TOOLS AND VERIFICATION TECHNIQUES FOR INTEGRATED FORMAL METHODS

SUN JING

(B.Sc. Nanjing University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgement

I am deeply indebted to my supervisor, Dr. Jin Song DONG, for his guidance, insight and encouragement throughout the course of my doctoral program and for his careful reading of and constructive criticisms and suggestions on drafts of this thesis and other works.

I would like to thank Dr. Martin Henz for his help on the better understanding of the Oz language. I owe thanks to Wang Hai, other office-mates and friends for their help, discussions and friendship. I would like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis. I would also like to thank Hugh Anderson for his comments on the English language of the thesis.

This study received financial support from the National University of Singapore. The School of Computing also provided the finance for me to present papers in several conferences overseas. In addition, I have been encouraged by receiving the Dean's Graduate Award 2001, the University Graduate Fellowship 2002 and the President's Graduate Fellowship 2003. For all this, I am very grateful.

I sincerely thank my parents Sun Lin Ping and Shen Jin Li for their love, encouragement and financial support in my years of study. Finally, I wish to express my love and thanks to my girl friend Yin Yin for her continuing love, patience, and understanding.

Contents

Summary

Formal techniques have been applied to the specification of software and system requirements. The well-defined semantics and syntax of formal specification languages make them suitable for precisely capturing and formally verifying system requirements. Integrated Formal Methods (IFM) combine different formalisms to capture the static and dynamic system properties in a highly structured way. Timed Communicating Object Z (TCOZ), builds on the strengths of Object-Z in modeling complex data and state with the strengths of Timed CSP in modeling real-time interactions, is potentially a good candidate for specifying composite systems. One weakness of IFM is the lack of tool support and connections to current industrial practice. This thesis demonstrates a series of developments intended to enhance tools and verification support for one of the IFM – TCOZ formal specification language. First, a customized markup language for a family of Z notations - ZML has been defined using the eXtensible Markup Language (XML) and serves as a standard interchange format between the various TCOZ support tools. Second, a web environment for browsing Z family specifications has been developed using XML and eXtensible Stylesheet Language (XSL) technology. Third, an executable semantics of TCOZ in a multi-paradigm programming language, Oz, has been defined for the animation of TCOZ models. Fourth, a combination and extension of state and event based proof systems has been established for formal reasoning about TCOZ specifications. In addition, a framework for the shallow embedding of TCOZ inference rules into the theorem prover Isabelle was presented to support

automatic proof assistance. The Z family web environment provides various browsing facilities such as auto type referencing, static syntax checking, Z schema calculus and Object-Z/TCOZ inheritance expansions. The idea for putting Z family on the web may create a new culture for constructing formal specifications as well as the education and resource sharing of formal methods through the Internet. The animation of TCOZ specifications in Oz provides an effective way of validating the consistency between a formal model and its real world requirements. The extension of Object-Z and TCSP's proof systems in TCOZ provides a rigorous reasoning system for TCOZ specifications. In addition, a formal reasoning of a three-layered Computer Aided Dispatch (CAD) system properties is demonstrated as a case study. Furthermore, the framework for a shallow embedding of TCOZ inference rules in the theorem prover Isabelle illustrates an automatic proof assistant to the TCOZ language. In summary, with the above tool support and verification techniques, TCOZ can be a potential candidate for industrial software engineering practice.

List of Figures

Chapter 1

Introduction and overview

1.1 Motivation and goals

Software engineering involves the design, implementation and maintenance of large software systems. It is unique among the engineering disciplines in that verifications are required as an essential part of professional practice. In order to show the faultlessness of the system design the first thing is to understand the requirements correctly. Requirement capture is a key activity in software and system engineering. A rapid increase in terms of size and complexity of software systems has led to a rising demand for high quality in the system analysis stage, which would reduce the cost of removing errors later in the software life cycle. Traditionally, requirements are specified textually in natural language, or by using hand-waving diagrammatical notations. However, requirements in this way are informal and imprecise, and tend to cause misunderstandings among clients, software designers and developers. Furthermore, such requirements may be inconsistent or incomplete since there is no way to formally verify their consistency. As a result, mathematical and logical approaches have been proposed to define better requirement specifications. Formal methods are well known for their preciseness and expressiveness in specifying software and system requirements [17, 18, 24, 43, 55]. Many formal specification languages have been proposed to accommodate various aspects and views. For example VDM [3], Z [92], Object-Z [88], and B [4] are state-oriented formalisms; ACT1 [28], CLEAR [11], OBJ [35], and Larch [47] are algebraic formalisms and CSP [44]; TCSP [82], CCS [48], and LOTOS [51] are process-oriented formalisms. The well-defined semantics and syntax of formal specification languages make them suitable for precisely capturing and formally verifying system requirements. In addition, there are some well developed tools to support the use of such formal notations, such as Z/EVES [13], Alloy [75], PVS [77], SPIN [46], FDR [60], UPPAAL [80] and so on. However, the design of complex systems requires powerful mechanisms for modeling data, state, communication, and real-time behavior; as well as for structuring and decomposing systems in order to control local complexity. One current research focus is on combining state based and event based formalisms, and many approaches have been reported at recent conferences on formal methods, i.e., IFM'02 [12, 36]. Integrated formal methods (IFM) combine different formalisms to capture the static and dynamic system properties in a highly structured way. One of these approaches, Timed Communicating Object Z (TCOZ) [67] builds on the strengths of Object-Z [25, 88] in modeling complex data and state with the strengths of TCSP [82, 83] in modeling real-time interactions.

TCOZ extended the inherited CSP's channel based communication mechanism, in which messages represent discrete synchronization between processes, to continuous function interface mechanisms inspired by process control theory: the sensor and actuator [66]. With such mechanisms TCOZ is capable of specifying both synchronous and asynchronous communication interactions of composite systems [30]. The current shortcoming of IFM on TCOZ in particular is the lack of tool and theoretical support and its connection to the current industrial best practice. The aim of this thesis is to provide various tool environments and verification techniques to the TCOZ formal specification language to enhance its practical usage. Four main areas of work will be addressed in the thesis: the Z family Markup Language (ZML) [1], the ZML web environment, the Oz animation environment, and TCOZ proof techniques and semantic embedding. These areas range from simple (light-weight) tools, through to more structured, complex and developed (heavy-weight) tools.

1.2 Thesis outline and overview

The structure of the thesis is as follows.

[1]Our ZML includes the Z/Object-Z/TCOZ languages. Recently we also participated in Utting's paper on "ZML: XML support for standard Z", which will appear in ZB2003.

1.2.1 Chapter 2

This chapter is devoted to an overview of the Z family languages, particularly the TCOZ integrated formal methods language. Z and CSP are two well known formal notations with their respective user groups. Recently there has been active investigation of the integration [31, 67, 89] of formal object-oriented methods (e.g. Object-Z [25, 88]) with process description languages (e.g. CSP [44]). One such approach, the Timed Communicating Object Z (TCOZ) [67] combines Object-Z's strengths in modelling complex data and state with TCSP's strengths in modelling real-time concurrency. The TCOZ communication interfaces, i.e., channel, sensor and actuator, are well suited for capturing communication between components. The introduction of a novel network topology operator allows the communications interfaces of complex processes to be visualized through simple network-topology graphs. This improves decoupling of class definitions by simplifying the interfaces between objects, thereby enhancing the modularity of system specifications. In this chapter we give a brief overview of the various aspects of TCOZ. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [68]. The formal semantics of TCOZ is also documented [65].

1.2.2 Chapter 3

This chapter presents an XML [101] approach to define a customized markup language for the Z family notations (Z/Object-Z/TCOZ). For a single formal notation there may exist many kinds of support tools for different usages, i.e., model constructing tools, animation tools, proof supporting tools, etc. Such tools demand a standard interchangeable common format among them. EXtensible Markup Language (XML) is a subset of the Standard Generalized Markup Language (SGML). It was designed to describe customized document structure. It is strongly believed that XML will become the most common tool for all data manipulation and data transmission. Thus a customized markup language for a particular formal language can be defined using XML technology. In this chapter, we present a Z Markup Language (ZML) for the Z family notations using W3C XML Schema [106].

1.2.3 Chapter 4

This chapter presents the development of a web environment for ZML and their projections to UML Diagrams. The World Wide Web (WWW) is a promising environment for software specification and design because it allows sharing design models and providing hyper textual links among the models [52]. Unified Modeling Language (UML) [81] is commonly regarded as one of the dominant graphical notations for industrial software system modeling. It is important to develop links and tools from FM to WWW and to UML so that FM technology transfer can be successful. In this chapter, we demonstrate the use of the eXtensible Stylesheet Language (XSL) [102] to develop a web environment that provides various browsing and syntax checking facilities for Z family languages; and a transformation tool for projecting TCOZ specifications (in ZML) to UML (in XMI).

1.2.4 Chapter 5

This chapter presents the development of an animation environment for the TCOZ notation. Specification animation plays an important role of validating the consistency between the formal model and the real world informal requirements. Even given the correctness of a formal specification, there may still be a gap between the formal model and the real world informal requirements. If the formal model does not truly reflect the real world requirements, it is useless to further verify its correctness. The purpose of animation is to exhibit the dynamic properties of a specification, and to bridge the gap between the real world problem and our interpretation of the informal requirements. In this chapter, we define executable semantics of TCOZ in a multi-paradigm programming language - Oz [39, 91, 42] for the animation of TCOZ models.

1.2.5 Chapter 6

This chapter presents a proof system for formally reasoning about TCOZ specifications. Based on TCSP semantics [82, 83], the denotational semantics of TCOZ has been developed [65]. However, in order to formally verify system properties, a proof system for TCOZ is needed. TCOZ preserves a large part of both the syntax and semantics of the base notations. Hence it can potentially benefit from existing reasoning systems of the individual notations. In this chapter we extend and link Smith's proof system of Object-Z [86] and Davies/Schneider's proof system of TCSP [82, 83] for reasoning about TCOZ models. The new proof rules for the TCOZ novel constructs, i.e., sensor/actuators, active objects, network topology, deadline and wait-until commands, etc., are developed in this chapter. Furthermore, a framework for encoding the inference rules into the theorem prover Isabelle/HOL is presented for automatic proof assistance of the TCOZ language.

1.2.6 Chapter 7

This chapter presents the formal verification process of a three-layered Computer Aided Dispatch (CAD) System generic architecture as a case study. Critical system properties are decomposed and proved by applying the inference rules presented in chapter 6. In addition, it also confirms that TCOZ could be useful a potential candidate of Architecture Description Language (ADL) for the specification of software architecture models.

1.2.7 Chapter 8

Chapter 8 concludes the thesis with a summary of the main contributions of this thesis, and some suggestions for further research.

1.3 Publications from the thesis

Most chapters of the thesis have been accepted in international refereed journals or conference proceedings. Chapter 3 has been presented at *The Tenth Interna-*

tional World Wide Web Conference (WWW-10, May 2001) [94] and it is used as a basis for the paper accepted by the *The Third Z and B International Conference (ZB2003, June 2003)* [99]. Chapter 4 has been published in the thirteenth volume of the *Annals of Software Engineering journal (ASE, June 2002)* [96] and *The Fourth International Conference on Formal Engineering Methods (ICFEM'02, October 2002)* [20]. Chapter 5 was presented at *Eighth Asia-Pacific Software Engineering Conference (APSEC'01, December 2001)* [93]. Chapter 6 and 7 were presented at the *Ninth Asia-Pacific Software Engineering Conference (APSEC'02, December 2002)* [95] and *The Tenth IEEE International Workshop on Software Specification and Design (IWSSD'00, November 2000)* [58]. In addition, partial contributions have been made to the ongoing research work on Semantic Web as noted in chapter 8, which were published at *The Eleventh International Formal Methods Europe Symposium (FME'02, July 2002)* [21] and *The Fourth International Conference on Formal Engineering Methods (ICFEM'02, October 2002)* [22].

Chapter 2

Background

This chapter sets the context for the later chapters, giving notations used and brief technical outlines of relevant Z family languages, in particular TCOZ features.

2.1 Z/Object-Z/TCSP overview

In this section, we will use a simple message queue system to give a brief introduction to the Z, Object-Z, TCSP and TCOZ notations.

2.1.1 Z

Z [92] is a formal specification language based on set theory and predicate logic. 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. Complex schema definitions can be composed from the simple ones by schema calculus. Z has been widely adopted to specify a range of software systems (see [40]). Various tools, i.e. editors, type/proof checkers and animators, for Z have been developed.

Consider the Z model of a FIFO message queue. Let the given type *MSG* represent a set of messages. The notation for this is:

$[MSG]$ [messages]

The queue contains operations to add elements to, and delete elements from, the queue. The total elements in the queue cannot be more than *max* (say, a number larger than 100). The global constant *max* can be defined using the Z axiomatic definition as:

$$\begin{array}{|l}
max : \mathbb{N} \\
\hline
max > 100
\end{array}$$

The state, potential state change and initial state of the queue system can be specified in Z as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, Queue\, \_\_\_\_\_\_\_\_\_\_} \\
items : \operatorname{seq} MSG \\
\hline
\#items \leq max \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\, QueueInit\, \_\_\_\_\_\_\_\_} \\
Queue \\
\hline
items = \langle\, \rangle \\
\hline
\end{array}$$

The operations to add messages to, and delete messages from, the queue can be modelled as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, Add\, \_\_\_\_\_\_\_\_\_\_} \\
\Delta Queue \\
item? : MSG \\
\hline
items' = items \frown \langle item? \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Delete\, \_\_\_\_\_\_\_\_\_\_} \\
\Delta Queue \\
item! : MSG \\
\hline
items \neq \langle\, \rangle \\
items = \langle item! \rangle \frown items' \\
\hline
\end{array}$$

More complex operations can be constructed by using schema calculus, e.g., a new message which pushes out an old message, say *Penguin*, can be specified by using the sequential composition schema operator $\fatsemi$ as:

$$Penguin \mathrel{\widehat=} Add \fatsemi Delete$$

which is an (atomic) operation with the effect of a *Add* followed by a *Delete*. Other forms of schema calculus include schema conjunction ‘ $\wedge$ ’, disjunction ‘ $\vee$ ’ implication ‘ $\Rightarrow$ ’, negation ‘ $\neg$ ’ and pipe ‘ $\gg$ ’, which have been discussed in many Z text books [92, 113].

2.1.2 Object-Z

Object-Z [25] 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. Object-Z has a type checker, but other tool support for Object-Z is limited in comparison to Z. The essential extension to Z in Object-Z is the *class* construct which groups the definition of a state schema with 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.

Consider the following specification of the *Queue* system in Object-Z.

Queue

$items : \mathrm{seq}\, MSG$

$\# items \leq max$

INIT

$items = \langle\ \rangle$

Add

$\Delta(items)$
$item? : MSG$

$items' = items \frown \langle item? \rangle$

Delete

$\Delta(items)$
$item! : MSG$

$items \neq \langle\ \rangle$
$items = \langle item! \rangle \frown items'$

Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value. The standard behavioral interpretation of Object-Z objects is as transition systems [87]. A behavior of a transition system consists of a series of state transitions each effected by one of the class operations. A *Queue* object starts with *items* empty then evolves by successively performing either *Add* or *Delete* operations. Operations in Object-Z are atomic, only one may occur at each transition, and there is no notion of time or duration. It is difficult to use the standard Object-Z semantics to model a system composed by multi-threaded component objects whose operations have duration.

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 $item?$ to the existing set provided the sequence has not already reached its maximum size (an identifier ending in '?' denotes an input). Operation *Delete* outputs a value $item!$ defined as one element of *items* and reduces *items* by deleting the last one from the original queue (an identifier ending in '!' denotes an output).

2.1.3 TCSP

TCSP [82] extends the well known CSP (Communicating Sequential Processes) notation of Hoare [44] with timing primitives. CSP is an event based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or communication) between processes. TCSP extends

CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization. New timing constructs such as timed prefix, timeout, delay, timed interrupt, etc., are introduced to capture the requirements related to a timed aspect. For instance, the timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline. The process

$$(a \rightarrow P) \triangleright \{t\}\ Q$$

will try to perform $(a \rightarrow P)$, but will pass control to Q if the event a has not occurred by time t , as measured from the invocation of the process.

A *Leave* process of the *Queue* example in TCSP can be constructed as follows.

$$\begin{aligned} \mathit{Queue}_{\mathit{Leave}}(\mathit{items}) = {} & \mathit{out}!\mathit{head}(\mathit{items}) \rightarrow \\ & ((\mathit{ack} \rightarrow \mathit{Delete}) \triangleright \{5\}\ \mathit{Queue}_{\mathit{Leave}}(\mathit{items})) \end{aligned}$$

It states that the *Leave* process will output the first element in the queue every 5 time units until an acknowledge message *ack* is received.

The language semantics of TCSP is based on considering a processes P as a set of timed failures ($\mathcal{TF}[\![P]\!]$), which represent the records of executions. A timed failure consists of timed traces and timed refusals. A timed trace contains the information about events performed according to their timing aspects, while a timed refusal contains the set of timed events which are refused by the execution. Timed failure semantics precisely capture the observation of an process execution. For example, one of the timed failures for the process $\mathit{Queue}_{\mathit{Leave}}(\mathit{items})$ could be:

$$(\langle(1, \mathit{out}.\mathit{head}(\mathit{items})), (3, \mathit{ack})\rangle, [1, 3) \times \{\mathit{ack}\})$$

It denotes one possible execution of the process that performs the output at time one and receives the acknowledgement at time three, while the refusal period for the *ack* event is between one and three. The $\mathcal{TF}[\![Queue_{Leave}(items)]\!]$ is a collection of all such executions.

2.2 TCOZ features

Timed Communicating Object Z (TCOZ) [67] is essentially a blending of Object-Z [26] with Timed CSP [82], 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 [68]. The formal semantics of TCOZ is also documented [65].

2.2.1 A model of time

In TCOZ, all timing information is represented as real valued measurements in *seconds*, the SI standard unit of time [49]. We believe that a mature approach to measurement and measurement standards is essential to the application of formal techniques to systems engineering problems. In order to support the use of standard units of measurement, extensions to the Z typing system suggested by Hayes and Mahony [41] are adopted. Under this convention, time quantities are represented by the type

$$\mathbb{T} == \mathbb{R} \odot \mathsf{T},$$

where $\mathbb{R}$ represents the real numbers and T is the SI symbol for dimensions of time. Time literals consist of a real number literal annotated with a symbol representing a unit of time. All the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

2.2.2 Interface – channels, sensors and actuators

CSP channels are given an independent, first class role in TCOZ. In order to support the role of CSP channels, the 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. Contrary to the conventions adopted for internal state attributes, channels are viewed as shared (global) rather than as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects. The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby enhancing the modularity of system specifications.

As a complement to the synchronizing CSP channel mechanism, TCOZ also adopts a non-synchronizing shared variable mechanism. 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 synchronization). The shift from closed to open systems necessitates close attention to issues of control, an area where both Z and CSP are weak [115]. We believe that TCOZ with the **actuator** and **sensor** can be a good candidate for specifying open control systems. Mahony and Dong [66] presented detailed discussion on TCOZ sensor and actuators.

2.2.3 Active objects

Active objects have their own thread of control, while passive objects are controlled by other objects in a system. In TCOZ, an identifier MAIN (indicating a non-terminating process) is used to represent the behavior of active objects of a given class [19]. The MAIN operation is optional in a class definition. It only appears in a class definition when the objects of that class are active objects. Classes for defining passive objects will not have the MAIN definition, but may contain CSP process constructors. If ob_1 and ob_2 are active objects of the class C, then the independent parallel composition behavior of the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\textsc{Main} \mid\mid\mid ob_2.\textsc{Main}$

2.2.4 Semantics of TCOZ

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [65]. 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 termination event $\checkmark$ 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.

2.2.5 Network topology

The syntactic structure of the CSP synchronization 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 [64]. 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. One CSP expression for such a network communication system is

$$\begin{array}{l}(A[bc'/bc] \mid[\, ab, ac \,]\mid (B[ac'/ac] \mid[\, bc \,]\mid C[ab'/ab]) \setminus ab, ac, bc) \\ \quad [ab, ac, bc/ab', ac', bc']\end{array}$$

The hiding and renaming is necessary in order to cover cases such as C being able to communicate on channel ab. The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology. This network topology of A, B and C may be described by

$$\|(A \xleftrightarrow{ab} B;\ B \xleftrightarrow{bc} C;\ C \xleftrightarrow{ca} A).$$

Other forms of usage allow network connections with common nodes to be run together, for example

$$\Vert(A \xleftrightarrow{ab} B \xleftrightarrow{bc} C \xleftrightarrow{ca} A),$$

and 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

$$\Vert(D \xleftrightarrow{df_1, df_2} F).$$

The syntactic implication of the above 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. For instance, an active Queue can be derived from the previous (Object-Z) *Queue* model as:

ActiveQueue

Queue

$t_j, t_l : \mathbb{T}$ [durations for Join/Leave operations]
$in, out : \textbf{chan}$ [channels for input and output]

$Join \mathrel{\widehat=} [item : MSG \mid \#items < max] \bullet in?item \rightarrow Add \bullet \textsc{Deadline}\, t_j$
$Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet out!head(items) \rightarrow Delete \bullet \textsc{Deadline}\, t_l$
$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet Join \,\Box\, Leave;\ Q$

where the TCOZ DEADLINE command is used to constrain the *Join* and *Leave* to be finished within their duration time.

As we can see that Object-Z and TCSP complement each other not only in their expressive capabilities, but also in their underlying semantics. Object-Z is an excellent notation for modeling data and states, but difficult for modeling real-time and concurrency. TCSP is good for specifying timed process and communication, but like CSP, cumbersome to capture the data states of a complex system. The combination of the two, TCOZ, treats data and algorithmic aspects in the Object-Z style and treats process control, timing, and communication aspects in the TCSP style. In addition, the object oriented flavor of TCOZ provides an ideal foundation for promoting modularity and separation of concerns in system design. With the above modeling abilities, TCOZ is potentially a good candidate for specifying composite systems in a highly constructed manner.

Chapter 3

Z family Markup Language – ZML

In this chapter, we present an XML approach to define a customized markup language for the Z family notations.

3.1 Introduction

Standard interchange format is important for various tool environment that share a common language. In this way tool developers can work in an open-source spirit, with the aim both of promoting interoperability and avoiding duplicate efforts. In this chapter we present the design and definition of an interchange format for the Z family languages.

3.2 Formal design model of ZML

In general, the requirement for a Z family interchange format is that it should be structured, complete and compact. The construction of such a format must start with formalizing the related syntax definitions of the Z family languages. The typing and dynamic semantics issues are not of concern here since our aim at the moment is to focus on syntax checks. Therefore, the static and dynamic semantics of Z family languages were deliberately left out in the following model. Pure Z notation can be used as the meta notation for the formal design of such a format. However, Object-Z is superior because it can construct a more compact and reusable design model. The Object-Z design model can be more easily extended when a new notation is considered to be included. TCOZ is more suited for modeling timed/concurrent interactive systems, and perhaps it is an overdo for the design even though Z family languages are computationally complex, when dealing with schema calculus and inheritance expansions.

Firstly, the character sets are defined by a Z free type definition as:

$$\mathit{Char} ::= \text{‘a’} \mid \text{‘b’} \mid ... \mid \text{‘1’} \mid \text{‘2’} \mid ... \mid \text{‘:’} \mid \text{‘/’} \mid \text{‘\#’} \mid ...$$

The string type is defined as a sequence of characters:

$$\mathit{String} == \operatorname{seq} \mathit{Char}$$

The URL type is defined as a string starting with “*http* : //” :

$$\mathit{URL} == \{s : \mathit{String} \mid \exists s_t : \mathit{String} \bullet s = \langle \text{‘h’}, \text{‘t’}, \text{‘t’}, \text{‘p’}, \text{‘:’}, \text{‘/’}, \text{‘/’} \rangle \frown s_t\}$$

The given type *Name* contains all the valid identifiers, such as names of type, schema, class and so on. It is assumed that only alphabets and ‘_’ can appear in an identifier.

$$\mathit{Name} == \{s : \mathit{String} \mid \operatorname{ran} s \subseteq \{\text{‘a’}, \text{‘b’}, ..., \text{‘1’}, \text{‘2’}, ..., \text{‘A’}, \text{‘B’}, ..., \text{‘\_’}\}\}$$

A type declaration contains either a given type or a combination of constructors and types such as $A \times B$. The constructors include binary constructors i.e. ‘ $\rightarrow$ ’, ‘ $\nrightarrow$ ’ and unary constructors i.e. ‘ $\mathbb{P}$ ’, ‘ $\mathbb{F}$ ’.

$$\begin{aligned}\mathit{TypeConst} == \{s : \mathit{String} \mid{}& \#s = 1 \land \\ & \operatorname{ran} s \subseteq \{\text{‘}\mathbb{P}\text{’}, \text{‘}\mathbb{F}\text{’}, \text{‘}\mathbb{P}_1\text{’}, \text{‘}\times\text{’}, \text{‘}\rightarrow\text{’}, \text{‘}\nrightarrow\text{’}, ...\}\}\end{aligned}$$

Syntactically, a type constructor, a type and a predicate constructor are similar, and are defined as:

TypeConstructor

$\mathit{content} : \mathit{TypeConst}$

Type

$\mathit{name} : \mathit{Name}$

PredConstructor

$\mathit{content} : \mathit{String}$

A declaration type *Dtype* is a sequence of a class-union of type constructors and defined types, A predicate is similarly defined.

$$\begin{array}{l} Dtype == \mathrm{seq}(TypeConstructor \cup \downarrow Type) \\ Predicate == \mathrm{seq}(PredConstructor \cup \downarrow Type) \end{array}$$

where $\downarrow Type$ denotes a union of all classes defined by inheriting *Type*.

The type definition *Typedef* is for defining user given types such as simple type, abbreviation and free types. The axiom definition *Axiomdef* is used to define global constants or functions such as liberal, generic and unique functions.

Typedef

$$\begin{array}{|l} Type \\ \hline \boxed{defs : Dtype} \end{array}$$

Axiomdef

$$\begin{array}{|l} Type \\ \hline \boxed{\begin{array}{l} decpart : Name \rightarrow Dtype \\ axpart : \mathbb{P}\, Predicate \end{array}} \end{array}$$

The declaration part *decpart* is a set of pairs, where the first element of a pair is a variable name and the second is the variable's type declaration. Note that the function is used here to indicate that one variable can only have one type declaration. The axiom part *axpart* consists a set of predicates, which states the properties of a particular schema.

There are three kinds of inclusions in Z: a direct (inc) form, a Δ (del) form and a Ξ (xi) form.

$$Inclusion \; == \; \{\text{‘inc’}, \text{‘xi’}, \text{‘del’}\} \; \twoheadrightarrow \; \mathbb{P}\, Name$$

Z language has two types of schema definitions: schema box (1) and schema calculus (2).

$$Schemdef \mathrel{\widehat=} Schemadef_1 \cup Schemadef_2$$

The schema box format is defined as:

```
┌─ Schemadef₁ ─────────────────────────────
│ Type
│ ┌────────────────────────────────────
│ │ incl : Inclusion
│ │ decpart : Name → Dtype
│ │ axpart : ℙ Predicate
│ └────────────────────────────────────
└──────────────────────────────────────────
```

For the second format *Schemadef*$_2$, a type *CalcOp* is introduced to model all the possible calculus operators, and the class *CalcConstructor* is for defining a single schema calculus. A *PredCalc* can be either a *Type* or a *CalcConstructor*. Note that recursive definitions are used to capture different combinations of expressions in the schema calculus.

$$CalcOp == \{s : String \mid \#s = 1 \land \operatorname{ran} s \subseteq \{`\land\text{'}, `\lor\text{'}, `\fatsemi\text{'}, `\gg\text{'}, `\neg\text{'}, ...\}\}$$
$$PredCalc == CalcConstructor \cup \downarrow Type$$

```
┌─ CalcConstructor ─────────────     ┌─ Schemadef₂ ──────────────────
│ ┌──────────────────────────        │ Type
│ │ op : CalcOp                      │ ┌──────────────────────────
│ │ items : ℙ PredCalc               │ │ calc : PredCalc
│ └──────────────────────────        │ └──────────────────────────
└───────────────────────────────     └───────────────────────────────
```

Object-Z/TCOZ languages are mainly composed of class definitions. Firstly, we define state, initial and operation schemas as follow.

```
┌─ Statedef ────────────────────     ┌─ Initdef ─────────────────────
│ ┌──────────────────────────        │ ┌──────────────────────────
│ │ decpart : Name → Dtype           │ │ axpart : ℙ Predicate
│ │ axpart : ℙ Predicate             │ └──────────────────────────
│ └──────────────────────────        └───────────────────────────────
└───────────────────────────────
```

$$\begin{array}{|l}
\textit{Opdef} \\
\hline
\textit{Statedef} \\
\quad \begin{array}{|l}
\hline
\textit{name} : \textit{Name} \\
\textit{delta} : \mathbb{P}\ \textit{Name} \\
\hline
\end{array} \\
\hline
\end{array}$$

An Object-Z/TCOZ class is defined as:

$$\begin{array}{|l}
\textit{Classdef} \\
\hline
\textit{Type} \\
\quad \begin{array}{|ll}
\hline
\textit{inherit} : \textit{Type} \nrightarrow (\textit{Name} \rightarrow \textit{Name}) & \\
 & \text{[inherit classes with rename list]} \\
\textit{state} : \textit{Statedef} & \text{[state schema]} \\
\textit{init} : \textit{Initdef} & \text{[initial schema]} \\
\textit{ops} : \mathbb{P}\ \textit{Opdef} & \text{[operation schemas]} \\
\hline
\end{array} \\
\hline
\end{array}$$

A *ZDefinition* is either a *Typedef*, *Axiomdef*, *Schemadef* or *Classdef*. *Description* was also included as an Object-Z class for defining documentation in a formal specification.

$$\textit{ZDefinition} \mathrel{\widehat=} \textit{Typedef} \cup \textit{Axiomdef} \cup \textit{Schemsdef} \cup \textit{Classdef}$$

3.3 XML and XML Schema

Having formally specified the Z family language requirement, the next step is related to implementation. EXtensible Markup Language (XML) [101] is a powerful publishing and document interchange format meta description language. It is a subset of the Standard Generalized Markup Language (SGML) and was designed to describe customized document structures. XML has become a World Wide Web

Consortium's (W3C) recommendation in 1998. It is strongly believed that XML will be the most common tool for all data manipulation and data transmission. A customized markup language for a particular formal language can be constructed using XML technology. Thus we define a Z Markup Language (ZML) for the Z family notations using XML. The customized ZML serves as a standard interchange format among the tools. Another benefit of using XML as an input medium is its close connections with the World Wide Web (WWW), which will be addressed in the next chapter.

With the formal definitions in the previous section we can encode the Z family syntax into a customized XML document structure. The World Wide Web Consortium (W3C) has provided two mechanisms for describing XML document structures: Document Type Definition (DTD) and XML Schema [106]. They are used for checking that each component of a document occurs in a valid place within the interchanged data stream. The former (DTD) originated from the SGML recommendation and used a different syntax. The XML Schema definition language is an XML language for describing and constraining the content of XML documents. W3C XML Schema has become a W3C Recommendation in May 2001. It is going to play the role of the DTD in defining customized XML structure in the future. It is consistent with XML syntax and easier to write than DTD. In addition, the XML Schema language allows better specification of the data types of elements than the DTD language. In addition to the built-in datatypes such as string, integer, boolean, float, data time and so on, XML Schema provides mechanisms to further constrain the allowable content of an element or attribute, such as setting a valid range of values or defining a regular expression to which the content must conform. Furthermore, since XML schemas are themselves written in XML, the document descriptions are far more extensible than they were in the original DTD syntax. Declarations can have richer and more complex internal structures than declarations in DTDs. Thus XML Schemas can be stored along with other XML documents in XML-oriented data stores, referenced, and even styled, using techniques like XML Linking Language (XLink) [107], XML Pointer Language (XPointer) [109], and eXtensibe Stylesheet Language (XSL) [102]. For our purposes, we prefer to use XML schema notation to define the ZML structure syntax for the Z family notations. As a result, we can obtain a tighter specification of the structure, and can take advantage of XML tools, such as XSL Transformations (XSLT).

The reason that we chose XML rather than MathML (Mathematical Markup Language) [103] is due to its extensibility. Though MathML is rich in writing mathematical expressions, the document structure is not suitable for authoring formal specification languages such as Z/Object-Z/TCOZ. For example, the Z schema box is more difficult to construct in MathML. Furthermore, MathML expressions are heavily loaded with defined tags, obscuring the content of the expression. This is difficult for authors, whose focus is on the abstraction of the model rather than the structure of the expressions themselves. In addition, we want to construct a web environment as close as possible to the LaTeX style files for Z/Object-Z/TCOZ (fuzz.sty, oz.sty and coz.sty) so that a simple translation tool can be developed to map existing Z/Object-Z/TCOZ specifications in LaTeX to our web ZML format.

3.4 The ZML syntax definition

The formal model defined in the previous section acted as a precise design reference document and provides clear guidelines to our XML implementations. The ZML syntax structure is derived from the model and encoded into the XML Schema definition. In this section, we go through each of the major constructs of the Z/Object-Z/TCOZ notations, and briefly describe our proposed ZML structure. The XML Schema was developed and validated by the XML Spy [6] tool suite.

