refuse any communication.
Note that the precondition requirement in the $Delete_0$ schema, though identical, could not achieve the desired restriction on the behaviour of *Delete*. Failure to satisfy a precondition when control is passed to an operation instead results in *divergence*, which is to say unspecified subsequent behaviour. $Delete_0$ places no restrictions at all on its behaviour when the initial queue is empty. The precondition is the state-based equivalent of process divergence and the guard is the state-based equivalent to process deadlock.
For every operation P (even those constructed using the process calculus) the collection of initial states for which the process will not diverge is called its *precondition* (written $\mathrm{pre}\, P$) and the collection of states for which it will not deadlock is called its *guard* (written $\mathrm{grd}\, P$).

10.1.2 Schemas and Processes

A schema expression describes a relationship on or between process state/s, whilst a process expression describes the overall behaviour or evolution of a process. The Z semantic model for operation schemas consists of sets of variable *bindings*, mappings from variable names to values. One semantic model for Timed CSP processes consists of sets of tuples consisting of a *trace* (a sequence of time stamped events), a *refusal* (a record of when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model the failures/divergences model. The basic approach taken in the TCOZ semantics is to adopt the Timed CSP semantic model and to provide an interpretation of the Z semantic model in terms of failures and divergences, though two additions are required to make this possible. Firstly, a variable binding is added to represent the initial values of all the process attributes. Secondly, a new class of events, referred to as *update* events, is introduced to represent changes to the process attributes. The resulting model is called the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is if a ✓ event occurs), then the state at the time of termination is called the *final* state.
The process model of an operation schema consists of all initial states and update traces (terminated with a ✓) such that the initial state and the final state satisfy the relation described by the schema. If no legal final state exists for a given initial state, the operation diverges immediately.
The process model for the state guard consists of replacing the trace part of every behaviour of the guarded process whose initial state does not satisfy the state guard with the empty trace. The empty event trace describes the process being blocked by

the failure of the state guard. In addition, divergence cannot occur if the state guard is not satisfied.

Since schema calculus operators cannot sensibly be applied to arbitrary CSP processes, it is necessary to strictly distinguish the schema calculus from the process calculus (see Appendix **??**). The two exceptions to this are the type-casting of operation schema expressions as terminating processes and of initial state schema expressions as state guards. In all other circumstance the schema and process calculi are separate and distinct. For example, if P and Q are operations schema expressions, the expression $a \rightarrow (P \land Q)$ is legal whilst the expression $(a \rightarrow P) \land Q$ is not. The full power of the schema calculus may be used to construct schema expressions, but once a schema expression has been cast into a process-like role it may no longer act in a schema-like role.

Some existing Object-Z schema calculus operators, such as $\_ \Box \_$, $\_ \parallel \_$, and $\_ ; \_$, have name-sakes with similar semantics in the CSP process calculus. The convention adopted in TCOZ is that the CSP operator is intended, only 'pure logic' schema calculus operators are allowed in TCOZ. This is justified by the superior algebraic properties of the CSP operators.

When operations are combined using the concurrency primitives $\_ \parallel \_$ and $\_ ||| \_$, the designer is exposed to all the usual dangers of shared variable concurrency. The operation $OS_1 \parallel OS_2$, where OS_1 and OS_2 are operation schema, will synchronise on all state update events on variables in the respective delta-lists. Thus $OS_1 \parallel OS_2$ will have much the same process properties as $OS_1 \land OS_2$, with the exception that when the operations are inconsistent for a given initial state, the concurrent composition will deadlock while the logical composition with diverge. For example, consider

$$SQRT == [x, x' : \mathbb{N} \mid x > 0 \land x'^2 = x]$$
$$HALF == [x, x' : \mathbb{N} \mid x' * 2 = x].$$

The operations $SQRT$ and $HALF$ can agree only in the case where $x = 4$. When either of the operations is undefined (for example when $x = 2$ $SQRT$ is undefined) $SQRT \parallel HALF$ will diverge. When both are defined but in disagreement (for example when $x = 16$) $SQRT \parallel HALF$ will deadlock at some unspecified time. The process $SQRT \land HALF$ is just $[x, x' : \mathbb{N} \mid x = 4 \land x' = 2]$ and will never deadlock. The concurrent composition $OS_1 ||| OS_2$ is even less well behaved, every variable may be updated in any way allowed by either OS_1 or OS_2. Such a situation is likely to be very difficult to analyse. We strongly recommend that concurrent composition of operations be used sparingly, preferably only in cases where the operations have disjoint delta-lists. Shared data structures should only be utilised when properly protected by the object encapsulation mechanism.

10.1.3 Active and passive objects

The definition in a class of the distinguished process name MAIN indicates that the class is being defined as *active*. The MAIN process is used to determine the behaviour of objects of an active class after initialisation. Initialisation is treated in the usual way through the INIT schema. Active objects have their own thread of control and their mutable state attributes and operation definitions are fully encapsulated (update events are hidden). Distinct objects, even of the same class, share no data and can experience no shared variable interference. Other objects can neither reference an active object's state attributes nor invoke any of its local operations. Only local constants, such as the object identity attribute *self*, may be accessed by other classes. All dynamic interactions with an active object must take place through the CSP channel communication mechanism. Active objects are considered to have the syntactic properties of process identifiers and may be composed using CSP operators. The MAIN operation is optional in a class definition. If a class is defined without a MAIN process it is called a *passive* class. Passive objects are controlled by other objects in a system and their state and operations are fully available to the controlling object (unless explicitly hidden). The appearance of MAIN clearly distinguishes the definition of active objects and passive objects in a system.
Returning to the timed-collection example, the existence of environmental obligations and the need to purge stale elements means that the timed-collection class must have its own thread of control. Assuming that the class operations are defined as in Section 10.1.1, the timed-collection behaviour is defined by a MAIN process similar to the Timed CSP version presented in Section **??**.

$$\begin{aligned} \text{MAIN} \mathrel{\widehat=} \mu\, TC \bullet\ & [mems = \varnothing] \bullet Add;\ TC \ \Box \\ & [mems \neq \varnothing] \bullet ((Add \ \Box\ Delete) \triangleright \{t\}\ Purge);\ TC \end{aligned}$$

The most striking difference lies in the use of the operation schemas to subsume the role of the complex annotations present in the Timed CSP version. This represents a clearer and more structured presentation of the basic control logic. A second difference lies in the use of the state guard construct to distinguish between the empty and non-empty behaviours of the timed-collection, thus saving the need to define separate empty and non-empty processes.

10.1.4 Communication channels

The class state-schema convention is extended to allow the declaration of communication channels. If c is to be used as a communication channel by any of the operations of a class, then it must be declared in the state schema to be of type **chan**. Channels are type heterogeneous and may carry communications of any type.
This convention can be criticised on the grounds that it violates the strong-typing of Z

and denies the reader possibly instructive information. We believe that the ability to send many forms of data over a channel plays a vital role in reducing the complexity of class interfaces and can be justified conceptually by the observation that it is more instructive to group logically related communications (such as those pertaining to a particular protocol) than to group communications with identical type but logically unrelated function. In any case, all data communicated on channels remains strongly-typed.

Contrary to the conventions adopted for internal state variables, channels are viewed as global rather then as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects. In the situation of multiple instances of objects of the same class in a system, those objects will all share the same channel. For example, if $\mathcal{O}$ is a sequence of objects of class C with channel c, then in the process

$$\left|\left|\right|\right| i : \operatorname{dom} \mathcal{O} \bullet \mathcal{O}(i)$$

each of the objects $\mathcal{O}(i)$ communicates with the environment by sharing channel c with every other object. In the general case there is no way for the environment to know which of the objects it is communicating with when using channel c. If it is necessary to know which object the environment is communicating with, the object identity attribute *self* [37] can be included in the communication, for example

$$c.(self, message).$$

(This technique is used frequently in the lift case study.) The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby further enhancing the modularity of system specifications.

Consider once again the timed-collection example. Using the TCOZ conventions, the class state can be significantly simplified from the Timed Object-Z version. The sole remaining primary class attribute is the actual collection itself, none of the timing attributes are required. In addition to the list of *mems*, the state schema must declare channels *left* and *right*. These channels serve much the same role as the corresponding environment variables in the Timed Object-Z version, but here that role is better defined in terms of the CSP process model. The secondary attributes *oldest* and t remain useful in simplifying the operation definitions and are retained.

$$
\begin{array}{|l}
\hline
mems : \mathbb{F}(\mathbb{T} \times X) \\
left, right : \textbf{chan} \\
\Delta \\
(t, oldest) : \mathbb{T} \times X \\
\hline
mems \neq \varnothing \Rightarrow (t, oldest) \in mems \land t = min\, \mathrm{dom}\, mems \\
\hline
\end{array}
$$

10.1.5 The timed-collection

Bringing together the various aspects of the TCOZ timed-collection introduced above, we are able to present the entire class definition.

TimedCollection[X]

$$
\begin{array}{|l}
\hline
mems : \mathbb{P}(\mathbb{T} \times X) \\
left, right : \textbf{chan} \\
\Delta \\
(t, oldest) : \mathbb{T} \times X \\
\hline
mems \neq \varnothing \Rightarrow (t, oldest) \in mems \land t = min\, \mathrm{dom}\, mems \\
\hline
\end{array}
$$

INIT

$$
\begin{array}{|l}
\hline
mems = \varnothing \\
\hline
\end{array}
$$

Add$_0$

$$
\begin{array}{|l}
\hline
\Delta(mems) \\
e? : X \\
t_i? : \mathbb{T} \\
\hline
\delta = t_a \land mems' = ps(t_i? + t_a, mems) \cup \{(t_i?, e?)\} \\
\hline
\end{array}
$$

Delete$_0$

$$
\begin{array}{|l}
\hline
\Delta(mems) \\
t_i : \mathbb{T} \\
\hline
mems \neq \varnothing \land \delta = t_d \\
mems' = ps(t_i + t_d, (mems \setminus \{(t, oldest)\}) \\
\hline
\end{array}
$$

$Add \mathrel{\widehat=} [e : X;\ t_i : \mathbb{T}] \bullet left?e\,@t_i \rightarrow Add_0$

$Delete \mathrel{\widehat=} [t_i : \mathbb{T} \mid mems \neq \varnothing] \bullet right!oldest\,@t_i \rightarrow Delete_0$

Purge

$$
\begin{array}{|l}
\hline
\Delta(mems) \\
\hline
\delta = t_p \land mems' = ps(t + t_p, mems) \\
\hline
\end{array}
$$

$\textsc{Main} \mathrel{\widehat=} \mu\, TC \bullet [mems = \varnothing] \bullet Add;\ TC \ \Box$
$[mems \neq \varnothing] \bullet ((Add \ \Box\ Delete) \triangleright \{t\}\ Purge);\ TC$

This specification represents a more concise, flexible, and scalable treatment of both process and state than is possible in either Object-Z or Timed CSP. The structure of the process' internal state and communications interfaces are prominently documented. The structured schema based approach to describing state transitions,

supported as it is by the full power of the Z toolkit and the schema calculus, is better able to handle large and complex process state than the essentially *ad hoc* state annotation conventions of CSP. Making use of the Timed CSP process definition conventions removes the need to consider process control matters in operation schemas. There is a clear separation of process control and algorithmic matters which simplifies the description of both.

10.1.6 Composing classes

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.
Essentially, all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialisation schemas of inherited classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.
TCOZ extends Object-Z with two new class constructs, channels '**chan**' and main behaviour 'MAIN'. Channels are treated as normal state constant attributes, therefore, they are pooled into the derived classes. Channel renaming is the same as state attribute renaming. Since new classes will generally have new behaviours, the MAIN class definition is never inherited. As for all other class definitions, a class extension must include a MAIN definition if the class is to be active. The rules for (active and passive) class inheritance are:

- A new active class may be derived from an existing active class by defining a new MAIN process.
- A new active class may be derived from an existing passive class by defining a MAIN process.
- A new passive class can also be derived from an existing active class by not defining a new MAIN process.
- A new passive class can be derived from an existing passive class by not defining a MAIN process.

A composite object which contains active objects is also an active object (a MAIN definition must appear in the composite object class). Active objects are responsible for their own intialisation, so a composite object will often not require an explicit INIT schema. An the example of this is the Buffered-Consumer/producer class in Section 10.1.7. Since it has no passive internal state it requires no explicit initialisation. Another result of the encapsulation of local state in active objects is the inability of a composite class to refer to the local state attributes and operations of a component object. Only class constants such as the object identity *self* may be accessed.

10.1.7 A multi-threaded example

The timed-collection example made no use of the multi-threaded capabilities of the TCOZ notation. In this section, a specification of a standard buffered consumer/producer process is presented as a demonstration these aspects of TCOZ.
A simple consumer/producer process accepts inputs of type L on its *left* channel, calculates some function

$$f : L \rightarrow R$$

of the inputs, and outputs the result on its *right* channel.

ConProd

$left, right : \textbf{chan}$

Process

$i? : L$
$o! : R$

$\delta = t_p \land o! = f(i?)$

$\textsc{Main} \mathrel{\widehat=} \mu\, C \bullet [i : L] \bullet left?i \rightarrow Process;\ right!o \rightarrow C$

To ensure that all inputs are accepted and outputs are received, the process is buffered left and right with timed-collection processes. The buffered process consists of a left buffer, a right buffer, and an internal consumer/producer.
Correct hookup of the timed-collection buffers to the consumer/producer is achieved by *renaming* the various internal channels. The renaming convention is the same as for Object-Z and Timed CSP, that is $P[a/b]$ is P with all occurrences of b replaced by a. Intermediate channels ml and mr are introduced as the internal interfaces to the left and right buffers respectively. In order to retain a consumer/producer like interface, the *right* channel of the left buffer is renamed to ml, the *left* channel of the right buffer renamed to mr and the *left* and *right* channels of the internal consumer/producer are renamed to ml and mr respectively.
The internal interfaces are protected from environmental interference by *hiding* them. The hiding notation is the same as for both Object-Z and Timed CSP, that is $(P \setminus c)$ is P with c protected from external influence. In the case where P is process-like and c is a channel this has the important result of freeing communications on c from the requirement of synchronising with the environment. Thus communications on mr and ml occur as soon as the local processes are ready and cannot be blocked by any other entity.

BufConProd

$l : TimedCollection[L][ml/right]$
$r : TimedCollection[R][mr/left]$
$cp : ConProd[ml/left, mr/right]$

$\textsc{Main} \mathrel{\widehat=} (l \,|[\, ml \,]|\, cp \,|[\, mr \,]|\, r) \setminus ml, mr)$

The *BufConProd* class definition allows true multithreading of the two buffers and the consumer/producer. For example, the left buffer may be accepting a new item at the same time as the consumer/producer is processing another item, at the same time as the right buffer is releasing yet another item to the environment. This concurrency is coerced into smooth co-operation through the requirement for synchronisation between the processes when communication occurs on the internal channels ml and mr.
The addition of these CSP process structuring features represents a significant advance in the scope and size of systems which may be addressed by the Object-Z approach.

10.1.8 Sensors and actuators

Complementary to the synchronising CSP channel mechanism and inspired by control thoery, TCOZ also adopts a non-synchronising shared variable mechanism [79]. A declaration of the form $s : X$ **sensor** provides a channel-like interface for using the shared variable s as an input. A declaration of the form $s : X$ **actuator** provides a local-variable-like interface for using the shared variable s as an output. Sensors and actuators may appear either at the system boundary (usually describing how global analog quantities are sampled from, or generated by the digital subsystem) or else within the system (providing a convenient mechanism for describing local communications which do not require synchronisations). The shift from closed to open systems necessitates close attention to issues of control, an area where both Z and CSP are weak [131]. We believe that TCOZ with the **sensor** (and **actuator**) can be a good design language for smart sensor based hybrid systems.

10.1.9 Semantics of TCOZ

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [81, 97]. In brief, the semantic approach is to identify the notions of operation and process by providing a process interpretation of the Z operation schema construct. TCOZ differs from many other approaches to blending Object-Z with a process algebra in that it does not identify operations with events.

Instead an unspecified, fine-grained collection of state-update events is hypothesized. Operation schemas are modelled by the collection of those sequences of update events that achieve the state change described by the schema. This means that there is no semantic difference between a Z operation schema and a CSP process. It therefore makes sense to also identify their syntactic classes.
The process model used by TCOZ consists of sets of tuples consisting of: an *initial* state; a *trace* (a sequence of time stamped events, including update-events), a *refusal* (a record of what and when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is if a ✓ event occurs), then the state at the time of termination is called the *final* state.
The process model of an operation schema consists of all initial states and update traces (terminated with a ✓) such that the initial state and the final state satisfy the relation described by the schema. If no legal final state exists for a given initial state, the operation diverges immediately. An advantage of this semantics is that it allows CSP process refinement to agree with Z operation refinement.

10.1.10 Network topologies

The syntactic structure of the CSP synchronisation operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology [78]. For example, consider that processes A and B communicate privately through the interface ab, processes A and C communicate privately through the interface ac, and processes B and C communicate privately through the interface bc. This network topology of A, B and C may be described by

$$\Big|\Big|(A \xleftrightarrow{ab} B;\ B \xleftrightarrow{bc} C;\ C \xleftrightarrow{ca} A).$$

Other forms of lax usage allow network connections with multiple channels above the arrow, for example if processes D and F communicate privately through the channel/sensor-actuator df_1 and df_2, then

$$\Big|\Big|(D \xleftrightarrow{df_1, df_2} F).$$

10.2 Case Study: Sensor Based Smart Systems

The recent development on sensor-based context-aware, ubiquitous and mobile computing has added new complexity and created new challenges in system design techniques for modelling and reasoning complex smart systems. The objectives of specifying and designing complex sensor based smart system is to precisely capture communicating behaviors, sensor constraints, and real-time system properties in a highly abstract, modular and integrated model on which important properties can be verified. In particular, the high level design models for the smart systems need to capture not only concurrent interactions between various software control units and physical sensing devices, but also environmental and requirement constraints on sensors and sensor relations. For example, a smart meeting room may have a number of autonomous control units that control lighting/security/aircondition devices, and even mobile devices (mobile phones attached with a RFID tag may automatically turn into the silence mode when they enter the meeting room). This kind of application consists of a number of different sensors. The above example may consist of a motion sensor for detecting any occupants, a light sensor for detecting the current outdoor lighting intensity and RFID tracking sensor to identify all mobile devices/users attached with a RFID tag. It is important for a design model to precisely capture the requirement constraints on sensors, e.g. room sensor value for outdoor light intensity will never reach 100 percent illumination requirement for the room, and sensor relations, e.g. whenever a RFID tag (attached to mobile device or user) is tracked by the tracking unit, the motion sensor must detect occupants.

To facilitate design analysis and review process effectively, the design model should focus on appropriate abstraction level. Certain low level implementation details can be abstracted away. For example, in our experience [122] the communications between mobile device and RFID tracking unit may involve a Java program in the mobile device to send an active query to a server (and then wait for a notification if the location is changed) through BlueTooth technology. However, those implementation details are best hidden from the abstract design so that more important system properties can be clearly reasoned.

In this section, we demonstrate that a sensor based formalism can be succinctly applied to the design of smart space with multiple functionalities. Based on the formal model, essential properties within a particular domain or across different domains can also be verified.

10.2.1 Smart Space: intelligent control over lighting and mobile phones

The aims of our smart space design are to save electric energy and provide autonomous control over the receiving modes of mobile phones. In most meeting rooms

(or libaray/museum) today, lights are controlled manually. Electrical energy is wasted by lighting rooms that are not occupied and by not adjusting light levels relative to the daylight. The first component of our smart room is an intelligent light control unit. It can detect the occupation of the building through a motion sensor, turn on or turn off the lights automatically. It is able to tune illumination in the building according to the outside light level. When users leave a room (leaving it empty) and the detector senses no movement, the light control unit waits a pre-defined absence time and then turns off the light group. The second concern is related to the widespread use of mobile phones which makes voice communication available anytime, anywhere, but also raises many social problems such as phones ringing during meetings. Normally users often have to configure the settings of mobile phones according to their circumstances to avoid inappropriate usage. However, manual configuration causes frequent interactions with mobile phone, imposing significant user distractions. To address this issue, the second component of our smart system is designed to automatically detecting mobile phone/user's location (via RFID tracking system). For example, if a mobile user (wearing a RFID tag) is determined to be in the meeting room then a control message will be sent from the mobile tracking system to the mobile phone so that a silent mode will be automatically selected. The silence mode can be either Vibration or Voice-mail mode depending on individual user's profile which can be updated dynamically.

10.2.2 Sensor Based Modelling

Sensors, Sensor Constraints and Sensing Patterns

The standard TCOZ communications interface is the CSP channel, which represents a sequence of discrete synchronization between system and environment. One common model for hybrid smart system interfaces is the continuous, differentiable function. We adopt an approach by providing standardized mechanisms in TCOZ for converting from the discrete to the continuous and vice versa, thus granting TCOZ process classes to present a continuous-function interface to their environments. This allows subsystems specified in TCOZ to fit seamlessly into the overall design of hybrid smart systems.
For example, the motion detector for a room can be specified by a declaration of the form

$$motion : OccupyState\ \mathbf{sensor}, \quad \text{where} \quad OccupyState ::= Occupied \mid Empty$$

which declares *motion* to be a continuous-function interface with public type $\mathbb{R} \rightarrow OccupyState$.
Internally, *motion* takes the syntactic role of a CSP channel. The relationship between

the public continuous-function variable and the internal channel is that whenever a value v is communicated on the internal channel at a time t, that value must be equal to the value of the continuous function at that time, that is $motion(t) = v$.
The light group intensity can be specified by a declaration of the form

$$i : Illumination \, \textbf{actuator}, \quad \text{where} \quad Illumination ::= 1 \,..\, 100$$

This declaration also introduces i as a public continuous-function variable, but in this case the internal role is that of the local state variable. Thus i may be updated and appear in the delta list of operations and any other place where a local variable may appear.

Sensor Constraints and Sensing Patterns

Various sensor constraints can be generally captured in the form of

$$\forall\, t : \mathbb{T} \bullet C(sensor_a(t))$$

where C can be an invariant constraint on sensor (or well tested environmental assumption). Inter-sensor relationships can also be captured in a similar form of

$$\forall\, t : \mathbb{T} \bullet R(sensor_a(t), sensor_b(t))$$

where R can be a relational constraint for $sensor_a$ and $sensor_b$ in predicate form.
Smart process involving sensors can have various behavior patterns. For example,

- *Periodic Sensing* pattern can be specified by the following frame:

  $$\mu\, P \bullet [v : \mathbb{R}] sensor_a?v \rightarrow (P_1 \,\textsc{Deadline}\, 1 \bullet \textsc{WaitUntil}\, 1);\ P$$

  where the $sensor_a$ will sense any real number value in one minute interval (provided the sub process P_1 must terminate within one second. Patterns for sensors to periodically wake up can also be easily specified by this kind of frame.

- *Conditional sensing* pattern can be specified by the following frame:

  $$...[v : \mathbb{R} \mid c(v)] \bullet sensor_b?v \rightarrow P_2$$

  where $c(v)$ is a condition on v (e.g. $5 < v < 10$) that $sensor_b$ wants to detect.

There are obviously other sensing patterns, e.g. timeout sensing and interrupt sensing. We will demonstrate the various sensing design patterns in the following smart space modelling case study.

Model of the Smart Space

LightGroup

$i : Illumination$ **actuator**
$dimmer$: **chan**

$\text{MAIN} \mathrel{\widehat=} \mu\, N \bullet [n : Illumination] \bullet dimmer?n \rightarrow i := n;\ N$

Class *LightGroup* defines the data state and behaviors of the *lights*. The light level is represented as a state variable $i : Illumination$. *dimmer* is defined as a channel connecting the light group to the light control unit. A MAIN process indicates that *LightGroup* defines an active object, which has its own thread of control. Whenever a message is received on the channel *dimmer*, the light level is updated with the new light level. The light level update can occur repeatedly, due to the recursion.

LightControlUnit

$dimmer$: **chan**
$motion : OccupyState$ **sensor**
$odsensor : Illumination$ **sensor**
$absenT : \mathbb{T}$

$\forall\, t : \mathbb{T} \bullet odensor(t) < 100$

Adjust

$n? : Illumination$
$dim! : Illumination$

$dim! + n? = 100$

$Ready \mathrel{\widehat=} motion?Occupied \rightarrow On$
$Regular \mathrel{\widehat=} \mu\, R \bullet [n : Illumination] \bullet odsensor?n \rightarrow$
$\quad (Adjust \bullet \text{DEADLINE}\, 0.5 \bullet \text{WAITUNTIL}\, 1);\ dimmer!dim \rightarrow R$
$On \mathrel{\widehat=} Regular \bigtriangledown motion?Empty \rightarrow OnAgain$
$OnAgain \mathrel{\widehat=} (motion?Occupied \rightarrow On) \triangleright \{absenT\}\ Off$
$Off \mathrel{\widehat=} dimmer!0 \rightarrow Ready$
$\text{MAIN} \mathrel{\widehat=} Off$

The light control unit communicates with the light group by declaring channels with the same name, monitors the signals from its sensors and sends proper signals to the light group. The sensor *motion* detects movement in the building. The sensor *odsensor* monitors the light level outside. The class invariant

$$\forall\, t : \mathbb{T} \bullet odensor(t) < 100$$

is a sensor constraint which captures an environmental assumption (i.e. due to the building window size, outdoor light intensity can never reach 100 percent room Illumination requirement). The process *Regular* is periodically (one second) sensing out-door lighting and outputting a proper light level to the light group.
At start-up, the light control unit sets the light level to 0 by sending value 0 through channel *dimmer*. Once some movement is detected by the motion sensor ($motion?Occupied$), the light control unit starts to repeatedly read the outside light level ($odsensor?n$) and then adjust the light level accordingly. If the motion sensor detects no movement ($motion?Empty$), the light control unit waits a pre-defined absence time ($absenT$, captured by the timeout operator $\triangleright$) and then turns off the light. However, if some movement is detected before the absence time, the control unit continues adjusting the light level.

MobileTrackingUnit

$tracking : \mathbb{P}\, RFID\_tag$ **sensor**
$recorded : \mathbb{P}\, RFID\_tag$
$signal$: **chan**

INIT
$recorded = \varnothing$

Update
$\Delta(recorded)$
$ms? : \mathbb{P}\, RFID\_tag$

$recorded' = ms?$

$$\textsc{Main} \mathrel{\widehat=} \mu\, MU \bullet [ms : \mathbb{P}\, RFID\_tag \mid ms \neq recorded] \bullet tracking?ms \rightarrow (\Big|\Big|_{m:(ms-recorded)} signal!(m, Silence) \rightarrow \textsc{Skip} \parallel \Big|\Big|_{m:(recorded-ms)} signal!(m, Back) \rightarrow \textsc{Skip}); \; Update; \; MU$$

The mobile tracking unit monitors all the RFID tags within range through the sensor *tracking*. It records the set of RFID tags that have been previously detected and communicates with individual mobile phones through channel *signal*. Initially, no RFID tag is recorded. The operation *Update* replaces the recorded RFID tags by the newly detected ones. Whenever a new set of RFID tags has been detected (conditional sensing pattern), for every RFID tag that has not been detected previously, a control message is sent to the corresponding mobile phone so that a silent mode will be automatically selected. For every RFID tag that the tracking sensor has lost track of, a control message is sent to the corresponding mobile phone so that the mobile phone is set to its previous mode.

$$MobilePMode \mathrel{\widehat=} Vibration \mid Ring \mid RingAndVibration \mid VoiceMail$$

MobileP

$id : RFID\_tag$
$signal$: **chan**
$mode, pre\_mode, silence\_mode : MobilePMode$

ManualUpdate

$\Delta(mode)$
$m? : MobilePMode$

$mode' = m?$

ChangeSilenceProfile

$\Delta(silence\_mode)$
$m? : Vibration \mid VoiceMail$

$silence\_mode' = m?$

AutoSilence

$\Delta(mode, pre\_mode)$

$pre\_mode' = mode$
$mode' = silence\_model$

SwitchBack

$\Delta(mode)$

$mode' = pre\_mode$

$\textsc{Main} \mathrel{\widehat=} \mu H \bullet (ManualUpdate \Box ChangeSilenceProfile \Box$
$signal?(id, Silence) \rightarrow AutoSilence;\ signal?(id, Back) \rightarrow SwitchBack);\ H$

MobilePMode defines the different modes of mobile phones. A mobile phone is associated with the RFID tag. It also records its current mode (*mode*), its previous mode (*pre_mode*) and its silence profile (*silence_mode*, either *Vibration* or *VoiceMail*). The mode of the mobile phone can be manually updated by the operation *ManualUpdate*. The silence profile can be updated by the operation *ChangeSilenceProfile*. Once a signal is received through channel *signal* from the mobile tracking system with the right RFID tag, if the message content is *Silence*, the mobile phone is set to silence mode and its previous mode is recorded in the variable *pre_mode*. If the message content is *Back*, the mobile phone is set back to its previous mode.

MobilePs

$phones : \mathbb{P}\ MobileP$
$ids : \mathbb{P}\ RFID\_tag$

$ids = \{p : phones \mid p.id\}$
$\#phones = \#ids$ [unique association of RFID tag with phones]

$\textsc{Main} \mathrel{\widehat=} |||\ p : phones$

Class *MobilePs* defines a group of mobile phones. Its behavior is the interleaving of all the mobile phones.

SmartMeetingSpace

$lg : LightGroup$
$lcu : LightControlUnit$
$mps : MobilePs$
$mtu : MobileTrackingUnit$

$mtu.recorded \subseteq mps.ids$
$\forall\, t : \mathbb{T} \bullet mtu.tracking(t) \neq \varnothing \Rightarrow lcu.motion(t) = Occupied$

$\textsc{Main} \mathrel{\widehat=} \left|\left|\right|(mps \overset{signal}{\longleftrightarrow} mtu;\ lcu \overset{dimmer}{\longleftrightarrow} lg)$

The smart meeting space consists of a light group (lg), a light control unit (lcu), a set of mobile phones (mps) and a mobile tracking unit (mtu). The system is constrained such that the mobile tracking unit tracks only those RFID tags associated with the mobile phones. The class invariant

$$\forall\, t : \mathbb{T} \bullet mtu.tracking(t) \neq \varnothing \Rightarrow lcu.motion(t) = Occupied$$

captures a relationship between two different sensors, i.e., whenever a mobile user is tracked by the tracking unit, the motion sensor must detect some movement.

10.3 Reasoning about the smart space properties

One of the promising advantages of using a formal modelling approach is that verifications of the specified system properties by means of sound proofs can be made possible. We demonstrate the reasoning of two key properties, one within the mobile domain and one across the mobile and the light domain.

1. Intra-domain property
   All the mobile phones in a smart meeting space should be in silence mode, which ensures that there should be no unexpect interruptions during the meeting time. This property can be formally expressed from the model as:

$$SmartMeetingSpace :: \forall\, m : mps.phones \bullet \quad [\mathsf{P1}]$$
$$m.id \in mtu.recorded \Rightarrow m.mode = m.silence\_mode$$

2. Inter-domain property

When there are mobile users in a smart meeting space, its lights should be on. This property can be formally expressed from the model as:

$$SmartMeetingSpace :: mtu.recorded \neq \varnothing \Rightarrow lg.i > 0 \qquad \text{[P2]}$$

In order to prove *P1*, we first need to show that *once a mobile phone is switched into a silent model it will remain in its status until a switch back signal is received.* This can be formally stated as follows.

$$\begin{array}{l} SmartMeetingSpace :: \forall\, m : mps.phones \bullet \\ \quad (m.id \in mtu.recorded \\ \quad \wedge\ mtu.(signal!(m.id, Silence)) \in (RFID\_tag \times \{Silence\}) \\ \quad \wedge\ mtu.(signal!(m.id, Back)) \notin (RFID\_tag \times \{Back\})) \\ \quad \Rightarrow m.mode = m.silence\_mode \end{array} \qquad \text{[P1.1]}$$

Proofs of the property *P1.1* can be constructed as follows.

$$\dfrac{\begin{array}{l} MobileTrackingUnit :: STATE \vdash signal \in \mathbf{chan} \wedge \textsc{Main} \vdash \\ \quad (signal!(id, Silence) \in (RFID\_tag \times \{Silence\}) \wedge \\ \quad signal!(id, Back) \notin (RFID\_tag \times \{Back\})) \\ MobileP :: STATE \vdash signal \in \mathbf{chan} \\ SmartMeetingSpace :: STATE \vdash \\ \qquad (mtu \in MobileTrackingUnit \wedge mps \in MobilePs) \\ \quad \wedge\ \textsc{Main} \vdash mps \xleftrightarrow{signal} mtu \\ MobilePs :: STATE \vdash phones \in \mathbb{P}\, MobileP \\ \quad \wedge\ \textsc{Main} \vdash (p \in phones \wedge ||| \ p) \end{array}}{\begin{array}{l} MobileP :: \textsc{Main} \vdash (signal?(id, Silence) \in (RFID\_tag \times \{Silence\}) \\ \quad \wedge\ signal?(id, Back) \notin (RFID\_tag \times \{Back\})) \\ MobileP :: MAIN \vdash \\ \quad (signal?(id, Silence) \in (RFID\_tag \times \{Silence\}) \wedge \\ \quad signal?(id, Back) \notin (RFID\_tag \times \{Back\})) \\ \qquad \Rightarrow (mode' = silence\_mode \wedge pre\_model' = mode) \end{array}}\ [\ Channel\ ]$$

$$MobileP :: MAIN \vdash mode' = silence\_mode$$

Therefore, the property *P1* can be proved using induction on the behaviors of the active object *MobileTrackingUnit* as follows.

Proof: Initially, the recorded *RFID* set is empty, i.e.,

$$\begin{array}{l} MobileTrackingUnit :: INIT \vdash recorded = \varnothing \wedge \textsc{Main} \vdash (ms \neq \varnothing \\ \quad \Rightarrow \forall\, m \in ms \bullet signal!(m, Silence) \in (RFID\_tag \times \{Silence\}) \end{array}$$

If a new set of *RFID* has been detected by the sensor, i.e., '$ms - \varnothing \neq \varnothing$', silence mode requests through the channel *signal!.(m, Silent)* will be sent to each individual mobile devices in the new record set *ms*. From the lemma *P1.1* we know that this will further trigger the *AutoSilence* operation in each mobile device, which set the post-state of recorded mobile phones to silent mode. Thus property *P1* holds for the base case.
Assuming the property *P1* holds at any particular sequence of the execution in the *MobileTrackingUnit* object. Let us consider its next possible behavior i.e. if a different set of *RFID* is detected by the sensor.

$$\frac{\begin{array}{l} MobileTrackingUnit :: \textsc{Main} \vdash ms \neq recorded \\ \Rightarrow (\forall\, m \in (ms - recorded) \bullet \\ \qquad signal!(m, Silence) \in (RFID\_tag \times \{Silence\} \wedge \\ \quad \forall\, n \in (recorded - ms) \bullet \\ \qquad signal!(n, Back) \in (RFID\_tag \times \{Back\} \wedge \\ \quad (recorded - ms) \cap ms = \varnothing) \end{array}}{\begin{array}{l} SmartMeetingSpace :: \forall\, m : mps.phones \bullet \\ \quad m.id \in mtu.recorded \Rightarrow m.mode = m.silence\_mode \end{array}} [\ P1.1\ ]$$

Note that at any point, the set of detected mobile phones and the ones that left the meeting place are disjoint, i.e., '*(recorded-ms)*$\cap$ *ms*=$\varnothing$'. Thus the original set of mobile phones that is still in the meeting space will remain in their silent mode status, since no switch back signals could be sent. Hence, we conclude that the property *P1* holds for the next continuous behavior of the mobile tracking unit. According to the induction rule, *P1* is true for all cases in the model.
Similarly, the proof of the property *P2* can be constructed as follows.

$$\frac{\begin{array}{l} SmartMeetingSpace :: STATE \vdash (lg \in LightGroup \land \\ \quad lcu \in LightControlUnit \land mtu \in MobileTrackingUnit \land \\ \quad \forall\, t : \mathbb{T} \bullet mtu.tracking(t) \neq \varnothing \Rightarrow lcu.motion(t) = Occupied) \\ \quad \land \textsc{Main} \vdash lcu \overset{dimmer}{\longleftrightarrow} l \end{array}}{\begin{array}{l} LigtControlUnit :: STATE \vdash (dimmer \in \textbf{chan} \land \\ \quad \forall\, t : \mathbb{T} \bullet odensor(t) < 100) \\ \quad Ready \vdash (motion?Occupied \in (Occupied \mid Empty) \land \\ \quad\quad motion?Empty \notin (Occupied \mid Empty)) \\ \quad \Rightarrow odsensor?n \in Illumination \land \\ \quad Regular \vdash odsensor?n \in Illumination \Rightarrow dim + n = 100 \land \\ \quad\quad dimmer!dim \in Illumination \\ LightGroup :: STATE \vdash dimmer \in \textbf{chan} \end{array}}\ [\ Invariant\ ]$$

$$\frac{\begin{array}{l} LightGroup :: \textsc{Main} \vdash dimmer?dim \in Illumination \land dim > 0 \\ LightGroup :: \textsc{Main} \vdash dimmer?n \in Illumination \Rightarrow i := n \end{array}}{LightGroup :: \textsc{Main} \vdash i > 0}\ [\ Channel\ ]$$

Because the RFID tracking sensor and the motion detecting sensor are tightly coupled with each other, whenever a *RFID* is detected in the meeting space the motion sensor in the room will be set to the '*Occupied*' status, which will further cause the lights in the room to remain on.