3.4.1 Root element definition

The Z family Markup Language mainly consists of eight types of definitions, i.e., given type, axiomatic definition, generic definition, abbreviation, free type, schema definition, class definition and predicate expressions. The ZML top level structure is depicted in Figure 3.1. The diagram is auto-generated by the XML Spy tool, and acts as a visual representation of its textual definition, assisting us in understanding the syntax structures. The corresponding W3C XML Schema text definition is as follows. It simply states the content and occurrence of each definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
```

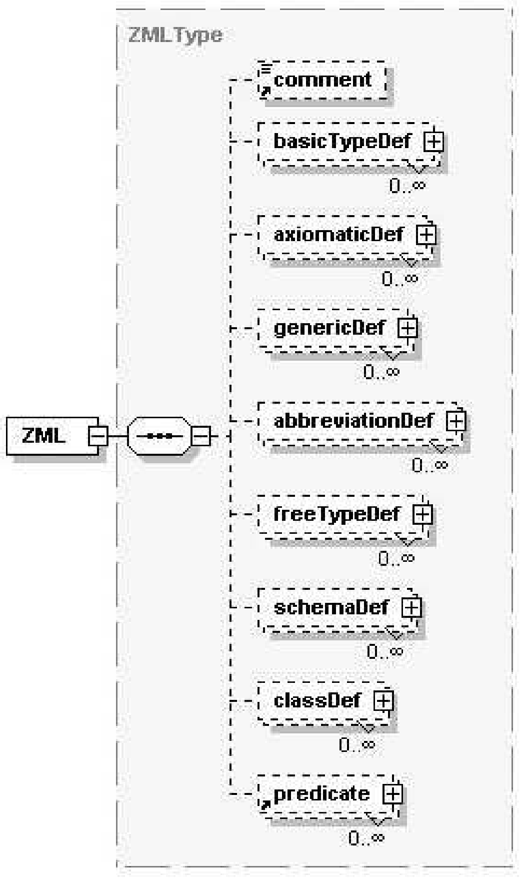


Figure 3.1: ZML top level structure

```
...
<xsd:element name="ZML" type="ZMLType"/>
<xsd:complexType name="ZMLType">
 <xsd:sequence>
  <xsd:element ref="comment" minOccurs="0"/>
  <xsd:element name="basicTypeDef" type="basicTypeDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="axiomaticDef" type="axiomaticDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="genericDef" type="genericDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="abbreviationDef" type="abbreviationDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="freeTypeDef" type="freeTypeDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="schemaDef" type="schemaDefType"
```

```
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="classDef" type="classDefType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element ref="predicate" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
 <xsd:attribute name="creator" type="xsd:string"/>
 <xsd:attribute name="date" type="xsd:date"/>
</xsd:complexType>
...
</Schema>
```

The `element` defines the markup tags for each syntax block in the Z/Object-Z/TCOZ notations. The content structure of each building block is defined as a complex type in XML Schema and is referred by the `type` definition of the element. The `occurrence` defines the possible appearance of each tag, which in this case is zero or many. In some sense, the XML Schema definition is similar to a BNF specification, although the former is more powerful and carries the data type definition as well.

3.4.2 Z related syntax definitions

A Z specification typically includes a number of given type, abbreviation, free type, axiomatic, generic and schema definitions. Their definitions are shown in Figure 3.2. Here a given type definition consists of one or more basic types. An axiomatic definition comprises of one or more variable declarations and optional predicate definitions to constrain the values. A generic definition consists of an optional formal parameter declaration, variable declarations and optional predicate constraint definitions. An abbreviation syntax introduces a new type definition

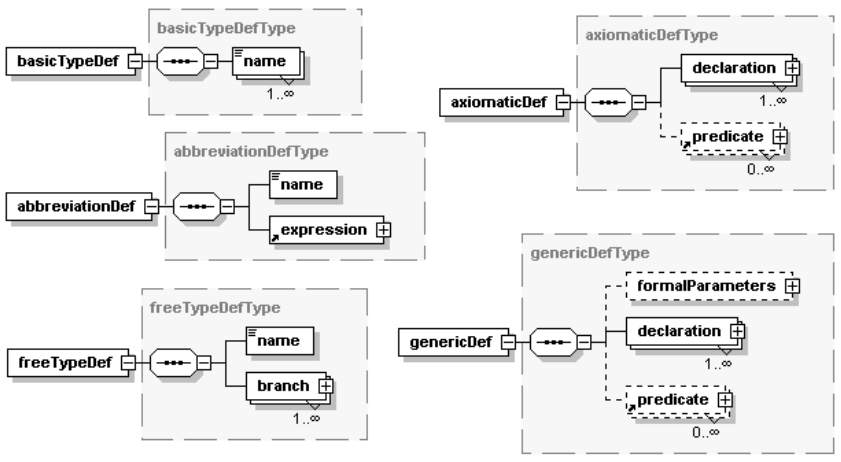


Figure 3.2: Given type, abbreviation, free type, axiomatic and generic definitions

that is the same as the type of the expression on the right. A Free type definition comprises one or more name labels and its branches, which denote a total injection from right to the new type on the left.

Schema definitions

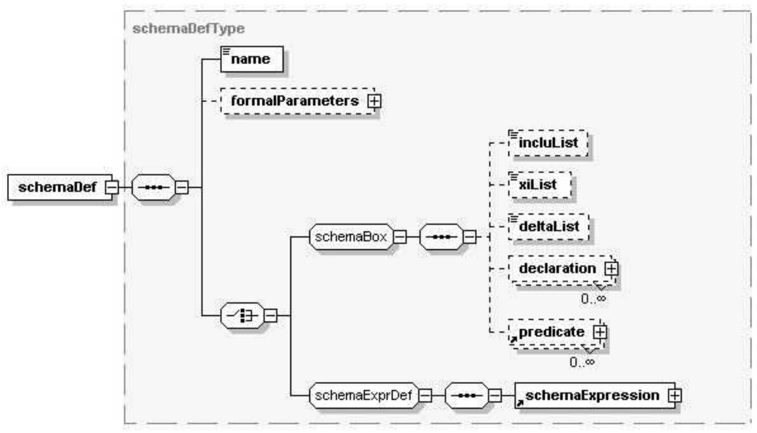


Figure 3.3: Schema definitions

The Z schema syntax consists of a name, optional generic parameters, and either schema box definitions or schema expression definitions, which are depicted in Figure 3.3. The schema box comprises an inclusion list, a Ξ-list, a Δ-list, some declarations and predicates.

The schema expression mainly includes six types of expression definitions, i.e., quantified expressions, schema text, unary schema expressions, binary schema expressions, bracket expressions and name conventions as showed in Figure 3.4. Note that here we use a recursive definition on the schema expression element.

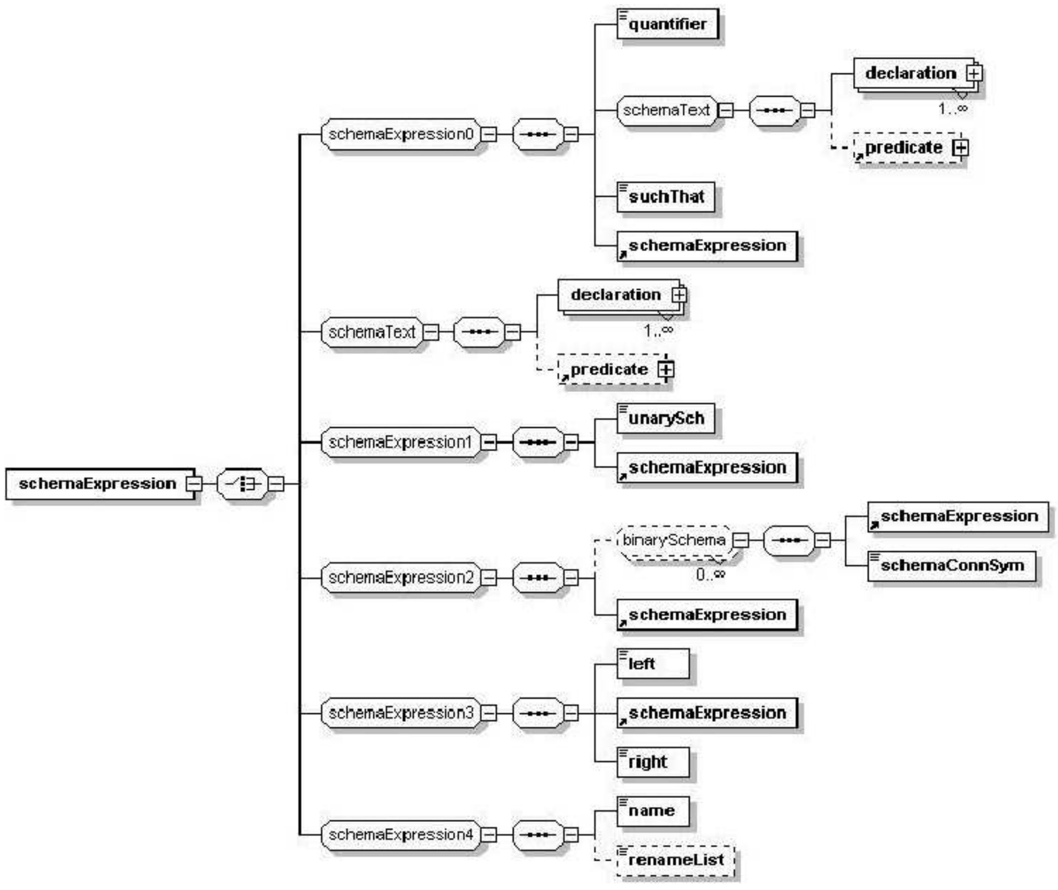


Figure 3.4: Schema expression definition

3.4.3 Object-Z related definitions

Object-Z [25] is an extension of the Z formal specification language to accommodate object orientation. 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. Syntactically, a class definition is a named box. In this box the constituents of the class are defined and related.

Class definitions

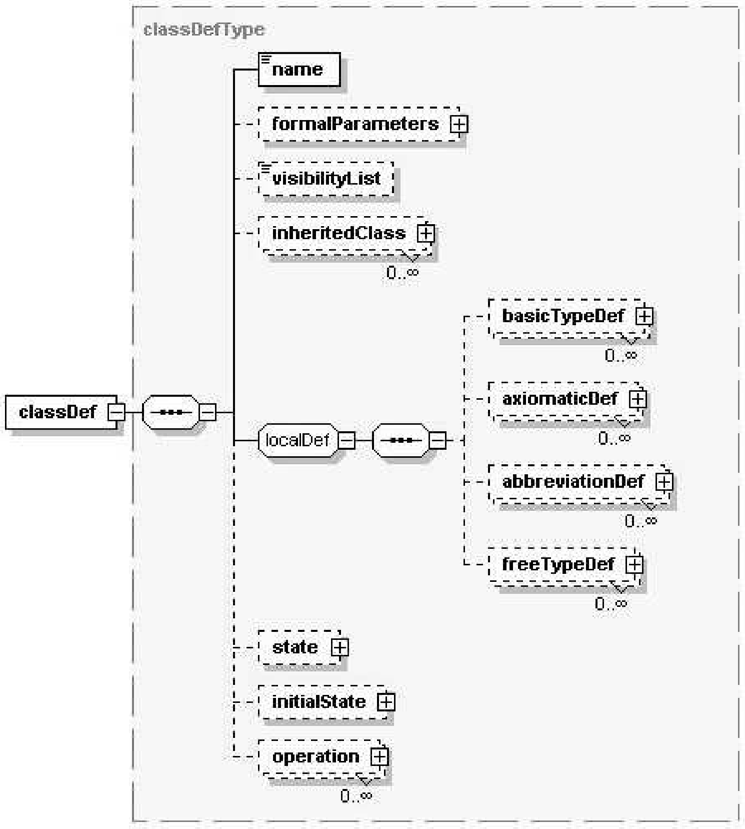


Figure 3.5: Class definitions

The *class* construct consists of a name, generic parameters, visibility list, inheritance list, local definitions, state schema definition, initial schema definition and some operation definitions, which are depicted in Figure 3.5.

An inherited class definition comprises of a class name, an optional parameter list and a rename list. An operation schema consists of a name, a choice between an

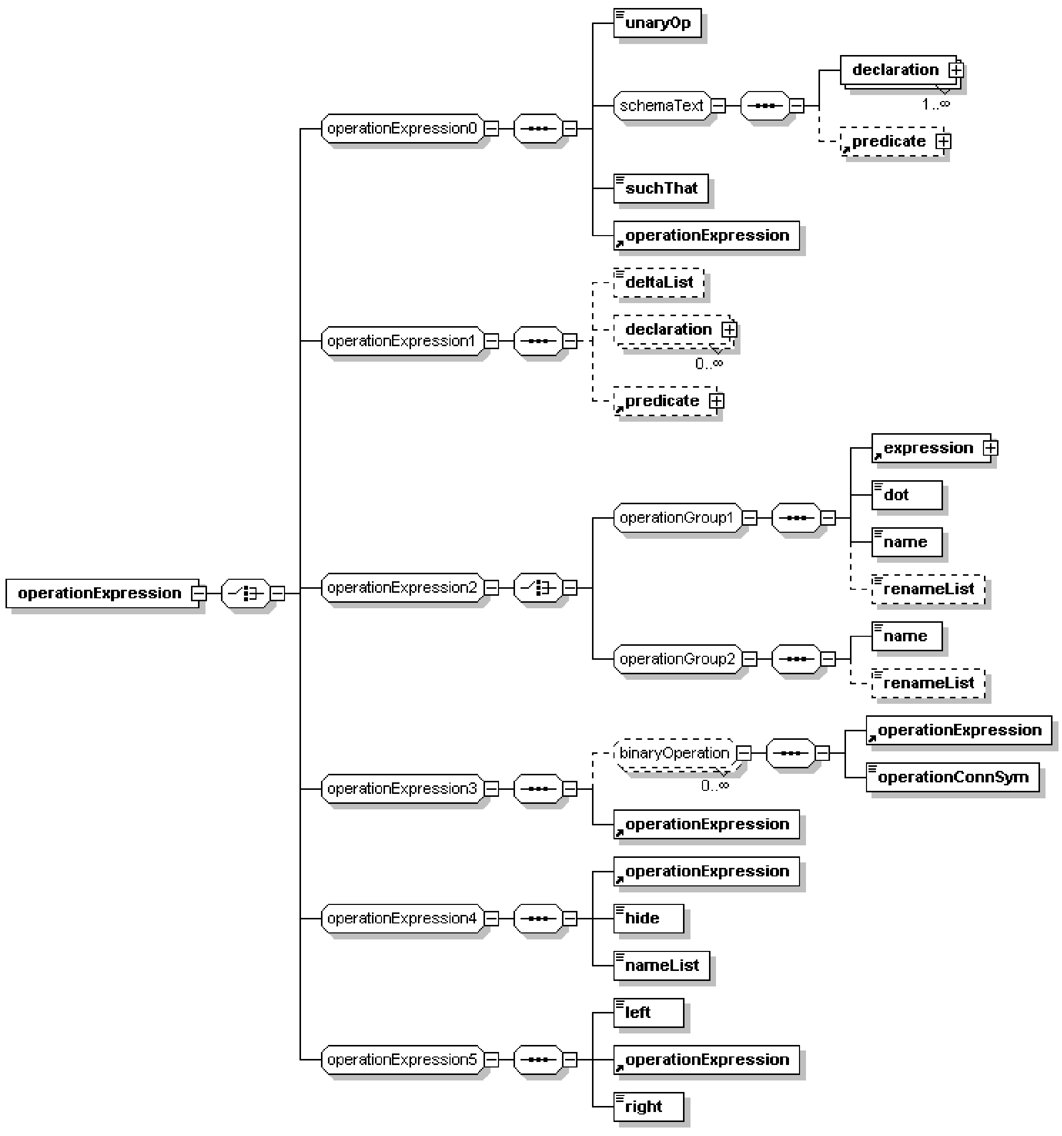


Figure 3.6: Operational expression definition

operation schema box or an operation expression definition, where the schema box contains a delta list, declarations and predicates. The structure of the operation expression is shown in figure 3.6.

3.4.4 TCSP related definitions

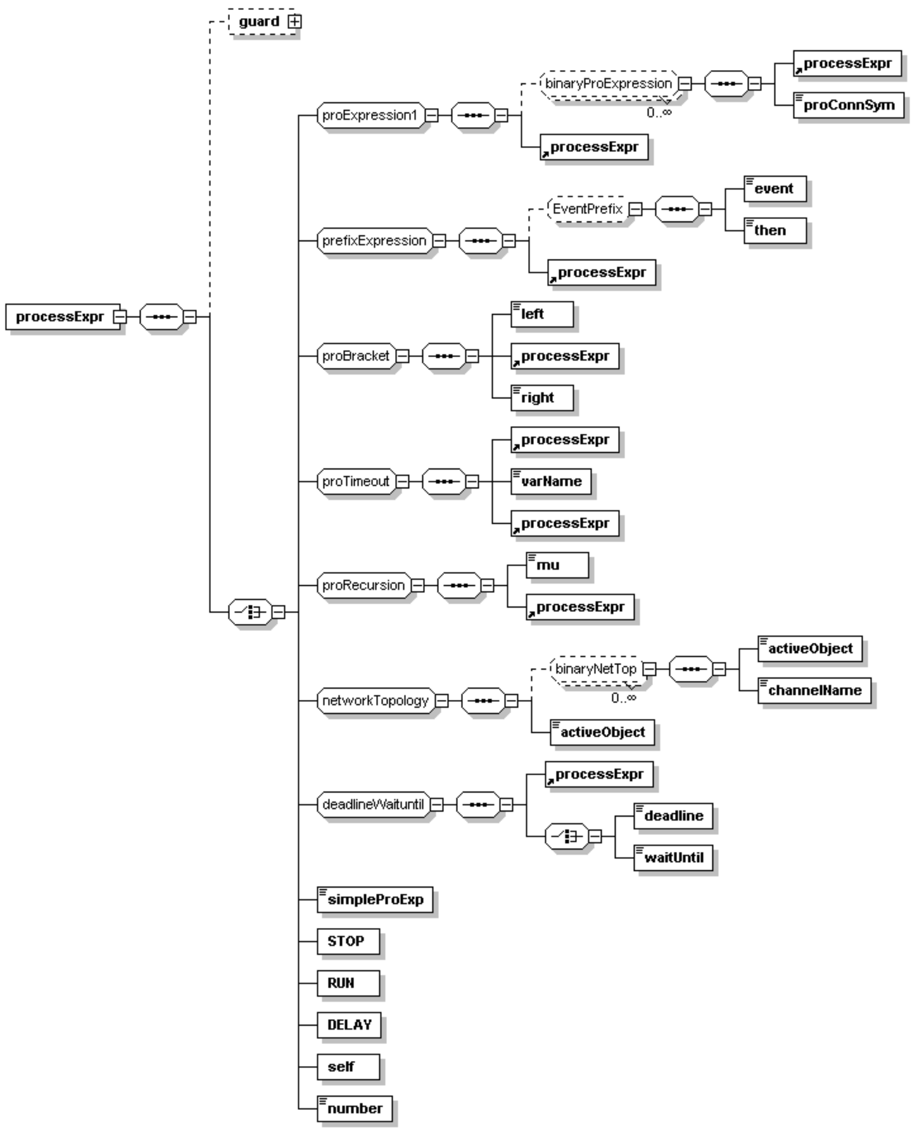


Figure 3.7: Process expression definition

TCSP [82, 83] is an extension of Hoare's Communicating Sequential Process (CSP) notation to accommodate the description of time-sensitive behaviors. The syntax structure of a TCSP process expression in TCOZ is illustrated in Figure 3.7. It defines simple processes as well as compound processes, e.g., prefix and binary process expressions, recursion and so on. Note that the DEADLINE, WAITUNTIL, Active Object and Network Topology expressions are related to the TCOZ extension syntax which will be addressed later.

3.4.5 TCOZ specific definitions

Timed Communicating Object-Z (TCOZ) [67] is an integration of Object-Z and the TCSP languages. It relates the notions of Object-Z operations and TCSP processes by providing a process interpretation of the Z operation schema construct. Thus TCSP primitives (process expressions) can be introduced inside Object-Z classes as operation definitions for modelling timing related aspects. TCOZ also extends the TCSP primitives as follows. TCOZ Active object [72] is an object that has its

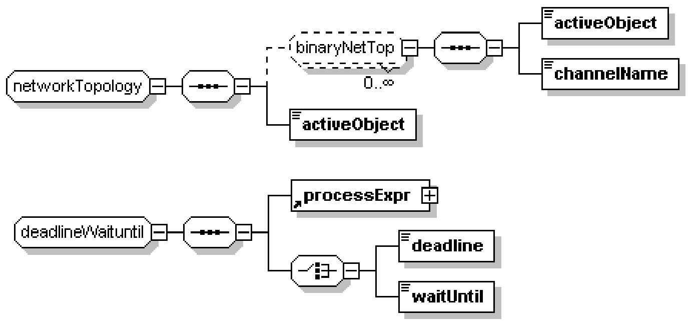


Figure 3.8: Network topology and WaitUntil definitions

own thread of control. In TCOZ, a graph-based approach is adopted to represent the network topology [64] for communication topologies between active objects. The DEADLINE and WAITUNTIL commands are TCOZ extensions to capture time sensitive behaviors. Their syntax structures are illustrated in Figure 3.8.

3.4.6 Other definitions

The ZML also consists of some basic structure definitions such as data type structures, predicate definition, expression syntax and so on.

Data types

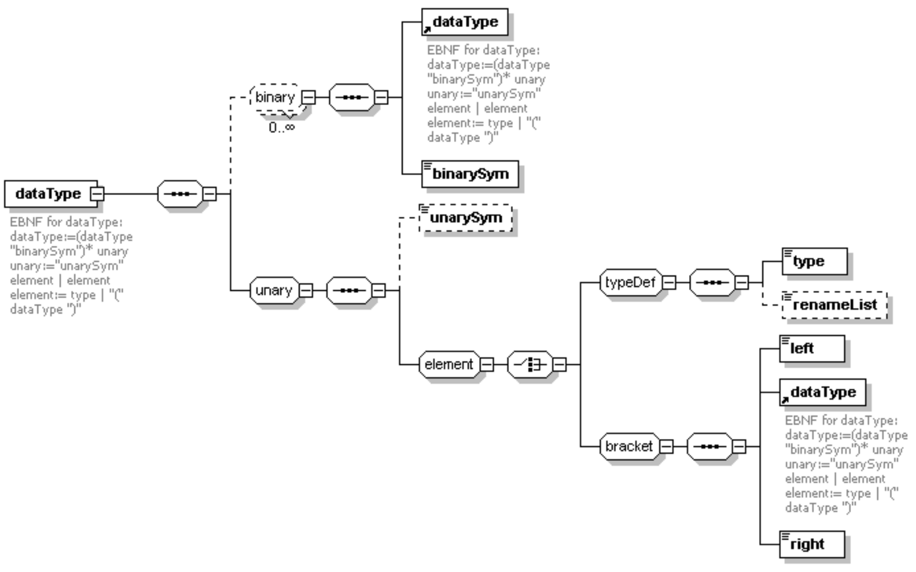

dataType
EBNF for dataType: dataType:=(dataType "binarySym")* unary unary:="unarySym" element | element element:= type | "(" dataType ")"
binary
0..∞
dataType
binarySym
unary
unarySym
element
typeDef
type
renameList
bracket
left
right

Figure 3.9: Data type definition

A variable declaration comprises a variable list and data type definition. A data type syntax consists of a recursive definition on the type constructs as in Figure 3.9.

It defines the unary and binary type constructs as well as the type substitution and bracket expressions. The EBNF for the data type syntax is as follows.

$dataType := (dataType\text{“}binarySym\text{”}) * unary$
$unary := \text{“}unarySym\text{”}\, element \mid element$
$element := type \mid \text{“(”}\, dataType \text{“)”}$

Predicate definitions

A predicate definition comprises five different structures such as quantified, binary, expression predicates. Its structure is depicted in Figure 3.10.

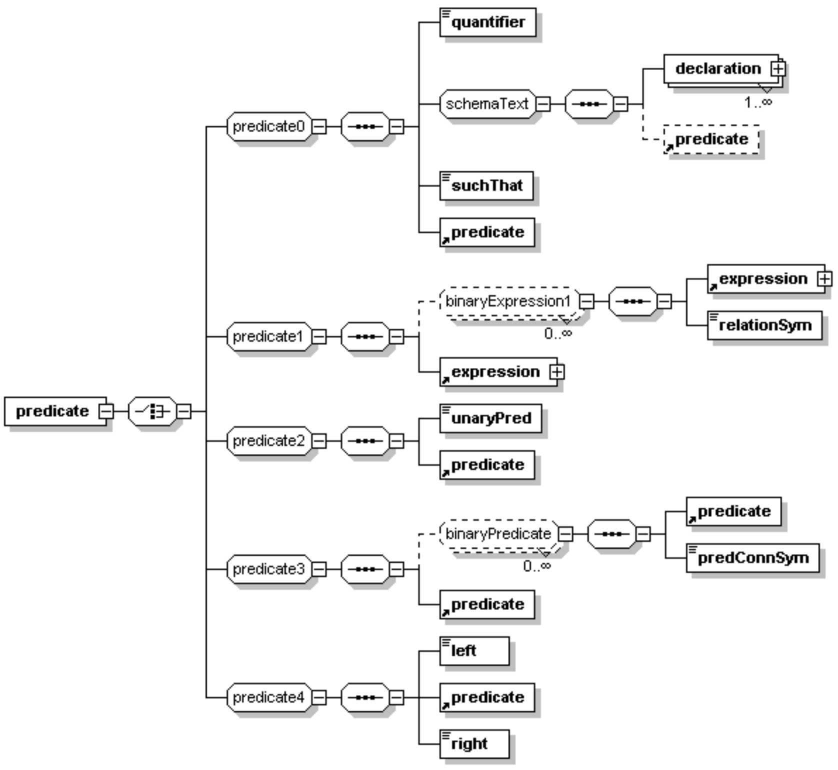


Figure 3.10: Predicate definitions

Expressions

The “expression” in ZML is the most complicated syntax structure. It consists of eight different type of expression categories, i.e., prefix, infix, postfix, object reference, name substitution, and numerical, which is illustrated in Figure 3.11.

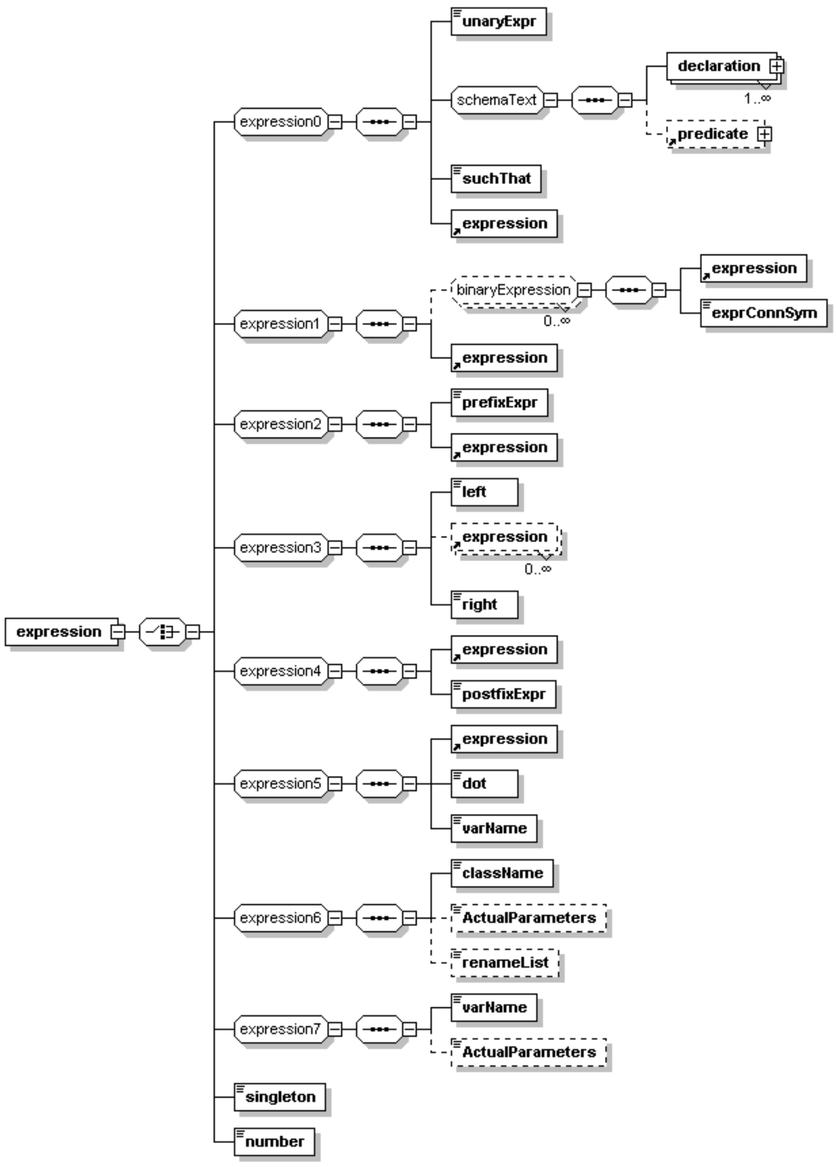


Figure 3.11: Expression definitions

In the outline just given, only a small portion of the ZML syntax definitions are presented. A complete syntax definition and full documentation of the XML Schema definitions for the ZML can be found at:

```
  http://nt-appn.comp.nus.edu.sg/fm/zml/zml.html
```

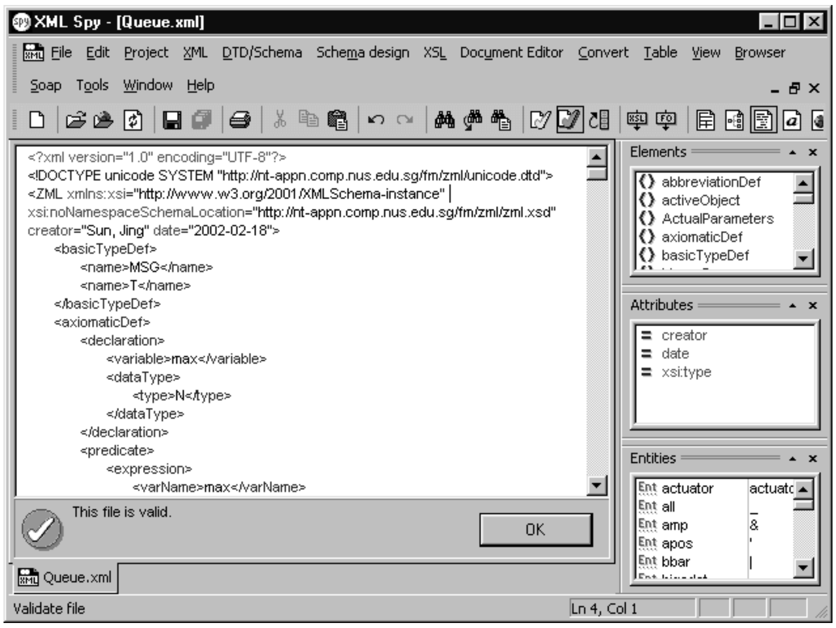
3.5 ZML example

In this chapter, we have defined a Z family markup language using the W3C XML Schema. The schema can be used as a validation document for checking the syntax correctness of the Z/Object-Z/TCOZ specifications in XML format. When authoring ZML files, the user simply declares the name space of the XML schema file as follows.

```
<?xml version="1.0" encoding="UTF-8"?
<ZML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation=
 "http://nt-appn.comp.nus.edu.sg/fm/zml/zml.xsd">
...
</ZML>
```

With the above namespace links, the XML editing tools can check the validity of the ZML file against the XML Schema definition. A validation process is shown in Figure 3.12. Any unspecified structures and entity symbols would be reported as a syntax error. The following represents part of the *Queue* example of chapter 2.1.2 in ZML.