10.3.1 Summary

In this section we have demonstrated that an integrated formalism, TCOZ with sensor and sensing pattern frameworks can be effectively applied to the design of smart space systems. The communicating behaviors, sensor constraints and real-time system properties can be captured in a highly abstract and modular style. Based on the formal model, essential properties within a particular domain or crossing different domains can also be verified. Future work may bring us to investigate complex sensor network systems, i.e., reconfiguration ability may be handled by introducing hybrid channels and hybrid sensors/actuators that can carry not only data but also process/programs.

10.3.2 Circus' Approach

10.3.3 Projections to TA

Chapter 11

OZTA

11.1 Introduction

Software system specification is an important activity in software engineering. Formalisms for specifying computer systems have been well researched. Logic-based formalisms, e.g. Z, have been popular in Europe. Graph-based formalisms, e.g. Automata, have been prevalent in North America. The design of complex systems requires techniques for capturing system functionalities and control behaviours. The system functionalities can be best captured in terms of operations and constraints — the ideal application for Z. The system control behaviours can be best captured in terms of visual flows between system functionalities — the ideal application for Automata. Furthermore, complex systems often have intricate system structures and real-time requirements. Object-Z [38, 107] is a structured extension to Z and can be supported by verification tools (e.g. [100, 76, 110, 126]). Timed Automata (TA) [2, **?**] is a real-time extension to Automata and can be effectively verified by model checkers (e.g. [9, 22, 111, 117]).
In our previous work [34], we investigated the projection techniques from the TCOZ [83] (extension to Object-Z) to TA and discussed the notion of timed patterns. In this paper, rather than taking the transformation point of view we propose to develop a novel integrated formal language which combines Object-Z with TA. An effective combination of Object-Z and TA can not only help Object-Z with real-time modeling capability but also help TA with enhanced structure and state modeling features. The result of such a combination can be a powerful unified method for designing complex computer systems. The challenge of achieving an effective combination of Object-Z and TA is to

- semantically and syntactically link the key language constructs so that the two notations can be used in a cohesive way;
- clearly separate system functionality aspects from time control behaviour pat-

terns so that separate tools can later be applied to check the related system properties;

- consistently unify the composition techniques from both Object-Z (class instantiation) and TA (automaton product) so that subsystem models can be easily and meaningfully composed;
- systematically develop the communication mechanisms so that various concurrent interactions between system components can be precisely captured.

In the remaining sections of this paper we will demonstrate how Object-Z and TA can be effectively combined.

11.2 Object-Z and Timed Automata

In this section, brief introductions to Object-Z and TA are presented together with motivating examples.

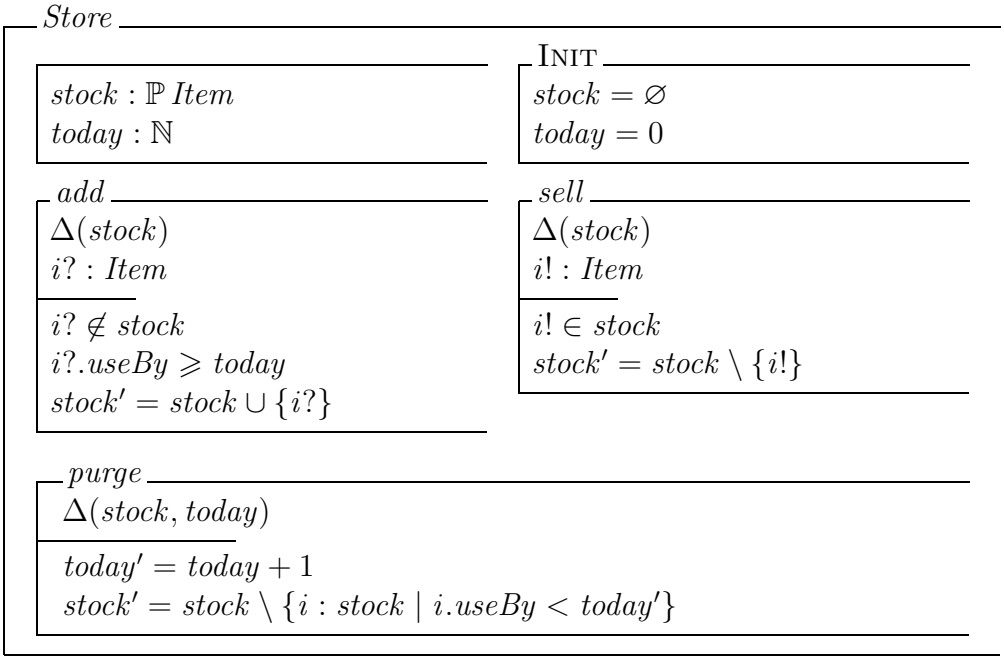
Stock Control System in Object-Z

In a simple stock-control system, the store stocks items which each have a fixed use-by date. An item can be added to the store's stock, but only if the use-by date of the added item is today's date or later. Any item can be sold by the store. At the beginning of each day those items whose use-by date is less than the current date are removed (i.e. purged) from the store.
To specify this system in Object-Z, first we specify an item as an object of the class *Item*:

Item

$$useBy : \mathbb{N}$$

Effectively, at this level of abstraction the only important thing about an item is its (fixed) use-by date.
The stock control system is specified by the class *Store*:



The semantics of Object-Z can be seen as a state transition system. For example, given a particular *Store* object state

$$\sigma = \{(store, \{item_a, item_b\}), (today, 20)\},$$

if operation *add* is then performed with a new input item $item_c$, the new object state would be

$$\sigma = \{(store, \{item_a, item_b, item_c\}), (today, 20)\}.$$

Notice that although there is an attribute *today* in this class and this attribute is incremented whenever the *purge* operation takes place, no notion of the progressive passing of time is captured by this specification. Conceptually, we think of the purge operation as taking place once a day, but this is not captured explicitly. Furthermore, in standard Object-Z the operations are assumed to be atomic, so there is no direct way of capturing the idea that an operation may take a specific time to complete.

Timed Automata

Timed Automata (TA) are finite state machines with clocks. It was introduced as a formal notation to model the behavior of real-time systems. Its definition provides a general way to annotate state-transition graphs with timing constraints using finitely many real-valued clock variables. Another interesting aspect of TA is that there exist

model checking methods for temporal logics with quantitative temporal operators which are directly applied to TA. Thus a variety of tools are available for specification and verification of real-time system modeled in TA.
In this paper, we follow the definitions given in [2]. Formally, for a set X of clock variables, the set $\Phi(X)$ of clock constraints φ is defined by the following grammar:

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2$$

where x is a clock in X and c is a constant in $\mathbb{R}$.
A clock interpretation ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to the set of nonnegative reals. We say that a clock interpretation ν for X satisfies a clock constraint φ over X iff φ evaluates to true according to the values given by ν. For $\delta \in \mathbb{R}, \nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$, and agrees with ν over the rest of the clocks.
A timed automaton A is a tuple $(S, S_0, \Sigma, X, I, E)$, where S is a finite set of states/locations; S_0, a subset of S, is a set of initial states; Σ is a set of actions/events; X is a finite set of clocks; I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$; E, a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, \lambda, \delta \rangle$ represents a transition from state s to state s' on input symbol a. The set λ gives the clocks to be reset with this transition, and δ is a clock constraint over X that specifies when the switch is enabled.
An example of a timed automaton is shown in Figure 11.1, a gate in a security system. The gate has three states: *closed*, *open* and *opened* (we assume the gate slams shut instantly when directed, so there is no *close* state). State *closed* is the initial state, as indicated by there being an action into *closed* that emanates from no state. Through action *open-s*, (i.e. open start) the gate starts the operation of opening which it completes within 2 time units. When the opening operation is completed, the action *open-e* (i.e. open end) occurs and the gate becomes opened. Through action *close* the gate closes instantly when exactly 10 time units have elapsed since the gate was opened.

11.3 Combining Object-Z and TA

In this section, the semantic and syntactic issues on integrating Object-Z and TA are discussed and a combined notation is proposed.
To illustrate how Object-Z and TA can be effectively integrated, consider the simple stock-control system we met in the last Section. What we want is to integrate into

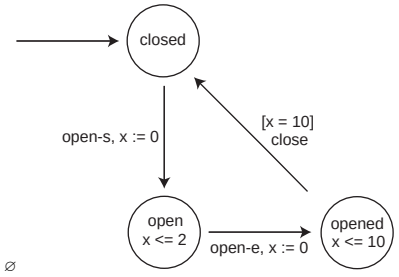


Figure 11.1: a gate

this specification a notion of the sequential passing of time.
Suppose time is a positive real number measured in days starting at 0 (so, for example, 1.5 is halfway through the second day). The use-by date associated with an item is a positive integer denoting the day by which the item must be sold or else purged from the store (e.g. a use-by date of 3 means that if the item is not sold on or before day 3, at the start of day 4 it is purged).
We shall suppose it takes at most Ta time units to add an item to the stock, at most Ts time units to sell an item in stock, and more than $Tp1$ but less than $Tp2$ time units ($Tp1 < Tp2$) to purge the stock at the beginning of the day, where each of Ta, Ts and $Tp2$ is much less than 1. Furthermore, the addition of any item to the stock or the selling of any item in stock must be started and completed within the same day. In our model, operations of the store will be disjoint, i.e. time-wise they do not overlap.
The store with this timing information incorporated is specified by adding a Timed Automaton to the class *Store* to get the class *TimedStore*:

TimedStore

$stock : \mathbb{P}\, Item$
$today : \mathbb{N}$

INIT
$stock = \varnothing$
$today = 0$

add
$\Delta(stock)$
$i? : Item$

$i? \notin stock$
$i?.useBy \geqslant today$
$stock' = stock \cup \{i?\}$

sell
$\Delta(stock)$
$i! : Item$

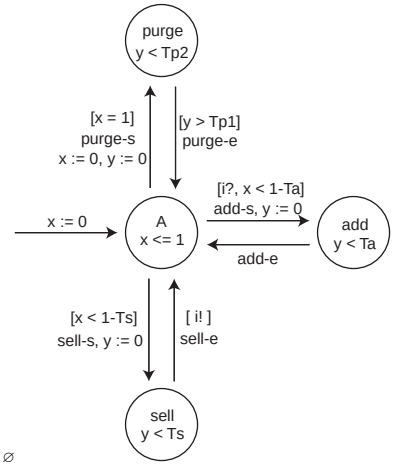
$i! \in stock$
$stock' = stock \setminus \{i!\}$

purge
$\Delta(stock, today)$

$today' = today + 1$
$stock' = stock \setminus \{i : stock \mid i.useBy < today'\}$

$Ta, Ts, Tp1, Tp2 : \mathbb{R}$

$x, y :$ **clock**



Consider the *TimedStore* class in detail. The top part of the class box is the standard Object-Z specification we met in the last section and contains no timing information. The bottom part of the class box contains a declaration of the timing constants and

the names of the clocks (in this case there are two clocks, x and y) as well as the associated automaton. Declaration x : **clock** means that x plays a dual role: it identifies (i.e. names) a clock and also records the time showing on the clock, i.e. it is a variable that takes positive real number values. In fact, as we shall see, the value of the clock x in this specification always lies between 0 and 1 inclusive and denotes the time that has passed in the current day. The clock y is used to ensure that the operations are completed within the specified time. It is assumed both clocks progress at the same rate, i.e. the passage of time is universally uniform.

The locations of the automata represent the various situations in which the store can find itself. A location, together with the switches to and from that location, specifies the timing limits (if any) for the corresponding situation. For each of the three operations specified in the Object-Z part there is a similarly-labeled location to capture the situation when the store is undergoing this operation; the store can undergo this operation only when in the corresponding location. The other location, A, represents the situation when the store is idle and no operation is being performed.

To illustrate the switches, consider those between locations A and *add*. The switch from A to *add* is labeled *add-s* (i.e. add start), while the switch from *add* to A is labeled *add-e* (i.e. add end). The expression in square brackets, i.e. $[i?, x < 1 - Ta]$ in the case of the switch labeled *add-s*, captures the requirements that must be met if the switch is to take place, i.e. the input item $i?$ (as defined in the Object-Z operation *add*) must be supplied, and in addition the time as recorded by the clock x must be less than $1 - Ta$ (so that the operation, which can take up to Ta time units to occur, can be completed within the same day). In addition, the precondition of the Object-Z operation *add* must hold for the *add-s* switch to occur. As the precondition of an operation must always hold before the switch to the place labeled by that operation's name can occur, this precondition is always implicitly conjoined with any specific additional requirements within the square brackets. When the switch from A to *add* occurs, the clock y is reset to 0; annotating the location *add* with the condition $y < Ta$ ensures that this location is exited within time duration Ta, as required.

For the operation *sell*, the supply of the output item $i!$ is a requirement that must be met for the switch *sell-e* to occur after the completion of the operation. Compare this with the *add* operation where the input $i?$ was required for the switch starting the operation to occur.

Looking now at the switch *purge-s*, this switch can occur only when x is 1. Furthermore, it must occur at this time because of the time restriction placed on location A. This ensures that the purge operation occurs precisely once a day (starting at the end of each day and the beginning of the next). When the switch does occur, the clock x is reset to 0 (ensuring that x always lies between 0 and 1 and hence denotes the time that has passed in the current day).

Location A is the automaton's initial location. The understanding is that the initial

conditions as specified by the INIT schema must hold when the automaton is started in location A, and at the same time the clock x is set to 0 (the initial value of the clock y can be arbitrary and so is not specified).
The fact that the 'start' switches associated with each operation emanate from location A and the 'end' switches each return to A, ensures that the operations *add*, *sell* and *purge* do not overlap time-wise.
Note that the naming of switches can be systematic, e.g., a switch pointing to an operation state can be labeled with the operation name follow by 's' (for start). If a switch is pointing from an operation state to an idle (control) state, then it can be labeled with the operation name follow by 'e' (for end).

Inheritance

Inheritance is a mechanism for incremental specification and reuse, whereby new classes may be derived from an existing class. Object-Z inheritance has a similar style as the Z schema inclusion. We propose that the control behaviour (expressed by the TA) can also be inherited and extended in a simple way. Consider a system *TimedStoreP* which has the same *sell* and *purge* functionalities with *TimedStore* except the *add* operation: only items with their expiration date at least 3 days ahead of the current day can be added into the store and the *add* operation takes less than a half of the *Ta* time units to finish. In addition, The system is able to identify the set *pItems* (promotion items) of items which have only two days left before their expiration. An extra operation *pSell* (promotion sell), which takes at most *Tpm* time units to execute, can sell a subset of these promotion items. The class *TimedStoreP* can be defined by inheriting the class *TimedStore*. For the behaviour (automaton) part, the *add* location refers to the redefined *add* operation and its local invariant changes to $y < Ta/2$ and enabling condition changes to $x < 1 - Ta/2$. And a new location *pSell* is introduced and is connected (by new switches) with the control (idle) location A from *TimedStore* The other locations and their connections remain unchanged as follows:

TimedStoreP

$TimedStore[\textbf{redef}\ add]$

Δ
$pItems : \mathbb{P}\ Items$

$pItems =$
$\quad \{i : store \mid$
$\quad\quad i.useBy < today + 2\}$

pSell

$\Delta(stock)$
$is! : \mathbb{P}_1\ pItems$

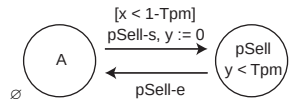
$stock' = stock \setminus is!$

add

$\Delta(stock)$
$i? : Item$

$i? \notin stock$
$i?.useBy \geqslant today + 3$
$stock' = stock \cup \{i?\}$

$Tpm : \mathbb{R}$



If we expand the inheritance, then *TimedStoreP* becomes:

TimedStoreP

$stock : \mathbb{P}\, Item$
$today : \mathbb{N}$
Δ
$pItems : \mathbb{P}\, Items$

$pItems = \{i : store \mid i.useBy < today + 2\}$

pSell

$\Delta(stock)$
$is! : \mathbb{P}_1\, pItems$

$stock' = stock \setminus is!$

INIT

[unchanged]

add

$\Delta(stock)$
$i? : Item$

$i? \notin stock$
$i?.useBy \geqslant today + 3$
$stock' = stock \cup \{i?\}$

sell

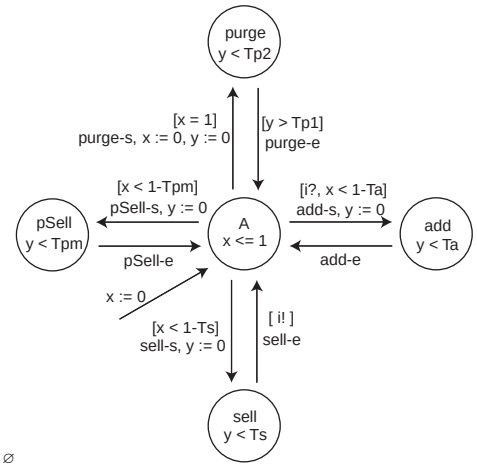
[unchanged]

purge

[unchanged]

$Ta, Ts, Tp1, Tp2, Tpm : \mathbb{R}$

x, y : **clock**



Note that *pItems* is modelled as a secondary attribute whose value is subject to change with each operation (implicitly it is included in every operation's Δ list).
For multiple inheritance cases, the rules are that all similarly named locations (and switches) are merged, with all corresponding invariants and conditions conjoined.