```
<basicTypeDef>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE unicode SYSTEM "http://nt-appn.comp.nus.edu.sg/fm/zml/unicode.dtd">
<ZML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nt-appn.comp.nus.edu.sg/fm/zml/zml.xsd"
creator="Sun, Jing" date="2002-02-18">
    <basicTypeDef>
        <name>MSG</name>
        <name>T</name>
    </basicTypeDef>
    <axiomaticDef>
        <declaration>
            <variable>max</variable>
            <dataType>
                <type>N</type>
            </dataType>
        </declaration>
        <predicate>
            <expression>
                <varName>max</varName>
```

This file is valid.

Figure 3.12: XML validation process

```
 <name>MSG</name>
 <name>T</name>
</basicTypeDef>
...
<schemaDef>
 <name>Queue</name>
 <declaration>
  <variable>items</variable>
  <dataType>
   <unarySym>seq</unarySym>
   <type>MSG</type>
  </dataType>
 </declaration>
 <predicate>
  <expression>
   <prefixExpr>#</prefixExpr>
   <expression>
    <varName>items</varName>
   </expression>
  </expression>
  <relationSym>leq</relationSym>
  <expression>
   <varName>max</varName>
  </expression>
 </predicate>
</schemaDef>
...
<schemaDef>
 <name>Add</name>
 <deltaList>Queue</deltaList>
 ...
</schemaDef>
<!-- other operation definitions -->
```

Note that the above example is related to Z syntax only. A more sophisticated example including Object-Z and TCOZ specifications will be demonstrated in the next chapter along with the ZML web browsing environment.

3.6 Conclusion

In this chapter, we have defined an XML mark-up language for the Z family notations. The XML Schema syntax has been created, validated, and found to be easy to use and practical. The ZML structure is mainly influenced by the syntax definitions in Spivey's Z reference manual [92], Smith's Object-Z book [25], and the TCOZ notation paper [67]. Some example specifications in ZML have been validated against the schema file. Furthermore, a web browsing environment [94], several projection tools and a Z/Object-Z/TCOZ type checker [93, 20] were built based on this ZML definition. In summary, the ZML serves as a standard interchange format among the various tool environments presented in the thesis.

Chapter 4

ZML environment for Z family notations

This chapter presents the development of a web environment for ZML and a tool for projecting a ZML specification into UML Diagrams.

4.1 Introduction

Most discussions related to “Web and Software Engineering” are centered around two main issues: how web technology assists software design and development and how software engineering techniques facilitate web applications. This chapter tries to address both issues within a specific context “XML[101]/XSL[102] and Formal/Graphical software modeling techniques”.

One reason for the slow adoption of Formal Methods (FM) is the lack of tool support and connections to the current industrial practice. Recent efforts and success in FM have been focused on building ‘heavy’ tools, such as theorem provers and model checkers. Although those tools are essential and important in supporting applications of formal methods, they are usually less used in practice due to the intrinsic difficulty involved in the technology. In order to achieve wider acceptance of formal methods, it is necessary to develop ‘light’ weight tools, such as easy-access browsers for formal specifications and projection/transformation tools from formal specifications to industry popular graphical notations. The World Wide Web (WWW) is a promising environment for software specification and design because it allows sharing design models and providing hyper textual links among the models [52]. The Unified Modeling Language (UML) [81] is commonly regarded as one of the dominant graphical notations for industrial software system modeling. It is important to develop links and tools from FM to WWW and to UML so that FM technology transfer can be successful.

Object-Z [25, 88], the object-oriented extension to Z, has an active research community but lacks tool support. TCOZ [67, 66] integrates Object-Z with process algebra Timed-CSP [82, 83]. In this chapter, we use XML and the eXtensible Stylesheet Language (XSL) [102] to develop a web environment that provides various browsing and syntax checking facilities for Z family languages. Second, with the emergence of XML Metadata Interchange (XMI) [38] as a standard, e.g., Rational Rose UML supports XMI input, it is possible to build a transformation link and projection tools from Object-Z/TCOZ specifications (in XML) to UML (in XMI) via XSLT [110] technology.

Since we believe that FM can improve software reliability for applications, Z family languages (particularly Object-Z) are used to formally specify the essential functionalities of the ZML. The Object-Z specification models are used as an initial design document to guide the XML/XSL implementation. In a sense, the chapter demonstrates a formal approach to modeling XML applications. Consequently, *"we take a dose of our own medicine"*.

The remainder of the chapter is organized as follows. Section 2 gives a brief introduction to the requirements of the Z family notations. Section 3 formally specifies the functionalities of the Z family web environment and UML projection tools in Object-Z itself. Section 4 outlines the main approach and techniques of the chapter, and discusses related work. Section 5 presents the implementation issues of the web environment and browsing facilities for Z family languages. Section 6 presents the implementation issues of the projection tools from Object-Z (in XML) to UML (in XMI). Section 7 concludes the chapter.

4.2 Z family languages requirements

In this section, we will outline some requirements for browsing Z family specifications on the web. The differences among Z, Object-Z and TCOZ notations are illustrated and Z schema calculus and Object-Z/TCOZ inheritance expansions (which is the challenge of the ZML development) are explained. Note that the essential requirements of building ZML are highlighted in *Italic* fonts.

4.2.1 Schema inclusion and calculus

Z specifications consist of schema inclusion and schema calculus, which are important constructs for composing complex schema definitions. Consider the Z model of a FIFO message queue in chapter 2 section 2.1.1. The expansions from [1] the schema inclusion of the *Queue* and *QueueInit* definitions are illustrated as below in $\Delta Queue$ and $QueueInit_e$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta Queue} \\ \hline
items : \mathrm{seq}\, MSG \\
items' : \mathrm{seq}\, MSG \\ \hline
\#items \leqslant max \\
\#items' \leqslant max \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{QueueInit_e} \\ \hline
items : \mathrm{seq}\, MSG \\ \hline
\#items \leqslant max \\
items = \langle\, \rangle \\ \hline
\end{array}$$

The expanded form of the schema calculus in *Penguin* is:

[1]The expanded form of a definition refers to provide a complete definition for the schema that merges all the definitions in its included, composed or inherited definitions.

$$
\begin{array}{|l}
\hline
\textit{Penguin}_e \\
\Delta \textit{Queue} \\
\textit{item}?, \textit{item}! : \textit{MSG} \\
\hline
\exists\, \textit{items}'' : \operatorname{seq} \textit{MSG} \bullet \textit{items}'' = \textit{items} \frown \langle \textit{item}? \rangle \\
\qquad \land\ \textit{items}'' \neq \langle\, \rangle \land \textit{items}'' = \langle \textit{item}! \rangle \frown \textit{items}' \\
\hline
\end{array}
$$

The schema calculus expansions such as $Penguin_e$ are useful for analysis, review and reasoning about Z specifications. ZML should support all schema inclusion and calculus expansions automatically.

4.2.2 Inheritance

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. Active classes can be defined by inheriting passive classes. TCOZ is a superset of Object-Z and all Object-Z classes are treated as passive classes (without MAIN operation) in TCOZ. For instance, the expanded form of the active queue example in section 2.2.5 is as follows:

$$
\begin{array}{|l}
\hline
\textit{ActiveQueue}_e \\
\quad
\begin{array}{|l}
\hline
\textit{items} : \operatorname{seq} \textit{MSG} \\
t_j, t_l : \mathbb{T};\ \textit{in}, \textit{out} : \textbf{chan} \\
\hline
\#\, \textit{items} \leq \textit{max} \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\text{INIT} \\
\textit{items} = \langle\, \rangle \\
\hline
\end{array}
\\
\quad
\begin{array}{|l}
\hline
\textit{Add} \\
\Delta(\textit{items}) \\
\textit{item}? : \textit{MSG} \\
\hline
\textit{items}' = \textit{items} \frown \langle \textit{item}? \rangle \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{Delete} \\
\Delta(\textit{items}) \\
\textit{item}! : \textit{MSG} \\
\hline
\textit{items} \neq \langle\, \rangle \\
\textit{items} = \langle \textit{item}! \rangle \frown \textit{items}' \\
\hline
\end{array}
\\
\textit{Join} \mathrel{\widehat{=}} [\textit{item} : \textit{MSG} \mid \#\textit{items} < \textit{max}] \bullet \textit{in}?\textit{item} \rightarrow \textit{Add} \bullet \text{DEADLINE}\ t_j \\
\textit{Leave} \mathrel{\widehat{=}} [\textit{items} \neq \langle\, \rangle] \bullet \textit{out}!\textit{head}(\textit{items}) \rightarrow \textit{Delete} \bullet \text{DEADLINE}\ t_l \\
\text{MAIN} \mathrel{\widehat{=}} \mu\, Q \bullet (\textit{Join} \mathbin{\Box} \textit{Leave}) \mathbin{⨾} Q \\
\hline
\end{array}
$$

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 initialization schemas of derived classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.

We believe the browsing facilities are particularly useful to Object-Z/TCOZ since the notations support cross references and various inheritance techniques for large specifications. It is necessary to view a full expanded version of an inheriting class for the purpose of reasoning and reviewing the class in isolation. It is desirable for ZML to automatically support the inheritance zoom-in/out features.

4.2.3 Instantiation and composition

Let C be the name of a class. The identifier C semantically denotes a collection of objects of the class. 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. 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. Both Object-Z and TCOZ support object composition, e.g., two queues and two active-queues classes can be constructed based on chapter 2 section 2.2.5's examples in Object-Z and TCOZ respectively as:

TwoQueues

$q_1, q_2 : Queue$

$Join \mathrel{\widehat=} q_1.Add$
$Leave \mathrel{\widehat=} q_2.Delete$
$Transfer \mathrel{\widehat=} q_1.Delete \parallel q_2.Add$

TwoActiveQueues

$q_1 : ActiveQueue[talk/out]$
$q_2 : ActiveQueue[talk/in]$

$\textsc{Main} \mathrel{\widehat=} q_1 \mathbin{|[\, talk \,]|} q_2$

The Object-Z parallel operator '$\parallel$' used in the definition of *Transfer* (in *TwoQueues*) 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 base name (i.e. the inputs and outputs are denoted by the same identifier apart from ? and ! decorations).

The CSP parallel operator '$|[\, talk \,]|$' used in the definition of Main (in TwoActiveQueues) captures the concurrent and synchronization behavior of the two communicating active processes q_1.Main and q_2.Main.

The models of *TwoQueues* and *TwoActiveQueues* appear to have similar behavior. However, the behavior of *TwoQueues* is purely sequential. For example, *Join* ($q_1.Add$) and *Leave* ($q2.Delete$) cannot concurrently operate or partially overlap (even assuming the duration of Object-Z operations can be explicitly modelled). This limitation is overcome in the (TCOZ) *TwoActiveQueues* (since two active queues have their own threads of control, only synchronizing through the *talk* channel).

Object-Z/TCOZ models of complex systems may involve complex composition hierarchies, it is useful to have hyper links for all defined types (particularly the class types) automatically created in the design document – a clear requirement for the ZML tool.

4.3 Formal model of ZML environment

4.3.1 Web browsing environment

In the previous chapter we have provided a formal model for the Z family language syntax. Based on that definition a Z family web browsing environment can be modelled as follows:

WebBE

$$\begin{array}{ll} zspec : \mathbb{P}\, ZDefinition & \text{[a specification]} \\ mainpage : URL & \text{[the main URL address]} \\ currpage : URL & \text{[the current page URL address]} \\ expandpos : Name \mathrel{+\!\!\!\!\rightarrow} \mathbb{B} & \text{[all expansible positions]} \end{array}$$

INIT

$$\begin{array}{l} currpage = mainpage \\ \mathrm{dom}\, expandpos = \{c : Classdef \cap zspec \mid c.inherit \neq \varnothing \bullet c.name\} \\ \quad \cup\{s_1 : Schemadef_1 \cap zspec \mid s_1.incl \neq \varnothing \bullet s_1.name\} \\ \quad \cup\{s_2 : Schemadef_2 \cap zspec \bullet s_2.name\} \\ \mathrm{ran}\, expandpos = \{false\} \end{array}$$

$$\begin{array}{|l}
\textit{Clicklink} \\
\hline
\Delta(\textit{currpage}) \\
l? : \textit{Name} \\
\hline
l? \in \{s : \textit{zspec} \bullet s.\textit{name}\} \\
\textit{currpage}' = \textit{mainpage} \frown \langle `\#' \rangle \frown l? \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Clickexpand} \\
\hline
\Delta(\textit{zspec}) \\
e? : \textit{Name} \\
\hline
e? \in \operatorname{dom} \textit{expandpos} \\
\exists_1 \, \textit{def} : (\textit{Classdef} \cup \textit{Schemadef}) \bullet \textit{def}.\textit{name} = e? \land \\
\quad \neg \, \textit{expandpos}(e?) \Rightarrow \textit{zspec}' = \textit{zspec} - \{\textit{def}\} \cup \{\textit{expand}(\textit{def})\}) \\
\quad \textit{expandpos}(e?) \Rightarrow \textit{zspec}' = \textit{zspec} - \{\textit{def}\} \cup \{\textit{expand}^{-1}(\textit{def})\}) \\
\textit{expandpos}' = \textit{expandpos} \oplus \{(e?, \neg \, \textit{expandpos}(e?))\} \\
\hline
\end{array}$$

Note that we introduced an attribute *expandpos* which stores the names of inherited classes and schemas defined by inclusion or schema-calculus for the purpose of expansion. There are two major operations for clicking on either type links or on the expansible positions. The *Clicklink* operation changes the current context to its corresponding type declaration context. The operation *Clickexpand* changes the status of the expansion mode and the content of the specification definitions.

The *expand* function is defined to handle all the class inheritance, schema inclusion and schema calculus expansions.

$$\begin{array}{|l}
\textit{expand} : (\textit{Classdef} \cup \textit{Schemadef}) \rightarrowtail (\textit{Classdef} \cup \textit{Schemadef}) \\
\hline
\forall \, \textit{def} : (\textit{Classdef} \cup \textit{Schemadef}) \bullet \\
\quad \textit{def} \in \textit{Classdef} \Rightarrow \textit{expand}(\textit{def}) = \textit{expand}_c(\textit{def}) \\
\quad \textit{def} \in \textit{Schemadef}_1 \Rightarrow \textit{expand}(\textit{def}) = \textit{expand}_{z1}(\textit{def}) \\
\quad \textit{def} \in \textit{Schemadef}_2 \Rightarrow \textit{expand}(\textit{def}) = \textit{expand}_{z2}(\textit{def})
\end{array}$$

where $expand_c$, $expand_{z1}$, $expand_{z2}$ and other auxiliary functions are defined as follows. The following auxiliary functions captures the semantics of schema calculus and class inheritance expansions.

The $expand_c$ function expands a class definition according to its inheritance list, and outputs the expanded version.

```
$expand_c : Classdef \rightarrowtail Classdef$
──────────────
$\forall c : Classdef \bullet$
    $c.inherit = \varnothing \Rightarrow expand_c(c) = c$
    $c.inherit \neq \varnothing \Rightarrow$
        $expand_c(c).name = c.name$
        $expand_c(c).inherit = \varnothing$
        $expand_c(c).state.decpart = \bigcup\{c_0 : classdef, t : Type \mid$
            $c_0.name = t.name \land t \in \operatorname{dom} c.inherit \bullet expand_c($
            $rename(c_0, c.inherit(t))).state.decpart\} \cup c.state.decpart$
        $expand_c(c).state.axpart = \bigcup\{c_0 : classdef, t : Type \mid$
            $c_0.name = t.name \land t \in \operatorname{dom} c.inherit \bullet expand_c($
            $rename(c_0, c.inherit(t))).state.axpart\} \cup c.state.axpart$
        $expand_c(c).init.axpart = \bigcup\{c_0 : classdef, t : Type \mid$
            $c_0.name = t.name \land t \in \operatorname{dom} c.inherit \bullet expand_c($
            $rename(c_0, c.inherit(t))).init.axpart\} \cup c.init.axpart$
        $expand_c(c).ops = \{opers : classify(\bigcup\{c_0 : classdef, t : Type \mid$
            $c_0.name = t.name \land t \in \operatorname{dom} c.inherit \bullet expand_c($
            $rename(c_0, c.inherit(t))).ops\} \cup c.ops) \bullet merge(opers)\}$
```

The function *rename* captures the class renaming facilities. Given a class and a renaming list, the function returns the renamed class.

```
| $rename : (Classdef \times (Name \rightarrow Name)) \rightarrow Classdef$
|------
| $\forall c : Classdef;\ l : Name \rightarrow Name \bullet$
|     $\operatorname{dom} l \in (\operatorname{dom} c.state.decpart \cup \{op : c.ops \bullet op.name\}) \Rightarrow$
|         $l = \varnothing \Rightarrow rename(c, l) = c$
|         $l \neq \varnothing \Rightarrow$
|             $rename(c, l).name = c.name$
|             $rename(c, l).inherit = \{i : c.inherit \bullet$
|                 $(\text{fst}(i), \{(a, b) : snd(i) \bullet (a, match_1(b, l))\})\}$
|             $rename(c, l).state.decpart = \{(na, dt) : c.state.decpart \bullet$
|                 $(match_1(na, l), dt)\}$
|             $rename(c, l).state.axpart = \{p : c.state.axpart \bullet$
|                 $\{(n, pred) : p \bullet (n, match_2(pred, l))\}\}$
|             $rename(c, l).init.axpart = \{p : c.init.axpart \bullet$
|                 $\{(n, pred) : p \bullet (n, match_2(pred, l))\}\}$
|             $rename(c, l).ops = \{op_2 : Opdef \mid op_1 : c.ops \bullet$
|                 $op_2.name = match_1(op_1.name, l)$
|                 $op_2.detla = \{d : op_1.delta \bullet match_1(d, l)\}$
|                 $op_2.axpart = \{p : op_1.axpart \bullet \{(n, pred) : p \bullet$
|                 $(n, match_2(pred, l))\}\}$
```

The $match_1, match_2$ function is used to find the corresponding item in an item list. Note that if an item is not in the given list it returns itself.

```
| $match_1 : (Name \times (Name \rightarrow Name)) \rightarrow Name$
|------
| $\forall old : Name;\ l : Name \rightarrow Name \bullet$
|     $old \in \operatorname{dom} l \Rightarrow match_1(old, l) = l(old)$
|     $old \notin \operatorname{dom} l \Rightarrow match_1(old, l) = old$
```

```
| $match_2 : ((PredConstructor \cup \downarrow Type) \times (Name \rightarrow Name)) \rightarrow$
|         $(PredConstructor \cup \downarrow Type)$
|------
| $\forall old : (PredConstructor \cup \downarrow Type);\ l : Name \rightarrow Name \bullet$
|     $old \in PredConstructor \Rightarrow$
|         $old.content \in \operatorname{dom} l \Rightarrow match_2(old, l).content = l(old.content)$
|         $old.content \notin \operatorname{dom} l \Rightarrow match_2(old, l).content = old.content$
|     $old \in \downarrow Type \Rightarrow$
|         $match_2(old, l) = old$
```

Function *classify* takes in a set of operation definitions and divides them into subsets, in which the name of the operation is the same.

$$
\begin{array}{l}
\mathit{classify} : \mathbb{P}\,\mathit{Opdef} \rightarrow \mathbb{P}(\mathbb{P}\,\mathit{Opdef}) \\
\hline
\forall (s, ss) : \mathit{classify} \bullet s = \bigcup ss \land \\
\qquad\qquad \forall\, ops : ss \bullet \forall\, op_1, op_2 : ops \bullet op_1.name = op_2.name
\end{array}
$$

The function *merge* merges a set of same named operations into a single operation definition.

$$
\begin{array}{l}
\mathit{merge} : \mathbb{P}\,\mathit{Opdef} \rightarrow \mathit{Opdef} \\
\hline
\forall\, ops : \mathbb{P}\,\mathit{Opdef} \bullet \\
\qquad merge(ops).name \in \{op : ops \bullet op.name\} \\
\qquad merge(ops).delta = \bigcup\{op : ops \bullet op.delta\} \\
\qquad merge(ops).decpart = \bigcup\{op : ops \bullet op.decpart\} \\
\qquad merge(ops).axpart = \bigcup\{op : ops \bullet op.axpart\}
\end{array}
$$

The *expand*$_{z1}$ function expands a schema box definition according to the inclusion of other schemas, and outputs the expanded schema.

$$
\begin{array}{l}
\mathit{expand}_{z1} : \mathit{Schemadef}_1 \rightarrowtail \mathit{Schemadef}_1 \\
\hline
\forall\, s : \mathit{Schemadef} \bullet \\
\qquad s.incl = \varnothing \Rightarrow expand(s) = s \\
\qquad s.incl \neq \varnothing \Rightarrow \\
\qquad\qquad expand_{z1}(s).name = s.name \\
\qquad\qquad expand_{z1}(s).incl = \varnothing \\
\qquad\qquad expand_{z1}(s).decpart = \bigcup\{name_i : s.incl(\text{`inc'});\ s_1 : Schemadef_1 \mid \\
\qquad\qquad\qquad s_1.name = name_i \bullet s_1.decpart\} \cup \bigcup\{name_{xd} : (s.incl(\text{`xi'}) \cup \\
\qquad\qquad\qquad s.incl(\text{`del'}));\ s_1 : Schemadef_1 \mid s_1.name = name_{xd} \bullet s_1.decpart \\
\qquad\qquad\qquad \cup\{(na, dt) : s_1.decpart \bullet (na \frown \langle \text{`}{'}\text{'} \rangle, dt)\}\} \cup s.decpart \\
\qquad\qquad expand_{z1}(s).axpart = \bigcup\{name_i : s.incl(\text{`inc'});\ s_1 : Schemadef_1 \mid \\
\qquad\qquad\qquad s_1.name = name_i \bullet s_1.axpart\} \cup \bigcup\{name_x : s.incl(\text{`xi'});\ s_1 : \\
\qquad\qquad\qquad Schemadef_1 \mid s_1.name = name_x \bullet s_1.axpart \cup \{p : s_1.axpart \bullet \\
\qquad\qquad\qquad \{(n, pred) : p \bullet (n, match_2(pred))\}\} \cup predxi(findlist(s_1))\} \cup \\
\qquad\qquad\qquad \bigcup\{name_d : s.incl(\text{`del'});\ s_1 : Schemadef_1 \mid s_1.name = name_d \\
\qquad\qquad\qquad \bullet\ s_1.axpart \cup \{p : s_1.axpart \bullet \{(n, pred) : p \bullet \\
\qquad\qquad\qquad (n, match_2(pred))\}\}\} \cup s.axpart
\end{array}
$$

The *findlist* function is used to find the pre-state and post-state for a schema box definition.

$$\begin{array}{|l}
\mathit{findlist} : \mathit{Schemadef}_1 \rightarrow (\mathit{Name} \rightarrow \mathit{Name}) \\
\hline
\forall\, s : \mathit{Schemadef}_1 \bullet \mathit{findlist}(s) = \{\mathit{decl} : s.\mathit{decpart} \bullet (\mathit{fst}(\mathit{decl}), \mathit{fst}(\mathit{decl}) \frown \langle \text{‘′’} \rangle)\}
\end{array}$$

The *predxi* function is used to get the implicit predicates for *xi* schema, that is, those with the post-state unchanged.

$$\begin{array}{|l}
\mathit{predxi} : (\mathit{Name} \rightarrow \mathit{Name}) \rightarrow (\mathbb{P}\, \mathit{Pred}) \\
\hline
\forall\, l : \operatorname{dom} \mathit{predxi} \bullet (\exists\, \mathit{post}, \mathit{pre}, \mathit{eq} : \mathit{PredConstructor} \bullet \mathit{post}.\mathit{content} = \mathit{snd}(l) \land \\
\qquad \mathit{eq}.\mathit{content} = \langle \text{‘=’} \rangle \land \mathit{pre}.\mathit{content} = \mathrm{fst}(l) \land \mathit{predxi}(l) = \mathit{post} \frown \mathit{eq} \frown \mathit{pre})
\end{array}$$

The expand_{z2} function expands a schema calculus definition, and outputs the definition with schema box format.

$$\begin{array}{|ll}
\mathit{expand}_{z2} : \mathit{Schemadef}_2 \rightarrowtail \mathit{Schemadef}_1 & \\
\hline
\forall\, s : \mathit{Schemadef}_2 \bullet & \\
\qquad \mathit{expand}_2(s).\mathit{name} = s.\mathit{name} & [\mathsf{Name}] \\
\qquad \mathit{expand}_2(s).\mathit{incl} = \mathit{formIncl}(s.\mathit{calc}) & [\mathsf{Incl}] \\
\qquad \mathit{expand}_2(s).\mathit{decpart} = \mathit{formDecpart}(s.\mathit{calc}) & [\mathsf{Decpart}] \\
\qquad \mathit{expand}_2(s).\mathit{axpart} = \{\mathit{formAxpart}(s.\mathit{calc})\} & [\mathsf{Axpart}]
\end{array}$$

Some auxiliary functions for the expansion of schema calculus are defined as follows. The *formIncl*, *formDepart*, *formAxpart* functions will generate the inclusion, type declaration and predicate part of the schema box correspondingly.

$$\begin{array}{|l}
\mathit{formIncl} : \mathit{PredCalc} \rightarrow \mathit{Inclusion} \\
\hline
\forall\, p : \mathit{PredCalc} \bullet \\
\qquad (p \in\ \downarrow \mathit{Type}) \Rightarrow \\
\qquad\qquad \mathit{formIncl}(p) = \{\exists_1\, s_1 : \mathit{Schemadef}_1 \mid s_1.\mathit{name} = p.\mathit{name} \bullet s_1.\mathit{incl}\} \\
\qquad (p \in \mathit{CalcConstructor}) \Rightarrow \\
\qquad\qquad \mathit{formIncl}(p) = \bigcup\{p_i : p.\mathit{items} \bullet \mathit{formIncl}(p_i)\}
\end{array}$$

$$\begin{array}{|l}
\mathit{formDecpart} : \mathit{PredCalc} \rightarrow \mathit{Decpart} \\
\hline
\forall\, p : \mathit{PredCalc} \bullet \\
\quad (p \in \ \downarrow \mathit{Type}) \Rightarrow \\
\qquad \mathit{formDecpart}(p) = \{\exists_1\, s_1 : \mathit{Schemadef}_1 \mid \\
\qquad\quad s_1.\mathit{name} = p.\mathit{name} \bullet s_1.\mathit{decpart}\} \\
\quad (p \in \mathit{CalcConstructor}) \Rightarrow \\
\qquad \mathit{formDecpart}(p) = \bigcup\{p_i : p.\mathit{items} \bullet \mathit{formDecpart}(p_i)\}
\end{array}$$

$$\begin{array}{|l}
\mathit{formAxpart} : \mathit{PredCalc} \rightarrow \mathit{Pred} \\
\hline
\forall\, p : \mathit{PredCalc} \bullet \\
\quad (p \in \ \downarrow \mathit{Type} \Rightarrow \\
\qquad \exists_1\, s_1 : \mathit{Schemadef}_1 \bullet s_1.\mathit{name} = p.\mathit{name} \wedge \\
\qquad\quad \mathit{formAxpart}(p) = \mathrm{tail}({}^\frown/\{\mathit{prd} : s_1.\mathit{axpart}; \\
\qquad\quad \mathit{op} : \mathit{PredConstructor} \mid \mathit{op}.\mathit{content} = \langle\text{` }\wedge\text{ '}\rangle \bullet \mathit{op} {}^\frown \mathit{prd}\})) \\
\quad (p \in \mathit{CalcConstructor} \Rightarrow \\
\qquad \mathit{formAxpart}(p) = \mathrm{tail}({}^\frown/\{p_i : p.\mathit{items};\ \mathit{op}, \mathit{op}_1, \mathit{op}_2 : \\
\qquad\quad \mathit{PredConstructor} \mid \mathit{op}.\mathit{content} = p.\mathit{op} \wedge \mathit{op}_1.\mathit{content} = \langle\text{`('}\rangle \\
\qquad\quad \wedge\ \mathit{op}_2.\mathit{content} = \langle\text{`)'}\rangle \bullet \mathit{op} {}^\frown \mathit{op}_1 {}^\frown \mathit{formAxpart}(p_i) {}^\frown \mathit{op}_2\}))
\end{array}$$

4.3.2 UML projection facilities

For the projections from Object-Z/TCOZ models into UML diagrams, we first give simplified models of UML class and diagrams. A UML class consists of a class name, a set of attributes and a set of operation names.

$$\begin{array}{|l}
\textit{UMLClass} \\
\hline
\begin{array}{|l}
\hline
\mathit{name} : \mathit{String} \\
\mathit{attris} : \mathit{String} \rightarrow \mathit{Dtype} \\
\mathit{ops} : \mathbb{P}\, \mathit{String} \\
\hline
\end{array} \\
\hline
\end{array}$$

A UML diagram *UMLDiagram* is a collection of UML classes, together with their relationships to each other such as *inheritance* and *aggregation*.

$$
\begin{array}{|l}
\hline
\textit{UMLDiagram} \\
\hline
\textit{classes} : \mathbb{P}\,\textit{UMLClass} \\
\textit{inh}, \textit{agg} : \textit{UMLClass} \leftrightarrow \textit{UMLClass} \\
\hline
\operatorname{dom}(\textit{inh} \cup \textit{agg}) \cup \operatorname{ran}(\textit{inh} \cup \textit{agg}) \subseteq \textit{classes} \\
\forall\, h : \textit{classes} \bullet (h, h) \notin \textit{inh}^{+} \\
\hline
\end{array}
$$

A function *project* models the transformation from an Object-Z/TCOZ specification to a UML class diagram, and is defined as follows:

$$
\begin{array}{|l}
\textit{project} : \mathbb{P}\,\textit{Classdef} \rightarrow \textit{UMLDiagram} \\
\hline
\forall (\textit{oz}, \textit{uml}) : \textit{project} \bullet \\
\quad \{c : \textit{oz} \bullet c.\textit{name}\} = \{c : \textit{uml}.\textit{classes} \bullet c.\textit{name}\} \bullet \\
\quad \forall\, c_1, c_2 : \textit{oz} \bullet \\
\qquad \exists_1\, c' : \textit{uml}.\textit{classes} \bullet \\
\qquad\quad c'.\textit{name} = c_1.\textit{name} \\
\qquad\quad c'.\textit{attris} = \{\textit{cls} : \textit{oz} \bullet \textit{cls}.\textit{name}\} \lhd c_1.\textit{state}.\textit{decpart} \\
\qquad\quad c'.\textit{ops} = \{o : \textit{Opdef} \mid o \in c_1.\textit{ops} \bullet o.\textit{name}\} \\
\qquad c_2.\textit{name} \in \{t : \operatorname{ran} c_1.\textit{state}.\textit{decpart} \bullet t.\textit{name}\} \Rightarrow \\
\qquad\quad \exists_1 (c_1', c_2') : \textit{uml}.\textit{agg} \bullet c_1'.\textit{name} = c_1.\textit{name} \\
\qquad\qquad \land\ c_2'.\textit{name} = c_2.\textit{name} \\
\qquad c_2.\textit{name} \in \{\textit{inh} : \operatorname{dom} c_1.\textit{inherit} \bullet \textit{inh}.\textit{name}\} \Rightarrow \\
\qquad\quad \exists_1 (c_1', c_2') : \textit{uml}.\textit{inh} \bullet c_1'.\textit{name} = c_1.\textit{name} \\
\qquad\qquad \land\ c_2'.\textit{name} = c_2.\textit{name}
\end{array}
$$

Note that our projection function is focused on the transformation from Object-Z/TCOZ specifications to UML class diagrams in this section. The projection to UML behavior diagrams such as statecharts may not be uniquely determined from an Object-Z/TCOZ specification. We will discuss the projection to statechart diagrams further in Section 4.6.

4.4 Main implementation issues and related background

Formal methods like the CafeOBJ system [32] have included an environment supporting formal specification over networks. Schemas using pure Z notation on the web based on HTML and Java Applet have also been investigated by Bowen and Chippington [8] and Ciancarini, Mascolo and Vitali [15]. HTML has been successful in presenting information on the Internet, however the lack of content information and the overburdened use of tags have made the efficient retrieval and exchange of information content more difficult to achieve.

Our work uses the latest technology of XML and XSL for displaying and transforming Z family notations on the web. The users only need to follow the defined syntax in writing the XML document, the layout part is user transparent. Our XML format is inspired by the work of Ciancarini et al [15] however we use different technology – XML/XSL. The developed XML/XSL web environment covers not only the pure Z notation but also Object-Z and TCOZ with various type referencing and expansion facilities. Furthermore, the projection tools from Object-Z/TCOZ to UML are built into our system. The conceptual projection techniques are derived from our research on linking UML with Object-Z [56, 57], which are similar to the translation rules developed by Kim and Carrington [53]. The difference is that we are working on the projection from Object-Z/TCOZ to UML whereas Kim and Carrington focus on translating UML to a partial Object-Z specification. We share the goal of visualizing Object-Z with the work of Wafula [111]. Other work (e.g. [29]) on linking Z and UML mainly concentrates on using Z to define the semantics for UML class diagrams.

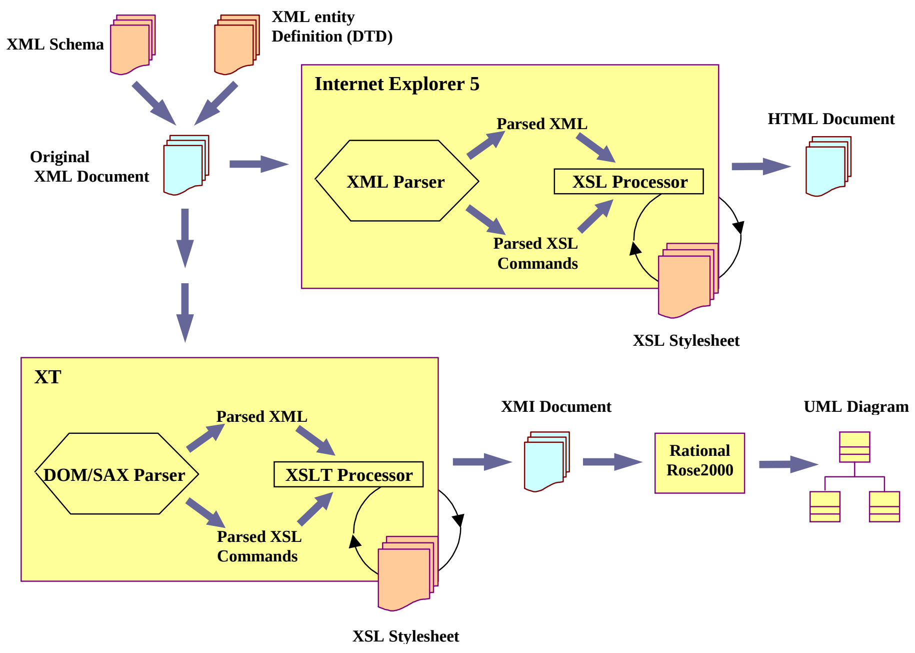


Figure 4.1: ZML overview diagram.

The main process and techniques for ZML are depicted in Figure 4.1. First, the formal specification model in ZML is validated against the Schema and DTD syntax definitions, then transformed into corresponding HTML or XMI format according to their style sheets. Finally, it is displayed in a web browser or in the Rational Rose UML suite. In the following sections, we use the *Queue* example to facilitate the detailed discussion of our implementation approaches.

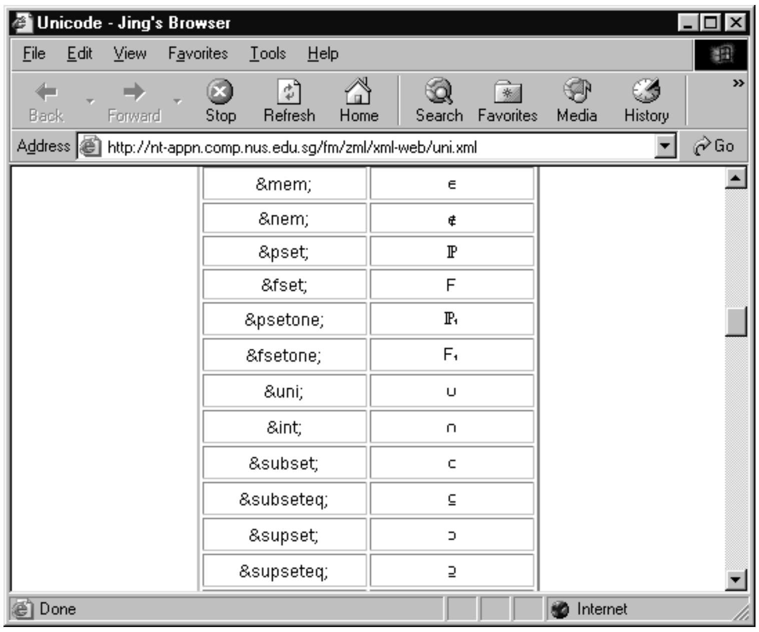
The formal model defined in Section 3.2 and Section 4.3 acts as a precise design reference document and provides clear guidelines for our XML/XSL implementations. For example, the ZML syntax structure is derived from the model; the XSL codes for implementing inheritance and schema calculus expansions in Section 4.5 is based on the *expand* function defined in Section 4.3.1; the XSLT codes for projecting Object-Z/TCOZ to UML in Section 4.6 is based on the *project* function defined in Section 4.3.2.

4.5 Web environment for Z family languages

4.5.1 Syntax definition and usage

In the previous chapter, a customized XML document syntax for the Z family language is defined according to the formal syntax definitions. Z family languages contain a rich set of mathematical symbols. Those symbols can be presented directly in Unicode [16] that is supported by XML. We have defined all entities in the DTD so that users do not have to memorize all the Unicode numbers when authoring their ZML documents. Some entity declaration DTD and symbol mappings (in Figure 4.2) are illustrated as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<!ENTITY emptyset "&#x2205;">
<!ENTITY mem "&#x2208;">
<!ENTITY pset "&#x2119;">
<!ENTITY nem "&#x2209;">
<!ENTITY uni "&#x222a;">
...
```


Unicode - Jing's Browser
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History
Address
Go
Done
Internet

http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/uni.xml

&mem;	∈
&nem;	∉
&pset;	ℙ
&fset;	F
&psetone;	ℙ₁
&fsetone;	F₁
&uni;	∪
&int;	∩
&subset;	⊂
&subseteq;	⊆
&supset;	⊃
&supseteq;	⊇

Figure 4.2: Unicode symbol mapping

It states the mapping information of Z related symbols to their corresponding Unicode, e.g., the empty set symbol in Z can be represented in Unicode number ‘x2205’. As most existing Z specifications were constructed in LaTeX, translating them to our format can be a trivial task as each entity may be given a Z LaTeX compatible name. DTD is chosen to define our entity declaration because XML Schema do not support entity declaration at the moment. The following is part of the ZML syntax for the *ActiveQueue* class.

```
<classDef>
 <name>ActiveQueue</name>
 <inheritedClass>
  <name>Queue</name>
 </inheritedClass>
 <state>
  <declaration>
   <variable>Tj Tl</variable>
   <dataType>
    <type>T</type>
   </dataType>
  </declaration>
  ...
 </state>
 <operation>
  <name>Join</name>
  <processExpr>...<processExpr>
 </operation>
 ...
</classdef>
```

4.5.2 XSL transformation

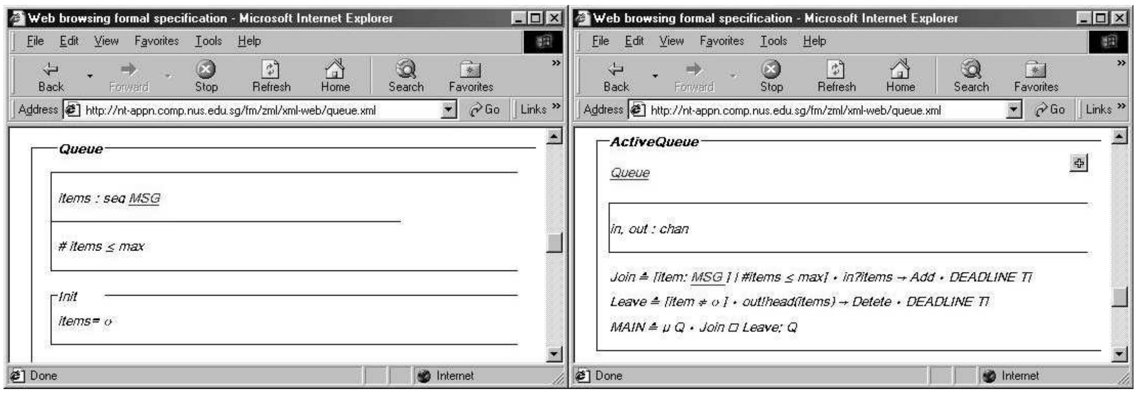
With a valid XML file in hand, the next step is to transform the XML file into HTML format and display it on the web. XSL is a stylesheet language to describe rules for matching and transforming XML documents. An XSL file is an XML document itself and it can perform the transformation from XML to HTML, XML to XML, XSL to XSL and so on. This kind of transformation can be done on the server side or the client side. Since common web browsers such as Internet Explorer 5 (IE5 or above) already support XSL technology, the current ZML environment is based on client side (browser) transformation. Server side transformation will be discussed later. A partial XSL stylesheet segment for displaying operation schema and class definitions is illustrated below.

```
<xsl:template match="operation">
<html>
  <tr>
    ...
    <td height="24" valign="middle" align="left" nowrap="true">
      <i><xsl:value-of select="name"/></i>
      ...
    </td>
    ...
  </tr>
  <xsl:for-each select="delta | declaration">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
  <xsl:apply-templates select="st"/>
  <xsl:for-each select="predicate">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
  ...
</html>
</xsl:template>

<xsl:template match="classdef[@layout='simpl'] |
  classdef[@layout='gen']">
<html>
  ...
  <a><xsl:attribute name="name"><xsl:value-of select="name"/>
    </xsl:attribute></a>
  ...
  <xsl:apply-templates select="state"/>
  <xsl:apply-templates select="init"/>
  <xsl:apply-templates select="op"/>
  ...
</html>
</xsl:template>
```

The XSL stylesheet defines `match` methods for each tag in the XML structure and describes the corresponding HTML codes. From the example above, in matching the 'operation' tag, the XSL will display the operation name, Δ-list, declaration and predicates accordingly; in matching the 'classdef' tag the XSL will first convert the class name into an HTML bookmark for the type reference usage and then apply the templates of drawing state schema, initiation schema, operations and so on. To apply a template in XSL is similar to making a function call in other programming languages, and each template will perform its own transformation. When authoring Z family specifications in the ZML format, the users only need to construct their ZML files and add a URL to the defined XSL stylesheet location as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
 href="http://nt-appn.comp.nus.edu.sg/fm/zml/objectzed.xsl"?>
```


Web browsing formal specification - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/queue.xml
Go Links
Done
Internet

Queue

items : seq MSG

items ≤ max

Init

items= ⟨⟩

ActiveQueue

Queue

in, out : chan

Join ≙ [item: MSG] | #items ≤ max] • in?items → Add • DEADLINE Ti

Leave ≙ [item ≠ ⟨⟩] • out!head(items) → Detete • DEADLINE Ti

MAIN ≙ μ Q • Join □ Leave; Q

Figure 4.3: Queue specification on web.

With this link, the browser will automatically transform a ZML document into the desired HTML output via the built-in XML parser. This process is totally user transparent and much faster than the Java applet approaches [8, 15]. For example, the *Queue* and *ActiveQueue* classes in ZML format specified previously is transformed into HTML as in Figure 4.3.

Note that by clicking the 'plus' button the expanded version of class "*ActiveQueue*" will be displayed. A full demonstration of the *Queue* specification example is available at

`http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/queue.xml.`

4.5.3 Extensive browsing facilities

In the previous section we introduced how the Z family notations can be elegantly and statically presented on the web. To make the environment more powerful and user friendly, some advance functionalities are developed. This section discusses the extensive browsing facilities for type reference, class inheritance expansion and schema calculus expansion.

Type referencing

When building a large formal model, which could include many type definitions and references, users often want to recall the definition of a particular type. Type referencing allows the user to browse back to the actual type definition and quickly access the corresponding type declarations. In a predicate or declaration, by clicking the name of the type, the user will be brought to the location where the type was declared. This is very useful for specification understanding.

This functionality is achieved in two steps. Firstly when a type definition node in XML is transferred to HTML, its name is converted into an HTML bookmark. Secondly, when the user needs to reference a type in a declaration or predicate, a hyper-link that points to the defined bookmark is created. The XSL template for the latter (`type` node) is shown as follows.

```
<xsl:template match="type">
  <xsl:choose>
    <xsl:when test="//classdef[$any$ name=context(-1)] |
      //basicTypeDef[$any$ name=context(-1)] |
      //schemadef[$any$ name=context(-1)]">
    <a>
      <xsl:attribute name="href">#<xsl:value-of/>
        </xsl:attribute><xsl:value-of/>
    </a>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

It searches whether any name of class definition, basic type definition or schema definition is equivalent to the current type name. If such a name exists, a type hyper-link is established.

Class inheritance and schema calculus expansions

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. The aim of the class inheritance expansion is to allow a user to view the full definition of a derived class. In the *ActiveQueue* class case (in the right hand side of the Figure 4.3), when a user clicks the button '+', the full definition of the class of *ActiveQueue* will be shown. This implementation is based on the inheritance expansion rules defined in the $expand_c$ function. Clicking button '−' is for going back to the un-expanded version.

The core part of the expansion techniques uses the XML Path Language (XPath) [108] facilities provided by XSL to match the corresponding definitions in the parent class and merge them into the derived class. Part of the XSL for merging the declarations in the state schema of a class is as follows.

```
<xsl:for-each select="//classdef[name=context(-1)
 /inheritedClass/name]/state/declaration">
 ...
</xsl:for-each>
<xsl:for-each select="state/declaration">
 ...
</xsl:for-each>
...
```

As we can see from the above, the `select` constraint will restrict a search through the entire ZML document for a match of the same named class definition corresponding to the name in its `inheritedClass` list. Following this, the state declaration of super class is merged with the current class. Thus the whole definition of `state` declarations in the derived class is completed. In addition, DHTML and JavaScript are used to control the visibility of the two versions of class definitions.

Schema inclusion and schema calculus expansions are similar to class inheritance expansion and can be constructed using the same mechanism.

4.5.4 Server side transformation

As mentioned in Section 4.5.2 the current ZML web environment is based on client (browser) side transformation. It is not compatible for browsers that do not support XSL technology presently such as Netscape. To make the ZML environment available to all kinds of browsers, we can perform the transform on the server side and send back pure HTML to the browsers. XSL transformation on the server is bound to be a major part of the Internet Information Server (IIS) work tasks in the future, as we will see a growth in the specialized browser market (for example the use of Braille, Speaking Web, Web Printers, Handheld PCs, Mobile Phones ... [84]). The following Active Server Pages (ASP) code for transforming the XML file to HTML on the server side can achieve this.

```
<%
    'Load the XML
    set xml = Server.CreateObject("Microsoft.XMLDOM")
    xml.async = false
    xml.validateOnParse = true
    xml.load(Server.MapPath("queue.xml"))
    'Load the XSL
    set xsl = Server.CreateObject("Microsoft.XMLDOM")
    xsl.async = false
    xsl.load(Server.MapPath("objectzednewnt.xsl"))
    'Transform the file
    Response.Write(xml.transformNode(xsl))
%>
```

The first block of code creates an instance of the Microsoft XML parser, and validates and loads the XML file into memory. The Microsoft XML parser is a COM component that implements the W3C XML Document Object Model (DOM) [100]. As a W3C specification, the objective for the XML DOM has been to provide a standard programming interface to a wide variety of applications for accessing and manipulating XML documents. The second block of code creates another instance of the parser and loads the XSL document into memory. The last line of code transforms the ZML document via the XSL style sheet, and then returns the resultant HTML to the browser.

The next section is focused on projecting Object-Z/TCOZ models (in XML) to UML diagrams (in XMI).

4.6 UML projection

As requirement specifications of software systems, formal models can be precise and elegant but difficult to read and interpret by software engineers without relevant mathematical background. In comparison, the most popular graphical notation UML is much easier to understand and widely accepted by the industry, but it lacks precise semantics. It is important to develop a transformation link/tool from the formal model to various UML diagrams. The key technique ideas in our approach are:

- Syntactically, UML (OCL) is extended with a TCOZ communication interface type – **chan**. As a result, TCOZ sub-expressions can be used (with the same role as OCL) in the statechart diagrams and collaboration diagrams.
- Semantically, UML class diagrams are identified with the signatures of the Object-Z/TCOZ classes. The states of the UML statechart diagram are identified with the TCOZ processes amd operations and the state transition links are identified with TCOZ events and guards. The classifier roles and communications are identified with TCOZ classes and their interactions respectively.
- Effectively, UML diagrams can be seen as visual projections from a unified formal Object-Z/TCOZ model.

4.6.1 Translation rules

An Object-Z/TCOZ model and a UML model are translated to each other from three views: static view, interaction view and behavior view, which are represented by class, collaboration and statechart diagrams respectively.

Static view

UML class diagrams are used to illustrate the static structure of a TCOZ model. Some guidelines are defined as:

- *Class* Each class in Object-Z/TCOZ is translated to a class in UML class diagrams and vice versa. In Object-Z/TCOZ, attributes and operations are encapsulated and private to classes. Therefore they are set to be private in UML class diagrams.
- *Active class* In UML, an active class is a class whose instances are active objects, and have their own thread of control. Classes for defining active objects in TCOZ will have the MAIN operation.
- *Inheritance* The inheritance relationship between two classes in Object-Z/TCOZ is directly translated into the inheritance relationship in UML.
- *Aggregation* If in a class there are one or more objects of another class as attributes, the relationship of the two classes projected to UML is aggregation, which means the second class is a constituent part of the first one.

Interaction view

In a composite system, objects of different classes interact with each other. The general arrangement of these interactions is captured with a network topology in TCOZ. In UML, collaboration diagrams are used to illustrate the system from this interaction view. A collaboration has a static part and a dynamic part. Objects/Classes in TCOZ are exactly the counterpart of static part–classifier roles in UML collaboration diagrams as the instantiation of the collaboration. They interact through a communication interface (**chan** for synchronized communications). The dynamic interactions of classifier roles in UML are illustrated as messages between them, and their properties can be set as synchronized communications, which happen to match well with the network topology in TCOZ. Based on such analysis, the rules are given as:

- Classes in TCOZ are projected to classifier roles in UML collaboration diagrams while their communications depicted by the network topology are projected to the messages between associated classifiers. The communications are indicated by the associated arrow's direction (indicating the data flow direction).
- If two classes in a TCOZ model communicate through a synchronous interface **chan**, the corresponding data flow direction is set according to the event definitions (from ! to ?).

Behavior view

In TCOZ, operations of a class specify its computation behaviors and interaction behaviors. The guidelines for the projection from TCOZ model to UML statechart diagram are:

- Consider each operation in TCOZ model as a state or substate, which may have its own actions or fix some values for a certain time span. Nested operations are translated into substates of the state representing the operation which calls them.
- Events and guards in a TCOZ model are viewed as triggers which cause transition of states in the statechart. They match the definition of triggers and guards in UML statechart diagrams.
- MAIN in TCOZ is modeled as the state in UML statechart diagrams that the startstate leads to, that is, the first state that the object lies in after the transition starts.
- In the case that an operation calls other operations, the called operations serve as the substates of the calling one, and they together compose a composite state in the statechart.
- *Interleaving* operations in TCOZ are translated into concurrent states in a composite state.

4.6.2 Implementation and examples

XML Metadata Interchange (XMI) [38] is an industry standard for storing and sharing object programming and design information. Unisys Corporation has implemented the XMI for the UML tool Rational Rose 2000. Rose can generate UML diagrams from imported XMI documents, and export XMI documents for any existing UML diagrams. Our implementation is based on the definition of ZML syntax for TCOZ; then via XSL [102] Transformations (XSLT) technology, define an XSL style sheet to capture all translation rules from TCOZ (in ZML) to UML (in XMI). XT is chosen as the XSLT processor and Rational Rose is used as the UML tool.

The XML file for formal specifications and the XMI file for UML diagrams have similar structures (an observation from their formal models defined in Section 4.3). Consistency has been considered when XSL and XML schema files were defined for Object-Z/TCOZ. An XMI file has a structure as follows:

```
<XMI xmi.version="1.0">
  <XMI.header>
  <XMI.content>
  <XMI.extensions>
</XMI>
```

The syntax definition of XMI for UML is specified as XMI 1.1 RTF UML DTD. This DTD file defines all entities and XMI syntax signatures for UML. The XMI file for UML diagrams consists of three parts: the header, content and extension. The `XMI.header` section includes some optional information about the UML model.

Elements in UML diagrams, such as classes in class diagrams and states in the statecharts, are specified in the `XMI.content` section, while their layout, colors and other displaying properties are specified in the `XMI.extensions` section.

The XSL file used in this section is the implementation of the transformation rules (abstractly defined in formal models, the *project* function, in Section 4.3.2) and the file is consistent with *UML.DTD*. The template technology plays a key role in implementing the translation rules. Considering the implementation issues and the translation rules based on the formal model, the following guidelines are formed:

- Each class in an Object-Z/TCOZ XML model corresponds to a class in the UML XMI model. They have the same name, attributes and operations.
- If a type value in the *InheritedClass* part of a class matches the name of any other class in the current ZML file, we regard that the former class inherits the second one and illustrate the inheritance relationship between these two classes in the UML class diagram. In the case of spelling mistakes or a missing reference of the *Inherit* type, we ignore the relationship.
- If a type value in the *decl* part, that is, the type of an attribute, matches the name of any class in the current ZML file, this is regarded as an aggregation relationship between these two classes. The cardinality of the aggregation will be calculated and classified into UML aggregation ranges.

Simplified XSL code for capturing the *aggregation* relationship is shown below. The *inheritance* relationship can be treated in a similar way.

```
<xsl:variable name="AggregationNo" select='position()'/>
<xsl:choose>
 <xsl:when test="//classdef[$classNo]/name=./type">
  <![CDATA[<Foundation.Core.AssociationEnd xmi.idref=']]>
  <xsl:value-of select="concat('G.',1+$AggregationNo*3)"
   /><![CDATA[ '/> ]]>
 </xsl:when>
 ...
</xsl:choose>
```

Due to the space limitation (XMI files for UML models are normally very large and complex with all details about property specifications), only the sketch of a simplified XMI unit – class *Queue*, is given as an example here.

```
<Foundation.Core.Class xmi.id = ' S.10001 '>
        <name> Queue </name>
        <namespace>
                <xmi.idref = 'G.1'/>
        </namespace>
        <GeneralizableElement.specialization>
                <xmi.idref = ' G.13 '/>
                <!-- { ActiveQueue -> Queue }-->
        </GeneralizableElement.specialization>
        <Classifier.feature>
                <Attribute xmi.id = ' S.10002 '>
                        <name> items </name>
                        <multiplicity>1..1</multiplicity>
                        <DataType xmi.idref = ' G.11 '/>
                                <!-- seq MSG -->
                </Attribute>
                <Operation xmi.id = ' S.10003 '>
                        <name>Init</name>
                </Operation>
                <Operation xmi.id = ' S.10004 '>
                        <name> Add </name>
                </Operation>
                <Operation xmi.id = ' S.10005 '>
                        <name> Delete </name>
                </Operation>
        </Classifier.feature>
</Foundation.Core.Class>
```

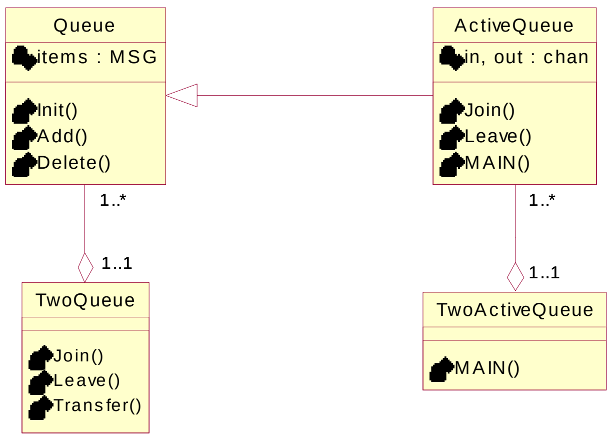


Figure 4.4: Generated class diagram.

The projection rules for translating a formal model to UML class diagrams are trivial. As in Figure 4.4, the UML class diagram depicts the static view of the four graph classes constructed from the previous sections. Note that this diagram was generated automatically from the XML model via the XSL transformation. All attributes and operations match their definitions in the formal model. Now we demonstrate how the relationships between classes are captured during the transformation.

The relationship between *ActiveQueue* and *Queue* is *Inheritence*. This relationship in the XMI segment is as follows.

```
<Foundation.Core.Generalization xmi.id = ' G.13 '>
        <name/>
        <Generalization.subtype>
                <Class xmi.idref = ' S.10006 '/>
                        <!-- ActiveQueue -->
        </Generalization.subtype>
        <Generalization.supertype>
                <Class xmi.idref = ' S.10001 '/>
                        <!-- Queue -->
        </Generalization.supertype>
</Foundation.Core.Generalization>
```

The relationship between *TwoQueues* and *Queue* is *Aggregation*. The aggregation relationship is illustrated in the following simplified XMI segment:

```
<Association xmi.id='G.2'>
  <name />
  <connection>
      <AssociationEnd xmi.id='G.3'>
        <name />
        <multiplicity>1</multiplicity>
        <type>
           <xmi.idref='S.10011'/>
           <!--  TwoQueues   -->
        </type>
      </AssociationEnd>
      <AssociationEnd xmi.id="G.4">
        <name />
        <multiplicity>1..*</multiplicity>
        <type>
           <xmi.idref="S.10001" />
           <!--  Queue   -->
        </type>
      </AssociationEnd>
  </connection>
</Association>
```

Currently we are investigating the dynamic view transformation. Based on semantic links defined in Section 4.6.1, a statechart diagram for the class *ActiveQueue* can be constructed as in Figure 4.5.

Brief structures of a `SimpleState Join` and a transition (from `Main` to `Join`) in the statechart in XMI are:

```
<State_Machines.SimpleState xmi.id="G.21">
        <name>Join</name>
```

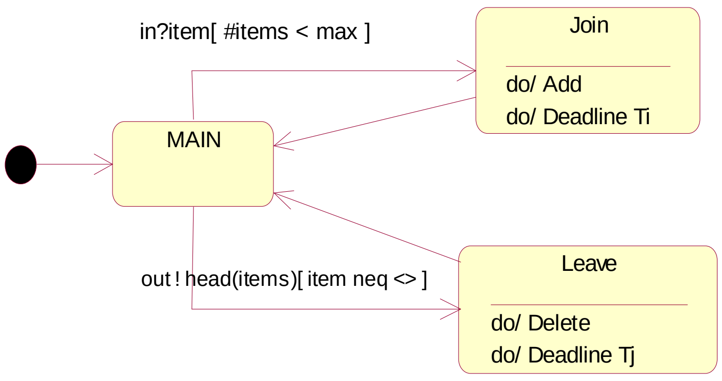


Figure 4.5: ActiveQueue statechart diagram.

```
</State_Machines.SimpleState>
<State_Machines.Transition xmi.id="G.24">
  <name />
  <source>
        <SimpleState xmi.idref="G.22" />
        <!-- Main  -->
  </source>
  <target>
        <SimpleState xmi.idref="G.23" />
        <!-- Join  -->
  </target>
  <trigger>
        <SignalEvent xmi.idref="G.28" />
        <!-- in?item   -->
  </trigger>
  <guard>
        <Guard xmi.id = 'G.30' />
        <expression>
               #items < max
        </expression>
  </guard>
</State_Machines.Transition>
```

The documentation about Object-Z/TCOZ to UML transformation and downloadable codes are available at:

`http://nt-appn.comp.nus.edu.sg/fm/zml/xmi-uml/xmi.htm`

4.7 Conclusion

The first contribution of this chapter is the demonstration of the XML/XSL approach to the development of a web environment for Z family languages. The ZML web environment includes the auto type referencing and browsing facilities such as the Z schema calculus and Object-Z/TCOZ inheritance expansions. Our ideas for putting Z family on the Web can be easily adopted by other formal specification notations, such as VDM and VDM++. In fact, since TCOZ includes most Timed CSP constructs, its web environment can be used for process algebra (CSP/Timed-CSP) specifications. Perhaps this may create a new culture for constructing formal specifications on the web in XML rather than in LaTeX. We hope it can be the starting point for developing a standard XML environment for all formal notations (including integrated formal notations, i.e., RAISE [71], SOFL [59] and so on): a Formal specification Markup Language (FML). This may also make an impact on formal methods education through the web.

The second contribution of this work is the investigation of the semantic links and web transformation environment (XSLT) between Object-Z/TCOZ (in XML) with UML diagrams (in XMI). In our approach, UML diagrams are visual projections from a formal Object-Z/TCOZ model, and they are consistent with the formal model. Recently, this work have been extended to support the auto-generation of UML statechart diagrams from Object-Z/TCOZ specifications using a Java XML parser [20]. Although we have some ideas on Object-Z/TCOZ behavior projections to statecharts, the development of the Web environment for systematic transformation from Object-Z/TCOZ to statechart/collaboration diagrams remains a challenge. The engineering work for developing further techniques and putting these techniques into commercial case tools perhaps requires involvement from industry partners.

The third contribution of this chapter is the demonstration of a formal design approach to modeling web applications. Object-Z has been used to specify and design the essential functionalities of the ZML environment. We have found that the formal model acts as a precise design document and has also provided clear guidelines for the XML/XSL implementations.

Since we have constructed a web XSL environment as close as possible to the LaTeX style files for Z/Object-Z (fuzz.sty and oz.sty), a LaTeX ZML translation tool was developed to map the existing Z/Object-Z specifications in LaTeX to their ZML format [74]. And a reverse process was also necessary as long as LaTeX is not totally replaced by XML technology.

Chapter 5

Animation of TCOZ specification

This chapter presents the development of an animation environment for the TCOZ specifications.

5.1 Introduction

Requirements capture is a key activity in software and system engineering. The challenge for the requirement specification of complex systems is how to precisely capture static and dynamic system properties in a highly structured way. The current research focus of combining integrated formal methods has led to a need for developing various support tools. TCOZ builds on the strengths of Object-Z [14, 88] in modeling complex data and state with the strengths of Timed CSP [82, 83] in modeling real-time concurrency. In addition to the investigation of the integrated formal methods, it is also important to develop transformation tools (from the integrated formal models) to animation tools for validating the formal models. Validation denotes the process of determining that the requirements are the right requirements and that they are complete. Animation is a means of performing such validation. Many approaches have been explored in animating Z using logic and functional programming languages, i.e., Prolog [112], Haskell [98] and so on. For integrated formal notations, i.e., TCOZ, the best candidates for such animation might be multi-paradigm programming languages, such as Oz [39].

In this chapter, we demonstrate the approach of animating TCOZ specifications in a multi-paradigm programming language Oz. The Oz programming system has been developed mainly by researchers from DFKI (the German Research Center for Artificial Intelligence). It is based on a concurrent constraint model and merges several paradigms of programming languages, such as object-orientation, constraint and logic programming, functional programming and concurrent programming into a single coherent design. Integrated formal notations such as TCOZ could find a majority of its corresponding features in Oz. In addition, XSLT is used as a transformation tool for the code generation from TCOZ (in XML) to Oz.

The remainder of the chapter is organized as follows. Section 2 and 3 present some general concepts about specification validation and animation languages. Section 4 presents the translation rules from TCOZ to Oz. Section 5 presents the implementation and a case study. Section 6 concludes the chapter.

5.2 Specification validation

Specification validation denotes the process of determining that the requirements are the right requirements and that they are complete. Animation of the specification is a means of performing such validation. Animation plays an important role in validating the consistency between the formal model and the real world informal requirements. System analysts or clients may wonder whether their specification correctly captures the real world problem that they want to solve. If the formal model does not truly reflect the real world requirements it is useless to further verify its correctness. One of the such validation techniques, namely specification animation, is to provide an executable version of the specification and validate the logic relationships inside formal model. The process of verifying the consistency between the formal model and real world model is difficult to formalize. Animation is an engineering process that brings us one step closer to this goal. It allows the system analysts to explore the behavior of the formal model and thus helps to clarify their interpretation and track down the misunderstandings with the clients since requirements at this stage may have not been fully developed and clearly understood. The purpose of animation is to exhibit the dynamic properties of a specification, and to bridge the gap between the real world problem and our interpretation of the informal requirements. Animation is a vital part in the early stages of formal modelling.

Programs are collections of detailed instructions to a computer. Implementation is the process of transforming a specification to produce a program (through refinement techniques). The product is a realistic computer system that meets the desired requirements. Prototyping is a rough and cheap version of implementation itself, perhaps with the non-functional requirement eased. Animation is a mapped and executable version of the specification that is concerned with an abstraction of the required system. It is not a real computer system that provides the detailed functionalities, but rather a system that focuses on the exploring of logical relationships within the specification. Some differences between animation and implementation lies in the following aspects. First, as type information defines a membership relation between the variable and its type set, data types inside an animation need not be actual data sets that are the same as those within an implementation. Because the primary purpose of an animation is to explore the consequences of a specification, rather than to produce a final implementation of the system or even a full scale prototype that is capable of handling realisticallysized data sets. It could be a virtual data set or even a subset as long as the type information could be demonstrated. In this way the focus is on the logical relationships and the behaviors of the specification. Second, animation should ensure that each animated operation is logically equivalent to its corresponding specification since specification is at a higher level of abstraction. Animation should not be a refinement from the formal specification. The underlying strategy of refinement is via weakening the precondition and strengthening the postcondition of a particular specification. These refinement steps would certainly make acceptable changes to the input and output domains of the system, which is appropriate to the implementation process but not adoptable in the animation stage. If the logical equivalence is not preserved it will not only have negative effects on the validation process of the formal model but also mislead the refinement process throughout the implementation stages. Thus the animation should be kept as close as possible to its original specification. The runnable code need not be highly efficient, as in the final implementation, since our focus here is to demonstrate the logical relationships and to maintain the logical preciseness. In summary, specification animation differs from implementation in data, level of abstraction, algorithm, efficiency, performance and so on.

5.3 Animation language - Oz a candidate for TCOZ

Generally speaking, any programming language could be used for animation. No matter what programming language the actual system is written in, the requirements stay the same, as does the specification. However, each programming language has its own specialized features which are most suitable for coding particular types of problems. For example, Java is good at web programming, Prolog is good at AI programming, PowerBuilder is good at database applications and so on. Coding in a language according to its desired features is much easier than that of the others. That is why so many different types of programming languages coexist: to meet all kinds of needs. An animation language has its own set of metrics as well. An animation system consists of a translator that translates original specifications into an animation language, and an evaluator that validates the corresponding executable specifications in the animation language. Thus the logic abstract level and degree of similarity in syntax and semantic with the formal notation should be a criteria of selection, e.g., animating Z using Prolog [112]. If the animation language were the formal notation itself then the translator would be unnecessary. Since most animation languages are different from formal specification notations, one solution is to provide an equivalent library that handles the specification constructs. The completeness of an exiting library compared to the formal notation could be another measure for selection. Once the specification has been turned into the format of the animation language it is time for validation. Running properties of the evaluator, such as efficiency, termination and so on, would be another criterion for choosing an animation language. Thus we select a programming language that has a high logic abstraction level, contains most of the features of the specification notation, along with properly designed library functions.

The programming language Oz [39, 91, 42] is a multi-paradigm language based on the concurrent constraint model. It is a high-level programming language that is designed for modern advanced, concurrent, intelligent, networked, soft real-time, parallel, interactive and pro-active applications. Oz provides the programmers and system developers with a wide range of programming abstractions to enable them to develop complex applications quickly and without the confinements of the underlying paradigm. It merges several paradigms of programming languages such as object orientation, constraint and logic programming, functional programming and concurrent programming into a single coherent design.

- Object-oriented programming – provides state, abstract data types, classes, objects, and inheritance.
- Functional programming – provides compositional syntax, first-class procedures, and lexical scoping.
- Logic & constraint programming – provides logic variables, disjunctive constructs, and programmable search strategies.
- Concurrent programming – provides thread invocation and interaction.
- Distributed programming – provides network-transparent distribution of Oz computations and language security; sharing variables, objects, classes, and procedures.

With the above features, Object orientation in Object-Z, concurrency in TCSP and the mixture of the two in TCOZ all find corresponding features in Oz. With the help of proper library functions and logic programming features, integrated formal notations such as TCOZ can be well animated in Oz.

5.4 TCOZ – Oz translation rules

We provide a translation guideline from TCOZ to Oz. This translation gives a runnable semantics for TCOZ in Oz. Some rules are defined as follows.

- Data types are translated into given sets in Oz. Because Oz is a dynamic typed language, each data type represents a set of possible values that the variable could have. For the purpose of animation, these data type contain only a small set of finite possible terms.
- Sequence is translated to the 'List' data type in Oz; set and its corresponding functions are translated to the appropriate library functions.
- TCOZ class is given the same signature as an Oz class with its inheritance section expanded. Because TCOZ class construct has different inheritance rules from that of Oz class [1] .
- Type and function definitions local to a class are translated to local declarations for an Oz class.
- The type declaration of the state schema in TCOZ class is translated to

[1] In TCOZ inheritance, the declarations and predicates of the same name operations in the super-class and sub-class are merged together instead of the case of overloading in Oz.

membership relations adding to the precondition of the state invariant or methods in Oz.

- Object reference in a class definition is regarded as a feature type in Oz, which later can be linked to a concrete class object. If the object reference is common to all the instances of the class, declaration in the feature via an anonymous variable '_' indicates that all instances of the class will share the same variable, in our case the common referred object.

- Operations that are not in the visibility list of the TCOZ class are translated to methods labelled by variables instead of literals in an Oz class, and are private to the class.

- Generic class definition is translated to function definition with type information as its parameter. It returns an Oz class declaration.

- Channel is treated as features of the cell type in an Oz class, which later can be assigned in the system specification according to the network topology.

- Active object class is translated to an Oz class that inherits the Oz 'Time.repeat' class, which is capable of setting up an action method (main) for continuous running as a non-terminating object.

5.5 Implementation and case study

5.5.1 TCOZ Oz library

As discussed earlier, an equivalent library for handling specification constructs can greatly benefit the translation process from FM specifications into the animation language. Part of the Oz library to manipulate TCOZ constructs, e.g., set operations and channel declarations, are defined as follows.

```
% set
fun {PowerSet A}
   case A
   of nil then [nil]
   [] H|T then
      {Union {PowerSet T} {Map {PowerSet T}
        fun {$ X} {Append [H] X} end}}
   end
end
...
%Channel
class Channel from BaseObject
   attr buffer signal
   meth init
      buffer <- {New OzChannel init}
      signal <- {New OzEvent init}
   end
   meth put(I)
      {@signal wait}
      {@buffer put(I)}
   end
   meth get(?I)
      {@signal notify}
      {@buffer get(I)}
   end
end
```

Firstly, a number of set functions such as subset, power set, union, intersection and so on are defined for matching the TCOZ set constructs. Secondly, TCOZ communication constructs such as channel are implemented using the concurrent programming aspects in Oz. The last example shows a TCOZ channel, which is shared among an arbitrary number of threads. Note that these functions are implemented using the logic programming aspects in Oz, which preserve the same abstraction level as the specification notation. We have completed the entire TCOZ set operations in Oz, and only a few are demonstrated in this chapter due to space limitations.

In the previous example, we programmed a signaling mechanism using a typical producers and consumers situation. This program relies on the use of logical variables to achieve the desired synchronization. A consuming thread has to wait until information exists in the channel. The *get* method notifies one producer at a time by setting the empty flag and signalling one producer. This is done as an atomic step. Any producing thread may put information in the channel synchronously. The *put* method does the reciprocal action. Most execution is done in an exclusive region. Multiple consuming threads will reserve their place in the channel, thereby achieving fairness.

5.5.2 TCOZ Oz projection

To animate TCOZ specifications in Oz, we first use XSL Transformation to project the TCOZ model into its Oz code frames, together with test cases and an auxiliary library to perform the validation. Customization of the code segments are needed during the process. The following is part of the XSL stylesheet for Oz projection.

```
<xsl:output method="text"/>
<xsl:template match="/">
  <xsl:apply-templates select="//classdef"/>
</xsl:template>
<xsl:template match="classdef">
  <xsl:text>class </xsl:text>
  <xsl:value-of select="name"/>
  <xsl:text> from </xsl:text>
  <xsl:apply-templates select="inherit"/>
  <xsl:if test="op/name[.='MAIN']">
   <xsl:text> Time.repeat </xsl:text>
  </xsl:if> ...
  <xsl:apply-templates select="state"/>
  <xsl:apply-templates select="init"/>
  <xsl:apply-templates select="op"/>
  ...
</xsl:template>
```

The above states that a projection will be made on each defined TCOZ class in XML to construct corresponding Oz classes, i.e., the inheritance relationships are captured through the `inherit` tags, the active objects are identified by their `MAIN` operations and so on.

5.5.3 Two communicating buffers example

Consider the TCOZ model of the *Buffer* and *TwoBuffers* below. Let the given type [*MSG*] represent a set of messages.

Buffer

$items : \mathrm{seq}\, MSG$
$left, right : \mathbf{chan}$

INIT
$items = \langle\ \rangle$

Add

$\Delta(items)$
$i? : MSG$

$items' = \langle i? \rangle ^\frown items$

Remove

$\Delta(items)$

$items = items' ^\frown \langle last(items) \rangle$

$Join \mathrel{\widehat=} [i : MSG \mid \#items < max] \bullet left?i \rightarrow Add$
$Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet right!last(items) \rightarrow Remove$
$\textsc{Main} \mathrel{\widehat=} \mu B \bullet Join \mathbin{\Box} Leave;\ B$

Two communicating buffers can be composed in TCOZ respectively as:

TwoBuffers

$l : Buffer[middle/right]$
$r : Buffer[middle/left]$

$\textsc{Main} \mathrel{\widehat=} \|(l \xleftrightarrow{middle} r)$

Note that the two buffers are communicating through the *middle* channel, which is depicted by the TCOZ network topology seen in Figure 5.1.