11.4 Composition and Communication

In this section, various composition and communication aspects of the combined language are discussed and synchronized communication links are systematically introduced.

Independent stores

Consider now a system consisting of two stores operating independently. This system is specified by the class *TwoIndStores*:

TwoIndStores

$s1, s2 : TimedStore$

$s1 \parallel s2$

The timed automaton of this class is simply the product [2] of the automata for the two stores $s1$ and $s2$. The timed automaton for $s1$ is just the automaton of the *TimedStore* class but with the label of each location, the label of each switch, and the names of the clocks distinguished by an '$s1$.' prefix, as illustrated in Figure 11.2. Notice that the input/output variables are not prefixed.

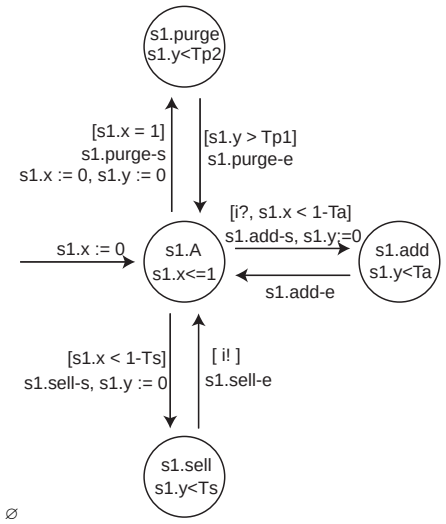


Figure 11.2: the automaton $s1$

The timed automaton for $s2$ is labeled similarly. The $s1 \parallel s2$ notation in the class *TwoIndStores* denotes the product of the associated automata. The implication here is that the two stores are not only completely independent, but operations in different stores can be executed concurrently. Indeed, when an object of the class *TwoIndStores* is instantiated, the two store objects start at the same time in their A position with $s1.x$ and $s2.x$ set to 0 synchronously. As time passes at the same rate for all clocks, both stores will always synchronise on the start of their respective *purge* operations, namely, at the start of the next day, but apart from that they run completely independently.
The two-stores example can be generalised to a collection of independent stores, as specified by the class *CollnIndStores*. In this class the expression $\parallel s : stores$ denotes the timed automata product $(s_1 \parallel s_2 \parallel \cdots)$ where the set *stores* is $\{s_1, s_2, \cdots\}$.

CollnIndStores

$stores : \mathbb{P}\, Store$

$\parallel s : stores$

Transferring between stores

Consider now a system consisting of two stores, where each item sold by the first store is added (i.e. transferred) to the second. Effectively, the first store sells items only to the second store. A specification of this system is given by the class *TransStores*:

TransStores

$s1, s2 : TimedStore$

$(s1 \parallel s2) \bullet \textbf{sync}\ \{s1.sell\text{-}e \leftrightarrow s2.add\text{-}s\}$

The **sync** clause indicates that the two switches labeled $s1.sell\text{-}e$ and $s2.add\text{-}s$ are to be treated as if these labels were identical, i.e. the automata must synchronize on these switches. As part of this synchronization, as the output $i!$ and the input $i?$ have the same base-name they are identified and hidden (just as is the case for the Object-Z parallel operator, i.e. they specify internal communication rather than communication with the environment). Apart from this synchronization, the product of the two timed automata effectively ensures that the two automata operate independently and concurrently.

Now consider a system like *TransStores* where again each item sold by the first store is added (i.e. transferred) to the second. However, an item from the environment may also be added to the second store, i.e. not all items added to the second store are necessarily transferring from the first. This system is specified in the class *Alt*1*TransStores*:

*Alt*1*TransStores*

$s1, s2 : TimedStore$

$(s1 \parallel s2) \bullet \textbf{sync}\ \{s1.\textit{sell-e} \rightarrow s2.\textit{add-s}\}$

The implication here is that whenever the switch $s1.\textit{sell-e}$ is taken then there must be synchronization with the switch $s2.\textit{add-s}$. However, the switch $s2.\textit{add-s}$ can occur independent of (i.e. without synchronizing with) the switch $s1.\textit{sell-e}$. With this notation, notice that the synchronization $\textbf{sync}\ \{s1.\textit{sell-e} \leftrightarrow s2.\textit{add-s}\}$ in the *TransStores* class could have been alternatively (but less elegantly) expressed as $\textbf{sync}\ \{s1.\textit{sell-e} \rightarrow s2.\textit{add-s}, s2.\textit{add-s} \rightarrow s1.\textit{sell-e}\}$.
Now consider the situation as before where an item sold by the first store can be transferred to the second, but in addition not only can an item from the environment be directly added to the second store, (i.e. not all items added to the second store are necessarily transferring from the first) but also an item sold by the first store can be passed to the environment (i.e. not all items sold by the first store are necessarily transferred to the second). This system is specified in the class *Alt*2*TransStores*:

*Alt*2*TransStores*

$s1, s2 : TimedStore$

$(s1 \parallel s2) \bullet \textbf{sync}\ \{s1.\textit{sell-e} \nleftrightarrow s2.\textit{add-s}\}$

The implication here is that when any of the switches $s1.\textit{sell-e}$ or $s2.\textit{add-s}$ is taken there may or may not (the choice is non-deterministic) be synchronization with the switch $s2.\textit{add-s}$ or $s1.\textit{sell-e}$ respectively.
The examples involving two stores given so far in this section can be generalised to a collection of stores. Consider a system consisting of a collection of stores where an item from the environment can be added to any store, an item sold by any store can be passed back to the environment, and given any two stores in the collection, an item sold by the first store can be added (transferred) to the second. Such a system is specified in the class *ColInTransStores*:

CollnTransStores

$stores : \mathbb{P}\ TimedStore$

$(\|\ s : stores) \bullet \mathbf{sync}\ \{s1, s2 : stores \mid s1 \neq s2 \bullet$
$s1.sell\text{-}e \nleftrightarrow s2.add\text{-}s\}$

More on synchronization

To further illustrate synchronization in timed automata, consider the three timed automata U, V and W illustrated in Figure 11.3.

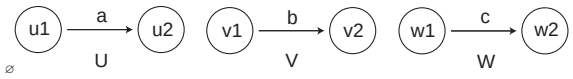


Figure 11.3: timed automata U, V and W

The timed automaton $(U \parallel V \parallel W) \bullet \mathbf{sync}\ \{a \leftrightarrow b\}$ is behaviorally equivalent to the product $U1 \parallel V1 \parallel W1$ of the timed automata $U1$, $V1$ and $W1$ illustrated in Figure 11.4. In this case the switches labeled a and b have been re-named to a common label d. As these labels are the same, the product automaton will synchronize on these switches. Consequently, the switch from location $u1$ to location $u2$ in $U1$ is always synchronized with the switch from location $v1$ to location $v2$ in $V1$, and conversely.

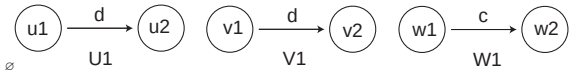


Figure 11.4: timed automata $U1$, $V1$ and $W1$

The timed automaton

$$(U \parallel V \parallel W) \bullet \mathbf{sync}\ \{a \leftrightarrow b, a \leftrightarrow c, b \leftrightarrow c\}$$

is behaviorally equivalent to the product $U2 \parallel V2 \parallel W2$ of the 3 automata $U2$, $V2$ and $W2$ illustrated in Figure 11.5. In this case the switch from location $u1$ to $u2$ in $U2$ must synchronize with either the switch from location $v1$ to $v2$ in $V2$ or from $w1$

to $w2$ in $W2$; the switch from location $v1$ to $v2$ in $V2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $w1$ to $w2$ in $W2$; and the switch from location $w1$ to $w2$ in $W2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $v1$ to $v2$ in $V2$.

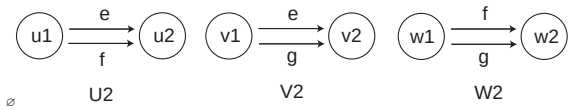


Figure 11.5: timed automata $U2$, $V2$ and $W2$

Compare this to the automaton

$$(U \parallel V \parallel W) \bullet \textbf{sync}\ \{a \leftrightarrow b \leftrightarrow c\}.$$

This automaton is behaviorally equivalent to the product $U3 \parallel V3 \parallel W3$ of the three automata $U3$, $V3$ and $W3$ illustrated in Figure 11.6. In this case the three switches from location $u1$ to $u2$ in $U3$, from location $v1$ to $v2$ in $V3$ and from location $w1$ to $w2$ in $W3$ must synchronize.

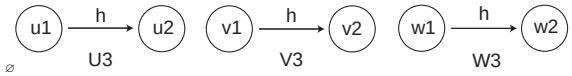


Figure 11.6: timed automata $U3$, $V3$ and $W3$

The timed automaton $(U \parallel V) \bullet \textbf{sync}\ \{a \rightarrow b\}$ is behaviorally equivalent to the product $U4 \parallel V4$ of the timed automata $U4$ and $V4$ illustrated in Figure 11.7. In $V4$ a switch labeled a is added to duplicate the switch labeled b. As the switch in automaton $U4$ is also labeled a, this ensures that the product automaton will synchronize on these two switches. Consequently, the switch from location $u1$ to location $u2$ in $U4$ is always synchronized with a switch from location $v1$ to location $v2$ in $V4$, but not conversely. The transition from location $v1$ to location $v2$ can use the switch labeled b in which case no synchronization takes place.
The timed automaton $(U \parallel V) \bullet \textbf{sync}\ \{a \nleftrightarrow b\}$ is behaviorally equivalent to the product $U5 \parallel V5$ of the timed automata $U5$ and $V5$ illustrated in Figure 11.8. In this case both of the switches labeled a and b are duplicated and a common name, d, is assigned to these new switches. This ensures that the product automaton will synchronize on these two switches. Consequently, a transition from location $u1$ to location $u2$ in $U5$ can synchronize with a transition from location $v1$ to location $v2$ in

Figure 11.7: timed automata $U4$ and $V4$

Figure 11.8: timed automata $U5$ and $V5$

$V5$ if the switch labeled d is used. However, a transition from location $u1$ to location $u2$ in $U5$ could use switch a, or a transition from location $v1$ to location $v2$ in $V5$ could use switch b; in either case no synchronization takes place.

11.5 Semantics

In this section we present a formal description of the operational behavior of this integrated language. The fundamental semantic links between Object-Z and TA are:

- Object-Z operations are identified as states in Timed Automata.
- Pre/Post-conditions of an Object-Z operation are identified to TA transition conditions.

The key novel idea of integrating the Object-Z semantics and TA semantics is to embed object state updates (of Object-Z) into the action transition semantics of TA. To facilitate the description of dynamic behaviors of a system, we introduce a set of locations A, called control locations, to coordinate the location switches from one Object-Z operation to another. Each location of a timed automaton specified in a class must be either a control location or an Object-Z operation location. A class has an Object-Z part *OZDefinition* which obeys the conventional definition [107]. *OZop* represents the set of Object-Z operations defined in the class. The original Object-Z operation operators: parallel composition, nondeterministic choice, sequential composition are replaced by *TADefinition*, which is a timed automaton: $\mathbb{S}_{\mathbb{OZTA}}$ is a tuple $(S, S_0, \Sigma, X, I, E)$, where

- S is a union of A and Op, in which A is a finite set of control (idle) states and Op is a finite set of operation states correspond to the Object-Z operations
- S_0, a subset of S, is a set of initial locations

- Σ is a set of labels
- X is a finite set of clocks
- I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$
- E, a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, r, \varphi \rangle$ represents a transition from location s to location s' on input symbol a. The set r gives the clocks to be reset with this transition, and φ is a clock constraint over X that specifies when the switch is enabled.

Operational Semantics

In this subsection, we present a timed transition system $\mathbb{S}_{\mathbb{OZTA}}$ to represent operational semantic models for this integrated language. Before we start to define the operational semantics, we need some definitions for the validity of Object-Z and TA expressions. The fact that a state guard G is valid under the semantic function $\sigma : Var \nrightarrow Value$ is denoted by the following notation:

$$\sigma \vDash G$$

which reads that G is valid under the semantic function σ. The fact that an operation Op is valid under the semantic functions σ_1, σ_2 is denoted by

$$\sigma_1, \sigma_2 \vDash Op$$

For example, in the context of the Store system,

$$\{(store, \{item_a, item_b\}), (today, 20)\},$$
$$\{(store, \{item_a, item_b, item_c\}), (today, 20)\} \vDash add[i? \mapsto item_c]$$

To keep track of the changes of clock values, we use functions known as clock assignments mapping X to the non-negative reals R_+. Let u, v denotes such functions, and use $u \vDash \varphi$ to mean that the clock values denoted by u satisfy the guard φ. For $d \in R_+$, let $u + d$ denote the clock assignment that maps all $x \in X$ to $u(x) + d$, and for $r \subseteq X$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and agree with u for the other clocks in $X \backslash r$.
To facilitate the description of operational semantics, let

$$OP : Location \nrightarrow OZop$$

denote the association between TA locations to Object-Z operations.
The operational semantics of this integrated language is an extension of TA transi-

tion semantics coupled with object states. The timed state transition system $\mathbb{S}_{\mathbb{OZTA}}$ consists of states which are tuples $\langle l, u, \sigma, \sigma_1 \rangle$ and state transitions are defined by the rules:

$$R_1 : \frac{l \xrightarrow{a,\varphi,r} l' \;\; \sigma,\sigma_1 \vDash OP(l) \;\; \sigma_1,\sigma_2 \vDash OP(l') \;\; u \vDash \varphi \;\; u' = [r \mapsto 0]u \;\; u' \vDash I(l') \;\; l,l' \in Op}{\langle l,u,\sigma,\sigma_1 \rangle \xrightarrow{a} \langle l',u',\sigma_1,\sigma_2 \rangle}$$

R_1 is an action transition from one operation (location) state l to another operation state l' where the post object state of l must be the same as the pre object state of l', the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_2 : \frac{\sigma_1,\sigma_2 \vDash OP(l) \;\; u \vDash I(l) \;\; u+d \vDash I(l) \;\; d \in R_+ \;\; l \in Op}{\langle l,u,\sigma_1,\sigma_2 \rangle \xrightarrow{d} \langle l,u+d,\sigma_1,\sigma_2 \rangle}$$

R_2 is a delay transition in a certain operation state where only time is progressed.

$$R_3 : \frac{l \xrightarrow{a,\varphi,r} l' \;\; u \vDash \varphi \;\; u' = [r \mapsto 0]u \;\; u' \vDash I(l') \;\; l \in A \;\; l' \in A}{\langle l,u,\sigma,\sigma \rangle \xrightarrow{a} \langle l',u',\sigma,\sigma \rangle}$$

R_3 is an transition from one control (location) state l to another control state l' where object states remain the same.

$$R_4 : \frac{u \vDash I(l) \;\; u+d \vDash I(l) \;\; d \in R_+ \;\; l \in A}{\langle l,u,\sigma,\sigma \rangle \xrightarrow{d} \langle l,u+d,\sigma,\sigma \rangle}$$

R_4 is a delay transition in a control state where time is progressed.

$$R_5 : \frac{l \xrightarrow{a,\varphi,r} l' \;\; \sigma_1,\sigma_2 \vDash OP(l) \;\; u \vDash \varphi \;\; u' = [r \mapsto 0]u \;\; u' \vDash I(l') \;\; l \in Op \;\; l' \in A}{\langle l,u,\sigma_1,\sigma_2 \rangle \xrightarrow{a} \langle l',u',\sigma_2,\sigma_2 \rangle}$$

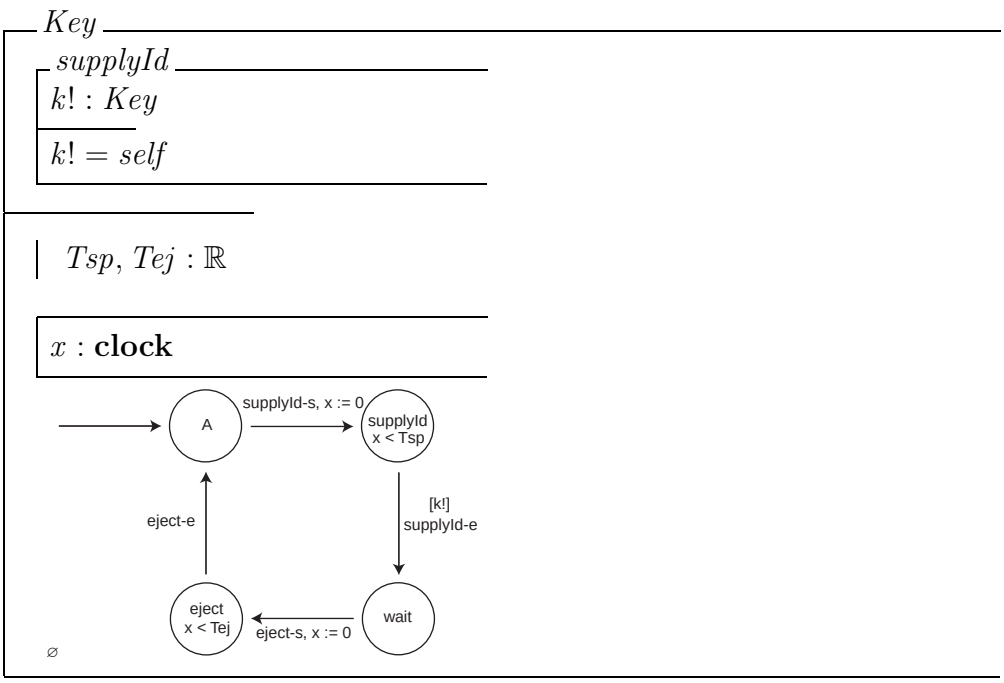
R_5 is an action transition from one operation (location) state l to a control state l' where the post object state of l must be the same as the object state of l', the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_6 : \frac{l \xrightarrow{a,\varphi,r} l' \;\; \sigma,\sigma_1 \vDash OP(l') \;\; u \vDash \varphi \;\; u' = [r \mapsto 0]u \;\; u' \vDash I(l') \;\; l \in A \;\; l' \in Op}{\langle l,u,\sigma,\sigma \rangle \xrightarrow{a} \langle l',u',\sigma,\sigma_1 \rangle}$$

R_6 is the inverse of R_5.
These rules define six types of transitions in $\mathbb{S}_{\mathbb{OZTA}}$. These rules are applied to a single timed transition system. A complex system can be described as a product of interacting timed transition systems. The communications between two transition systems are obtained by synchronizing the transition with identical labels.