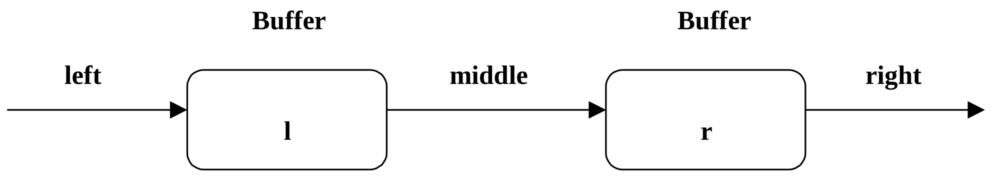


Figure 5.1: Two communicating buffers example

The translated specifications in Oz are as follows.

```
%Buffer
class Buffer from Time.repeat
  feat
    left
    right
  attr
    items
  meth Invariants($)
    {All @items fun {$ X} {Member X MSG} end}
  end
  meth init
    items <- nil
  end
  meth Add(I)
    cond
      ({Member I MSG} andthen
      {self Invariants($)}) = true
    then
      items <- {Append @items [I]}
    else
       {Browse 'Pre-condition not satisfied.'}
    end
  end
  ...
end
```

Note that the preconditions in the TCOZ schema are treated as the logical conditional statements `cond` in Oz. The '`else`' statement is introduced for execution purposes only. Without the statement, the process will hang when the preconditions are not satisfied. An Oz `cond` statement has the following semantics. Assume a thread is executing the statement in space SP [2] and has the following form.

```
  cond X1 ... XN in S0 then S1 else S2 end
```

where Xi are newly introduced variables, and Si are statements. X1 ... XN in S0 then S1 is the clause of the conditional, and S2 is the alternative.

- The thread is blocked.

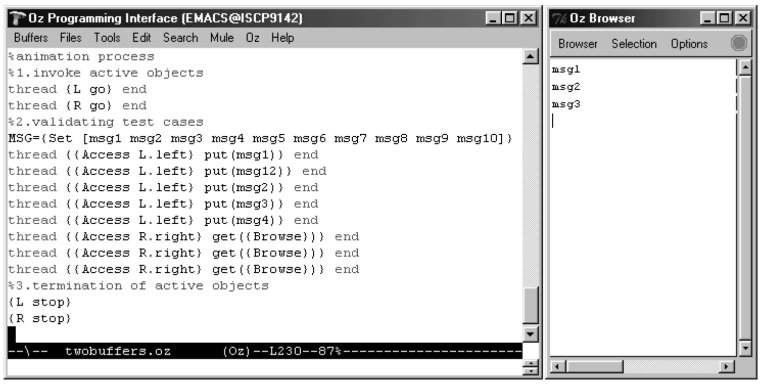
[2]*SP* denotes an Oz computation space, which consists of a computation store and a set of executing threads.

- A space SP_1 is created, with a single thread executing the guard *cond* $X1$ ... XN in $S0$.
- Execution of the parent thread remains blocked until SP_1 is either entailed or disentailed. Notice that these conditions may never occur, e.g. when some thread is suspended or running forever in SP_1.
- If SP_1 is disentailed, the parent thread continues with statement $S2$.
- If SP_1 is entailed, assume it has been reduced to the store θ and the set of local variables SX. In this case, the space is merged with the parent space. θ and SX added to the parent store, and the parent thread continues with the execution of $S1$.

The *TwoBuffers* example depicted by the TCOZ network topology can be translated into the following Oz segment.

```
%network topology
L = {New Buffer init}
R = {New Buffer init}
Left = {New Channel init}
Middle = {New Channel init}
Right = {New Channel init}
L.left = {NewCell Left}
L.right = {NewCell Middle}
R.left = {NewCell Middle}
R.right = {NewCell Right}
%active objects
{L setRepAll(action: main)}
{R setRepAll(action: main)}
```

From the translation rules defined in the previous section, we first create the instances of the *left*, *middle* and *right* channels; then associate these channels to its


Oz Programming Interface (EMACS@ISCP9142)
Buffers Files Tools Edit Search Mule Oz Help
Oz Browser
Browser Selection Options

```
%animation process
%1.invoke active objects
thread {L go} end
thread {R go} end
%2.validating test cases
MSG={Set [msg1 msg2 msg3 msg4 msg5 msg6 msg7 msg8 msg9 msg10]}
thread {{Access L.left} put(msg1)} end
thread {{Access L.left} put(msg12)} end
thread {{Access L.left} put(msg2)} end
thread {{Access L.left} put(msg3)} end
thread {{Access L.left} put(msg4)} end
thread {{Access R.right} get({Browse})} end
thread {{Access R.right} get({Browse})} end
thread {{Access R.right} get({Browse})} end
%3.termination of active objects
{L stop}
{R stop}

--\--  twobuffers.oz      (Oz)--L230--87%--------------------
```

```
msg1
msg2
msg3
```

Figure 5.2: Animation of the two communicating buffers example

corresponding feature variables in the `Buffer` definition according to the network topology of the TCOZ specification. The function `setRepAll` is to set up an action for the TCOZ active objects.

After the translation from TCOZ specification into Oz, it is time to build up test cases and carry out the validating process. As seen from Figure 5.2, we firstly invoked the two active objects and let them run concurrently in their own threads. Then, five inputs along the *left* channel of the *TwoBuffers* were put into the system. Note that one of them, `msg12`, is outside of the `MSG` type range. When obtaining three outputs through the *right* channel the results are `msg1`, `msg2` and `msg3`. Note that `msg12` was checked by the state invariants and ignored. Furthermore, the desired output is the consequence of the *TwoBuffers* communicating through its internal *middle* channel performed by two active `Buffer`s, which match perfectly with the corresponding TCOZ specification as well as the user requirements.

5.6 Conclusion

The contribution of this chapter is the demonstration of an approach to animate TCOZ specifications in a multi-paradigm language - Oz. With the availability of all kinds of programming concepts in Oz, e.g., OO, logic and concurrency, we defined a TCOZ constructs library so that animating TCOZ model in Oz can be easily and effectively achieved. We also constructed an XSLT stylesheet for the automatic transformation from TCOZ specifications into Oz code frames. However, our translating and validating processes still need human interaction for complicated predicate expressions at the moment. A more sophisticated translation tool can be built based on the TCOZ XML format to Oz syntax. This will be part of our future work.

Chapter 6

Proof techniques for TCOZ

This chapter presents a proof system for the TCOZ specification language and a framework of logical embedding of inference rules into the theorem prover Isabelle.

6.1 Introduction

Formally reasoning about properties of a system specification involves showing that the properties can be derived from the specification using the rules of a mathematically sound logic. This logic is given by a formal system which defines a set of axioms and a set of inference rules. The approach is to provide a complete logic and a set of inference rules for the particular specification language. The work done by Graeme Smith, Jim Davis and Steve Schneider takes this approach. Smith extended the W logic of Z to Object Z with class features [86]. He presented a set of inference rules for reasoning about classes including inheritance, parallelism, class membership and so on. Davis/Schneider extended the proof system of Hoare's Communicating Sequential Process (CSP) to accommodate reasoning about complex timing constraints for TCSP [82, 83]. Thus, in order to formally verify system properties, a proof system for TCOZ is needed. TCOZ preserves a large part of both the syntax and semantics of the base notations and hence can potentially benefit from existing reasoning systems of the individual notations. In this chapter we extend and link Smith's proof system of Object-Z [1] and Davies/Schneider's proof system of TCSP for reasoning about TCOZ models. The new proof rules for the TCOZ novel constructs, i.e., active objects, sensor/actuators, network topology, deadline and wait-until commands, etc, are developed. Furthermore, a framework for the embedding of TCOZ event reasoning rules into the generic theorem prover Isabelle/HOL is demonstrated.

[1]The proof system of Object-Z [86] extends the $\mathcal{W}$ logic [114] of Z.

The remainder of the chapter is organized as follows. In section 2, we briefly introduce the TCOZ inference rules. Section 3 presents the encoding of TCOZ event reasoning rules in Isabelle/HOL. Section 4 concludes the chapter.

6.2 TCOZ inference rules

Timed Communicating Object-Z (TCOZ) [67], an integration of object oriented Z and TCSP, introduces CSP primitives inside Object-Z class definitions for modelling timing related aspects. The proof systems of Object-Z and TCSP can be adopted and extended to facilitate reasoning about both state and event oriented properties of a TCOZ specification.

6.2.1 State oriented reasoning

Adopted Object-Z rules

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. From a system point of view, it also enables modular verification. Smith extends the W logic of Z to Object-Z [86] for reasoning about object models and type systems [78]. The fundamental logic in Object-Z is presented in the sequent form, defined as follows:

$$A :: d \mid \Psi \vdash \Phi$$

where A is the name of a class, d is a list of declarations and Ψ and Φ are lists of predicates in the local content of class A. Inference rules are also restricted in the local environment of a class context.

$$\frac{A :: d_1 \mid \Psi_1 \vdash \Phi_1}{A :: d_2 \mid \Psi_2 \vdash \Phi_2} \; [\, A :: p \,]$$

The upper part is called a premise which contains zero or more sequents; the middle part is called a proviso which is a predicate that makes the rule applicable; and the lower part is called the conclusion, which is a single sequent which must be valid when the proviso and the premise are true.

Some of the inference rules adopted from Object-Z are listed as follows. Detailed information of Object-Z inference rules can be found in Smith's logic for Object-Z [86].

- State definition – For a state definition of class $A[X_1, ..., X_n]$, the state schema inference rule is defined as follows:

$$\begin{array}{|l}
\hline
d_1 \\
\Delta \\
d_2 \\
\hline
p \\
\hline
\end{array}$$

$$\frac{A[t_1, ..., t_n] :: STATE = [\uparrow STATE;\ b \odot d_1;\ b \odot d_2;\ \mid b \odot p] \vdash}{A[t_1, ..., t_n] :: \ \vdash} \; [\, q \,]$$

  $STATE$ refers to the state definition of a class, $\uparrow STATE$ stands for the inherited state definitions from its super classes, and the proviso q is in the form of

  $$q \equiv b = \langle\!|\ X_1 \leadsto t_1, ..., X_n \leadsto t_n\ |\!\rangle$$

where t_i is the actual parameter substituted to X_i through substitution operator $\odot$.

- Initialization definition – For a generic initial schema definition Op of class $A[X_1, ..., X_n]$, the initial schema inference rule is defined as follows:

  $$\begin{array}{|l} \hline \textit{INIT} \\ \quad p \\ \hline \end{array}$$

  $$\frac{A[t_1, ..., t_n] :: \textit{INIT} = \uparrow \textit{INIT} \wedge [\textit{STATE} \mid b \odot p] \vdash}{A[t_1, ..., t_n] :: \ \vdash} [\ q\ ]$$

  $\textit{INIT}$ refers to the initial definition of a class, $\uparrow \textit{INIT}$ stands for the inherited initial definitions from its super classes, and the proviso q is in the form of $q \equiv b = \llparenthesis\ X_1 \leadsto t_1, ..., X_n \leadsto t_n\ \rrparenthesis$.

- Operation definition – For a generic operation schema of class $A[X_1, ..., X_n]$, the inference rule for operation is defined as follows:

  $Op \mathrel{\widehat{=}} OP$

  $$\frac{A[t_1, ..., t_n] :: Op = \Delta \textit{STATE} \bullet (\uparrow Op \wedge b \odot OP) \vdash}{A[t_1, ..., t_n] :: \ \vdash} [\ q\ ]$$

  The Op refers to an operation definition in a class, $\uparrow Op$ stands for the inherited operation definitions from its super classes, and the proviso q is in the form of

  $q \equiv b = \llparenthesis\ X_1 \leadsto t_1, ..., X_n \leadsto t_n\ \rrparenthesis$

- Inheritance related rules

  $$\frac{\begin{array}{l} A_1 :: \ \vdash \textit{STATE} = S_1 \\ ... \\ A_n :: \ \vdash \textit{STATE} = S_n \end{array}}{B :: \ \vdash \ \uparrow \textit{STATE} = S_1 \wedge ... \wedge S_n} [\ p\ ]$$

$$\frac{\begin{array}{l} A_1 :: \ \vdash INIT = S_1 \\ ... \\ A_n :: \ \vdash INIT = S_n \end{array}}{B :: \ \vdash \ \uparrow INIT = S_1 \wedge ... \wedge S_n} [\, p \,]$$

$$\frac{\begin{array}{l} A_1 :: \ \vdash Op = S_1 \\ ... \\ A_n :: \ \vdash Op = S_n \end{array}}{B :: \ \vdash \ \uparrow Op = S_1 \wedge ... \wedge S_n} [\, p \,]$$

$\uparrow STATE$, $\uparrow INIT$ and $\uparrow Op$ stand for the inherited state, initial and operation definitions respectively. The proviso p is in the form of $p \equiv \iota(B) = \{A_1, ..., A_n\}$, where the meta function ι returns the set of inherited classes of B.

TCOZ extension rules

TCOZ [67] extends Object-Z class definitions in two aspects. Firstly, the state schema convention is extended to allow the declaration of object communication interfaces, i.e., channels, sensors and actuators. If c is to be used as a communication interface by any of the operations of a class, then it must be explicitly declared in the state schema. Channels are type heterogeneous and may carry communications of any type, while sensors/actuators are type specific. These communication interfaces are connected by the network topologies in TCOZ. The second extension is that in addition to operations (terminating processes), non-terminating processes named *MAIN* are introduced to represent the behavior of active classes. The inheritance mechanism of active classes differs from the normal passive classes as the *MAIN* operation must always be redefined explicitly. For a complete TCOZ semantics refer to paper [65].

Based on the Object-Z logics just outlined, new extension rules in TCOZ are defined below:

- Non-terminating process (MAIN) – For a generic *MAIN* definition of class $A[X_1, ..., X_n]$, the inference rule is defined as follows:

  $$MAIN \mathrel{\widehat{=}} OP$$

  $$\frac{A[t_1, ..., t_n] :: MAIN = \Delta STATE \bullet (b \odot OP) \vdash}{A[t_1, ..., t_n] :: \ \vdash} \ [\ q\ ]$$

  MAIN refers to the non-terminating process definition in an active class. Note that there is no need to consider the inherited *MAIN* definitions from its super-classes since the process *MAIN* must always be redefined in the subclasses if it appears. The proviso q is in the form of $q \equiv b = \llparenthesis\ X_1 \leadsto t_1, ..., X_n \leadsto t_n\ \rrparenthesis$.

- Synchronous communication (Channel) – For a generic network topology definition of classes A, B and AB, the channel inference rule is defined as follows:

```
A
  c : chan
  ...
  MAIN ≙ ...c!x...

B
  c : chan
  ...
  MAIN ≙ ...c?x...

AB
  a : A
  b : B
  ...
  MAIN ≙ ...a ←c→ b...
```

  $$MAIN \mathrel{\widehat{=}} ...a \xleftrightarrow{c} b...$$

$$\frac{\begin{array}{l} A[t_1, ..., t_n] :: STATE \vdash c \in \mathbf{chan} \wedge MAIN \vdash c!x \in X \\ B[t_1, ..., t_n] :: STATE \vdash c \in \mathbf{chan} \\ AB[t_1, ..., t_n] :: STATE \vdash a \in A \wedge b \in B \wedge MAIN \vdash a \xleftrightarrow{c} b \end{array}}{B[t_1, ..., t_n] :: MAIN \vdash c?x \in X} \; [\, q \,]$$

The above states that if classes A and B are communicating through channel c, synchronization will be enforced on the input and outputs, i.e., outputs from A through c will lead to inputs to B.

- Asynchronous communication (Sensor and Actuator) – For a generic network topology definition of classes A, B and AB, the sensor/actuator inference rule is defined as follows:

A

$s : X$ **actuator**
...

...

B

$s : X$ **sensor**
...

$MAIN \mathrel{\widehat=} ...s?x...$

AB

$a : A$
$b : B$
...

$MAIN \mathrel{\widehat=} ...a \xleftrightarrow{s} b...$

$$\frac{\begin{array}{l} A[t_1, ..., t_n] :: STATE \vdash s \in X\ \mathbf{actuator} \\ B[t_1, ..., t_n] :: STATE \vdash s \in X\ \mathbf{sensor} \\ AB[t_1, ..., t_n] :: STATE \vdash a \in A \wedge b \in B \wedge MAIN \vdash a \xleftrightarrow{s} b \end{array}}{B[t_1, ..., t_n] :: MAIN \vdash s.x \in X} \; [\, q \,]$$

The rule states that if classes A and B are communicating through the sensor and actuator mechanism s, synchronization will be enforced on the input and outputs, i.e., implicit continuous outputs from A through s will lead to inputs to B when needed.

6.2.2 Event oriented reasoning

TCSP [82] is an extension of Hoare's Communicating Sequential Process (CSP) to accommodate the description of time-sensitive behaviors. A requirements specification $S(s, \aleph)$ of TCSP processes in TCOZ is the possible observations that can be made for their executions. These are described in terms of the timed failure model $(s, \aleph)$, which consists of timed traces and timed refusals. Timed trace s is the sequence of events occurring during the execution according to their timing aspects, while the timed refusal set $\aleph$ is the timed events which are refused by the execution. A process Q meets a specification $S(s, \aleph)$ if S holds for every timed failure associated with Q.

$$Q \ \mathbf{sat} \ S(s, \aleph) \Leftrightarrow \forall (s, \aleph) \in \mathcal{TF}[\![Q]\!] \bullet S(s, \aleph)$$

TCSP rules in TCOZ

The approach taken in the TCOZ notation is to identify operations as terminating CSP processes and to model active objects as non-terminating CSP processes. With operations given the same semantics as processes, TCSP primitives are adopted in the class constructs with satisfaction of the timed failure model restricted to the class constructs. Furthermore, the combination of simple operations with CSP operators makes it possible to represent true multi-threaded computation at the operation level. Therefore the satisfaction properties in a TCOZ specification with respect to TCSP aspects are extended to be restricted inside the local environment of a class context as follows:

$$A :: \ Q \ \mathbf{sat} \ S(s, \aleph) \Leftrightarrow A :: \ \forall (s, \aleph) \in \mathcal{TF}[\![Q]\!] \bullet S(s, \aleph)$$

Some of the TCSP inference rules adopted in the TCOZ context are listed below. For a detailed view of TCSP inference rules, please refer to the TCSP proof system [82].

- Conjunction – If a process satisfies two different specifications they also satisfy its conjunction.

$$\frac{\begin{array}{l} A :: Q \ \mathbf{sat}\ S(s, \aleph) \\ A :: Q \ \mathbf{sat}\ T(s, \aleph) \end{array}}{A :: Q \ \mathbf{sat}\ (S(s, \aleph) \wedge T(s, \aleph))}$$

- Weaken – If a specification S logically implies another specification T, then every process that satisfies S also satisfies the weaker specification T.

$$\frac{\begin{array}{l} A :: Q \ \mathbf{sat}\ S(s, \aleph) \\ A :: S(s, \aleph) \Rightarrow T(s, \aleph) \end{array}}{A :: Q \ \mathbf{sat}\ T(s, \aleph)}$$

- Sequential composition – The behavior of the process can be divided into two aspects. If control has not been transferred from Q_1 to Q_2, then the trace of the composition is the trace of Q_1 during which the termination ✓ is not performed and would be refused if offered. Otherwise, the trace is a concatenation of s_1 and s_2 performed by S_1 and S_2 respectively.

$$\frac{\begin{array}{l} A :: Q_1 \ \mathbf{sat}\ S_1(s, \aleph) \\ A :: Q_2 \ \mathbf{sat}\ S_2(s, \aleph) \end{array}}{\begin{array}{l} A :: Q_1 \mathbin{;} Q_2 \ \mathbf{sat}\ \checkmark \notin \sigma(s) \wedge S_1(s, \aleph \cup [0, \infty) \times \{\checkmark\}) \\ \qquad \vee \exists\, s_1, s_2, t \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) \\ \qquad\qquad \wedge S_1(s_1 \frown \langle (t, \checkmark) \rangle, \aleph \upharpoonright t \cup [0, t) \times \{\checkmark\}) \\ \qquad\qquad \wedge S_2((s_2, \aleph) - t) \end{array}}$$

- External choice – The combination behavior of the term is either Q_1 or Q_2. Any event refused before the first observable event occurs must be refused by both processes.

$$\frac{\begin{array}{l} A :: Q_1 \ \mathbf{sat} \ S_1(s, \aleph) \\ A :: Q_2 \ \mathbf{sat} \ S_2(s, \aleph) \end{array}}{\begin{array}{l} A :: Q_1 \ \Box \ Q_2 \ \mathbf{sat} \ (S_1(s, \aleph) \vee S_2(s, \aleph)) \\ \qquad \wedge \ S_1(\langle \, \rangle, \aleph \restriction \mathit{begin}(s)) \wedge S_2(\langle \, \rangle, \aleph \restriction \mathit{begin}(s)) \end{array}}$$

- Recursion – To prove that a recursive process $Y = F(Y)$ satisfies a requirement specification $S(s, \aleph)$, it is sufficient to show that under the hypothesis that Y satisfies $S(s, \aleph)$ and its definition $F(Y)$ also satisfies $S(s, \aleph)$.

$$\frac{A :: \forall \, Y \bullet (Y \ \mathbf{sat} \ S(s, \aleph) \Rightarrow F(Y) \ \mathbf{sat} \ S(s, \aleph))}{A :: \ Y = F(Y) \ \mathbf{sat} \ S(s, \aleph)} \ [\, q \,]$$

where $q = \exists \, Q_0 \bullet Q_0 \ \mathbf{sat} \ S(s, \aleph), S(s, \aleph)$ is admissible.

TCOZ extension rules

In this section, we develop the proof rules for the new TCOZ constructs, i.e., Deadline, WaitUntil commands and Network Topology. In presenting the inference rules, we first use the timed labelled transition system notation to provide operational semantics for each language constructs. Based on their operational semantics, timed failure models of the language constructs can be calculated. Finally, inference rules are derived from their corresponding timed failure semantic (denotational). In doing so, the soundness property of the inference rules can be preserved directly from the denotational semantics of each language constructs.

Deadline command

The DEADLINE operator ($Q \bullet \textsc{Deadline}\, d$) allows the successful termination of process Q to be restricted within the d units of time starting from the beginning of first occurrence in Q. The operational semantics for this operator are given as follows:

$$\frac{Q \xrightarrow{a} Q'}{Q \bullet \textsc{Deadline}\, d \xrightarrow{a} Q' \bullet \textsc{Deadline}\, d}\,[\, a \neq \checkmark \,]$$

$$\frac{Q \xrightarrow{\checkmark} Q'}{Q \bullet \textsc{Deadline}\, d \xrightarrow{\checkmark} Q'}$$

$$\frac{}{Q \bullet \textsc{Deadline}\, 0 \xrightarrow{\tau} \textsc{Stop}}$$

$$\frac{Q \overset{d'}{\rightsquigarrow} Q'}{Q \bullet \textsc{Deadline}\, d \overset{d'}{\rightsquigarrow} Q' \bullet \textsc{Deadline}(d - d')}\,[\, d' \leqslant d \,]$$

where '$\rightarrow$' refers to an event transition and '$\rightsquigarrow$' refers to an evolution transition. The above states that the process has the same effect as Q, but is constrained to terminate no later than d. If it fails to terminate by time d, it deadlocks. According the above operational semantics, its timed failure computation is as follows:

$$\begin{aligned}&\mathcal{TF}[\![Q \bullet \textsc{Deadline}\, d]\!] = \{(s, \aleph) \mid end(s) \leqslant d \wedge \checkmark \in \sigma(s) \wedge (s, \aleph) \in \mathcal{TF}[\![Q]\!]\}\\ &\quad \cup \{(s, \aleph) \mid end(s) > d \wedge \checkmark \notin \sigma(s) \wedge (s, \aleph) \in \mathcal{TF}[\![\textsc{Stop}]\!]\}\end{aligned}$$

The inference rule for the DEADLINE constructor can be derived from the timed failure semantics as follows:

$$\frac{A :: Q \ \mathbf{sat}\ S(s, \aleph)}{\begin{aligned}&A :: Q \bullet \textsc{Deadline}\, d \ \mathbf{sat}\ (end(s) \leqslant d \wedge \checkmark \in \sigma(s) \wedge S(s, \aleph \upharpoonright d))\\ &\quad \vee (end(s) > d \wedge S(\langle\rangle, (d, \infty) \times \Sigma^{\checkmark}))\end{aligned}}$$

WaitUntil command

The $\textsc{WaitUntil}$ operator ($Q \bullet \textsc{WaitUntil}\, d$) allows the period of execution of the process Q to be extended to d units of time starting from the first occurrence in Q, if the process terminates before d. The operational semantics for this operator are given as follows:

$$\frac{Q \xrightarrow{a} Q'}{Q \bullet \textsc{WaitUntil}\, d \xrightarrow{\mu} Q' \bullet \textsc{WaitUntil}\, d}\ [\ a \neq \checkmark\ ]$$

$$\frac{Q \xrightarrow{\checkmark} Q'}{Q \bullet \textsc{WaitUntil}\, d \xrightarrow{\checkmark} Q' \mathbin{;} \textsc{Wait}\, d}$$

$$\frac{}{Q \bullet \textsc{WaitUntil}\, 0 \xrightarrow{\tau} Q}$$

$$\frac{Q \overset{d'}{\rightsquigarrow} Q'}{Q \bullet \textsc{WaitUntil}\, d \overset{d'}{\rightsquigarrow} Q' \bullet \textsc{WaitUntil}(d - d')}\ [\ d' \leqslant d\ ]$$

The above states that the process has the same effect as Q, but it will not terminate until at least time d. According the above operational semantics, its timed failure semantics can be defined as follows:

$$\begin{aligned}
\mathcal{TF}[\![Q \bullet \textsc{WaitUntil}\, d]\!] = \{(s_1 \frown s_2, \aleph) \mid\ & end(s_1) < d \\
& \wedge (s_1, \aleph \upharpoonright end(s_1)) \in \mathcal{TF}[\![Q]\!] \wedge \\
& ((s_2, \aleph) - end(s_1)) \in \mathcal{TF}[\![\textsc{Wait}(d - end(s_1))]\!])\} \\
& \cup \{(s, \aleph) \mid end(s) \geqslant d \wedge (s, \aleph) \in \mathcal{TF}[\![Q]\!]\}
\end{aligned}$$

The inference rule for the $\textsc{WaitUntil}$ constructor can be derived from its timed failure semantics (denotational) as follows:

$$\frac{A :: Q \ \mathbf{sat}\ S(s, \aleph)}{\begin{aligned} A :: Q \bullet \textsc{WaitUntil}\, d \ \mathbf{sat}\ & (end(s) > d \wedge S(s, \aleph)) \\ & \vee (end(s) \leqslant d \wedge S(s \frown \langle d, \checkmark \rangle, \aleph \cup [end(s), d) \times \Sigma^{\checkmark})) \end{aligned}}$$

Network topology

The TCOZ network topology construct is a graphically-based representation of the TCSP parallel operator, where communications are made through common interfaces such as channels and sensor/actuators. Two types of communication mechanisms are introduced in the network topology structure: synchronous and asynchronous. In the case of synchronous communication, an output and input relationship needs to be explicitly specified along the common channel. For asynchronous communication, an actuator acts as continuously outputting its value to the environment, and the sensor acquires the value when needed. The operational semantics for the network topology operator are developed as follows:

$$\frac{Q_1 \xrightarrow{a} Q_1'}{Q_1 \overset{c}{\longleftrightarrow} Q_2 \xrightarrow{a} Q_1' \overset{c}{\longleftrightarrow} Q_2}\ [\ a \in \Sigma - \{c.v\}\ ]$$

$$\frac{Q_2 \xrightarrow{a} Q_2'}{Q_1 \overset{c}{\longleftrightarrow} Q_2 \xrightarrow{a} Q_1 \overset{c}{\longleftrightarrow} Q_2'}\ [\ a \in \Sigma - \{c.v\}\ ]$$

$$\frac{Q_1 \xrightarrow{c.v} Q_1' \quad Q_2 \xrightarrow{c.v} Q_2'}{Q_1 \overset{c}{\longleftrightarrow} Q_2 \xrightarrow{c.v} Q_1' \overset{c}{\longleftrightarrow} Q_2'}\ [\ c.v \in \Sigma\ ]$$

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1' \quad Q_2 \overset{d}{\rightsquigarrow} Q_2'}{Q_1 \overset{c}{\longleftrightarrow} Q_2 \overset{d}{\rightsquigarrow} Q_1' \overset{c}{\longleftrightarrow} Q_2'}$$

Note that $c.v$ is a compound common event indicating that the value v being communicated along synchronous channel or a pair of asynchronous sensor and actuator labelled c. The above is represented in terms of the timed failure model as follows:

$$\begin{aligned}\mathcal{TF}[\![Q_1 \xleftrightarrow{c} Q_2]\!] = \{(s, \aleph) \mid \exists\, \aleph_1, \aleph_2 \bullet \aleph = \aleph_1 \cup \aleph_2 \\ \wedge\ (s \upharpoonright \sigma(Q_1), \aleph_1) \in \mathcal{TF}[\![Q_1]\!] \wedge (s \upharpoonright \sigma(Q_2), \aleph_2) \in \mathcal{TF}[\![Q_2]\!]\}\end{aligned}$$

where $\sigma(Q)$ denotes the alphabet of the process Q. Note that both Q_1 and Q_2 must agree on the communication event $c.v$. If either of them refuses the event then the communication will be refused. The inference rule for the network topology constructor is presented as follows:

$$\frac{\begin{array}{l} A :: Q_1 \ \mathbf{sat}\ S_1(s, \aleph) \\ B :: Q_2 \ \mathbf{sat}\ S_2(s, \aleph) \end{array}}{\begin{array}{l} AB :: Q_1 \xleftrightarrow{c} Q_2 \ \mathbf{sat}\ \exists\, \aleph_1, \aleph_2 \bullet S_1(s \upharpoonright \sigma(Q_1), \aleph_1) \\ \qquad \wedge\ S_2(s \upharpoonright \sigma(Q_2), \aleph_2) \wedge \aleph = \aleph_1 \cup \aleph_2 \end{array}}$$

6.3 Towards automated proof assistance

In the above section we presented a combination and extension of state (Object-Z) and event based (TCSP) proof systems for formal reasoning about TCOZ specifications. As the proof process is manual, one immediate work is to investigate the encoding of TCOZ proof rules into theorem provers such as Isabelle/HOL [73] to support automatic proof assistance. There are previous research works done in embedding Z and CSP into theorem provers, such as HOL-Z [54], HOL-CSP [97]. Both approaches are based on the shallow embedding of the language semantics into the generic prover Isabelle. HOL-Z is a structure preserving encoding of Z into the higher-order logic, which allows the deduction to be performed at the schema level. HOL-CSP presented a machine verified failure divergence model for the CSP language in Isabelle/HOL. There are also previous attempts to embed CSP into the PVS prover [10, 27].

Isabelle is a generic system for implementing a logical formalism. It consists of many logics, such as First Order Logic (FOL), High Order Logic (HOL), Zermelo Frankel set theory (ZF), Constructive Type Theory (CTT), the Logic of Computable Functions (LCF), and so on. The Isabelle system is implemented in the functional language ML. It provides powerful mechanisms to define hierarchical theories (object logics). New object logics can be built from Isabelle metalogic, by means of constructing and proving new theories. Its fundamental inference techniques are based on higher order unification and term rewriting. Isabelle/HOL is the specialization of Isabelle for high order logic.

TCOZ is essentially a blending of Object-Z with TCSP. Therefore the encoding of TCOZ language to the theorem prover can be divided into two stages: first, the encoding of state (Object-Z) and event based (TCSP) semantics into Isabelle/HOL, then the extending and developing of proof rules to accommodate the TCOZ language. Recently, attempts have been made to encode Object-Z into Isabelle [90]. In this section we present some tentative approaches for the encoding of the TCOZ event reasoning rules into Isabelle/HOL.

6.3.1 Timed failure and process

TCSP's timed failure semantic is used as a basis for the logic embedding. A timed failure is a pair that consist of a timed trace and a timed refusal as mentioned in section 6.2.2. The corresponding representations in Isabelle/HOL are as follows:

```
datatype 'a event = ev 'a | tick
typedef (time)
```

```
  time = "{t. t : nat & 0 < t}"
  by (auto)
types
  'a t_event = "('t time * 'a event)"
consts
  is_ttrace :: "('a t_event) list => bool"
typedef (t_trace)
  'a t_trace = "{tt. is_ttrace tt}"
  by (auto)
types
  'a t_refusal = "('a t_event) set"
  'a t_failure = "'a t_trace * 'a t_refusal"
consts
  is_process  :: "'a t_failure set => bool"
typedef (process)
  'a process = "{p. is_process p}"
  by (auto)
```

The above defines the fundamental concepts of the timed failure semantic, i.e., trace, refusal, failure and process. Note that the timed trace and process were defined as sets of timed event and timed failures that satisfy desired properties. Before we move on to the definition of process and rules, let us look at some auxiliary functions for such definitions.

$s \mid\mid\upharpoonright t$	[timed trace that strictly before t $\longrightarrow$ befs]
$s \upharpoonright t$	[timed trace that before and at t $\longrightarrow$ befeqs]
$\aleph \mid\mid\upharpoonright t$	[timed refusal that strictly before t $\longrightarrow$ befx]
$\aleph \upharpoonright t$	[timed refusal that before and at t $\longrightarrow$ befeqx]
$\sigma(s)$	[alphabet of events in a timed trace s $\longrightarrow$ sigmas]
$\sigma(\aleph)$	[alphabet of events in a timed refusal $\aleph$ $\longrightarrow$ sigmax]

Some of their corresponding Isabelle representations are as follows.

```
constdefs
  befs :: "['tt t_trace, 't time] => 'tt t_trace"
    "befs tt t == "{tt'::t_trace. tt' <= tt &
      ! t'::time a::event. (t', a) : tt & t' < t
        --> (t', a) : tt'}"
  ...
  sigmax :: "'a t_refusal => 'a event set"
    "sigmax tr == "{a::event. (t,a) : tr}"
  ...
```

It states that the '`befs s t`' function maps the original timed trace `s` onto the time interval `[0, t)`. Other functions can be encoded similarly. The constraints on the timed trace and process are defined as follows.

```
defs
  is_ttrace_def : "is_ttrace TT == ! t1 t2 a1 a2 .
    [(t1::time, a1::event), (t2, a2)] <= TT
      --> (t1 <= t2 & a1 ~= tick)"
  is_process_def : "is_process P == ([],{}) : P &
    (! s t X Y. (t, Y) : P & (s X) <= (t, Y) --> (s, X) : P) &
    (! s X. (s, X) : P & ? X'. X <= X' & (s, X') : P &
      ! t::time a::event . (t, a) ~: X' -->
      ((befs s t)@[(t,a)], (befx X' t)) : P)"
```

For the timed trace, it is a sequence of timed events in which times are non-decreasing. A well-timed CSP process is a set of timed failures that represents the execution records. These timed failures should meet certain properties. First, it must contain the empty observation which denotes no event occurring. Second, it must be downwards closed, which means that any prefix order of a timed failure should be included in its timed failure set. A prefix order of a timed failure $\preccurlyeq$ is defined as follows:

$$(s', \aleph') \preccurlyeq (s, \aleph) \Leftrightarrow \exists\, s'' \bullet s = s' \frown s'' \wedge \aleph' \subseteq \aleph \mid\upharpoonright \mathit{begin}(s'')$$

where $\mathit{begin}(s)$ denotes the first (earliest) time value in a timed trace s. Third, the timed event should be either possible or refusable, which means that if a timed event $(t,\ a)$ does not appear in the refusal set it must be included in the timed execution trace. All these properties are encoded in the function `is_process_def` as shown.