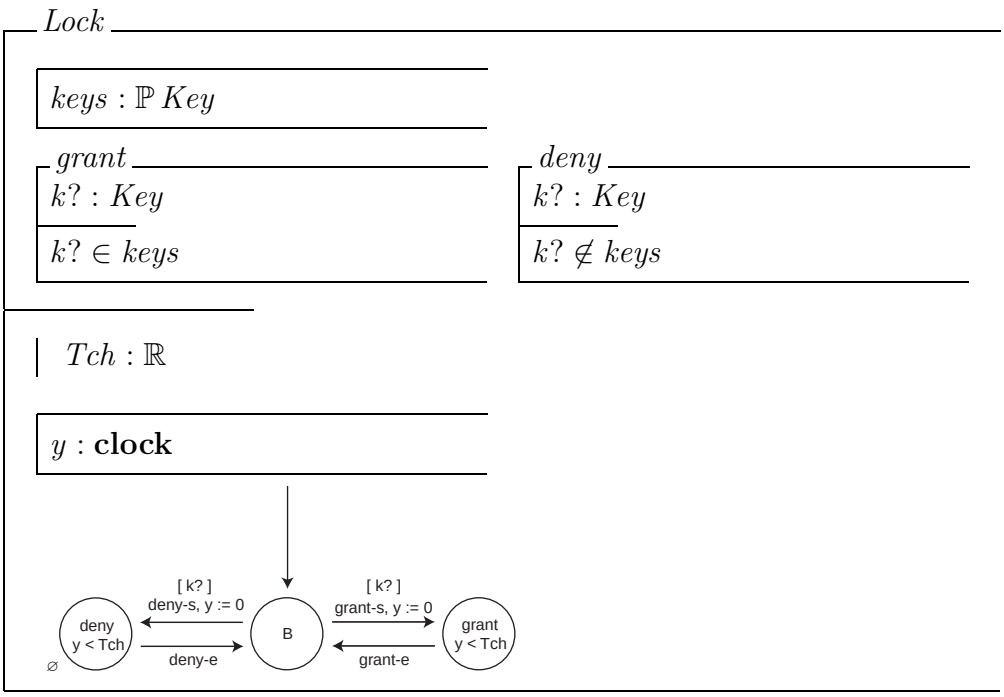
11.6 A Case Study

As an illustration of how Object-Z and TA can be successfully integrated in practice, we present here a case study of an electronic key system.
A room can be accessed through a sliding door. To open the sliding door, an electronic key is inserted into the door's electronic lock. The identity of the key (as encoded as part of the key) is passed to the lock that then checks to see if the key has permission to access the room. When access permission has been checked the key is ejected from the lock. If the key has access permission the door is opened (or remains open); otherwise the door is closed (or remains closed).
We shall suppose that it takes less than Tsp time units from the time the key is inserted in the door's lock for the key to supply its identity to the lock, less than Tch time units for the lock to check if the key has permission to access the room, less than Tej time units for the key to be ejected from the lock, and less than Top time units for the door to satisfy an 'open' request. Also, if the door has been open Tto time units since the last 'open' request, a time-out occurs and the door is closed. It takes less than Tcl time units for the door to satisfy a 'close' request.
A key is specified by the class Key:

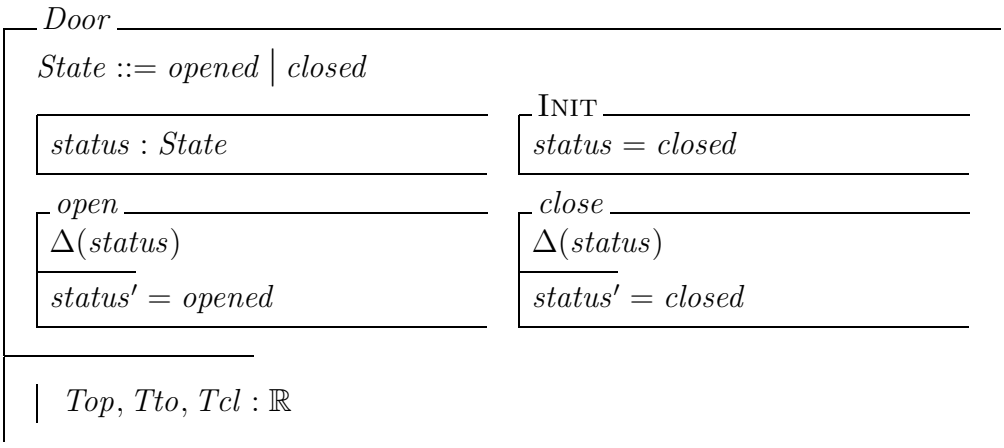


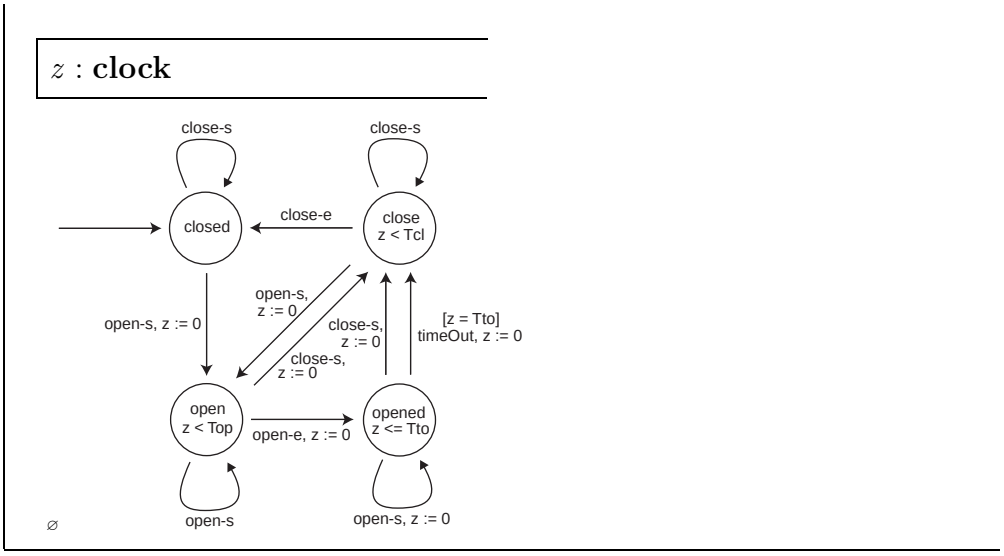
The only operation in the Object-Z section of this class is *supplyId* specifying the situation in which a key supplies its identity (to the lock). When considering time aspects, however, other situations arise. A key will be in location *wait* after it has

supplied its identity and is waiting to see whether or not access is granted. A key will be in location *eject* when it is being ejected from the lock once access permission has been decided.
The lock is specified by the class *Lock*:



The attribute *keys* in this class denotes the set of keys that have permission to access the room. The operations *grant* and *deny* capture whether or not any supplied key is in this set, and hence whether or not access to the room is granted or denied.
The door is specified by the class *Door*:

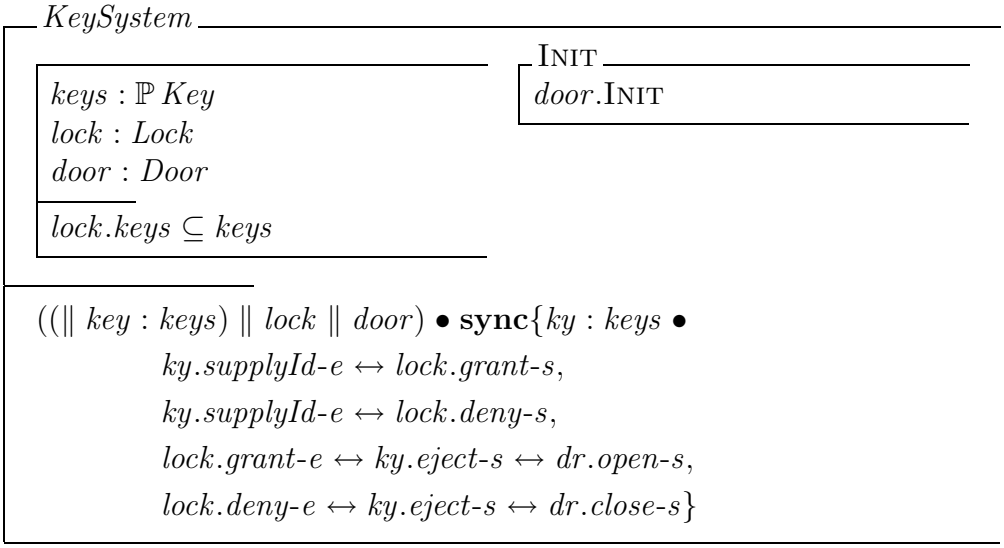




A door can be in any of four situations: closed (location *closed*) opening (location *open* where the operation *open* occurs) opened (location *opened*) and closing (location *close* where the operation *close* occurs). In each situation, the door can receive an instruction to open or close the door. In all cases, when an instruction to open the door is received the switch *open-s* is taken, while if an instruction to close the door is received the switch *close-s* is taken.

In locations *closed* or *close*, if the instruction to open is received the operation *open* is invoked, while if the instruction to close is received effectively the door continues as if nothing had happened. In location *open*, if the instruction to open is received effectively the door continues as if nothing had happened, while if the instruction to close is received the operation *close* is invoked. In location *opened*, if the instruction to open is received the door remains open but the timing is reset to 0, while if the instruction to close is received the operation *close* is invoked.

The complete electronic key system can now be specified by the class *KeySystem*. In this class, the attribute *keys* denotes the set of all keys in the system; the set of keys that have permission to access the room will be a subset of *keys*. The synchronization conditions ensure that the key identity output by a key is passed to the lock and used to determine whether or not that key has permission to access the room, and that once the access permission has been decided the key is ejected and the door requested to open or close, depending on whether access was granted or denied.



11.7 Related Works and Conclusion

This research can be classified as one of the integrated formal methods (IFM). The IFM research area has been active for a number of years (e.g. [5, 44, 16, 11]) with a particular focus on integrating state based and event based formalisms (e.g. [42, 115, 108, 116, 127]).

One closely related area to ours is the research work on integration of Object-Z with various timed calculi. For example, Object-Z is combined with Timed CSP [101] in [83, 25], with timed refinement calculus [82, 40] in [109] and with duration calculus [132] in [60]. Indeed, those combinations have made improvements in comparison to some early conservative framework approaches [28, 93].

The technical difference between our approach to the others has been the way to clearly separate functionalities and timed behaviour and the use of graph based Timed Automata instead of the timed calculi to capture the behaviour. One clear benefit of our approach is that many existing well developed tools [9, 22, 111, 117] for TA can be used to check the timed behaviour of the design models. In addition to the benefit of bring graphical appeal in capturing the object behaviour of Object-Z classes, our approach also provide a way to structure TA automatons using Object-Z classes so that the scale up problem of TA can be managed. The novel communication mechanism developed in our approach is also more flexible and expressive than CSP channels. For example, arbitrary communication between various objects can be captured at the composite class level with the elegant communication links.

Another related work is the combination of Z with graphical diagrams, i.e. statecharts and petri nets. For example, in [15] a framework is presented to link Z with statecharts and treats Z operation schemas as state transition links in statecharts. Similarly, the

language OZS [47] blends Object-Z with statecharts and treats Object-Z operations as state transition links. Combinations of Z and Petri Nets have also been investigated in [57, 55]. All those approaches suffer a common drawback that the states in the graph have no systematic correspondence in Z or Object-Z parts. The issues of object composition and timing have not been addressed. Our approach is different: we treat Object-Z operations as states (TA locations) instead of state transitions (TA switches) and furthermore we have a systematic naming convention for all switches. Object composition and real-time issues are the main focus points in our approach.
In our future work, we plan to investigate the refinement techniques for this combination of Object-Z and TA. We also plan to develop various tools to support the combination, e.g. to develop an OZTA editor and various linking programs to the Object-Z tools and TA model checkers. For the OZTA editor, we chose to represent the OZTA specification information in an XML format to enable easy linkage with Object-Z and TA tools. Some of our on-going tool development can be accessed here [**?**].

Chapter 12

Insert Z into LSC and Synthesis

Behavior modelling plays an important role in software engineering. It is the basis of system development methods like system specification, design, code generation, testing and verification. Two complementary approaches for modelling behavior have been proven useful in practice. One is interaction-based, which focuses on global interactions between system components, e.g., MSC, LSC. The other is state-based modelling, which concentrates on the internal states of individual components, e.g., Z and VDM. Industrial scale systems often have not only complex data structure but also intensive interactive behaviors. In this chapter, we combine the modelling power of Z and LSC and investigate how to synthesize implementations all the way from LSC models equipped with complex data structures.

12.1 Introduction

In order to formally specify complex system, we propose a combination of interaction and state-based modelling, namely Live Sequence Chart and Z specification. That is, a complete system specification shall consist of two separate parts: an LSC part for capturing interactions between system components and a Z part for modelling the data and functional aspects. The significant and novel aspect of the combination is that it combines the modelling power of both and thus specifies systems beyond the capability of either one. Moreover, such combined specifications contain sufficient information for synthesis of distributed implementable system design.
State-based modelling naturally complements interaction-based modelling, and thus it is no doubt that a smooth integration of them shall be beneficial. LSC is a rather rich extension to MSC that allows specification of not only possible behaviors, but also mandatory behaviors. We choose Z over other state-based modelling language because Z is the widely known and accepted as well as well-developed in terms of specification, refinement, etc. The Z language is favored over Object-Z because Z

is relatively simple structured and the class structure (as well as inheritance and polymorphism) in Object-Z may serve as an unnecessary complication.
Synthesis from specifications like scenario-based diagrams or various automata is showed to be extremely hard [94, 95, 50, 69]. Our problem is further complicated by the complex data structure underlying the scenarios. Due to the high complexity of the problem, our primary aim is to discover a practical way of synthesizing sound (and not necessarily complete) implementations. To our best knowledge, our work is the first attempt to synthesize low-level implementations from combination of interactive-based modelling and state-based modelling.
We take a step-by-step approach. Firstly, a distributed object system is synthesized from the LSC universal charts. The local actions in the charts are treated as abstract events, as we did in Chapter 8. The global state machine is never constructed during the step so as to avoid state space explosion. Meanwhile, an abstract finite state machine is constructed from the Z model using automated predicate abstraction [8, 75], which allows us to grasp the behaviors of the objects based on a finite set of assertions. The abstraction method presented in Chapter 6 is augmented to cope with Z semantics. Secondly, the distributed object system is refined on an object basis to satisfy data-related requirements. Thus, the preconditions of the local actions (Z operations) and hot conditions in the LSC model will never be violated. Additional crucial properties for open systems, like nonblocking and uncontrollability of the environment, are also taken into account. Finally, we may synthesize executable implementation by generating code from the refined finite state machine (the design). Our method is experimented using a Java application.

12.2 Integrating Live Sequence Chart and Z

State-based modeling language like Z and interaction-based modeling language like LSC naturally complement each other. LSC lacks the expressiveness to capture complicated data and functional model. Local actions are often ignored or treated as abstract events in the study of the verification and synthesis problem of LSC. Examples are the works in [50, 12] and our work in Chapter 8. Local data variables are often implicitly associated with the objects. They may appear in the conditions or get updated by the local actions. However, there is no way to specify exactly how the local actions update the local variables and what the data space of the object is, except using concrete implementations, which we think is undesirable as sequence diagrams are used in the early stage of system development. On the other hand, in Z specification, the system behavior patterns are often implicitly embedded within various state/operational constraints. Without explicit system behavior representation, it is difficult to analyze or implement those abstract models. Z is not intended for timed or concurrent behaviors [128]. It lacks the expressiveness to capture dynamic

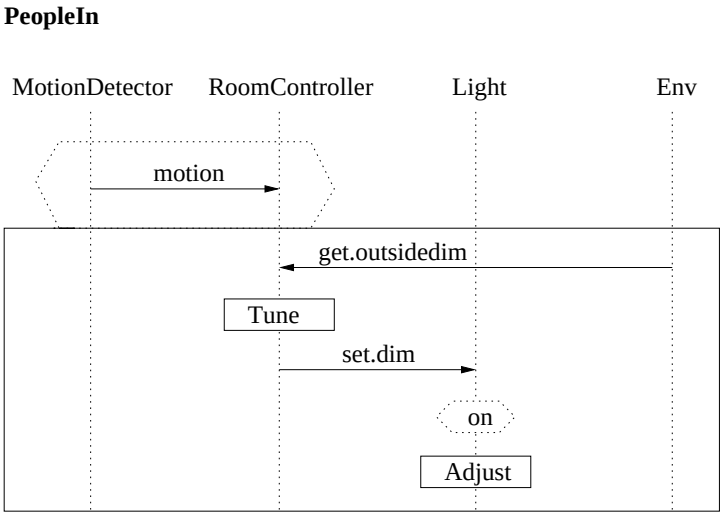


Figure 12.1: A scenario of the LCS: *PeopleIn*

interactive behaviors between the components in the system.

Combination of LSC and Z shall constitute a powerful modelling language covering a wider range of systems. Thus, we propose a simple yet effective integration of LSC and Z. We require that a combined system specification shall consist of two parts. One is a set of LSC universal charts, which specify mandatory interaction scenarios between system components. The other is a Z specification, which specifies the data and functional models associated with the objects in the system. In particular, each object in the LSC model with non-trivial data states is associated with a Z package in the Z part. Each local action in the LSC model is defined in the respective Z package as a Z operation schema. Conditions in LSC model may only mention variables defined in the respective Z state schema in addition to external inputs.