6.3.2 Language constructs

With the above concepts, TCSP language constructs can be defined such as:

```
constdefs
  STOP :: "'a process"
    "STOP == Abs_process {(s, X). s = []}"
  SKIP :: "'a process"
    "SKIP == Abs_process ({(s, X). s = [] & tick ~: (sigmax X)}
      Un {(s,X). s = [(t, tick)] & tick ~: (sigmax (befx X t))})"
  Seq :: "['a process,'a process] => 'a process"
    "Seq P Q == Abs_process ({(s, X). tick ~: (sigmas s) &
        (s, X Un (refAll infty {tick})) : Rep_process P}
      Un {(s, X). ? s1 s2. s = s1@s2 & tick ~: (sigmas s) &
        (s1@[(t,tick)], (befx X t) Un
          (refAll t {tick})) : Rep_process P &
        ((s2, X) - t) : Rep_process Q})"
  ...
```

The above illustrates the encoding of the ‘Stop’, ‘Skip’, and the composition ‘$\fatsemi$’ constructs. They are all strictly based on the timed failure semantics presented in Section 6.2.2. Other TCSP primitives can be defined similarly. In addition, having the language constructs in Isabelle/HOL, we can machine verify the correctness of the TCSP semantics by deriving lemmas and applying tactics.

6.3.3 Specification satisfaction and inference rules

With the TCSP language constructs embedded in Isabelle/HOL, the satisfaction of a process to its specification can be defined as follows.

```
constdefs
  Sat :: "['p process, 'a => bool] => bool"
    "Sat P S == ! s X. (s, X) : (Rep_process P) & S (s, X)"
```

It states that a process `P` meets a specification `S`, if and only if, for every timed failure associated to `P`, `S` is true. Thus the inference rules can be represented as lemmas or theorems in Isabelle/HOL, such as:

```
lemma sequential : " [| Sat Q1 S1 ; Sat Q2 S2 |] ==>
  Sat (Seq Q1 Q2) ((tick ~: (sigmas s) &
      S1 (s, X Un (refAll infty {tick})))
    | (? s1 s2. s = s1@s2 & tick ~: (sigmas s) &
      S1 (s1@[(t,tick)], (befx X t) Un (refAll t {tick})) &
      S2 ((s2, X) - t)))"
  ...
```

The above is the inference rule for sequential composition as introduced in the early section. By applying appropriate tactics, we can prove these inference rules automatically and use them for future reasoning. Alternatively, we could assert these rules as axioms into the theory and apply them directly. However, it is recommended to take the definitional approach rather than the axiomatic approach, for the latter may put forward arbitrary and inconsistent axioms. Finally, the TCOZ theory file can be an extension of both Object-Z and TCSP theory files with new language constructs and theorems.

```
Theory TCOZ = ObjectZ + TCSP:
consts
 Deadline :: "['a process, 't time] => 'a process"
 WaitUntil :: "['a process, 't time] => 'a process"
 Network :: "['a process, 'a event set, 'a process] => 'a process"
defs
  WaitUntil_def : "WaitUntil P t == Abs_process ({(s, X). ? s1 s2.
    s = s1@s2 & (end s1) < t & (s1, (befx X (end s1))) : Rep_process P
    & ((s2, X) - (end s1)) : Rep_process (Wait (t - (end s1)))}
    Un {(s, X). (end s1) >= t & (s, X) : Rep_process P}"
 ...
lemma waituntil : " [| Sat Q S |] ==>  Sat (WaitUntil Q d) (...)"
```

Note that '`Wait (t - (end s1))`' denotes the delay construct and the '`end s1`' computes the finish time of the timed trace `s1`. New theorems of TCOZ inference rules can be constructed accordingly. In summary, by such framework of encoding, TCOZ proof system can be verified and applied automatically for future reasoning tasks using Isabelle/HOL.

6.4 Conclusion

In this chapter, we combined and extended both state-based Object-Z and event-based TCSP proof systems for formally reasoning about TCOZ specifications. New inference rules for TCOZ novel constructs are introduced based on their underlying language semantics. A case study of applying these rules for the verification of a generic Computer Aided Dispatch System architecture will be demonstrated in the next chapter. Furthermore, in this chapter we also presented an initial framework for encoding the TCOZ language into the theorem prover Isabelle/HOL for automated proof support. The amount of work of constructing the theory files, verifying the proof rules and deduce new theorems from the theory files could be large. In addition, a parsing program can be written to take in the standard ZML format of a TCOZ specification and produce the corresponding Isabelle theory representations for formal verification.

Chapter 7

Verifying and reasoning about generic CAD system architecture - a case study

This chapter presents the formal modeling and verification of a generic Computer Aided Dispatch (CAD) System architecture.

7.1 Introduction

Software architecture is an important level of description for software systems [79]. It involves the definition of system elements, the interaction among the elements, patterns of the compositions, and the constraints on the patterns [85]. The current practice of software architecture mainly relies on diagrams and textural explanations. Several Architectural Description Languages (ADL) have been proposed, such as Darwin [63] and Rapide [61]. These ADLs offer approaches to describe software architectures explicitly as hierarchical structures. Formal modeling techniques have been applied to the software architecture descriptions. The well-defined semantics and syntax make them suitable for precisely specifying and formally verifying architecture designs. Many researchers [2, 85] have used Z to formalize the computational data/state aspects of software architectures. Allen and Garlan [5] have also applied a CSP-like notation called Wright to formalize the interactive communication aspects of software architectures. Both approaches are beneficial and provide some formal foundations to a software architecture description. However, the formal link and consistency issues between the models represented in different formalisms remain as a challenge. In general, Z is a state-based formalism which may not be suitable for specifying interactions; Wright is designed only for architecture communication definitions. From a system designer point of view, he/she might prefer to use a single and coherent language that can capture both static and dynamic properties of the system architecture. Thus the consistency issues between the models represented in different formalisms can be resolved. Recent advances in formal specification techniques and integrated formal methods [12, 36] may provide some promising solutions to the problem. In this chapter, we also show that integrated formal notations, i.e., Timed Communicating Object Z (TCOZ) [67] could be a good candidate for such architecture description through the design and verification of a generic Computer Aided Dispatch (CAD) system. The class construct in TCOZ is an ideal encapsulation mechanism for composing and extending architecture models. The synchronous and asynchronous communication interfaces in TCOZ are well suited for capturing various interactions between the components. The network topology of TCOZ is a good mechanism for depicting the architectural configurations.

Computer Aided Dispatch (CAD) System is a generic family system that provides automatic dispatching of the requested tasks within their critical timing requirements. In our current project, "Software Reuse Framework for Reliable Mission-Critical Systems" [1], one goal is to develop the reuse-based design and development methods of reliable Computer Aided Dispatch (CAD) systems. We have found that high level reuse can be best achieved through software architecture models. An effective approach to reuse requires a generic CAD architecture that defines the overall structure and a common base of customizable software assets to be reused across CAD systems. In this chapter, we apply TCOZ to represent an incremental three layered architecture model of the CAD systems [34]. These three layers include the following:

[1]The project, "Software Reuse Framework for Reliable Mission-Critical Systems", is supported by Singapore-Ontario Joint Research Programme.

- Style – an architectural style for CAD systems.
- Generalization – a CAD system generic architecture based on the style model. Critical system properties of the generic layer can be formulated and decomposed into state-based and event/time-based properties. The proofs of those properties are presented.
- Customization – the specific system architecture models are derived from the generic model.

The main benefits of having a three layered approach are reusability, simplicity and reliability. The upper layers represent common paradigms among the family systems, i.e., generic patterns of components and connectors, so that high level relationships in the system can be understood. The lower layers characterize the specific requirements within the new domain, i.e., specific topology of components and connectors, so that new systems can be built as variations and extensions on old systems. This allows us to describe a system architecture as an open-ended collection of reusable architectural elements. Formal specifications of architecture models permit us to reason about important properties at each desired level. Good understanding and precise representation of architecture models lead to reliable system implementations based on the architectures.

The remainder of the chapter is organized as follows. Section 2 presents CAD system architecture style model. In section 3, we develop a CAD system generic architecture model by extending the style model. Section 4 presents the verification of

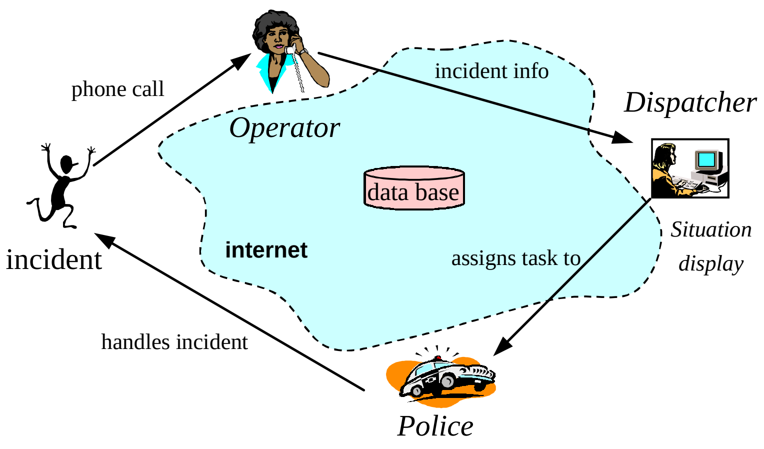


Figure 7.1: An operational scenario in CAD system for police.

critical properties in the CAD system. In section 5 we illustrate the customizations of the generic architecture model to various specific systems architecture models. Section 6 concludes the chapter.

7.2 CAD systems and architecture style model

7.2.1 Overview of CAD system family

Computer Aided Dispatch (CAD) Systems are used by the police, fire & rescue, health service and in many other contexts. Figure 7.1 depicts a basic operational scenario as well as the roles and elements of a CAD system for the Police. An *Operator* receives information about an incident and informs a *Dispatcher* about the incident. The *Dispatcher* examines the "Situation Display" that shows a map of the area where the incident happened. Then, the *Dispatcher* assigns a task of handling the incident to the *Police Unit*, i.e., a *Police Car* that is closest to the place of the incident. The *Police Unit* approaches the place of the incident and handles the problem. The information about current and past incidents is stored in the database. At the basic operational level, CAD systems for Fire & Rescue or Health Services are similar to CAD for Police. These systems support the dispatch of units to incidents. However, there are also differences across CAD systems. The specific context of the operation (such as Police or Fire & Rescue) results in many variations on the basic operational scheme. For example, CAD systems differ in specifics of how resources are assigned to tasks, monitoring, reporting and timing requirements, information to be stored in a database, system component deployment strategies, reliability and availability requirements, and so on. If we ignore commonalities, each CAD system must be developed from scratch and maintained as a separate product - an expensive and inefficient solution. The reuse-based approach may radically cut development and maintenance cost. An effective approach to reuse requires a generic CAD architecture that defines the overall structure and a common base of customizable software assets to be reused across CAD systems. The CAD systems mentioned above form an important product line developed by our industrial partner Singapore Engineering Software Pte Ltd. However, we can further extend the domain analysis [62] and view CAD systems as instances of a general task-resource allocation problem. Then we can observe a similar pattern in the CAD systems mentioned above and the Teleservice and Remote Medical Care System (TRMCS) [50] that supports transition patients from hospital care to home care. In fact, in our examples illustrating CAD architecture specifications, we shall

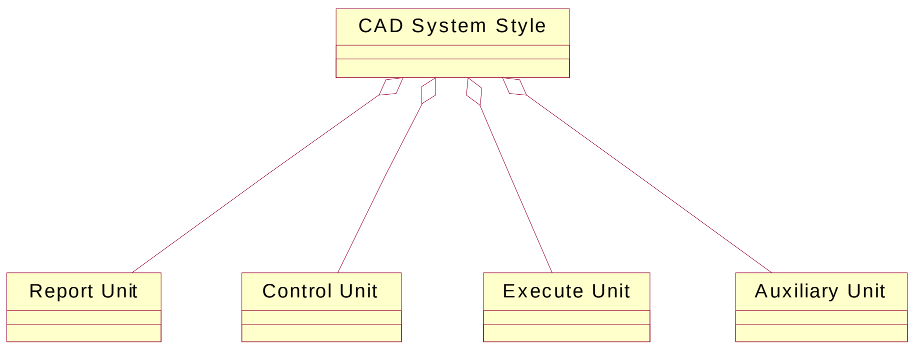


Figure 7.2: CAD system components.

show how CAD for Police and CAD for TRMCS can be derived from a common generic model of CAD architecture.

In architectural descriptions, the three basic elements are components, connectors and configuration of the system [2]. An architectural style defines the properties that are shared by a family of systems. A style concentrates on the commonalities of communication interfaces, interaction mechanisms and architectural configurations of a family of systems but ignores the details of component functionalities and communications.

We have encountered many CAD Systems in our project, "Software Reuse Framework for Reliable Mission-Critical Systems". From a high-level architectural view, the core components and communication of these CAD Systems are depicted in Figure 7.2 and listed as follows:

- Report Unit – A group of reporting units serve as information collectors for the central controller.

- Control Unit – A central control unit manages and dispatches the tasks of the system. This unit makes crucial decisions and assigns tasks to executable resources for engagement against the emergencies. The central controller communicates with all other main units of the system.

- Execute Unit – A group of executing units execute the tasks assigned by the dispatcher. All of them communicate directly with the central dispatcher while working independently from each other.

- Auxiliary Unit – A group of auxiliary units assist the central dispatcher or other main units by taking some less important tasks such as collecting and storing auxiliary information.

- CAD System Style – A system level configuration acts as a collection of related units which perform the desired functionalities. Note that critical timing requirements are important in the units' computation behavior and their interactive communication.

7.2.2 Components and connectors

As pointed out by Garlan and Perry [33], components are the primary elements for computation in a system. Each component has an interface specification that define its properties, which include the signature and functionality of its resources together with global relation, performance properties and so on [85]. TCOZ views components in terms of internal computations and interactions with the rest of the system. The internal computations are context-independent, encapsulated behaviors of the components, while the context-setting interaction patterns are accomplished by the communication interfaces. In our approach, we use both implicit and explicit connectors to depict the communication patterns between the system components. TCOZ provides a fixed set of connector types for component interactions, i.e., **chan**s handle synchronous communication and **sensor** / **actuator**s handle asynchronous communication. At the component level, interactions patterns are captured using implicit connectors such as **chan**s and **sensor** / **actuator**s together with its component definitions. At the configuration level, system components are defined using the TCOZ *Network Topology* construct, and act as explicit connectors to establish the overall system configuration. Note that in some circumstances the use of explicit connectors can bring a number of benefits [5] such as reuse. With TCOZ, it is easy to model a new connector type by creating a new type of component, which is similar to the approach of Rapide [61].

There are four types of software components in our CAD system architecture, i.e., the report unit, the control unit, the execute unit and the auxiliary unit. Their formal definition are presented as follows.

$[\mathit{ReportInfo}]$	[Emergency report type]
$[\mathit{AuxInfo}]$	[Auxiliary information type]
$[\mathit{Task}]$	[Task type]

The four components (units) are modeled as:

$ReportUnit[X]$

$listenport : \textbf{chan}$
$reportport : \textbf{chan}$
$synauxport : \textbf{chan}$
$asynauxport : X\ \textbf{sensor}$

$\textsc{Main} \mathrel{\widehat=} \mu R \bullet [r : ReportInfo;\ a : AuxInfo, t : Task] \bullet$
$\quad listenport?(self, r) \rightarrow (synauxport?(self, a) \rightarrow \textsc{Skip} \;\Box$
$\quad asynauxport?(self, a) \rightarrow \textsc{Skip});\ reportport!t \rightarrow \textsc{Skip};\ R$

Note that in the architecture style level the focus is on the identification of the commonalities of components and their communication interfaces. As from the above, the interaction behavior of the *ReportUnit* is captured by the non-terminating process MAIN in the active object. The *ReportUnit* collects the device information from the synchronous input channel *listenport* (e.g., phones, monitors, alarms, etc. for reporting the incidents) and some additional information from both auxiliary synchronous input channel *synauxport* and asynchronous input **sensor** *asynauxport* (e.g., locations, time, etc. determined from the reports); generates reporting information and pass through the synchronous output channel *reportport* to the Control Unit for the purpose of dispatch.

$ControlUnit$

$reportport : \textbf{chan}$
$dispatchport : \textbf{chan}$

$\textsc{Main} \mathrel{\widehat=} \mu C \bullet ((([t : Task] \bullet reportport?t \rightarrow \textsc{Skip}) \;\Box\; ([t :$
$\quad tasks, e : ExecuteUnit] \bullet dispatchport!(e, t) \rightarrow \textsc{Skip}));\ C$

The *ControlUnit* receives the reporting information from the synchronous input channel *reportport*; generates proper tasks and dispatches them through the synchronous output channel *dispatchport* to the Execute Unit for the purpose of execution.

ExecuteUnit

$dispachport : \mathbf{chan}$

$\textsc{Main} \mathrel{\widehat=} \mu\, E \bullet [t : Task] \bullet dispatchport?(self, t) \rightarrow \textsc{Skip};\ E$

The *ExecuteUnit* receives the dispatched task information from the synchronous input channel *dipatchport* and performs the actual task execution.

AuxiliaryUnit[*X*]

$synauxport : \mathbf{chan}$
$asynauxport : X\ \mathbf{actuator}$

$$\begin{aligned}&\textsc{Main} \mathrel{\widehat=} \mu\, A \bullet [a : AuxInfo, r : ReportUnit] \bullet ((synauxport!(r, a)\\&\quad \rightarrow \textsc{Skip}) \,\square\, (asynauxport!(r, a) \rightarrow \textsc{Skip}));\ A\end{aligned}$$

The *AuxiliaryUnit* provides the addition information to other components through the synchronous output channel *synauxport* and asynchronous output **actuator** *asynauxport*. Note that the communications in the *AuxUnit* may be synchronous or asynchronous, so we give two options in the style.

Each component has its own interfaces for communication with the rest of the system. The details of encapsulated behaviors of the components are deliberately suppressed here in the architectural style since each component of the same type may have different computation behaviors. In the Main operation of each component, we defines the communication patterns.

7.2.3 Configuration and style

A configuration is a collection of interacting component instances and their connectors in a system. The instances of components are distinguished from component types. An architectural style defines the common properties of a family of systems that are shared by any configuration in the style. In the TCOZ approach, configurations are specified by the *Network Topology* construct in the system component and act as an explicit connector.

CADStyle$[X]$

$$
\begin{array}{l}
c : \downarrow \mathit{ControlUnit} \\
rs : \mathbb{F}_1 \downarrow \mathit{ReportUnit}[X] \\
es : \mathbb{F}_1 \downarrow \mathit{ExecuteUnit} \\
as : \mathbb{F} \downarrow \mathit{AuxiliaryUnit}[X]
\end{array}
$$

$$
\textsc{Main} \mathrel{\widehat=} \Vert_{(a,r,e):as \times rs \times es} (a \xleftrightarrow{synauxport,asynauxport} r; \; r \xleftrightarrow{reportport} c \xleftrightarrow{dispatchport} e)
$$

In the example above, all components comprise a CAD System style. The *Network Topology* construct in the MAIN operation clearly identifies the interaction between the components in the system, where the lines connecting components depict the interactive communication relationships and the labels on the lines correspond to the implicit connectors (communication interfaces). For example, the auxiliary units communicate with the report units through the *synauxport* channel and *asynauxport sensor/actuator*; the report units communicate with the control unit through the *reportport* channel; the control unit communicate with the execution unit through the *dispatchport* channel. The objects interaction through the communication interfaces can also be visualized as in the UML diagram Figure 7.3.

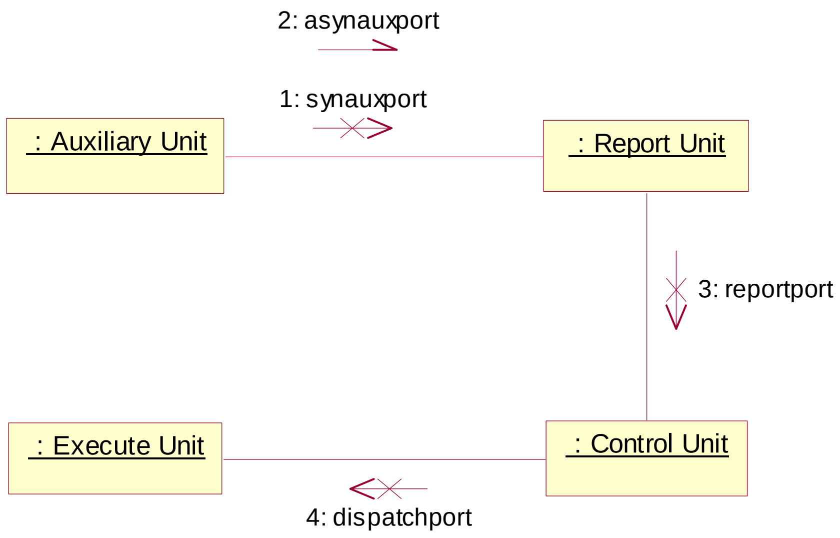


Figure 7.3: CAD system style communication.

7.3 A generic architecture for CAD systems

In this section, we will present a CAD System generic architecture specified in TCOZ. We inherit, extend and instantiate the architectural style presented in the previous section. Unlike the style, a generic model defines crucial computation and communication details of the components in CAD Systems.

Based on the *ReportUnit*, *ControlUnit* and *ExecuteUnit* in the architectural style, we further decompose a generic CAD System into three main types of components (not including auxiliary components):

- The emergency report receivers – obtain emergency information, create de-

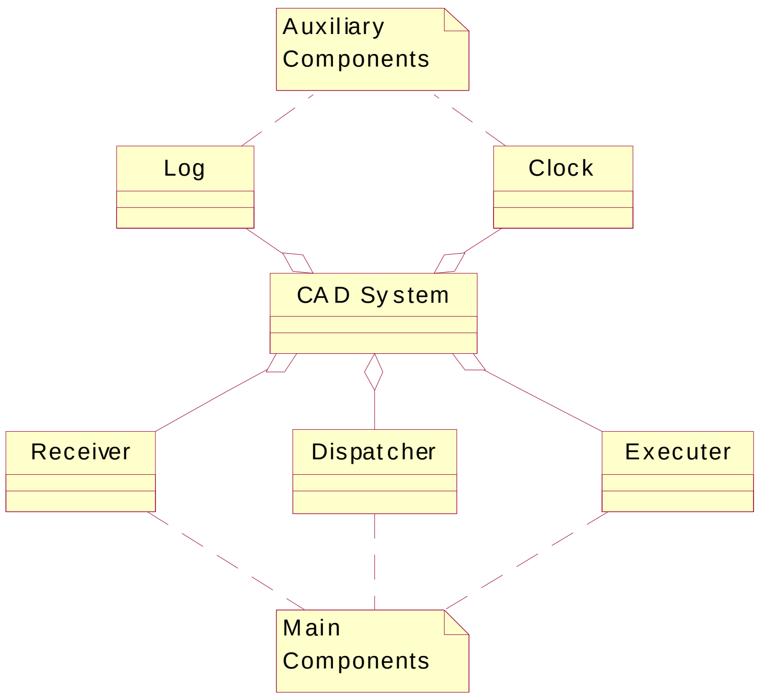


Figure 7.4: The overall structure of CAD system.

tailed tasks and send the tasks to the central dispatcher.

- The central dispatcher – stores the tasks, updates the tasks and dispatches tasks to related task executers according to the business logic.

- The task executers – execute the tasks that dispatched to them. The role of executers may vary in different CAD Systems, such as police offices in police system, hospitals in medical system, etc.

The hierarchical structure is illustrated in Figure 7.4. The *Clock* and *Log* are two auxiliary components extended from the *AuxiliaryUnit* in the style model. They offer time information and logging of important system actions respectively. The subscriber's role also vary in different systems, from patients in the medical system to case locations in the police system. Since most CAD Systems are time-critical, we make the timing requirement an important feature in our generic model. Furthermore, some type variants and common functions were introduced for the purpose of easy customization into specific CAD Systems. The computation behaviors of components are self-encapsulated while implicit connectors are also specified inside relative components. As mentioned previously, a system can be viewed as any one of its components interacting with the rest of the system through the *Network Topolgy*. Therefore, it is natural for us to study the overall system by analyzing the components individually first.

7.3.1 Clock

In order to record the system information at each particular time, a calendar clock is constructed as follows.

Calendar time type is defined as:

$CalT == \mathbb{N}\,\mathsf{yr} \times \mathbb{N}\,\mathsf{mn} \times \mathbb{N}\,\mathsf{dy} \times \mathbb{N}\,\mathsf{hr} \times \mathbb{N}\,\mathsf{min} \times \mathbb{N}\,\mathsf{s}$

The clock stores the total elapsed seconds since some reference date, and the function

$cal : \mathbb{N}\,\mathsf{s} \rightarrowtail CalT$

$\ldots$ [detail of function omitted]

is used to convert the elapsed seconds to a calendar-time.

Clock

$AuxiliaryUnit[CalT][time/asynauxport]$

$total : \mathbb{N}\,\mathsf{s}$

Inc

$\Delta(total, time)$

$total' = total + 1\,\mathsf{s} \land time = cal(total)$

$\textsc{Main} \mathrel{\widehat{=}} \mu\, C \bullet (Inc \bullet \textsc{Deadline}\, 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\, 1\,\mathsf{s};\ C$

The *Clock* component inherits the *AuxiliaryUnit* in the CAD style, where its asynchronous **actuator** *asynauxport* is renamed to *time* and generic type X is substituted by the calendar time type *CalT*. Note that the *time* value increases every second and the display screen updates in less than 50 milliseconds.

7.3.2 System logs

Most CAD Systems require strict persistent repository of data and history log. A generic active object of *Log*$[X]$ is defined as follows, where X is the data structure type of the records in the log.

Log$[X]$

$AuxiliaryUnit[X][record/synauxport]$

$log : \mathrm{seq}\, X$

Add

$\Delta(log)$
$x? : X$

$log' = log \frown \langle x? \rangle$

$\textsc{Main} \mathrel{\widehat{=}} \mu\, L \bullet [x : X] \bullet record?x \rightarrow Add;\ L$

The *Log* component inherits the *AuxiliaryUnit* in the CAD style, where its synchronous channel *synauxport* is renamed to *record*. The system logs consist of two types of logs. One is for the logined reports; and the other is for the dispatched tasks. These can also be customized according to various requirements respectively. The content in the log file is modelled as a variant of type X, which varies according to each particular system.

7.3.3 Emergency receiving part

The system receives emergency reports from its environment. Components in this part inherit *ReportUnit* in the style. When the receiving part of the system receives an emergency report, it generates a *Task* from the reported information *ReportInfo* by the function *GenTask* and sends the task to the central dispatcher.

$$GenTask : ReportInfo \rightarrow Task$$

Receiver

$$\begin{array}{l}
ReportUnit[CalT][listen/listenport, record/synauxport, \\
\qquad time/asynauxport, login/reportport] \\
WriteLog \mathrel{\widehat=} [t : CalT;\ r_i : ReportInfo] \bullet time?t \rightarrow \\
\qquad record!(t, GenTask(r_i)) \rightarrow \textsc{Skip} \\
\textsc{Main} \mathrel{\widehat=} \mu\, R \bullet [r_i : ReportInfo] \bullet listen?r_i \rightarrow \\
\qquad (login!(GenTask(r_i)) \rightarrow WriteLog);\ R
\end{array}$$

The *Receiver* component inherits the *ReportUnit* in the CAD style, where its synchronous channel *listenport* is renamed to *listen*, synchronous channel *synauxport* is renamed to *record*, asynchronous **sensor** *asynauxport* is renamed to *time*, synchronous channel *reportport* is renamed to *login*, and generic type X is substituted by the calendar time type $CalT$. The behavior of the *Receiver* is to collect the emergency information from the synchronous input channel *listen* (e.g., phones, monitors, alarms, etc. for reporting the incidents); generate task information and pass it through the synchronous output channel *login* to the *Dispatcher* for the dispatch purpose, and at the same it records the login information into the system log by the *WriteLog* operation. While recording to log file, it gets the time information from the asynchronous input **sensor** *time* and passes the log information through the synchronous output channel *record* for the repository purpose.

7.3.4 Central dispatcher

All tasks will be stored and assigned through the *Dispatcher*. It is the central and crucial part of the system, actively communicating with other parts. Component in this part inherits *ControlUnit* in the style.

Each task has its own severe level, which means it has its own critical timing requirement. In a generic way, we define a function $Task_T$ to denote the latest time before passing it to an executer.

$$Task_T : Task \rightarrow \mathbb{T}$$

A generic function pt is defined to purge the time out items from the original set into the second set corresponding to the time elapsed and update the time stamps accordingly:

$$[X]$$
$$pt : (\mathbb{T} \times \mathbb{F}(X \times \mathbb{T})) \rightarrow (\mathbb{F}(X \times \mathbb{T}) \times \mathbb{F}\, X)$$
$$\begin{aligned} &\forall\, t : \mathbb{T};\ s : \mathbb{F}(X \times \mathbb{T}) \bullet pt(t, s) = \\ &\quad (\{(e, t_o) : s \mid t_o > t \bullet (e, t_o - t)\}, \{(e, t_o) : s \mid t_o \leqslant t \bullet e\}) \end{aligned}$$

e.g.

$$pt(2\,\mathsf{s}, \{(a, 1\,\mathsf{s}), (b, 3\,\mathsf{s}), (c, 7\,\mathsf{s})\}) = (\{(b, 1\,\mathsf{s}), (c, 5\,\mathsf{s})\}, \{a\})$$

which means that after the elapsing of 2 seconds the time stamp of b and c would become 1 and 5, and the time out item a is purged into the second set.

The most critical system component is the *Dispatcher* class:

Dispatcher

$ControlUnit[login/reportport, dispatch/dispatchport]$

$$\begin{array}{l}
ex : \mathbb{F}_1\, Executer \\
tasks : \mathbb{F}(Task \times \mathbb{T}) \\
\Delta \\
t : \mathbb{T} \\
timeup : \mathbb{F}\, Task \\
\hline
tasks \neq \varnothing \Rightarrow 0 \leqslant t \leqslant min\, \mathrm{ran}\, tasks
\end{array}$$

INIT

$$\begin{array}{l}
tasks = \varnothing
\end{array}$$

Add

$$\begin{array}{l}
\Delta(tasks) \\
task? : Task \\
t_i? : \mathbb{T} \\
\hline
task' = \mathrm{fst}(pt(t_i?, tasks)) \cup (task?, Task_T(task?)) \\
timeup' = snd(pt(t_i?, tasks))
\end{array}$$

Purge

$$\begin{array}{l}
\Delta(tasks) \\
\hline
pt(t, tasks) = (tasks', timeup')
\end{array}$$

$$\begin{array}{l}
AddTask \mathrel{\widehat=} [task : (Task - \mathrm{dom}\, tasks);\ t_i : \mathbb{T}] \bullet \\
\qquad\qquad login?task@t_i \rightarrow Add \\
Dispatch \mathrel{\widehat=} [f : timeup \rightarrow ex] \bullet \\
\qquad\qquad |||_{(task,e):f}\ dispatch!(e, task) \rightarrow \textsc{Skip} \\
\textsc{Main} \mathrel{\widehat=} \mu\, D \bullet ([tasks = \varnothing] \bullet AddTask \,\Box\, [tasks \neq \varnothing] \bullet \\
\qquad\qquad (AddTask \rhd\{t\}\ (Purge;\ Dispatch)));\ D
\end{array}$$

The *Dispatcher* component inherits the *ControlUnit* in the CAD style, where its synchronous channel *reportport* is renamed to *lgoin* and synchronous channel *dispatchport* is renamed to *dispatch*. The behavior of the *Dispatcher* is to receive the task login information from the synchronous input channel *login* and dispatch the tasks according to their critical timing requirements through the synchronous output channel *dispatch* to the execute units for the purpose of execution.

Note the secondary attribute t records the time value which is less than or equal to the minimum time stamp in the task set. This constraint is captured by the class invariant, which must be preserved by all operations. Attribute *timeup* stores all the time-out tasks after each purge operation. The behavior of the MAIN process of the dispatcher is basically either adding or dispatching tasks. If the task set is empty, only adding is performed; while for the non-empty task set, both adding and dispatching are enabled. A *Purge* process is placed when element(s) of *task* is timed out. A *Dispatch* operation is defined (in a flexible way, i.e. any function f) to assign every time-out task to an execution unit in parallel.

Note that the TCSP expression in the form $a@t \to P(t)$ is a process primitive, where a denotes the event initially enabled by the process and t denotes the timing relative to the occurrence of event a. The expression $(a \to P) \rhd \{t\}\ Q$ describes the timed interrupt primitive, where the process will try to perform $a \to P$ and would pass control to Q if the event a has not occurred by time t. According to this semantic, when $tasks \neq \varnothing$, if the operation *AddTask* (when $t_i < t$) is performed, right after the operation, $timeup = \varnothing$ must hold because of the definition of the function pt and class invariant $0 \leqslant t \leqslant min\,\mathrm{ran}\,tasks$ (simplified when $tasks \neq \varnothing$).

This is the reason for designing Main with *Dispatch* operation only after *Purge*, which means that the dispatch will happen exactly at the corresponding timing requirement of each task.

It is reasonable to assume that the time durations t_a, t_d , t_p and t_b of the operations *AddTask* and *Purge* are far less than t or t_i (as $t_a, t_d, t_p, t_b \ll t, t_i$). For instance, t could be in the scale of seconds and t_a might be in microseconds. On the other hand, if the time durations such as t_a are considered, the *AddTask* schema can be modified.

7.3.5 Executers

Tasks are dispatched to the executers for execution via the central dispatcher. A dispatch log file keep the records of all dispatched tasks. Components in this part inherit *ExecuteUnit* in the style.

$$
\begin{array}{l}
\textit{Executer} \\
\hline
\textit{ExecuteUnit}[\textit{dispatch}/\textit{dispatchport}] \\
\boxed{\begin{array}{l} \textit{time} : \textit{CalT}\ \mathbf{sensor} \\ \textit{record} : \mathbf{chan} \end{array}} \\
\textit{WriteLog} \mathrel{\widehat{=}} [t : \textit{CalT};\ \textit{task} : \textit{Task}] \bullet \\
\qquad\qquad \textit{time}?t \rightarrow \textit{record}!(t, \textit{task}, \textit{self}) \rightarrow \textsc{Skip} \\
\textit{Driven} \mathrel{\widehat{=}} [\textit{task} : \textit{Task}] \bullet \textit{dispatch}?(\textit{self}, \textit{task}) \rightarrow \textit{WriteLog} \\
\textsc{Main} \mathrel{\widehat{=}} \mu\, E \bullet \textit{Driven};\ E \\
\hline
\end{array}
$$

The *Executer* component inherits the *ExecuteUnit* in the CAD style, where its synchronous channel *dispatchport* is renamed to *dispatch*. The behavior of the *Executeer* is to receive the dispatched task from the synchronous input channel *dispatch*; execute it and record the dispatched information into the system log by the *WriteLog* operation. While recording to log file, it gets the time information from the asynchronous input **sensor** *time* and passes the log information through the synchronous output channel *record* for the repository purpose.

7.3.6 Generic system architecture configuration

The overall system is a composition of all components that communicate with each other. We organize the interactive relationships through TCOZ network topologies. This system component *CADSystem* plays the role of explicit connector in establishing the configuration of the system.

CADSystem

$CADStyle[CalT][d/c]$

$clock : Clock$
$inlog : Log[CalT \times Task]$
$dispatchlog : Log[CalT \times Task \times Executer]$

$d \in Dispatcher \land d.ex = es$
$\forall r : rs \bullet r \in Receiver$
$\forall e : es \bullet e \in Executer$
$\{clock, inlog, dispatchlog\} \subseteq as$

$$\textsc{Main} \mathrel{\widehat{=}} \big\|_{(r,e):rs\times es}(r \overset{login}{\longleftrightarrow} d \overset{dispatch}{\longleftrightarrow} e;\ inlog \overset{record}{\longleftrightarrow} r \overset{time}{\longleftrightarrow} clock \overset{time}{\longleftrightarrow} e \overset{record}{\longleftrightarrow} dispatchlog)$$

The *CADSystem* component inherits the *CADStyle* connector in the CAD style, where its *ControlUnit* object c is renamed to the *Dispatcher* object d and generic type X is substituted by the calendar time type $CalT$. New instances of auxiliary components such as *clock*, *inlog* and *dispatchlog* are introduced to the system

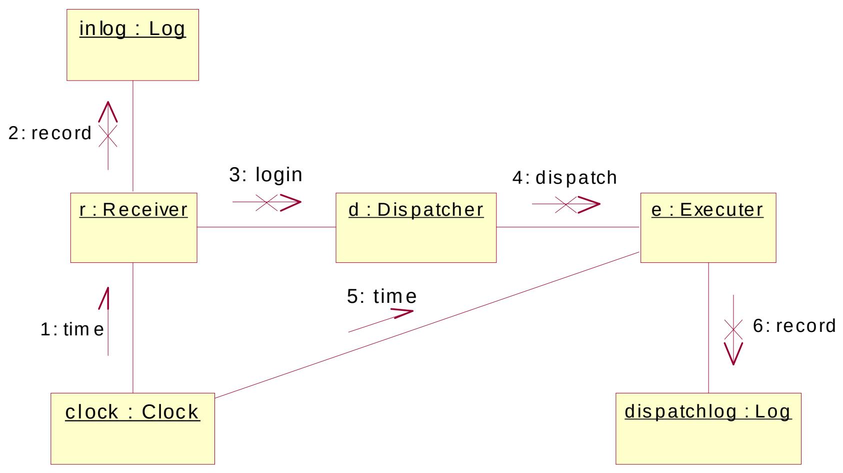


Figure 7.5: The configuration of CAD system.

together with the constraints upon them. From a communication point of view, the *CADSystem* connector specifies that the receiver communicates with the dispatcher through the *login* channel; the dispatcher communicates with the executer through the *dispatch* channel; the receiver communicates with the clock through the *time* sensor/actuator; the receiver communicates with the input log file through the *record* channel; the executer communicates with the clock through the *time* sensor/actuator; the executer communicates with the dispatch log file through the *record* channel.

The UML collaboration diagram in Figure 7.5 also visualizes the configuration of the system defined in the formal model.

7.4 CAD system architecture analysis and verification

From a safety critical perspective, the key point of the CAD system architecture is to provide guaranteed time critical service to all the valid tasks. This critical property can be formally interpreted from the formal model as:

$$\begin{array}{l}\textbf{Theorem:}\ \mathit{CADSystem} :: \forall\, task_o : Task;\ ct_1 : CalT \bullet \\ \quad (ct_1, task_o) \in \operatorname{ran} inlog.log \Rightarrow \exists\, ct_2 : CalT;\ e : es;\ \bullet \\ \qquad (ct_2, task_o, e) \in \operatorname{ran} dispatchlog.log \\ \qquad \wedge\ (cal^{\sim}(ct_2) - cal^{\sim}(ct_1)) = Task_T(task_o)\end{array} \qquad [P]$$

The above simply states that any task which logged into the system will be dispatched at its critical time requirement. In order to prove the validity of the theorem P, the first thing is to show that the *Clock* component in the system correctly models the behavior of a physical timing device – the global clock. This property can be interpreted into the following timed failure specification as below.

$$\begin{array}{lcl}\textbf{Lemma:}\ L_0(s, \aleph) & = & Clock :: \forall\, total : \mathbb{N}\,\mathsf{s};\ t_0, t_1 : \mathbb{T} \bullet \\ & & \quad time!cal(total)\ live\ [t_0, t_1) \Rightarrow (t_1 - t_0 = 1\,\mathsf{s})\end{array}$$

Note that the *live* expression is a specification macro for the TCOZ actuator construct defined as follows:

$$\begin{array}{l}a\ live\ [t_1, t_2) = \forall\, t \in [t_1, t_2) \bullet a\ at\ t \wedge \forall\, t_i : \mathbb{T} \bullet \\ \qquad\qquad (t_i < t_1 \Rightarrow \neg(a\ at\ t_i) \wedge t_i \geq t_2 \Rightarrow \neg(a\ at\ t_i))\end{array}$$

This macro simply expresses that the event a is continuously recorded in the trace as having occurred at every point on a maximal interval I, where I is in the form of $[t_1, t_2)$.

Proof:

Base case: The specification is trivially satisfied by *STOP*.

Assuming the C **sat** $L_0(s, \aleph)$, it is sufficient to show that

$$(\mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s} \fatsemi C \ \mathbf{sat}\ L_0(s, \aleph).$$

Let:

$$\begin{aligned} L_1(s, \aleph) = \mathit{Clock} :: \forall\, \mathit{total} : \mathbb{N}\,\mathsf{s};\ t_0, t_1 : \mathbb{T} \bullet \\ \mathit{time!cal(total)}\ \mathit{live}\ [t_0, t_1) \Rightarrow (t_1 - t_0 \in [0, \infty)) \\ L_2(s, \aleph) = \mathit{Clock} :: \forall\, \mathit{total} : \mathbb{N}\,\mathsf{s};\ t_0, t_1 : \mathbb{T} \bullet \\ \mathit{time!cal(total)}\ \mathit{live}\ [t_0, t_1) \Rightarrow (t_1 - t_0 \in [0, 50\,\mathsf{ms})) \end{aligned}$$

The proof of $[L_0]$ can be constructed as follows:

$$\begin{array}{l|l}
\mathit{Clock} :: \mathit{Inc}\ \mathbf{sat}\ L_1(s, \aleph) & \\
\hline
\mathit{Clock} :: \mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}\ \ \mathbf{sat}\ (\mathit{end}(s) \leqslant 50\,\mathsf{ms} \wedge & [\ \mathit{Deadline}\ ] \\
\quad\quad \checkmark \in \sigma(s) \wedge L_1(s, \aleph \restriction 50\,\mathsf{ms})) \vee (\mathit{end}(s) & \\
\quad\quad > 50\,\mathsf{ms} \wedge L_1(\langle\rangle, (50\,\mathsf{ms}, \infty) \times \Sigma^{\checkmark})) & \\
\hline
\mathit{Clock} :: \mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}\ \ \mathbf{sat}\ L_2(s, \aleph) & [\ \mathit{Weaken}\ ] \\
\hline
\mathit{Clock} :: (\mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s} & [\ \mathit{WaitUntil}\ ] \\
\quad \mathbf{sat}\ ((\mathit{end}(s) > 1\,\mathsf{s} \wedge L_2(s, \aleph)) \vee (\mathit{end}(s) \leqslant 1\,\mathsf{s} & \\
\quad\quad \wedge L_2(s \frown \langle 1\,\mathsf{s}, \checkmark\rangle, \aleph \cup [\mathit{end}(s), 1\,\mathsf{s}) \times \Sigma^{\checkmark}))) & \\
\hline
\mathit{Clock} :: (\mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s} & [\ \mathit{Weaken}\ ] \\
\quad \mathbf{sat}\ L_0(s, \aleph) & \\
\mathit{Clock} :: C\ \mathbf{sat}\ L_0(s, \aleph) & \\
\hline
\mathit{Clock} :: ((\mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s}) \fatsemi C & [\ \mathit{Sequential}\ ] \\
\quad \mathbf{sat}\ \checkmark \notin \sigma(s) \wedge L_0(s, \aleph \cup [0, \infty) \times \{\checkmark\}) & \\
\quad\quad \vee \exists\, s_1, s_2, t \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) & \\
\quad\quad \wedge L_0(s_1 \frown \langle(t, \checkmark)\rangle, \aleph \restriction t \cup [0, t) \times \{\checkmark\}) & \\
\quad\quad \wedge L_0((s_2, \aleph) - t) & \\
\hline
\mathit{Clock} :: ((\mathit{Inc} \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s}) \fatsemi C & [\ \mathit{Weaken}\ ] \\
\quad \mathbf{sat}\ L_0(s, \aleph) &
\end{array}$$

According to the recursion induction rule, the behavior specification $L_0(s, \aleph)$ is satisfied, therefore **Lemma** L_0 has been proved.

After showing that the *Clock* component is consistent with the global clock, we are now ready to prove the correctness of theorem P. First, theorem P can be rewritten into state-based and event/time-based properties as follows:

- No message lost – This property claims that no tasks will be lost once they are in the system. It can be translated into the statement that any task in the login log would be eventually in the dispatched log:

  Theorem 1: $CADSystem :: \forall\, task : Task \bullet$ $[P_1]$
  $task \in \mathrm{ran}\,\mathrm{ran}\, inlog.log \Rightarrow task \in \mathrm{ran}\,\mathrm{ran}\, dispatchlog.log$

- Dispatching at critical time range – This property claims that all tasks in the system will be dispatched to an execution unit at their required critical time range. It can be translated into the statement that the duration from login to the system to its dispatch of each task should be exactly equal to its time requirement $Task_T(task)$:

  Theorem 2: $CADSystem :: \forall\, task : Task;\ t_0 : \mathbb{T};\ e : es \bullet$
  $login?task\ at\ t_0 \Rightarrow dispatch!(e, task)\, at\ (t_0 + Task_T(task))$
  $[P_2]$

As from above, theorem P can be formally translated into a data (state-based) property P_1 and a timing (event-based) property P_2, which later can be proved by the TCOZ inference rules.

7.4.1 Proof of theorem P_1

First, we use structural induction to prove the following property holds by the *Dispatcher* class.

$$\textbf{Lemma :}\ \textit{Dispatcher} :: \forall\, \textit{task} : \textit{Task} \bullet (\textit{task}, \textit{Task}_T(\textit{task})) \in \textit{tasks} \\ \Rightarrow \textit{dispatch}.(e, \textit{task}) \in (\textit{Executer} \times \textit{Task}) \qquad [P_{1.1}]$$

Proof:

Initially: $\textit{Dispatcher} :: \textit{INIT} \vdash \textit{tasks} = \varnothing$, therefore predicate $[P_{1.1}]$ holds (trivial).

Assume the pre-state of the operations in class *Dispatcher* is true, which is $[\forall\, \textit{task} : \textit{Task} \bullet (\textit{task}, \textit{Task}_T(\textit{task})) \in \textit{tasks} \Rightarrow \textit{dispatch}.(e, \textit{task}) \in (\textit{Executer} \times \textit{Task})]$. The post-state of *Dispatcher* is depicted by two kinds of behaviors, *AddTask* and $(\textit{Purge} \fatsemi \textit{Dispatch})$, which are associated with the timeout constraint as follows:

- If no new task is added after the minimum time stamp of all tasks – t, the $(\textit{Purge} \fatsemi \textit{Dispatch})$ operation will perform, which will reduce the number of tasks in the *tasks* set. According to the assumption, $[P_{1.1}]$ holds for the post-state.

- If a new task is added to the *tasks* set before t, by the definition of the *pt* function, the time stamp of this particular task will decrease in a monotonic manner as either the *AddTask* or $(\textit{Purge} \fatsemi \textit{Dispatch})$ operation would perform. Thus the task will eventually be purged from the *tasks* set and dispatched to the *Executor*s. Therefore, $[P_{1.1}]$ holds for the post-state.

According to the structural induction, **Lemma** $P_{1.1}$ is proved.

The proof of $[P_1]$ can be constructed via state reasoning rules as follows:

$$\frac{\begin{array}{l} \mathit{CADSystem} :: \mathit{STATE} \vdash d \in \mathit{Dispatcher} \wedge rs \in \mathbb{F}_1\, \mathit{Receiver} \\ \quad \wedge\ \mathit{inlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task}] \\ \mathit{Receiver} :: \mathit{STATE} \vdash \mathit{listen}, \mathit{login}, \mathit{record} \in \mathbf{chan}\ \wedge \\ \quad \mathit{MAIN} \vdash \mathit{listen.task} \in \mathit{Task} \Rightarrow \mathit{login.task} \in \mathit{Task} \\ \qquad \wedge\ \mathit{record}.(t, \mathit{task}) \in (\mathit{CalT} \times \mathit{Task}) \\ \mathit{Dispatcher} :: \mathit{STATE} \vdash \mathit{login} \in \mathbf{chan} \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan} \\ \mathit{CADSystem} :: \mathit{MAIN} \vdash r \in rs \wedge d \xleftrightarrow{\mathit{login}} r \xleftrightarrow{\mathit{record}} \mathit{inlog} \end{array}}{\begin{array}{l} \mathit{Dispatcher} :: \mathit{MAIN} \vdash \mathit{login.task} \in \mathit{Task} \Rightarrow \\ \quad (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks}\ \ \wedge \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{MAIN} \vdash \mathit{record}.(t, \mathit{task}) \in \\ \quad (\mathit{CalT} \times \mathit{Task}) \Rightarrow (t, \mathit{task}) \in \operatorname{ran} \mathit{log} \qquad\qquad [P_{1.2}] \end{array}}\ [\ \mathit{Channel}\ ]$$

$$\frac{\begin{array}{l} \mathit{CADSystem} :: \mathit{STATE} \vdash d \in \mathit{Dispatcher} \wedge es \in \mathbb{F}_1\, \mathit{Executer} \\ \quad \wedge\ \mathit{dipatchlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executor}] \\ \mathit{Dispatcher} :: \mathit{MAIN} \vdash \mathit{login.task} \in \mathit{Task} \Rightarrow \\ \quad (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks} \\ \mathit{Dispatcher} :: \mathit{STATE} \vdash \mathit{dispatch} \in \mathbf{chan} \\ \mathit{Dispatcher} ::\ \ \vdash (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks} \Rightarrow \\ \quad \mathit{dispatch}.(e, \mathit{task}) \in (\mathit{Executer} \times \mathit{Task}) \qquad\qquad [P_{1.1}] \\ \mathit{Executer} :: \mathit{STATE} \vdash \mathit{dispatch} \in \mathbf{chan} \\ \mathit{CADSysyem} :: \mathit{MAIN} \vdash e \in es \wedge d \xleftrightarrow{\mathit{dispatch}} e \end{array}}{\mathit{Executer} :: \mathit{MAIN} \vdash \mathit{dispatch}.(e, \mathit{task}) \in (\mathit{Executer} \times \mathit{Task})}\ [\ \mathit{Channel}\ ]$$

$$\frac{\begin{array}{l} \mathit{Executer} :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan}\ \wedge \\ \quad \mathit{MAIN} \vdash \mathit{dispatch}.(e, \mathit{task}) \in (\mathit{Executer} \times \mathit{Task}) \Rightarrow \\ \qquad \mathit{record}.(t, \mathit{task}, \mathit{self}) \in (\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}) \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan} \\ \mathit{CADSysyem} :: \mathit{MAIN} \vdash e \in es \wedge e \xleftrightarrow{\mathit{record}} \mathit{dispatchlog} \end{array}}{\begin{array}{l} \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{MAIN} \vdash \\ \quad \mathit{record}.(t, \mathit{task}, e) \in (\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}) \\ \qquad \Rightarrow (t, \mathit{task}, e) \in \operatorname{ran} \mathit{log} \qquad\qquad [P_{1.3}] \end{array}}\ [\ \mathit{Sensor}\ ]$$

Thus P_1 can be clearly derived from $P_{1.2}$ and $P_{1.3}$ above as follows:

$$\frac{\begin{array}{l} \mathit{CADSystem} :: \mathit{STATE} \vdash \mathit{inlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task}]\ \wedge \\ \quad \mathit{dipatchlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executor}] \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{MAIN} \vdash \mathit{record}.(t, \mathit{task}) \in (\mathit{CalT} \times \mathit{Task}) \\ \quad \Rightarrow (t, \mathit{task}) \in \operatorname{ran} \mathit{log} \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{MAIN} \vdash \mathit{record}.(t, \mathit{task}, e) \in \\ \quad (\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}) \Rightarrow (t, \mathit{task}, e) \in \operatorname{ran} \mathit{log} \end{array}}{\begin{array}{l} \mathit{CADSystem} ::\ \ \vdash \forall\, \mathit{task} : \mathit{Task} \bullet \mathit{task} \in \operatorname{ran} \operatorname{ran} \mathit{inlog.log} \\ \quad \Rightarrow \mathit{task} \in \operatorname{ran} \operatorname{ran} \mathit{dispatchlog.log} \end{array}}$$

7.4.2 Proof of theorem P_2

P_2 can be interpreted as the following timed specification in terms of the timed failure model.

$$\begin{array}{l}
P_2(s,\aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : \mathit{es} \bullet \\
\quad \mathit{login}?\mathit{task}\ \mathit{at}\ t_0 \Rightarrow \mathit{dispatch}!(e, \mathit{task})\ \mathit{at}\ (t_0 + \mathit{Task}_T(\mathit{task}))
\end{array}$$

Proof:

Base case: The specification is trivially satisfied by $STOP$.

Assuming the D **sat** $P_2(s,\aleph)$, it is sufficient to show that $([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \ \Box\ [\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask} \rhd\{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) \fatsemi D$ **sat** $P_2(s,\aleph)$.

Let $P_{2.1}$,$P_{2.2}$ be two time failure expressions represented as follows:

$$\begin{array}{l}
P_{2.1}(s,\aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : \mathit{es} \bullet \\
\quad \mathit{login}?\mathit{task}\ \mathit{at}\ t_0 \Rightarrow t_0 < t \wedge \mathit{timeup} = \varnothing \wedge \\
\qquad \neg\ (\mathit{dispatch}!(e, \mathit{task})\ \mathit{at}\ t_0) \\
P_{2.2}(s,\aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : \mathit{es} \bullet \\
\quad (\mathit{dispatch}!(e, \mathit{task})\ \mathit{at}\ t_0 \Rightarrow t_0 = t \wedge \mathit{timeup} \neq \varnothing \wedge \\
\qquad \exists\, \mathit{ts} \subseteq \mathit{tasks} \bullet \forall (\mathit{task}_1, t_1), (\mathit{task}_2, t_2) \in \mathit{ts} \bullet t_1 = t_2 = t \\
\qquad\quad \wedge\ \mathit{Task}_T(\mathit{task}_1) = \mathit{Task}_T(\mathit{task}_2))
\end{array}$$

In our model, the behavior of adding and assigning valid tasks is determined by function *pt*, *Add* and *Purge* operations in the nonterminating process MAIN of the class *Dispatcher*. Considering each non-recursive transaction trace of the MAIN process as one cycle, the possible actions of the *Dispatcher* within the cycle are as follows:

$$\begin{array}{l}
A_1 : \mathit{AddTask}\ \mathit{when}\ \mathit{tasks} = \varnothing \\
A_2 : \mathit{AddTask}\ \mathit{when}\ \mathit{tasks} \neq \varnothing \wedge t_i < t \\
A_3 : \mathit{Purge} \fatsemi \mathit{Dispatch}\ \mathit{when}\ \mathit{tasks} \neq \varnothing \wedge t_i = t
\end{array}$$

Therefore it is trivial to show that $P_{2.1}$ and $P_{2.2}$ are satisfied by *AddTask* and $(\mathit{Purge} \fatsemi \mathit{Dispatch})$ respectively. The proof of $[P_2]$ can be constructed via event reasoning rules as follows:

$$
\begin{array}{ll}
\mathit{Dispatcher} :: ([\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask}) \ \mathbf{sat}\ P_{2.1}(s, \aleph) & \\
\mathit{Dispatcher} :: ([\mathit{tasks} \neq \varnothing] \bullet (\mathit{Purge} \fatsemi \mathit{Dispatch})) \ \mathbf{sat}\ P_{2.2}(s, \aleph) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) & [\ \mathit{Timeout}\ ] \\
\quad \mathbf{sat}\ (\mathit{begin}(s) \leqslant t \wedge P_{2.1}(s, \aleph)) & \\
\qquad \vee\ (\mathit{begin}(s) \geqslant t \wedge P_{2.1}(\langle\ \rangle, \aleph \upharpoonright t) \wedge P_{2.2}((s, \aleph) - t)) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) & [\ \mathit{Weaken}\ ] \\
\quad \mathbf{sat}\ P_2(s, \aleph) & \\
\mathit{Dispatcher} :: ([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask}) \ \mathbf{sat}\ P_2(s, \aleph) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \Box [\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask} & [\ \mathit{External}\ ] \\
\quad \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) \ \mathbf{sat}\ P_2(s, \aleph) \wedge P_2(\langle\ \rangle, \aleph \upharpoonright \mathit{begin}(s)) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \Box [\mathit{tasks} \neq \varnothing] \bullet & [\ \mathit{Weaken}\ ] \\
\quad \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) \ \mathbf{sat}\ P_2(s, \aleph) & \\
\mathit{Dispatcher} :: D \ \mathbf{sat}\ P_2(s, \aleph) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \Box [\mathit{tasks} \neq \varnothing] \bullet & [\ \mathit{Sequential}\ ] \\
\quad \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) \fatsemi D \ \mathbf{sat}\ \checkmark \notin \sigma(s) & \\
\qquad \wedge\ P_2(s, \aleph \cup [0, \infty) \times \{\checkmark\}) & \\
\quad \vee\ \exists\, s_1, s_2, t_i \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) \wedge P_2(s_1 \frown \langle (t_i, \checkmark) \rangle, & \\
\qquad \aleph \upharpoonright t_i \cup [0, t_i) \times \{\checkmark\}) \wedge P_2((s_2, \aleph) - t_i) & \\
\hline
\mathit{Dispatcher} :: ([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \Box [\mathit{tasks} \neq \varnothing] \bullet & [\ \mathit{Weaken}\ ] \\
\quad \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge} \fatsemi \mathit{Dispatch})) \fatsemi D \ \mathbf{sat}\ P_2(s, \aleph) &
\end{array}
$$

According to the recursion induction rule, the behavior specification $P_2(s, \aleph)$ is satisfied, therefore **Theorem** P_2 has been proved. Thus from the proofs of P_1 and P_2 we can see that the critical timing requirement of the generic CAD system architecture **Theorem** P is formally verified.

7.5 Architecture customization

A generic system architecture must be easily customizable to meet the requirements of specific systems. The customization includes customizing computation behaviors of components and customizing architectural configuration in terms of connectors. There are two common approaches in achieving the customization. One is to model the generic architecture in as compact a manner as possible, which includes only the intersection parts among all system family members. In this way, specific system architectures can be derived from the generic model through inheriting and expanding the components. The other approach is to cover most common functionalities of the system family in the generic model, and then model specific system architectures through cutting down and modifying relevant components. The first approach is suitable for system families in which most systems share not only the main structure but also many component behavior and communication details. The second one, in a sense, is better for the system family in which among systems there are only minor differences in architectural configuration while the component inner behaviors are not very interactive. Real world systems are usually complex and cannot be simply classified into any one of the above two approaches. Therefore, the customization approach might be a blend of the two approaches above. Most CAD Systems share common architecture features on a large scale. However the types and functionalities differ from system to system and need to be specifically redefined in particular systems. We demonstrate the customization of the generic architecture into specific systems through a police system and a

Teleservices and Remote Medical Care System [50] case studies.

7.5.1 CAD system for police

Generic types and functions are defined abstractly in the generic model. During the customization, we need to specify the types and functions to meet the requirements of the particular system since these requirements are meant to be different within each system. So the first step of customization is to redefine the types and functions.

- *ReportInfo* – In the police system, the *ReportInfo* describes the disaster status and other helpful information.

  $$ReportInfo == Situation \times Location$$

- *GenTask* – The function generates tasks according to the incident report information. This is performed by the professional staff with the receiver operators.

- *Convert* – The function converts an automatically detected case into an incident report. This is performed by the auto-alarm devices.

- $Task_T$ – This function generates timing requirements according to emergency severity levels. In the police system, all accidents reported must be handled immediately, so that we define the $Task_T$ to set the latest time before passing each task to be 0, which means that each task will be forced into *Executer*s – policemen, right after its storage in the task queue.

$$\begin{array}{|l} \mathit{Task}_T : \mathit{Task} \rightarrow \mathbb{T} \\ \hline \forall\, t : \mathit{Task} \bullet \mathit{Task}_T(t) = 0 \end{array}$$

Secondly, as most police systems provide auto-alarm services such as bank alarm bells, shop alarms and high security building alarm signals for their customers for emergency case reporting, the receiving units of the police system should include auto-alarm equipment. These devices continuously read their environment and will raise the alarm immediately if any violations are detected. The alarm device is modeled as follows:

$$\begin{array}{|l} \textit{Alarm} \\ \hline \mathit{Receiver}[\mathit{sensor}/\mathit{listen}] \\ \textsc{Main} \mathrel{\widehat=} \mu\, A \bullet [c : \mathit{CASE}] \bullet \mathit{sensor}?c \rightarrow \\ \qquad\qquad (\mathit{login}!(\mathit{GenTask}(\mathit{convert}(c))) \rightarrow \mathit{WriteLog}) \mathbin{;} A \\ \hline \end{array}$$

The *Alarm* inherits the *Executer* component from the generic CAD architecture. The system reporting device is a collection of the *Alarm*s and *Receiver*s. Since the police system is very similar to the generalized CAD system, our customization here is mainly focused on substituting type variants and redefining functions. We will demonstrate a more complicated customization procedure of a Teleservices and Remote Medical Care System in the CAD System family.

7.5.2 Teleservices and remote medical care system

The Teleservice and Remote Medical Care System (TRMCS) provides services for the transition of patients from hospital care to home care. In the TRMC system, the *ReportInfo* includes the patient's symptoms and the place of the incident.

$$\mathit{ReportInfo} == \mathit{Symptom} \times \mathit{Location}$$

Secondly, the TRMCS system consists of a number of help centers for performing the emergency job execution. For the sake of urgency, a task might be put up for open bid, and the help centers compete to answer it. At the same time, the system must guarantee that at least one help center responds. Therefore, we offer two mechanisms for help centers to be assigned tasks. First, the executers are aware of what tasks are available at the current time and they can actively select tasks from the dispatcher. Second, tasks are passively dispatched to the executers for execution in the case that some tasks are not selected by any help center within a certain deadline. Thus the *HelpCenter* and $\mathit{Dispatcher}_{\mathit{TRMCS}}$ components that inherit the *Executer* and *Dispatcher* components from the generic CAD architecture are modelled as follows:

HelpCenter

Executer

$d : \mathit{Dispatcher}_{\mathit{TRMCS}}$
$\mathit{select}, \mathit{choose} : \mathbf{chan}$

$$\mathit{Select} \mathrel{\widehat=} [\mathit{task} : \mathrm{dom}\, d.\mathit{tasks}] \bullet \mathit{select}?\mathit{task} \rightarrow \mathit{choose}!\mathit{task}$$
$$\rightarrow \mathit{dispatch}?(\mathit{self}, \mathit{task}) \rightarrow \mathit{WriteLog}$$
$$\textsc{Main} \mathrel{\widehat=} \mu\, H \bullet (\mathit{Select} \mathbin{\Box} \mathit{Driven}) \mathbin{;} H$$

$\mathit{Dispatcher}_{\mathit{TRMCS}}$

Dispatcher

$\mathit{choose} : \mathbf{chan}$

Delete

$\Delta(tasks)$
$task? : Task$
$t_i? : \mathbb{T}$

$tasks \neq \varnothing$
$pt(t_i?, task? \lhd tasks) = (tasks', timeup')$

$$\begin{array}{l}
Assign \mathrel{\widehat=} [task : tasks;\ e : ex;\ t_i : \mathbb{T}] \bullet choose?(e, task)@t_i \\
\qquad \rightarrow dispatch!(e, task) \rightarrow Delete \\
\textsc{Main} \mathrel{\widehat=} \mu\, D \bullet ([tasks = \varnothing] \bullet AddTask \;\Box\; [tasks \neq \varnothing] \bullet \\
\qquad ((AddTask \;\Box\; Assign) \rhd\{t\}\ (Purge \mathbin{;} Dispatch))) \mathbin{;} D
\end{array}$$

By customizing a task selection property into the system, the TRMCS configuration is modelled with additional components as follows:

TRMCSystem

$CADSystem$

$d \in Dispatcher_{TRMCS}$
$\forall\, h : es \bullet h \in HelpCenter \wedge h.d = d$

$$\begin{array}{l}
\textsc{Main} \mathrel{\widehat=} \Big\|_{(r,h):rs \times es} (r \xleftrightarrow{login} d \xleftrightarrow{choose, dispatch} h; \\
\qquad inlog \xleftrightarrow{record} r \xleftrightarrow{time} clock \xleftrightarrow{time} h \xleftrightarrow{record} dispatchlog)
\end{array}$$

Note that the $Dispatcher_{TRMCS}$ and *HelpCenter* components are also communicating through the synchronous channel *choose*. From the above system architecture, by means of active selection and passive assignment the tasks are dispatched within their critical timing requirement. Thus the **Theorem** P is modified as follows:

Theorem: 3 $TRMCSystem :: \forall\, task_o : Task;\ ct_1 : CalT \bullet$

$$\begin{array}{l}
\quad (ct_1, task_o) \in \operatorname{ran} inlog.log \Rightarrow \exists\, ct_2 : CalT;\ e : es; \bullet \\
\qquad (ct_2, task_o, e) \in \operatorname{ran} dispatchlog.log \wedge \\
\qquad\quad (cal^{\sim}(ct_2) - cal^{\sim}(ct_1)) \leqslant Task_T(task_o) \qquad [P']
\end{array}$$

The above states that the dispatching of a task should be performed within its timing requirement $Task_T(task)$ due to the active selections, while in the generalized CAD system this should perform exactly at $Task_T(task)$. Note that theorem P' can also be proved similarly as illustrated in Section 7.4. Hence, the TRMCS is customized from the general architecture of CAD system to its own special requirements.

7.6 Conclusion

In this chapter, we demonstrated the verification of an incremental three layer architecture model for the CAD system family, i.e., the style, the generalization and the customization, by applying TCOZ proof rules. The CAD style captures the most common patterns among the CAD systems. The generalization layer models the essential functionalities of the CAD systems. The customization characterizes the additional specific requirements within each particular system. Thus new systems are built as variations and customizations of the upper-level designs, and the whole family architecture is depicted as an open-ended design for reuse. In this chapter, we also found that TCOZ could be a potential candidate for an Architecture Description Language for the formal specifications of software architecture models. The class constructs in TCOZ are well suited for component declaration. The communication interfaces, i.e., channel, sensor and actuator, act as implicit connecters for modeling the communications between components. The network topology is used as explicit connectors for defining the overall configuration of the system. All these features may provide a more consistent and flexible way of specifying software architectures. Furthermore, in this chapter we have demonstrated the verification of architecture properties via formal reasoning. We applied both state and event based inference rules defined in the previous chapter for the verification of TCOZ architecture specifications. Complex system properties are decomposed into state and event related properties and proved respectively. In summary, this chapter demonstrates that integrated formal modeling techniques (i.e. TCOZ) can be a good candidate for modeling and formal reasoning about complex software systems – in this case the software architecture descriptions.

Chapter 8

Conclusions and directions for further research

This chapter summarizes the main contributions of the thesis and discusses possible directions for further research.

8.1 Thesis main contributions and influence

The content of the thesis addresses a spectrum of tools and verification techniques for the integrated formal notation – TCOZ. The spectrum ranges from the light-weight through middle-weight to heavy-weight tool support.

- This thesis successfully applied the XML technology to define a customized markup language for the Z family notations (Z/Object-Z/TCOZ). The ZML serves as a standard interchange format between various tool environments. The schema also acts as a syntax checker for validating the content of the specifications written in XML.
- This thesis developed a web environment for the Z family languages based on XML/XSL transformation. The ZML web environment provides a feasible means of constructing, displaying and resource sharing formal specification models on the web. It includes the auto type referencing, static syntax checking and browsing facilities such as the Z schema calculus and Object-Z/TCOZ inheritance expansions. This will also make an impact on formal methods education through the internet.
- This thesis demonstrated the investigation of the semantic links between Object-Z/TCOZ specifications and UML diagrams via XSL transformation. UML is commonly regard as one of the dominant graphical notations for industrial software system modeling. In our approach, UML diagrams are visual projections from a formal Object-Z/TCOZ model, therefore they are more

consistent and precise. Thus our projection environment provides a means of visualization for the formal specification models through XML/XSLT.

- This thesis developed a specification animation environment for the TCOZ specifications in a multi-paradigm language - Oz. Oz provides various programming constructs such as object orientation, constraint and logic programming, functional programming, concurrent programming and so on. By presenting an executable semantic of TCOZ in Oz, with a well defined TCOZ construct library, animation of TCOZ models can be easily and effectively achieved. Furthermore, an XSLT stylesheet for the automatic transformation from TCOZ specification into Oz code frames is constructed. This provides an effective way of validating the consistency between the TCOZ formal model and its real world requirements.

- This thesis presented an approach of combining and extending the state-based (Object-Z) and event based (TCSP) proof systems for formally verifying TCOZ specifications. Complex system properties can be decomposed into state and event related properties and proved respectively. In general, it provides a rigorous means of reasoning for the integrated formal notations such as TCOZ. In addition, a framework for the shallow embedding of TCOZ inference rules into the theorem prover Isabelle/HOL was illustrated to support automatic proof assistance.

- This thesis also demonstrated the formal design approaches to the modelling of various applications as well as system architectures, such as ZML web environment, UML projections and CAD system family architectures. These formal models act as precise design documentation and provide clear guidelines to the implementations.

In summary, with the above tool support and verification techniques, TCOZ is a viable potential candidate for industrial software modelling.

8.2 Directions for further research

The following topics, arising out of this thesis, seem worthy of further research.