System modelling shall start with identifying scenario-based system requirements, from which the universal charts are constructed. During the process, the system engineer slowly decides the data variable and local actions for each object. The designer's intension of the local action can be naturally documented as pre/postcondition pairs. Later, the designer may specify each local action using Z operation schema to formally state how each local action updates the data state. This way, a complete system specification is built. In the following, the same Light Control System is used as a running example to show how it may be specified using a combination of LSC and Z packages, and how implementation may be synthesized from the specification.

Figure 12.1 captures a typical scenario of the LCS. When a user enters a room: the motion detector senses the presence of the person, and the room controller reacts by

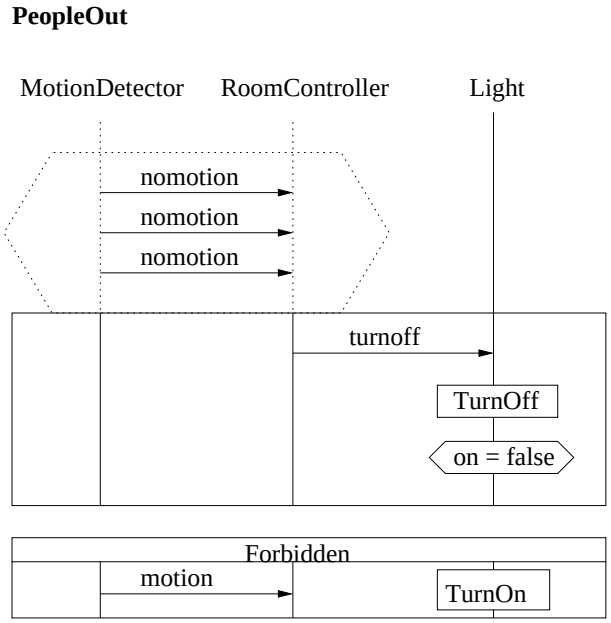


Figure 12.2: Scenario of the LCS: *PeopleOut*

sensing the current daylight level and tuning the light with appropriate illumination if the light is already on. Figure 12.2 illustrates another scenario of the LCS. Whenever a user leaves a room (leaving it empty), the detector senses no movement. The room controller waits for a safe number of *nomotion* to make sure the room is empty and then turns off the light. There are a number of important features of LSC presented in the chart, i.e., hot location, hot condition and forbidden events. Basically, the forbidden events require that in order to complete this scenario, no movement should be detected before the chart ends and the light is eventually turned off before it is turned on again. The rest of the scenarios are presented in Figure 12.3 and 12.4, in which the occupant may directly turn on/off the light by pushing the button or the system may adjust the illumination of the light.

After identifying the universal charts, the data variables and local computation of each object become clear. No local action is associated with instance *MotionDetector*, which suggests that it has trivial data state. The Z package associated with the *Light* and *RoomController* are illustrated in the following.

The Z package of the *light* contains the following schemas.

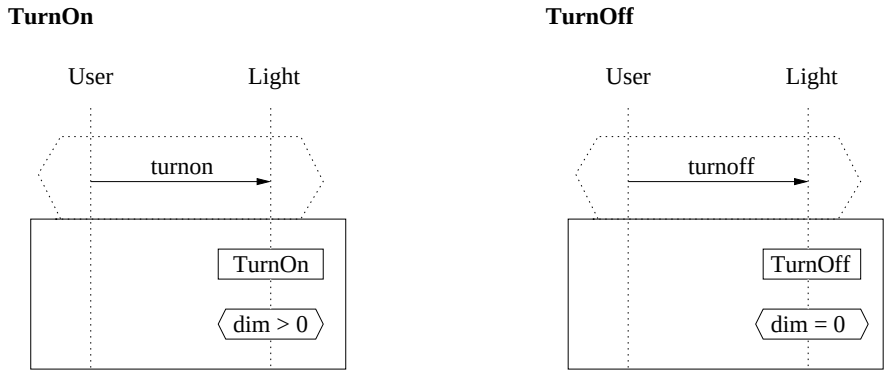


Figure 12.3: Scenarios of the LCS

Light

$dim : Illumination$

$on : \mathbb{B}$

$dim > 0 \Leftrightarrow on = true$

Adjust

$\Delta Light$

$dim? : Illumination$

$on = true \land dim' = dim?$

TurningOn

$\Delta Light$

$on = false \land dim' = 100$

$on' = true$

TurningOff

$\Delta Light$

$on = true \land dim' = 0$

$on' = false$

LightInit

$Light'$

$dim' = 0 \land on' = false$

The Z package of the *room controller* contains the following schemas.

RoomController

$dim : 0 \,..\, 100$

RoomControllerInit

$RoomController'$

$dim' = 0$

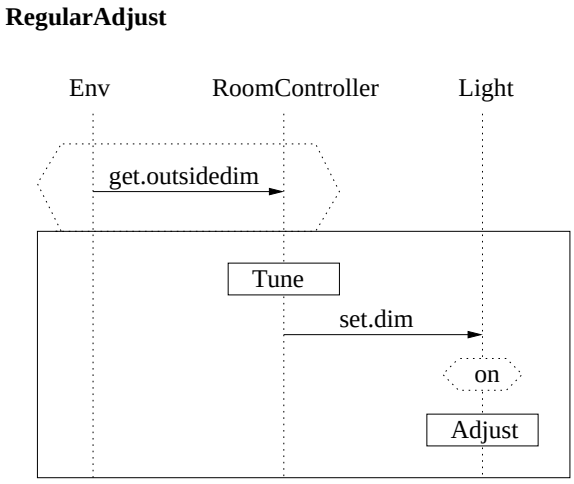


Figure 12.4: Scenario of the LCS

Tune

$\Delta RoomController$

$outsidedim? : 0 \,..\, 100$

$(outsidedim? \leq 20 \land dim' = 100) \lor$
$\quad (outsidedim? > 20 \land dim' + outsidedim? = 100)$

The variable *dim* in the state schema represents the light level (in *room controller*'s knowledge). Initially, it is of value 0. The operation *Tune* computes the desired light level according to the outside light level.

All instances in Figure 12.1,12.2,12.3,12.4 with non-trivial data states are associated with Z packages, i.e., the *Light* package for the *Light* object and the *RoomController* package for the *RoomController* object. Local actions like *Adjust*, *TurnOn*, *TurnOff*, *Tune*, are defined as operation schemas in the respective package. Therefore, the Z specification and the LSC model constitute an integrated specification of the LCS.

The result is a rigid system architecture, which has its advantages: the data and functional model and the interaction-based model remain orthogonal throughout development, and so can be analyzed or refined separately using existing tools and methods. Once both parts stabilize, the integrated specifications shall contain sufficient information on both data and control aspects of the system, which allows us to automatically synthesize implementable designs. Graphically, links from an instance in the chart to its Z state schema, and links from local actions to Z operation schemas shall be provided, e.g., the Z schema is showed in the popup window once the instance

is highlighted and so are the operation schemas.

12.3 Synthesis of Distributed Object System

In this section, a distributed object system is synthesized from the universal charts. For the time being, local actions are treated as abstract events. The synthesized object system is refined in the next section to handle data-related requirements. The synthesis is closely related to the construction in Chapter 8. However, because we have to store the data-related requirement for later refinement, finite state machines instead of CSP processes are constructed. State invariants are used to store data requirements. Moreover, using of finite state machine allows us to reuse our work in Chapter 6.

There are a number of principles to identify a good synthesis strategy. Firstly, the synthesis should be robust with the notion of data refinement [128] so that the synthesized design remains valid after refinement of the Z operations. Secondly, the global state machine should never be constructed so that state explosion is avoided. This is essential for notations like LSC, which has a distributed nature and an underlying partial ordering semantics. Above all, the synthesized design should be consistent with the specification.

12.3.1 Synthesizing Local State Machines

We start with constructing a state machine for each instance in a single chart. Given a basic chart m (a main chart or a sub-chart of a main chart without hierarchy), let $M_m^i \mathrel{\widehat=} (S, S_0, F, \Sigma, T, I)$ be state machine synthesized from instance i in chart m. The basic idea is to construct one state for each location. Thus, S is the set of states corresponding to the set of locations along the instance. S_0 contains exactly the state corresponding to the first location. F contains the states corresponding to the cold locations. For each location labelled with a cold condition, an additional state labelled with the negation of the condition is constructed so that if the condition is violated, the additional state is reached. The only transition enabled at the additional state is labelled with a synchronization barrier, which is used to terminate the (activation of) the chart. For each location labelled with a hot condition, the condition is labelled with the respective state and no additional state is added. This prevents behaviors that might violate the hot condition from happening. There is a transition (s_1, e, s_2) in T if the location corresponding to s_2 is next to the location corresponding to s_1 which is labelled with e. After reaching the very last location of the chart (the bottom line), the state machine behaves freely so that it puts no further constraint over the system. Such a state machine constrains single activation of the basic chart.

Hierarchical chart can be flattened as finite state machines straightforwardly. Fig-

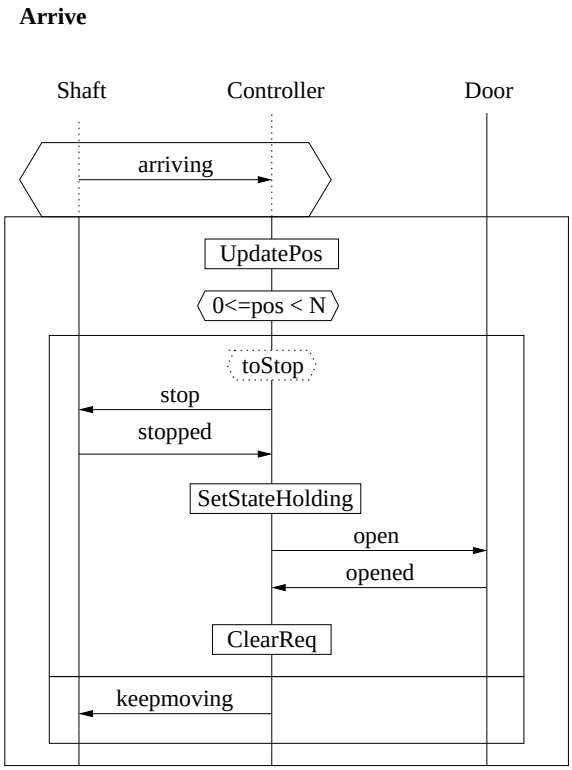


Figure 12.5: Scenario of the Lift Control System

ure 12.5 presents a universal chart containing a conditional branch. It is part of the LSC specification of a lift control system. Whenever the lift approaches the next floor, the *shaft* sends a message *arriving* to *controller*. The *controller* refreshes its knowledge of the current level by updating its local variable *pos*. A hot condition stating that the value of *pos* (a local variable representing the current level) must be within its range is asserted. The *controller* decides whether to stop at the next floor. If the condition *toStop* is true, i.e., the next level is requested internally or requested externally with the right direction, the *shaft* stops and the *door* is opened and the respective request is cleared. Otherwise, the lift continues travelling in the same direction.

The state machine presented in Figure 12.6 captures the behaviors of *Shaft* in the main. Events $Arrive.x.main$ and $Arrive.x.sub1$ are barriers used to synchronize the entering or exiting of the main chart or a sub-chart among all participating instances. Variable x is an identifier which distinguishes different activation of the same chart. Therefore, only participating instances in the same activation of the chart are synchronized. Whenever the chart completes (reaching the filled circle), all events in Σ_m^i

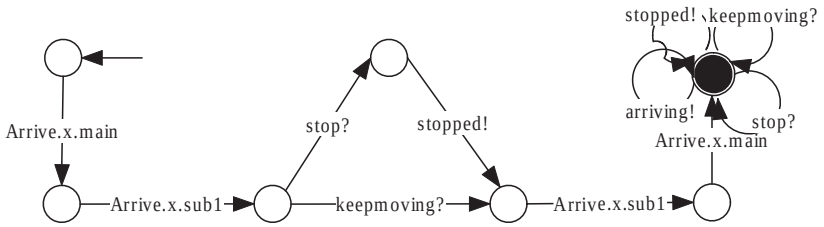


Figure 12.6: State machine for *Shaft*

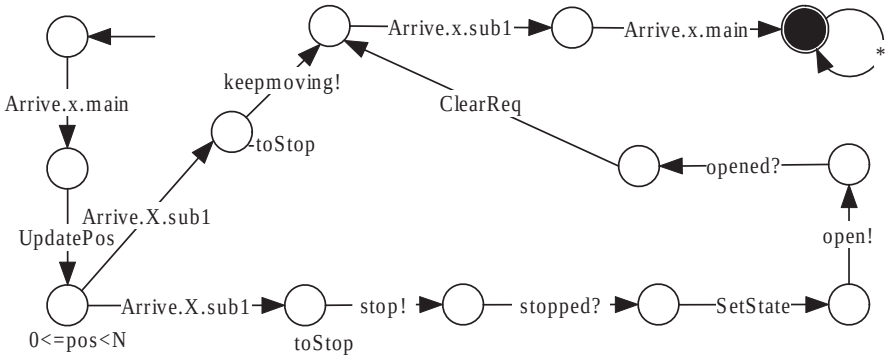


Figure 12.7: State machine for *Controller*

can be engaged freely (indicated by a transition labelled with *). Only transitions labelled with visible events are constructed since transitions concerning invisible events are free to occur by the definition of parallel composition.

The state machine presented in Figure 12.7 is synthesized for instance *Controller*. The hot condition is labelled with the state right after local action *UpdatePos*. After entering the sub-chart, two states are reached, one labelled with condition *toStop* and the other labelled with its negation. Thus, the conditional branch is effectively flattened. In general, state machines for hierarchical charts can be constructed from the state machines for the sub-charts.

A universal chart u is associated with two sets of synchronous barriers, namely $u.x.y.conVio$ and $u.x.y$ where x is a counter uniquely identifying an activation of chart u, and y is the identifier of a sub-chart. The x component is necessary because there could be multiple or even infinite overlapping activation of the same chart. For instance, trace $\langle nomotion, nomotion, nomotion \rangle$ triggers three overlapping activation of the chart *PeopleOut*. Event $u.x.y$ is used to synchronize the entering or exiting of

sub-chart y in chart u among those participating instances. Event $u.x.y.conVio$ is engaged if and only if a cold condition in sub-chart y is violated in the x-activation of u. It is the only event which can be engaged at the state labelled with the negation of a cold condition. Other instances in the chart are ready to engage in this event all the time (a transition labelled with this transition is enabled at every state in the state machine for other instances).

The state machine for an instance in the pre-chart is similarly constructed. However, because a universal chart puts no constraint over the system before entering the main chart, the state machine synthesized from the pre-chart shall allow all possible behaviors, and at the same time monitor communication sequences that may match the pre-chart. Let $M_p^i \triangleq (S, S_0, F, \Sigma, T, I)$ be the state machine synthesized from instance i in the pre-chart p. There is a transition (s_1, e, s_2) in $M_p^i.T$ if the location corresponding to s_2 is next to the location corresponding to s_1, which is labelled with e. In addition, a transition (s_1, e', s_{max}) is constructed for every event e' in $\Sigma_u^i \setminus \{e\}$, where s_{max} is the state corresponding to the last location on instance i in the *main* chart (the filled one). Intuitively, the pre-chart progresses whenever an expected event is engaged, whereas an unexpected event aborts the activation of the chart. Because hot condition in pre-chart has no semantic meaning, all conditions in pre-charts are treated as cold conditions. Lastly, the state corresponding to the last location in the pre-chart is identified with the state corresponding to the first location in the main chart so that once the pre-chart is completed, the main chart is reached.

Figure 12.8 shows the state machines synthesized for instances in the chart showed in Figure 12.2. The alphabet of each state machine includes the forbidden events. The forbidden events are allowed to occur before entering the main chart. Once a communication sequence matches the pre-chart, the state machine synchronizes entering of the main chart. All states in the pre-chart are accepting as the state machine shall not constrain the system execution before entering the main chart.

The state machines constructed so far only monitors a single activation of the chart. A trace which triggers multiple activation of the same chart is not properly constrained. For instance, the state machines in Figure 12.8 may execute the following trace:

$$\langle nomotion, motion, nomotion, nomotion, nomotion, TurnOff \rangle$$

It is however not allowed by the chart in Figure 12.2 because the three consecutive *nomotion*? triggers another activation of the chart. The remedy is to identify the filled state with the initial state so that the state machine is reused for later activation. However, such state machines still can not constraint overlapping activation. In general, it is necessary to have infinite copies of such state machines to monitor possible infinite (overlapping) activation of the same chart. For simplicity, we make the assumption that the number of overlapping activation of any chart is always bounded. Under the assumption, only finite copies of state machines are necessary

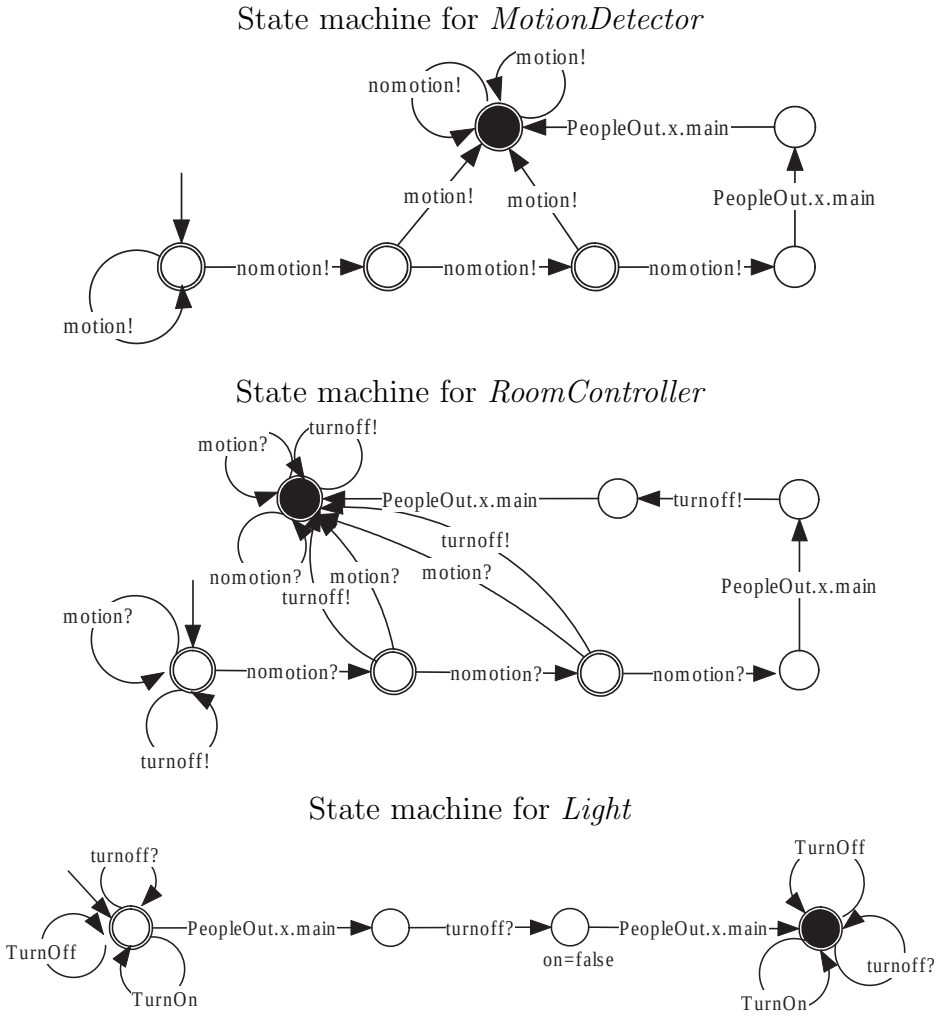


Figure 12.8: State machines for instances in *PeopleOut*

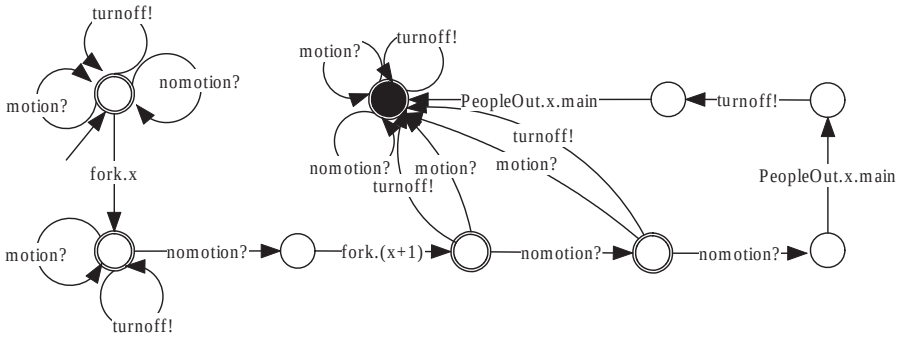


Figure 12.9: State machines for instance *RoomController*

for monitoring overlapping activation, and they can be reused for non-overlapping activation. In practice, large number of overlapping activation is unlikely because system behaviors are increasingly restricted as the number of overlapping activation increases. There is often a natural limit on the number of overlapping activation. For instance, there could be at most three overlapping activation of chart *PeopleOut* because the main chart shall complete before the fourth *nomotion* event. A simple analysis shall tell the maximum number of activation allowed by a chart.

The state machine presented in Figure 12.9 is synthesized from *RoomController* in scenario *PeopleOut*. It monitors the x-activation of the chart. The state machine is augmented with a special synchronization barrier $fork.x$, which is used internally to activate a new copy of the state machine whenever it moves beyond the initial state. Because there are at most three overlapping activation of the chart, three copies of the state machine with x ranging from 0 to 2 are constructed. The copy with $x = 0$ does not have the first state. The copy with $x = 2$ does not have the state where $fork.3$ can be engaged because there is no fourth copy to be forked. The product of the three copies are computed as showed in Figure 12.10. The very last state (the one composed by three filled state) is identified with the initial state so as to allow non-overlapping activation. We remark that the final state machine can be further reduced using standard technique like bi-simulation reduction, etc. For instance, all states labelled with event *fork* are removed since they contribute nothing to system behaviors.

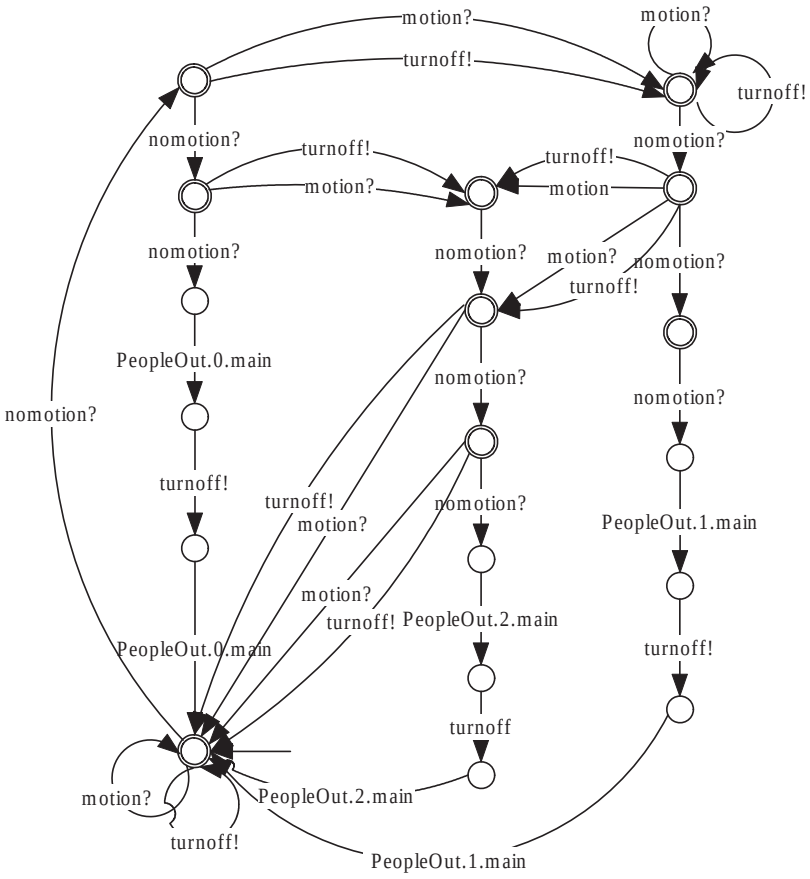


Figure 12.10: State machine synthesized for instance *RoomController*

We remark that the product of the state machines for all instances in the chart, $\|_i M_u^i$, refines the chart, i.e., all accepting runs of the state machine satisfy the chart. An immediate consequence is that product of the state machines for all the universal charts, $\|_u \|_i M_u^i$, refines the LSC specification, i.e., only behaviors satisfying all the universal charts are allowed. Because the parallel composition operator is symmetric and associative, the following rule is established. Let M_{LSC}^i be the local behaviors of an object i.

$$\|_u \|_i M_u^i \widehat{=} \|_i \|_u M_u^i \widehat{=} \|_i M_{LSC}^i$$

Due to the above transformation, the local behaviors of an object are determined without constructing the global state machine. For example, the behaviors of the *RoomController* are captured by the product of the state machines synthesized from all the universal charts. We skip the formal soundness proof. In previous chapters, we have formally defined a trace-based denotational semantics for LSC, and then developed a sound interpretation of LSC in the classic notion of CSP [59]. By transforming CSP interpretation of the LSC model using its algebraic laws, the local behaviors of each object are grouped together as a set of distributed processes. A bisimulation relation between the synthesized state machine and the transition system interpretation of the distributed processes would prove the soundness of the synthesis. Alternatively, we may define a similar set of algebraic laws in terms of finite state machines and prove the soundness directly.
So far, environmental objects are not distinguished from system objects. In other words, we handle only closed systems but not open systems. Synthesis for open systems asks whether there is an implementation that can be deployed in any malevolent environment. To avoid the undecidability of distributed synthesis for open systems, the same lightweight approach presented in Chapter 8 is adopted. The synthesized state machine for the environment (parallel composition of all state machines for environment objects) is verified to be equivalent to the user-supplied modelling of the environment.

12.4 Refinement of the Distributed Object System

In our combined specification, local actions could be complicated computation constrained by pre/post-condition. It is necessary to refine the distributed object system so as to guarantee that a local action is only engaged with its precondition satisfied, a hot condition shall be satisfied in all circumstance, etc. However, it is difficult to tell if certain assertion is true after a series of local computations simply because the state space of a Z specification may often be infinite. The problem is further complicated as Z operation schema may take inputs from the environment, which can

not be controlled by the system. Our remedy is predicate abstraction, as applied in Chapter 6 for extracting finite state realization of Object-Z specifications. Predicate abstraction allows us to interpret and then restrict the behaviors of an object based on abstract view of the data variables, which is essential for our synthesis since an implemental control structure may only contain a finite number of control states.
The abstraction method used in Chapter 6 is amended for abstracting Z packages. In Z semantics, the result of applying an operation outside its precondition is divergence. Thus, in abstraction of a Z package, an operation must be applied at states where its precondition is satisfied. Moreover, in the abstraction interpretation, we guarantee that applying an operation may reach all states where the postcondition may be satisfied. This way, our abstraction is robust with Z data refinement, i.e., weakening precondition and strengthening postcondition. The abstract machine is then used to refine the distributed object system synthesized from the LSC model on an object basis. Invocation of operations that might violate its precondition or result in a state violating a hot condition is systematically pruned.
In order to guarantee the correctness of the synthesized design, we require that the set of predicates for abstraction includes all conditions in the universal charts (as well as the predicate in the initial schema for simplicity). A finite state abstraction of a Z package is built by abstracting both its initial schema and its operation schemas. Because only sound designs are of interest, a local action shall be invoked only when we are certain no assertions will be violated. Thus, the precondition of the operation is abstracted as $\mathcal{W}(\text{pre}(\mathit{Operation}))$ and its postcondition is abstracted as $\mathcal{S}(\text{post}(\mathit{Operation}, s_a))$, where s_a is an abstract state satisfying the abstract precondition. Intuitively, by replacing the precondition with a more restrictive one, we make sure no precondition shall be violated. By replacing the postcondition with a less restrictive one, we make sure that no hot conditions shall be violated in all circumstance.

Definition 12.4.1 *Given a set of predicate P, $M_Z^i \mathrel{\widehat{=}} (S, S_0, F, \Sigma, T, I)$ is an abstraction of the Z package associated with object i only if $S \mathrel{\widehat{=}} S_a$ and $S_0 \mathrel{\widehat{=}} \mathcal{W}(\mathit{initial\ condition})$ and $F \mathrel{\widehat{=}} S$ and Σ is the set of operation schemas in the package and I labels a state with itself and $T \mathrel{\widehat{=}} \{(s_1, e, s_2) : S \times \Sigma \times S \mid s_1 \in \mathcal{W}(\mathit{pre}(e)) \wedge s_2 \in \mathcal{S}(\mathit{post}(e, s_1))\}$.* □

Assume the set of predicates for abstracting the *Light* package is $\{\mathit{dim} = 0, \mathit{on} = \mathit{false}, \mathit{dim} > 0\}$, the set of abstract states contains two states: $S_a \mathrel{\widehat{=}} \{\mathit{dim} = 0 \wedge \mathit{on} = \mathit{false}, \mathit{dim} > 0 \wedge \mathit{on} = \mathit{true}\}$. The abstract initial state is exactly the state where $\mathit{dim} = 0 \wedge \mathit{on} = \mathit{false}$. Operation *Adjust* is abstracted by computing the following:

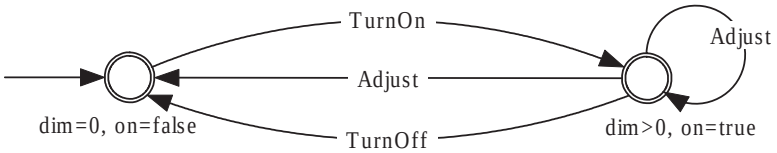


Figure 12.11: Abstraction of the *Light* package

$$
\begin{aligned}
&\mathcal{W}(\mathrm{pre}(\mathit{Adjust}))\\
&\quad \widehat{=}\ \mathcal{W}(\exists\, \mathit{dim}' : \mathit{Illumination};\ \mathit{on}' : \mathbb{B} \mid \mathit{dim}' > 0 \Leftrightarrow \mathit{on}' = \mathit{true} \bullet\\
&\qquad \mathit{on} = \mathit{true} \wedge \mathit{dim}' = \mathit{dim}?)\\
&\quad \widehat{=}\ \{\mathit{dim} > 0 \wedge \mathit{on} = \mathit{true}\}\\
&\mathcal{S}(\mathrm{post}(\mathit{Adjust}, \mathit{dim} > 0 \wedge \mathit{on} = \mathit{true}))\\
&\quad \widehat{=}\ \mathcal{S}(\exists\, \mathit{dim}, \mathit{dim}? : \mathit{Illumination};\ \mathit{on} : \mathbb{B} \mid\\
&\qquad \mathit{dim} > 0 \Leftrightarrow \mathit{on} = \mathit{true} \bullet\\
&\qquad\quad \mathit{dim} > 0 \wedge \mathit{on} = \mathit{true} \wedge \mathit{on} = \mathit{true} \wedge \mathit{dim}' = \mathit{dim}?)\\
&\quad \widehat{=}\ \{\mathit{dim}' = 0 \wedge \mathit{on}' = \mathit{false}, \mathit{dim}' > 0 \wedge \mathit{on}' = \mathit{true}\}
\end{aligned}
$$

Thus, the abstract operation *Adjust* is enabled only at the abstract state where *on* is true, from which both abstract states can be reached. We skip the abstraction of the other operations in the package. Figure 12.11 shows the resultant state machine. After constructing the abstract state machine from the Z package, the product of M^i_{LSC} and M^i_Z is computed. By removing states labeled with contradiction, we guarantee that no precondition or hot condition is violated. However, the problem is complicated by the uncontrollability of the environment because removing states may put restriction over inputs from the environment, which is problematic. For instance, if we allow the user to adjust the illumination by setting it to certain value, captured by the universal chart in Figure 12.12. It requires that after operation *Adjust*, $\mathit{dim} > 0$ must hold. Intuitively, we know that this hot condition may not be satisfied because the user may set the dim to 0 and hence accidentally turns off the light (due to the state invariant). Another important property for open systems is nonblocking, i.e., the design should not introduce any fresh deadlock. The pruning algorithm presented in Chapter 6 is reused to determine whether there is a satisfying design, and synthesizes one if possible by refining the product state machine.
Figure 12.12 presents the state machine for instance *Light* from scenario **UserAdjust** assuming there is no overlapping activations of the chart for simplicity. The product of the state machines in Figure 12.11 and Figure 12.12 is shown in Figure 12.14 (where one state labelled with *false* has been removed). Applying the pruning algorithm to the product removes all but the two accepting states and the transitions between

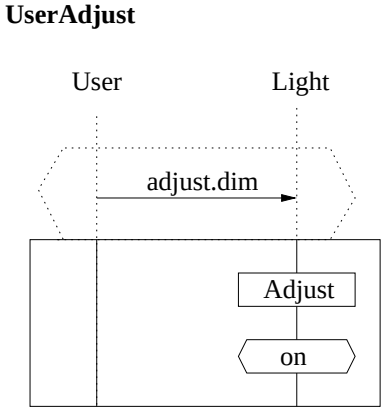


Figure 12.12: Scenario of the LCS: *UserAdjust*

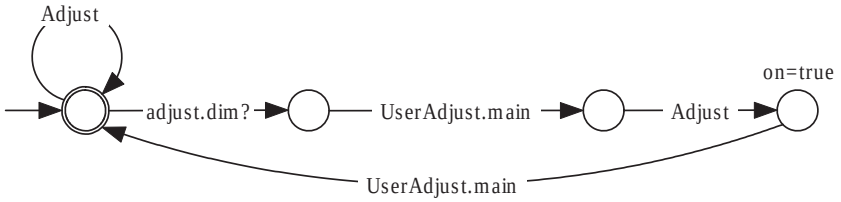


Figure 12.13: State machine synthesized from instance *Light* in *UserAdjust*

them. The * state is removed because *Adjust* is an uncontrollable event at the state and the state labelled with $on = false \wedge dim = 0$ is not reachable from the * state by engaging *Adjust* while it does in the abstract state machine. Thus, line 10 of the algorithm applies so that the transitions labelled with *Adjust* are removed. The *** state is removed because it is not reachable any more. The ** state is removed because it becomes a fresh deadlock state and thus line 14 of the algorithm applies so that the state is pruned. The rest are removed because they can not reach an accepting state. The resultant state machine is a valid design for closed systems because there are initial and accepting states. Intuitively, the resultant state machine guarantees that the chart **UserAdjust** is satisfied by requiring that it is never activated. However, for open systems, *user* is considered as part of the environment and therefore there is no way to prevent users from activating the chart through sending message *adjust*.*dim*?. In our approach, the synthesized modelling of the *User* is failed to simulate of the default modelling (where users can initiate any communication at any time) and, thus, there is no design satisfying the chart.

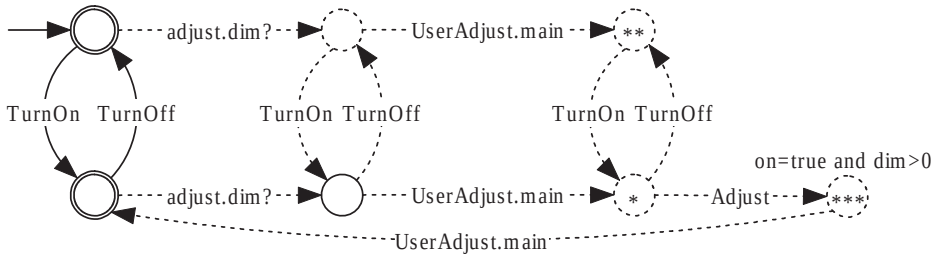


Figure 12.14: Product state machine

12.5 Automation

We implemented a prototype to experiment our approach with standard case studies. The input to our experimental tool is an XML representation of the Z model and an XML representation of the LSC model. A transition in the pruned state machine may be constrained by restricting its postcondition in the pruned state machine, which is not implementable. Two different remedies have been explored. The first remedy is to guard each invocation of the action with a proper guard condition. For partially pruned nondeterministic choices, the transitions shall be guarded with the weakest precondition that guarantees the reach of the desired state. After that, executable implementation can be synthesized straightforwardly with the implementation of each local actions supplied by users. As long as the implementation of local actions conforms to its precondition/postcondition specification, our synthesized prototype remains sound. However, a reasonable guard condition must not involve any primed variables. Computing the weakest precondition requires elimination of the primed variables, which is in general undecidable. Therefore, this remedy is unlikely fully automated. The other remedy is to generate a set of proof obligations for nondeterministic choices which are partially pruned. When the user provides an implementation of the operation, the proof obligations are verified (or tested) in addition to the pre/post-condition so as to make sure the operation satisfies the more restrictive post-condition at the system states.