8.2.1 Z Markup Language standardization

In chapter 3, a customized Markup Language for the Z family notations is presented. Recently it is common for tools to interact using XML. The ZML is to serve as a standard interchange format among the TCOZ tool environments. This idea can be easily adopted by other formal specification notations. Thus defining standard markups for each formal language is essential. By doing so, different tools for the same language can share a common interchangeable input/output. We are currently involved in the definition of a standard markup language [99] for the ISO Z standard [1], contributed to the Community Z Tools (CZT) initiative [69]. Hopefully it will become part of the ISO Z standard in the future. By providing XSL style sheets for each formal notation, we can create a new culture for constructing formal specifications on the web in XML rather than in LaTeX. Furthermore, with the help of XSLT, transformations between different formal notations can be made possible. Thus projections and translations of specification models among various formal languages can be easily achieved. We hope the above can be a starting point for developing a standard XML environment for all formal notations – Formal specification Markup Language (FML). It will certainly make an impact on formal methods education through the internet.

8.2.2 Semantic web

The XML web environment presented in chapter 4 is mainly based on structured syntax. Recently the W3C proposed a new mechanism for presenting information on the web – Semantic Web (SW) [105]. It is commonly regarded as the next generation of the web, and is an emerging technology between the Knowledge Representation and the XML Communities. SW proposed the idea of having data on the web defined and linked in such a way that it can be used for automation, extension and integration. The success of the Semantic Web may have profound impact on the web environment for formal specifications. The DARPA Agent Markup Language (DAML) [37] is a semantic markup language based on RDF/RDF-Schema [104] and XML for web resources. The diversity of various formal specification techniques and the need for their effective combinations requires an extensible and integrated supporting environment. Various formal notations can be used in an effective combination if the semantic links between those notations can be clearly established. By using the RDF/DAML, a semantic web environment can be constructed for supporting, extending and integrating different formalisms based on their language semantics. Some recent initial works have been presented in the papers [21, 22]. Such a meta integrator may bring together the strengths of various formal methods communities in a flexible and widely accessible fashion. A SW environment for the Standard Z and transformations to/from ZML could be one of our future works.

8.2.3 UML transformation

In chapter 4, we have defined an XSLT stylesheet for automatically transforming the Object-Z/TCOZ models in XML into UML class diagrams [94]. The XSLT encodes the projection rules from the formal notations into their corresponding UML counterparts. Recently this work has been extended to support the auto-generation of UML statechart diagrams from Object-Z/TCOZ specifications using a Java XML parser [20]. Both implementations take the ZML format as a standard input and perform XML transformation into XMI (XML Metadata Interchange) format for visualization in the Rational Rose tool suite. Further investigations can be made between the projection of TCOZ models into other UML diagrams such as sequence, collaboration, activity diagrams and so on. In addition, by using the Rational script, it is possible to integrate these projections into the Rose interface as part of the plug-in menu and run together with the Rational UML tool suite. In addition, the idea of projecting TCOZ models into UML diagrams can be easily adapted into other modeling languages, such as Timed Automata [23, 76], Spin model checker and so on, for the verification of corresponding system properties.

8.2.4 Animation and testing

In chapter 5 we presented an approach of animating TCOZ specifications in the Oz language. Based on the ZML representation of TCOZ models, corresponding Oz code frames can be generated via XSLT. However, our translating and validating processes still need human interaction at the moment. A more sophisticated translation tool can be built based on the TCOZ XML format to Oz syntax. Furthermore, specification animation plays a role in validating the consistency between the user's informal requirements and the formal specification. Validation denotes the process of determining that the requirements are the right requirements and that they are complete. Thus software testing strategies can be introduced into the specification animation process. Recently there has been much research focused on combining software testing and formal specification [45, 70]. The process of generating test cases from a formal specification is a form of analysis that helps to validate the specification, because the tests are concrete instantiations of the specification. We believe that the combination of animation and formal testing approaches can provide a more rigorous process for specification validation. In addition, with the help of the ZML structured format, automation can be more easily achieved.

8.2.5 Automated formal verification

In chapter 6, we presented a sketched framework for the embedding of the TCOZ language into the generic theorem prover Isabelle/HOL. By doing so, we could support automated proof assistance for the verification of TCOZ specifications. One immediate work is to complete the theory files and machine verify the TCSP's timed failure semantics. Based on the correct semantic model, we can further verify inference rules and deduce new theorems from the system. Finally, one other goal is to combine and extend the embedding of Object-Z and TCSP to accommodate the automated reasoning for TCOZ language. Furthermore, a parsing program can be built to translate ZML format of TCOZ specifications into its corresponding Isabelle/HOL representations for automated formal verifications.

Finally, our ultimate goal is to provide an integrated tool support for the TCOZ formal specification language, which includes all the topics presented in this thesis such as model constructing, syntax and type checking, web publishing, UML visualization, animating and formal verification functions in a coherent environment to fulfill its potential industry usage.

Bibliography

[1] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.

[2] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[3] J. Abrial. The B tool (abstract). In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM – The Way Ahead*, volume 328 of *Lect. Notes in Comput. Sci.*, pages 86–85. Springer-Verlag, 1988.

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

[5] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 1997.

[6] Inc. Altova. XML SPY. http://www.xmlspy.com/.

[7] K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.

[8] J. P. Bowen and D. Chippington. Z on the Web using Java. In Bowen et al. [9], pages 66–80.

[9] J. P. Bowen, A. Fett, and M. G. Hinchey, editors. *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1998.

[10] P. Brooke. *A Timed Semantics for a Hierarchical Design Notation.* PhD thesis, University of York, 1999.

[11] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. *Lecture Notes in Computer Science*, 86:293–329, 1980.

[12] M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods, Turku, Finland*, Lect. Notes in Comput. Sci. Springer-Verlag, May 2002.

[13] ORA Canada. Z/EVES. http://www.ora.on.ca/z-eves/, 2002.

[14] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. North-Holland, 1990.

[15] P. Ciancarini, C. Mascolo, and F. Vitali. Visualizing Z notation in HTML documents. In Bowen et al. [9], pages 81–95.

[16] The Unicode Consortium. Unicode Home Page. http://www.unicode.org/.

[17] J. Crow and B. D. Vito. Formalizing space shuttle software requirements: four case studies. *ACM Trans. Software Engineering and Methodology*, 7(3):296–332, July 1998.

[18] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[19] J. S. Dong and B. Mahony. Active Objects in TCOZ. In J. Staples, M. Hinchey, and S. Liu, editors, *the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 16–25. IEEE Computer Society Press, December 1998.

[20] J. S. Dong, Y. F. Li, J. Sun, J. Sun, and H. Wang. XML-based static type checking and dynamic visualization for TCOZ. In *4th International Conference on Formal Engineering Methods*, pages 311–322. Springer-Verlag, October 2002.

[21] J. S. Dong, H. Wang, and J. Sun. Semantic Web for Extending and Linking Formalisms. In *Formal Methods Europe (FME'02 - FLoC)*, pages 587–606. Springer-Verlag, July 2002.

[22] J. S. Dong, H. Wang, and J. Sun. Z Approach to Semantic Web. In *The 4th International Conference on Formal Engineering Methods*, pages 156–167. Springer-Verlag, October 2002.

[23] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Y. Wang. TCOZ to Timed Automata. Technical report, School of Computing, National University of Singapore, 2003. http://nt-appn.comp.nus.edu.sg/fm/tcoz2ta/tr.pdf.

[24] E. Dubois, E. Yu, and M. Petit. From Early to Late Formal Requirements: a Process-Control Case Study. In *The 9th IEEE International Workshop on Software Specification and Design (IWSSD'98)*, pages 34–42. IEEE Computer Society Press, 1998.

[25] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[26] 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.

[27] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Theorem Proving in Higher Order Logics*, pages 121–136, 1997.

[28] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. *EATCS Monographs on Theoretical Computer Science*, 6, 1985.

[29] A. S. Evans and A. N. Clark. Foundations of the unified modeling language. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, 1998.

[30] M. Feather. Language Support for the Specification and Development of Composite Systems. *ACM Trans. Prog. Lang. and Syst.*, 9(2):198–234, 1987.

[31] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [7].

[32] K. Futatsugi and A. Nakagawa. An Overview of CAFE Specification Environment. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

[33] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.

[34] D. Garlan and N. Delisle. Formal specification of an architecture for a family of instrumentation systems. In M. Hinchey and J. Bowen, editors, *Applications of formal methods*, pages 55–72. Prentice-Hall, 1995.

[35] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, pages 391–420. Addison-Wesley, 1985.

[36] W. Grieskamp, T. Santen, and B. Stoddart, editors. *IFM'00: Integrated Formal Methods, Dagstuhl Castle, Germany*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.

[37] DAML Research Group. The DARPA Agent Markup Language Homepage. http://www.daml.org/.

[38] Object Management Group. XML Metadata Interchange (XMI) Specification, May 2003.

[39] Mozart Research Groups. The Mozart Programming System, 2002. http://www.mozart-oz.org/.

[40] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[41] I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3), 1995.

[42] M. Henz. *Objects in Oz.* PhD thesis, Universitat des Saarlandes, Fachbereich Informatik, Saarbrucken, Germany, June 1997.

[43] J. Hesketh, D. Robertson, N. Fuchs, and A. Bundy. Lightweight formalisation in support of requirements engineering. *Automated Software Engineering*, 5:183–210, 1998.

[44] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[45] H.-M. Hoercher and J. Peleska. Using formal specification to support software testing. *Software Quality Journal*, 4(4):309–327, 1995.

[46] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[47] J. Horning. Combining algebraic and predicative specifications in Larch. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85 (part II): Formal Methods and Software Development*, volume 186 of *Lect. Notes in Comput. Sci.*, pages 12–26. Springer-Verlag, 1985.

[48] C. Hung. CCS used as a proof-assistant tool. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 387–398. North-Holland, 1986.

[49] International Organization for Standardization, Geneva. *Units of measurement: handbook on international standards for units of measurement*, 1979.

[50] P. Inverardi, H. Muccini, D. Richardson, and S. Ficks. The Teleservices and Remote Medical Care System (TRMCS), 2000.

[51] ISO 8807. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[52] G. Kaiser, S. Dossick, W. Jiang, and J. Yang. An Architecture for WWW-based Hypercode Environments. In R. Adrion, A. Fuggetta, and R. Taylor, editors, *The 19th International Conference on Software Engineering (ICSE'97)*, pages 3–13, Boston, USA, 1997. IEEE Press.

[53] S. K. Kim and D. Carrington. An Integrated Framework with UML and Object-Z for Developing a Precise Specification. In *The 7th Asia-Pacific Software Engineering Conference (APSEC'00)*, pages 240–248. IEEE Press, 2000.

[54] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

[55] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Trans. Software Eng.*, 20(9), September 1994.

[56] J. Liu, J. S. Dong, B. Mahony, and K. Shi. Linking UML with integrated formal techniques. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, pages 210–223. Idea Group Publishing, 2001.

[57] J. Liu. Linking Integrated Formal Methods with UML. Master's thesis, National University of Singapore, 2001.

[58] J. Liu, J. S. Dong, and J. Sun. TRMCS in TCOZ. In *Proceedings of the Tenth International Workshop on Software Specification and Design (IWSSD'00)*, pages 63–72, San Diego, USA, November 2000. IEEE Press.

[59] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1), January 1998.

[60] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual. http://www.formal.demon.co.uk/FDR2.html, May 2000.

[61] D. Luckham and J. Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9), September,1995.

[62] C. Lung and J. Urban. An approach to the classification of domain models in support of analogical reuse. In *Proc. ACM SIGSOFT 1995 Symposium on Software Reusability*, pages 169–178. ACM Press, 1995.

[63] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference*, 1994.

[64] 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.

[65] B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In Araki et al. [7], pages 66–85.

[66] 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.

[67] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[68] 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.

[69] A. P. Martin. Community Z Tools Initiative, 2001. http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/.

[70] T. Miller and P. Strooper. Combining the Animation and Testing of Abstract Data Types. In *The Second Asia-Pacific Conference on Quality Software (APAQS'01)*, pages 249–259. IEEE Press, December 2001.

[71] M. Nielsen, K. Havelund, R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[72] O. Nierstrasz. Active objects in hybrid. In *Proc. 2nd ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, 1987.

[73] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[74] Formal Specification Research (NUS). Z notation in LaTeX to/from XML translator, April 2003. http://www-appn.comp.nus.edu.sg/ rpfm/LTFZ/.

[75] MIT Lab of Computer Science. The Alloy Analyzer, 2002. http://sdg.lcs.mit.edu/alloy/.

[76] J. Ouaknine and J. Worrell. Timed CSP = Closed Timed Safety Automata. In Uwe Nestmann and Prakash Panangaden, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier Science Publishers, 2002.

[77] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[78] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[79] D. Perry and A. Wolf. Foundations for the study of software architecture, 1992. ACM SIGSOFT Software Engineering Notes, 17:40–52, October 1992.

[80] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[81] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Languauge Reference Manual.* Addison-Wesley, 1999.

[82] S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[83] 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.

[84] W3C Schools. XSL - On the Server. http://www.w3schools.com/xsl/xsl_server.asp.

[85] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[86] 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.

[87] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[88] G. Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[89] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[90] G. Smith, F. Kammller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In *2nd International Conference of Z and B Users (ZB'02)*, LNCS. Springer, 2002.

[91] G. Smolka, M. Henz, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, 1995.

[92] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[93] J. Sun, J. S. Dong, J. Liu, and H. Wang. An XML/XSL Approach to Visualize and Animate TCOZ. In *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 453–460. IEEE Press, 2001.

[94] J. Sun, J. S. Dong, J. Liu, and H. Wang. Object-Z Web Environment and Projections to UML. In *WWW-10: 10th International World Wide Web Conference*, pages 725–734. ACM Press, May 2001.

[95] J. Sun and J. S. Dong. Specifying and Reasoning about Generic Architecture in TCOZ. In P. Strooper and P. Muenchaisri, editors, *The 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 405–414. IEEE Press, December 2002.

[96] J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13(1-4):329–356, June 2002.

[97] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.

[98] M. Utting. Animating Z: interactivity, transparency and equivalence. In *Proc. Asia Pacific Software Engineering Conference '95 (APSEC'95)*, pages 294 –303. IEEE Computer Society Press, April 1995.

[99] M. Utting, I. Toyn, J. Sun, A. Martin, J. S. Dong, N. Daley, and D. Currie. Zml: XML support for standard Z. In *3nd International Conference of Z and B Users (ZB'03)*, LNCS. Springer, June 2003.

[100] World Wide Web Consortium (W3C). Document Object Model (DOM). http://www.w3.org/DOM/.

[101] World Wide Web Consortium (W3C). Extensible Markup Language (XML). http://www.w3.org/XML.

[102] World Wide Web Consortium (W3C). Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL.

[103] World Wide Web Consortium (W3C). Mathematical Markup Language (MathML). http://www.w3.org/TR/2002/WD-MathML2-20021219/.

[104] World Wide Web Consortium (W3C). Resource Description Framework (RDF). http://www.w3.org/RDF/.

[105] World Wide Web Consortium (W3C). Semantic Web. http://www.w3.org /2001/sw/.

[106] World Wide Web Consortium (W3C). W3C XML Schema. http://www.w3. org/XML/Schema.

[107] World Wide Web Consortium (W3C). XML Linking Language (XLink). http://www.w3.org/TR/xlink/.

[108] World Wide Web Consortium (W3C). XML Path Language (XPath). http://www.w3.org/TR/xpath/.

[109] World Wide Web Consortium (W3C). XML Pointer Language (XPointer). http://www.w3.org/TR/xptr/.

[110] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999.

[111] E. N. Wafula and P. A. Swatman. FOOM: A Diagrammatic Illustration of Inter-Object Communication in Object-Z Specifications. In *The 1995 Asia-Pacific Software Engineering Conference (APSEC'95)*. IEEE Computer Society Press, December 1995.

[112] M. M. West and B. M. Eaglestone. Software development: two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4):264 – 276, July 1992.

[113] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[114] 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.

[115] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Software Engineering and Methodology*, 6(1):1–30, January 1997.

Appendix A

A.1 Z glossary

Definitions and declarations

Let x, x_k be identifiers and let T, T_k be non-empty, set-valued expressions.

$LHS == RHS$	Definition of LHS as syntactically equivalent to RHS.
$LHS[X_1, X_2, \ldots, X_n] == RHS$	Generic definition of LHS, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets.
$x : T$	A declaration, $x : T$, introduces a new variable x of type T.
$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$	List of declarations.
$x_1, x_2, \ldots, x_n : T$	$== x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$
$[X_1, X_2, \ldots, X_n]$	Introduction of free types named $X_1, X_2, \ldots, X_n$.

Logic

Let P, Q be predicates and let D be a declaration or a list of declarations.

$true, false$	Logical constants.
$\neg P$	Negation: "not P".
$P \land Q$	Conjunction: "P and Q".
$P \lor Q$	Disjunction: "P or Q or both".
$P \Rightarrow Q$	$== (\neg P) \lor Q$ Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	$== (P \Rightarrow Q) \land (Q \Rightarrow P)$ Equivalence: "P is logically equivalent to Q".
$\forall x : T \bullet P$	Universal quantification: "for all x of type T, P holds".
$\exists x : T \bullet P$	Existential quantification: "there exists an x of type T such that P holds".
$\exists_1 x : T \bullet P$	Unique existence: "there exists a unique x of type T such that P holds".

$\forall x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

"For all x_1 of type T_1, x_2 of type T_2, ..., and x_n of type T_n, P holds."

$\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	
	Similar to $\forall$.
$\exists_1\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	
	Similar to $\forall$.
$\forall\, D \mid P \bullet Q$	$\Leftrightarrow \forall\, D \bullet P \Rightarrow Q$
$\exists\, D \mid P \bullet Q$	$\Leftrightarrow \exists\, D \bullet P \land Q$
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\Leftrightarrow \neg\,(t_1 = t_2)$

Sets

Let X be a set; S and T be subsets of X; t, t_k terms; P a predicate; and D declarations.

$t \in S$	Set membership: "t is a member of S".
$t \notin S$	$\Leftrightarrow \neg\,(t \in S)$
$S \subseteq T$	$\Leftrightarrow (\forall\, x : S \bullet x \in T)$
	Set inclusion.
$S \subset T$	$\Leftrightarrow S \subseteq T \land S \neq T$
	Strict set inclusion.

$\varnothing$	The empty set.
$\{t_1, t_2, \ldots, t_n\}$	The set containing the values of terms $t_1, t_2, \ldots, t_n$.
$\{x : T \mid P\}$	The set containing exactly those x of type T for which P holds.
$(t_1, t_2, \ldots, t_n)$	Ordered n-tuple of $t_1, t_2, \ldots, t_n$.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the kth component is of type T_k.
$\mathit{first}(t_1, t_2, \ldots, t_n)$	$== t_1$ Similarly, $\mathit{second}(t_1, t_2, \ldots, t_n) == t_2$, etc.
$\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$	The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.
$\{D \mid P \bullet t\}$	The set of values of the term t for the variables declared in D ranging over all values for which P holds.
$\{D \bullet t\}$	$== \{D \mid \mathit{true} \bullet t\}$
$\mathbb{P}\, S$	Powerset: the set of all subsets of S.

$\mathbb{P}_1 S$	$== \mathbb{P}\, S \setminus \{\varnothing\}$ The set of all non-empty subsets of S.
$\mathbb{F}\, S$	$== \{\, T : \mathbb{P}\, S \mid T \text{ is finite } \}$ Set of finite subsets of S.
$\mathbb{F}_1 S$	$== \mathbb{F}\, S \setminus \{\varnothing\}$ Set of finite non-empty subsets of S.
$S \cap T$	$== \{x : X \mid x \in S \land x \in T\}$ Set intersection.
$S \cup T$	$== \{x : X \mid x \in S \lor x \in T\}$ Set union.
$S \setminus T$	$== \{x : X \mid x \in S \land x \notin T\}$ Set difference.
$\bigcap SS$	$== \{x : X \mid (\forall\, S : SS \bullet x \in S)\}$ Intersection of a set of sets; SS is a set containing as its members subsets of X, i.e. $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\bigcup SS$	$== \{x : X \mid (\exists\, S : SS \bullet x \in S)\}$ Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\#S$	Size (number of distinct members) of a finite set.

Numbers

$\mathbb{R}$	The set of real numbers.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$\mathbb{N}$	$== \{n : \mathbb{Z} \mid n \geq 0\}$ The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	$== \mathbb{N} \setminus \{0\}$ The set of strictly positive natural numbers.
$m \mathinner{..} n$	$== \{k : \mathbb{Z} \mid m \leq k \wedge k \leq n\}$ The set of integers between m and n inclusive.
$\mathit{min}\, S$	Minimum of a set; for $S : \mathbb{P}_1 \mathbb{Z}$, $\mathit{min}\, S \in S \wedge (\forall x : S \bullet x \geq \mathit{min}\, S)$.
$\mathit{max}\, S$	Maximum of a set; for $S : \mathbb{P}_1 \mathbb{Z}$, $\mathit{max}\, S \in S \wedge (\forall x : S \bullet x \leq \mathit{max}\, S)$.

Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$	$== \mathbb{P}(X \times Y)$ The set of relations between X and Y.

$x \underline{R} y$	$== (x, y) \in R$ x is related by R to y.
$x \mapsto y$	$== (x, y)$
$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$	$== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ The relation relating x_1 to y_1, x_2 to y_2, …, and x_n to y_n.
$\operatorname{dom} R$	$== \{x : X \mid (\exists\, y : Y \bullet x \underline{R} y)\}$ The domain of a relation: the set of x components that are related to some y.
$\operatorname{ran} R$	$== \{y : Y \mid (\exists\, x : X \bullet x \underline{R} y)\}$ The range of a relation: the set of y components that some x is related to.
$R_1 \mathbin{;} R_2$	$== \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \land y\, R_2\, z)\}$ Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$.
$R_1 \circ R_2$	$== R_2 \mathbin{;} R_1$ Relational composition. This form is primarily used when R_1 and R_2 are functions.
$R^{\sim}$	$== \{y : Y;\ x : X \mid x \underline{R} y\}$ Transpose of a relation R.

$\mathrm{id}\, S$	$== \{x : S \bullet x \mapsto x\}$
	Identity function on the set S.
R^k	The homogeneous relation R composed with itself k times:
	given $R : X \leftrightarrow X$,
	$R^0 = \mathrm{id}\, X$ and $R^{k+1} = R^k \fatsemi R$.
R^+	$== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$
	$= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
	Transitive closure.
R^*	$== \bigcup\{n : \mathbb{N} \bullet R^n\}$
	$= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
	Reflexive transitive closure.
$R(\!\mid S \mid\!)$	$== \{y : Y \mid (\exists x : S \bullet x \; \underline{R} \; y)\}$
	Image of the set S through the relation R.
$S \lhd R$	$== \{x : X;\ y : Y \mid x \in S \land x \; \underline{R} \; y\}$
	Domain restriction: the relation R with its domain restricted
	to the set S.
$S \ndres R$	$== (X \setminus S) \lhd R$
	Domain subtraction: the relation R with the elements of S
	removed from its domain.
$R \rhd T$	$== \{x : X;\ y : Y \mid x \; \underline{R} \; y \land y \in T\}$
	Range restriction to T.

$R \unrhd T$	$== R \rhd (Y \setminus T)$ Range subtraction of T.
$R_1 \oplus R_2$	$== (\operatorname{dom} R_2 \unlhd R_1) \cup R_2$ Overriding; $R_1, R_2 : X \leftrightarrow Y$.

Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$	The function f applied to t.
$X \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} Y$	$== \{f : X \leftrightarrow Y \mid (\forall\, x : \operatorname{dom} f \bullet (\exists_1\, y : Y \bullet x \underline{f}\ y))\}$ The set of partial functions from X to Y.
$X \rightarrow Y$	$== \{f : X \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} Y \mid \operatorname{dom} f = X\}$ The set of total functions from X to Y.
$X \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrowtail} Y$	$== \{f : X \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} Y \mid (\forall\, y : \operatorname{ran} f \bullet (\exists_1\, x : X \bullet x \underline{f}\ y))\}$ The set of partial one-to-one functions (partial injections) from X to Y.
$X \rightarrowtail Y$	$== \{f : X \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrowtail} Y \mid \operatorname{dom} f = X\}$ The set of total one-to-one functions (total injections) from X to Y.

$X \psurj Y$	$== \{f : X \pfun Y \mid \operatorname{ran} f = Y\}$ The set of partial onto functions (partial surjections) from X to Y.
$X \surj Y$	$== (X \psurj Y) \cap (X \fun Y)$ The set of total onto functions (total surjections) from X to Y.
$X \bij Y$	$== (X \surj Y) \cap (X \inj Y)$ The set of total one-to-one onto functions (total bijections) from X to Y.
$X \ffun Y$	$== \{f : X \pfun Y \mid f \in \mathbb{F}(X \times Y)\}$ The set of finite partial functions from X to Y.
$X \finj Y$	$== \{f : X \inj Y \mid f \in \mathbb{F}(X \times Y)\}$ The set of finite partial one-to-one functions from X to Y.
$(\lambda\, x : X \mid P \bullet t)$	$== \{x : X \mid P \bullet x \mapsto t\}$ Lambda-abstraction: the function that, given an argument x of type X such that P holds, gives a result which is the value of the term t.
$(\lambda\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet t)$	$== \{x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$
disjoint$[I, X]$	$== \{S : I \pfun \mathbb{P}\, X \mid \forall\, i, j : \operatorname{dom} S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \varnothing\}$

	Pairwise disjoint; where I is a set and S an indexed family of subsets of X (i.e. $S : I \pfun \mathbb{P}\, X$).
$S\ \underline{\text{partitions}}\ T$	$== S \in \text{disjoint} \wedge \bigcup \operatorname{ran} S = T$

Sequences

Let X be a set; A and B be sequences with elements taken from X; and $a_1, \ldots, a_n$ terms of type X.

$\operatorname{seq} X$	$== \{A : \mathbb{N}_1 \pfun X \mid (\exists\, n : \mathbb{N} \bullet \operatorname{dom} A = 1..n)\}$ The set of finite sequences whose elements are drawn from X.
$\operatorname{seq}_\infty X$	$== \{A : \mathbb{N}_1 \pfun X \mid A \in \operatorname{seq} X \vee \operatorname{dom} A = \mathbb{N}_1\}$ The set of finite and infinite sequences whose elements are drawn from X.
$\#A$	The length of a finite sequence A. (This is just '#' on the set representing the sequence.)
$\langle\rangle$	$== \{\}$ The empty sequence.
$\operatorname{seq}_1 X$	$== \{s : \operatorname{seq} X \mid s \neq \langle\rangle\}$ The set of non-empty finite sequences.
$\langle a_1, \ldots, a_n\rangle$	$= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$

$\langle a_1, \ldots, a_n \rangle \frown \langle b_1, \ldots, b_m \rangle$	
	$= \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$ Concatenation. $\langle\rangle \frown A = A \frown \langle\rangle = A$.
$\mathit{head}\ A$	The first element of a non-empty sequence: $A \neq \langle\rangle \Rightarrow \mathit{head}\ A = A(1)$.
$\mathit{tail}\ A$	All but the head of a non-empty sequence: $\mathit{tail}\ (\langle x \rangle \frown A) = A$.
$\mathit{last}\ A$	The final element of a non-empty finite sequence: $A \neq \langle\rangle \Rightarrow \mathit{last}\ A = A(\#A)$.
$\mathit{front}\ A$	All but the last of a non-empty finite sequence: $\mathit{front}\ (A \frown \langle x \rangle) = A$.
$\mathit{rev}\ \langle a_1, a_2, \ldots, a_n \rangle$	
	$= \langle a_n, \ldots, a_2, a_1 \rangle$ Reverse of a finite sequence; $\mathit{rev}\ \langle\rangle = \langle\rangle$.
$\frown/\ AA$	$= AA(1) \frown \ldots \frown AA(\#AA)$ Distributed concatenation; where $AA : \mathrm{seq}(\mathrm{seq}(X))$. $\frown/\langle\rangle = \langle\rangle$.
$A \subseteq B$	$\Leftrightarrow \exists\, C : \mathrm{seq}_\infty X \bullet A \frown C = B$ A is a prefix of B. (This is just '$\subseteq$' on the sets representing the sequences.)

$\mathrm{squash}\, f$	Convert a finite function, $f : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} X$, into a sequence by squashing its domain. That is, $\mathrm{squash}\{\} = \langle\rangle$, and if $f \neq \{\}$ then $\mathrm{squash}\, f = \langle f(i)\rangle \frown \mathrm{squash}(\{i\} \lhd f)$, where $i = min(\mathrm{dom}\, f)$. For example, $\mathrm{squash}\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C\rangle$.
$A \upharpoonright T$	$== \mathrm{squash}(A \rhd T)$ Restrict the range of the sequence A to the set T.

Bags

$\mathrm{bag}\, X$	$== X \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{N}_1$ The set of bags whose elements are drawn from X. A bag is represented by a function that maps each element in the bag onto its frequency of occurrence in the bag.
$[\![\ ]\!]$	The empty bag $\varnothing$.
$[\![x_1, x_2, \ldots, x_n]\!]$	The bag containing $x_1, x_2, \ldots, x_n$, each with the frequency that it occurs in the list.
$items\ s$	$== \{x : \mathrm{ran}\, s \bullet x \mapsto \#\{i : \mathrm{dom}\, s \mid s(i) = x\}\}$ The bag of items contained in the sequence s.

Axiomatic definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D. These variables must satisfy the predicate P. The scope of the variables is the whole specification.

$$\begin{array}{|l}
D \\
\hline
P
\end{array}$$

Generic definitions

Let D be a list of declarations, P a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$\begin{array}{|l}
\hline\hline
[X_1, X_2, \ldots X_n] \\
D \\
\hline
P \\
\hline
\end{array}$$

The declared variables must be uniquely defined by the predicate P.

Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$\begin{array}{|l}
\hline
S \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \#y \\
\hline
\end{array}$$

and horizontally, for the same example,

$$S == [x : \mathbb{N};\ y : \operatorname{seq}\mathbb{N} \mid x \leq \#y]$$

Schemas can be used in signatures after $\forall$, λ, $\{...\}$, etc.:

$$(\forall S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall x : \mathbb{N};\ y : \operatorname{seq}\mathbb{N} \mid x \leq \#y \bullet y \neq \langle\rangle)$$

$\{S\}$	Stands for the set of objects described by schema S. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$.

Schema operators

Let S be defined as above and $w : S$.

$w.x$	$== (\lambda S \bullet x)(w)$ Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \#w.y$' holds.
θS	The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y.
Compatibility	Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same

name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must have the same declared sets) and the predicates of S and T are conjoined. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
S \\
z : \mathbb{N} \\ \hline
z < x \\ \hline
\end{array}$$

is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
x, z : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\ \hline
x \leq \# y \land z < x \\ \hline
\end{array}$$

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (T).

Decoration Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example, S' is

$[x' : \mathbb{N};\ y' : \operatorname{seq} \mathbb{N} \mid x' \leq \# y']$.

$\neg\, S$ — The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid \neg\,(x \leq \#y)]$.

$S \land T$ — The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above).

Given $T == [x : \mathbb{N};\ z : \mathbb{P}\,\mathbb{N} \mid x \in z]$, $S \land T$ is

$$\begin{array}{|l}
\hline
S \land T \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \land x \in z \\
\hline
\end{array}$$

$S \lor T$ — The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \lor T$ is

$$\begin{array}{|l}
\hline
S \lor T \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\,\mathbb{N} \\
\hline
x \leq \#y \lor x \in z \\
\hline
\end{array}$$

$S \Rightarrow T$ — The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Rightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \Rightarrow T} \\ \hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\ \hline
x \leq \#y \Rightarrow x \in z \\ \hline
\end{array}$$

$S \Leftrightarrow T$

The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Leftrightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \Leftrightarrow T} \\ \hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\ \hline
x \leq \#y \Leftrightarrow x \in z \\ \hline
\end{array}$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The paranthe-ses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are re-moved from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{(S \land T) \upharpoonright (x, y)} \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
(\exists\, z : \mathbb{P}\,\mathbb{N} \bullet \\
\qquad x \leq \#y \land x \in z) \\
\hline
\end{array}$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists\, D \bullet S$

Existential quantification of a schema.

The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is existentially quantified over D. For example, $\exists\, x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, x : \mathbb{N} \bullet S} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
\exists\, x : \mathbb{N} \bullet \\
\qquad x \leq \#y \\
\hline
\end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, S \bullet T} \\
z : \mathbb{N} \\
\hline
\exists\, S \bullet \\
\qquad x \leq \#y \land z < x \\
\hline
\end{array}$$

$\forall\, D \bullet S$

Universal quantification of a schema.

The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is universally quantified over D. For example, $\forall\, x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall x : \mathbb{N} \bullet S} \\
\hline
y : \operatorname{seq} \mathbb{N} \\
\hline
\forall x : \mathbb{N} \bullet \\
\qquad x \leq \# y \\
\hline
\end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall S \bullet T} \\
\hline
z : \mathbb{N} \\
\hline
\forall S \bullet \\
\qquad x \leq \# y \wedge z < x \\
\hline
\end{array}$$

Operation schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

undashed state before the operation,

dashed state after the operation,

ending in "?" inputs to (arguments for) the operation, and

ending in "!" outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS	$\widehat{=} S \wedge S'$
	Change of state schema: this is a default definition for ΔS.

In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\quad \widehat{=} [\Delta S \mid \theta S' = \theta S]$

No change of state schema.

Operation schema operators

$\mathrm{pre}\, S$ Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

$$\begin{array}{|l}
\hline
S \\
x?, s, s', y! : \mathbb{N} \\
\hline
s' = s - x? \land y! = s' \\
\hline
\end{array}$$

$\mathrm{pre}\, S$ is,

$$\begin{array}{|l}
\hline
\mathrm{pre}\, S \\
x?, s : \mathbb{N} \\
\hline
\exists\, s', y! : \mathbb{N} \bullet \\
\qquad s' = s - x? \land y! = s' \\
\hline
\end{array}$$

$S;\ T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T

that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S;\ T$ is,

$$\begin{array}{|l}
\multicolumn{1}{l}{S;\ T} \\
\hline
x?, s, s', y! : \mathbb{N} \\
\hline
(\exists\, ss : \mathbb{N} \bullet \\
\qquad ss = s - x? \land y! = ss \\
\qquad \land\ ss \leq x? \land s' = ss + x?) \\
\hline
\end{array}$$

A.2 TCOZ glossary

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
$\perp$	divergent process
$\textsc{Stop}$	deadlocked process
$\textsc{Skip}$	terminate immediately
$\textsc{Wait}\, t$	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

continued on next page

Notation	Explanation
$P;\ Q$	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
$P \mid[A]\mid Q$	synchronise P and Q on events from A

continued on next page

Notation	Explanation
$(\lVert\ p_1, \ldots, p_n \bullet \ldots;\ p_i \overset{A}{\longleftrightarrow} p_j;\ \ldots)$	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
$P \rhd\{t\}\ Q$	if P does not begin by time t, perform Q instead
$P \swarrow\!\!\!\diagup\{t\}\ Q$	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