Our approach is designed to handle complex systems. During the first step, we synthesize a distributed object system from the LSC model without constructing the global state machine. Later, we limit the number of overlapping activation of the same chart as a way to further reduce the size of the local state machines. For instance, all universal charts except *PeopleOut* allow no overlapping activation in the LCS example. Computing the product of multiple state machines ($\|_i M_u^i$) explicitly are expensive, e.g., the state machine for instance *Light* contains 760 states without any reduction. Therefore we reuse existing CSP-based process oriented design patterns

for concurrency [124] to generate structural prototypes.
To handle systems with infinite data space, we adopt predicate abstraction to construct abstract view of system behaviors in terms of finite assertions. In general, the size of the abstract state machine is exponential to the number of predicates for abstraction. It is the most time-consuming operation in our method. However, it remains affordable because only one Z package is abstracted at a time and there are unlikely to be large number of conditions concerning one object. Our abstraction method constructs an abstract state graph by paying a reasonable price. In our prototype, sound approximation of the function $\mathcal{W}$ and $\mathcal{S}$ is used. To further speed up the abstraction as well as to guarantee termination of the proving, every lemma is proved in a limited amount of time. The time limit is set as a user option. The date aspect of the LCS example is slightly trivial. As for reference, in a vending machine example where there are state variables with infinite domain and multiple operation schemas, all together 190 lemmas are generated and all 105 provable lemmas are proved without user interaction in minutes. The lift control system is also experimented to handle system with arrays of variables (refer to `http://www.comp.nus.edu.sg/~sunj/liftsystem` for detail). In addition, a number of tricks are used to reduce the abstract state space, for instance removing false state by considering co-relation between the predicates and the state invariant before abstract. The complexity of our pruning algorithm is polynomial time in terms of the number of states. So are the operations we perform over the state machine. Thus, they are carried out in reasonably speed fashion.

12.6 Summary and Discussion

In this work, we present a systematic way of synthesizing designs from a combination of state-based modelling and interaction-based modelling, namely Z and LSC. Our contribution is threefold. Firstly, we propose an intuitive integration of Z model and LSC model, which is capable of modelling systems with not only complicated data structures but also complex interactive behaviors. Secondly, we develop a systematic way of synthesizing distributed finite state designs all the way from the combined specifications. Thirdly, we developed an experimental tool to automate our method. Our method does suffer from being over-restrictive sometimes. One of the reasons has already been mentioned in Section 12.3.1, i.e., we do not allow infinite overlapping activation of the same chart. Another reason is that because our pruning applies on object basis, valid designs requiring cooperation of multiple system objects are not possible. For instance, inputs to an operation from other system components are controllable if we consider the global state machine. For example, in Figure 12.1, the value of *dim* from *RoomController* is actually never 0 from the whole system's view. Disallowing such design is a sacrifice we have to make if we do not construct the global state machine. The third reason is the limited power of proving. The effectiveness of

the predicate abstraction, e.g., fewer spurious behaviors, depends the proving power. Spurious behaviors may result in pruning valid designs. For instance, if the abstraction suggests that applying an uncontrollable operation may result in an undesired state from a given state where as in fact it cannot, then the uncontrollable operation will be prohibited from happening. Nevertheless, our approach serves as a promising method to apply synthesis techniques to complicated system specifications, and it can be applied to other integrations of state-based and interaction-based modelling as well.

Bibliography

[1] A. Alencar and J. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lect. Notes in Comput. Sci.*, pages 180–199. Springer-Verlag, 1991.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 304–313. ACM Press, 2000.

[4] Rajeev Alur, Costas Couroubetis, and David L. Dill. Model-checking for Real-time Systems. In *proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[5] K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.

[6] R. Arthan. On free type definitions in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 40–58, University of York, Dec 1991.

[7] J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 18(1), 1990.

[8] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI 2001*, pages 203–213, 2001.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems 1995*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[10] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, May 2001.

[11] E. Boiten, J. Derrick, and G. Smith, editors. *IFM'04: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, April 2004.

[12] Y. Bontemps, P. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, July 2004.

[13] G. Booch. *Object-Oriented Design with Applications*. Addison-Wesley, 1991.

[14] S. Brien, T. King, J. Nicholls, J. Woodcock, and J. Wordsworth. Z Base Standard — Version 1.0. Technical report, Programming Research Group, Oxford Univ., Nov 1992.

[15] R. Bussow and W. Grieskamp. A Modular Framework for the Integration of Heterogeneous Notations and Tools. In Araki et al. [5], pages 211–230.

[16] M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2002.

[17] F. Civello. Role for composite objects in object-oriented analysis and design. In *Proc. 8th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'93)*, pages 376–393, 1993.

[18] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, 1991.

[19] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[20] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[21] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[22] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, pages 208–219. Springer, 1996.

[23] D. de Champeaux, D. Lea, and P. Faure. *Object-Oriented System Development*. Addison-Wesley, 1993.

[24] M. Dean and G. Schreiber (editors). OWL Web Ontology Language Reference. W3C Recommendation, http://www.w3.org/TR/owl-ref/, 2004.

[25] J. Derrick. Timed csp and object-z. In *3nd International Conference of Z and B Users (ZB'03)*, LNCS. Springer, June 2003.

[26] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994.

[27] T. Dillon and P.L. Tan. *Object-Oriented Conceptual Modeling*. Prentice-Hall, 1993.

[28] J. S. Dong, J. Colton, and L. Zucconi. A Formal Object Approach to Real-Time Specification. In *the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, Seoul, Korea, December 1996. IEEE Press.

[29] J. S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12*, pages 181–190. Prentice-Hall, November 1993.

[30] J. S. Dong, P. Hao, and B. Mahony. Formal designs for embedded and hybrid systems. *International Journal on Software Engineering and Knowledge Engineering*, 15(2):373–378, April 2005.

[31] J. S. Dong, P. Hao, S.C. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In J. Davies, W. Schulte, and M. Barnett, editors, *the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, pages 483–498. Springer-Verlag, November 2004.

[32] J. S. Dong, C. H .Lee, Y. F. Li, and H. Wang. Verifying DAML+OIL and Beyond in Z/EVES. In *The 26th International Conference on Software Engineering (ICSE'04)*. ACM/IEEE Press, May 2004.

[33] J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *Proceedings of 12th Internation Symposium on Formal Methods Europe: FM'03*, pages 796–813, Pisa, Italy, September 2003. LNCS, Springer-Verlag.

[34] J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi. Timed Patterns: TCOZ to Timed Automata. In Jim Davies and Wolfram Schulte, editors, *The 6th International Conference on Formal Engineering Methods (ICFEM'04)*, Seattle, U.S.A., November 2004.

[35] D. Duke. *Object-Oriented Formal Specification*. PhD thesis, University of Queensland, 1991.

[36] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lect. Notes in Comput. Sci.*, pages 242–262. Springer-Verlag, 1990.

[37] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[38] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z.* Cornerstones of Computing Series. Macmillan, March 2000.

[39] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[40] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In *Mathematics of Program Construction*, 1998.

[41] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[42] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [5].

[43] A. Goldberg and D. Robson. *Smalltalk 80: The Language.* Addison-Wesley, 1989.

[44] W. Grieskamp, T. Santen, and B. Stoddart, editors. *IFM'00: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.

[45] A Griffiths. An Extended Semantic Foundation For Object-Z. Technical Report 95-39, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, 1995.

[46] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *Object-Oriented Systems*, 2:195–215, Chapman & Hall 1995.

[47] J. P. Gruer, V. Hilaire, A. Koukam, and P. Rovarini. Heterogeneous formal specification based on object-z and startecharts: semantics and verification. *J. Systems and Software*, 2004.

[48] Volker Haarslev and Ralf Möller. *RACER User's Guide and Reference Manual: Version 1.7.6*, December 2002.

[49] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5), 1988.

[50] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *International Journal on Foundations of Computer Science*, 13(1):5–51, 2002.

[51] D. Harel and R. Marelly. *Come, Let's Play - Scenario-Based Programming Using LSCs and Play-Engine*. Springer, 2003.

[52] D. Harel and R. Marelly. *Play-Engine User's Guide*, 2003.

[53] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[54] Jifeng He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[55] X. He. PZ nets a formal method integrating Petri nets with Z. *Information & Software Technology*, 43(1):1–18, 2001.

[56] K. M. V. Hee, editor. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, 1994.

[57] M. Heiner and M. Heisel. Modeling safety-critical systems with Z and Petri nets. In *International Conference on Computer Safety, Reliability and Security, LNCS, Springer*, pages 361–374, 1999.

[58] B. Henderson-Sellers. *A Book of Object-Oriented Knowledge*. Object Oriented Series. Prentice-Hall, 1992.

[59] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[60] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes, Data and Time. In Butler et al. [16].

[61] J. Hogg. Islands: Aliasing Protection In Object-Oriented Languages. In *Proc. 6th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, pages 271–285, 1991.

[62] I. Horrocks. The FaCT system. *Tableaux'98, Lecture Notes in Computer Science*, 1397:307–312, 1998.

[63] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, 2004.

[64] ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.

[65] D. Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Available: http://sdg.lcs.mit.edu/alloy/book.pdf (an early version has been published in TOSEM Vol-11), 2002.

[66] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering: ICSE'2000*, pages 730–733, Limerick, Ireland, June 2000. ACM Press.

[67] C. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice-Hall, 1986.

[68] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach*. Object Oriented Series. Prentice-Hall, 1994.

[69] I. Kruger. Modeling and Synthesis with MSC Extensions for Broadcasting, Overlapping, Preemptive, and Triggered Collaborations. In *Workshop on Scenarios and State Machines at ICSE 2003*, 2003.

[70] K. Lano. Z++. In *Proc. 1990 Z User's Meeting*, Oxford, December 1990.

[71] K. Lano and H. Haughton. A Comparative Description of Object-oriented Specification Language. In K. Lano and H. Haughton, editors, *Object Oriented Specification Case Studies*, Object Oriented Series. Prentice-Hall, 1993.

[72] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.

[73] O. Lassila and R. R. Swick (editors). Resource description framework (rdf) model and syntax specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, Feb, 1999.

[74] R.H. Liffers. Inheritance versus Containment. *ACM SIGPLAN Notices*, 28(9):36–38, 1993.

[75] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6:11–44, Jan 1995.

[76] C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff. Hol-Z in the UniForM-workbench – a case study in tool integration for Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lect. Notes in Comput. Sci.*, pages 116–134. Springer-Verlag, 1998.

[77] Nancy A. Lynch and Frits W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[78] B. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In J. Bowen, A. Fett, and M. Hinchey, editors, *The 11th International Conference of Z Users*, volume 1493 of *Lecture Notes in Computer Science*, pages 308–327, Berlin, Germany, September 1998. Springer-Verlag.

[79] B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., pages 1166–1185, Toulouse, France, September 1999. Springer-Verlag.

[80] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[81] B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

[82] B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, University of Queensland, 1991. Available as `ftp://ftp.it.uq.edu.au/pub/Thesis/brendan_mahony.ps.Z`.

[83] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Press.

[84] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proceedings of OOPSLA'02*, pages 83–100, 2002.

[85] S. Mauw and M. A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[86] S.L. Meira and A.L.C. Cavalcanti. Modular object-oriented z specifications. In *Proc. 1990 Z User's Meeting*, pages 173–192, December 1991.

[87] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.

[88] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[89] R. Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, 1999.

[90] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[91] O. Nierstrasz. Composing Active Objects — The Next 700 Concurrent Object-Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 151–171. MIT Press, 1993.

[92] J. Odell. Managing object complexity, part II: composition. *Journal of Object-Oriented Programming*, 5(6):17–20, 1992.

[93] K. Periyasamy and V.S. Alagar. Adding Real-Time Filters to Object-Oriented Specification of Time Critical Systems. In *the 1998 IEEE Workshop on Industrial-strength Formal specification Techniques*, Boca Raton, Florida, USA, October 1998. IEEE Press.

[94] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM Symposium Principles of Programming Languages (POPL 1989)*, pages 179–190, 1989.

[95] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesize. In *Proceedings 31st IEEE Symposium on Foundation of Computer Science*, 1990.

[96] Ben Potter, Jane Sinclair, and David Till. *An introduction to formal specification and Z*. Prentice-Hall, 1991.

[97] S. C. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation of TCOZ in Unifying Theory of Programming. In *Proceedings of 12th Internation Symposium on Formal Methods E urope: FM'03*, Pisa, Italy, September 2003. LNCS, Springer-Verlag.

[98] A. W. Roscoe. *Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[99] J. Rumbaugh. Derived information. *Journal of Object-Oriented Programming*, 5(1):57–61, 1992.

[100] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.

[101] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*,

volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.

[102] A. Smith. On recursive free types in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 3–39, University of York, December 1991.

[103] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.

[104] G. Smith. A Logic for Object-Z (Additional Rules). Technical Report 95-26, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.

[105] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

[106] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of FME'97: Industrial Benefit of Formal Methods*, Graz, Austria, September 1997. Springer-Verlag.

[107] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[108] G. Smith and J. Derrick. Specification, Refinement and Verification of Current Systems — An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[109] G. Smith and I. Hayes. Towards Real-Time Object-Z . In Araki et al. [5].

[110] G. Smith, F. Kammuller, and T. Santen. Encoding object-z in isabelle/hol. In *2nd International Conference of Z and B Users (ZB'02)*, LNCS. Springer, 2002.

[111] M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proceedings of Workshop on Real-time Tools*. Aalborg, August 2001.

[112] J.M. Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Comput. Sci.* Cambridge University Press, UK, 1988.

[113] J.M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.

[114] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[115] C. Suhl. RT-Z: An integration of Z and timed CSP. In Araki et al. [5].

[116] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Press.

[117] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *Proceedings of the 7th Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 546–562. Springer, 1996.

[118] UML Group. OMG UML Version 1.5. http://www.uml.org/, June 2002.

[119] World Wide Web Consortium (W3C). Extensible markup language (xml). http://www.w3.org/XML.

[120] World Wide Web Consortium (W3C). Xml schema. http://www.w3.org/XML/Schema.

[121] H. Wang, J. S. Dong, and J. Sun. Reasoning Support for SWRL-FOL Using Alloy. In *17th International Conference on Software Engineering and Knowledge Eng ineering (SEKE'05)*, July 2005.

[122] X. Wang, D. Zhang, J. S. Dong, C. Chin, and S. Hettiarachchi. Semantic Space: An Infrastructure for Smart Spaces. *IEEE Pervasive Computing*, 3(3):32–39, July 2004.

[123] J. Warmer and A. Kleppe. *The object constraint language: precise modeling with UML*. Addison-Wesley, Boston, MA, USA, 1999.

[124] P. H. Welch, J. R. Aldous, and J. Foster. CSP Networking for Java (JCSP.net). In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (2)*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer, 2002.

[125] A. Wills. Capsules and types in Fresco. In P. America, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lect. Notes in Comput. Sci.*, pages 59–76. Springer-Verlag, 1991.

[126] K. Winter and R. Duke. Model Checking Object-Z Using ASM. In Butler et al. [16], pages 165–184.

[127] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *2nd International Conference on Z and B*, volume 2272 of *Lect. Notes in Comput. Sci.*, pages 184–203. Springer-Verlag, 2002.

[128] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[129] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *the Sixth Annual Z User Meeting, York, UK.*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.

[130] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

[131] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Software Engineering and Methodology*, 6(1):1–30, January 1997.

[132] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

Notation	Explanation
c : **chan**	declare c to be a channel
a : **actuator**	declare a to be a actuator
s : **sensor**	declare s to be a sensor
$\bot$	divergent process
STOP	deadlocked process
SKIP	terminate immediately
WAIT t	delay termination by t
$a \rightarrow P$	communicate a then do P
$a@t \rightarrow P$	communicate a at time t then do P
$[t : \mathbb{T}] \bullet a@t \rightarrow P$	record time of a event in variable t
$c.a$	communicate a on channel c
$c?a$	input a on channel c
$c!a$	output a from channel c
$[b] \bullet P$	enable P only if b
$P;\ Q$	perform P until termination, then perform Q
$P \Box Q$	perform the first enabled of P and Q
$[i : I] \bullet P$	perform P with first enabled value of i (indexed external choice)
$P \sqcap Q$	perform either of P and Q
$[i! : I];\ P$	perform P with any value of i (indexed internal choice)
$v := e$	syntactic sugar for $[\Delta v \mid v' = e]$
$P \setminus A$	hide the events A from the environment of P
$P \,\|[\, A \,]\|\, Q$	synchronise P and Q on events from A

continued on next page

Notation	Explanation
$(\!\!\parallel p_1, \ldots, p_n \bullet \ldots;\ p_i \overset{A}{\longleftrightarrow} p_j;\ \ldots)$	network topology abstraction with parameters $p_1, \ldots, p_n$ and network connections including p_i communicating with p_j on private channels from A
$P \mid\mid\mid Q$	P and Q running without sychronisations
$P \triangleright\{t\}\ Q$	if P does not begin by time t, perform Q instead
$P \swarrow\{t\}\ Q$	perform P until time t, then transfer control to Q
$P \bigtriangledown\ e \rightarrow Q$	perform P until exception e, then transfer control to Q
$P \bullet \textsc{Deadline}\ t$	behaviours of P which terminate before time t
$P \bullet \textsc{WaitUntil}\ t$	after P idle until time t