Annals of Software Engineering 13, 329–356, 2002

A Formal Object Approach to the Design of ZML

JING SUN, JIN SONG DONG, JING LIU and HAI WANG dongjs@comp.nus.edu.sg
Department of Computer Science, School of Computing, National University of Singapore,
10 Kent Ridge Crescent, Singapore 119260, Republic of Singapore

Abstract. This paper addresses two issues: how formal object modeling techniques facilitate the XML application development and how XML technology helps formal/graphical software design process. In particular, the paper presents a XML/XSL approach to the development of a web environment for Z family languages (Z/Object-Z/TCOZ). The projection techniques and tools from object-oriented Z (in XML) to UML (in XMI) are developed using XSL Transformations (XSLT). Furthermore, object-oriented Z is used to specify and design the essential functionalities of the web environment and the projection tools to UML. In a sense, the paper also demonstrates a formal object approach to modeling XML applications.

Keywords: formal specification, Z/Object-Z/TCOZ, XML/XSL/XMI, UML

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 paper tries to address both issues within a specific context "XML [(W3C) 2000a]/XSL [(W3C) 2000b] and Formal/Graphical software modeling techniques."

One reason for the slow adoption of formal methods (FM) is the lack of tools 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. 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 [Kaiser *et al.* 1997]. Unified Modeling Language (UML) [Rumbaugh *et al.* 1999] is commonly regarded as one of the dominate 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.

Z [Woodcock and Davies 1996] is a state-oriented formal specification language based on set theory and predicate logic. Object-Z [Duke and Rose 2000; Smith 2000]

is an object-oriented extension to Z. TCOZ [Mahony and Dong 2000; Mahony and Dong 1999] integrates Object-Z with process algebra Timed-CSP [Schneider and Davies 1995]. In this paper, we plan firstly to use eXtensible Markup Language (XML) and eXtensible Stylesheet Language (XSL) to develop a web environment that provide various browsing and syntax checking facilities for Z family (Z/Object-Z/TCOZ) languages. Second, with the emergence of XML Metadata Interchange (XMI) as a standard, e.g., Rational Rose UML supports XMI input, it is possible to build a transformation link and projection tools from object-oriented Z specifications (in XML) to UML (in XMI) via XSLT [(W3C) 1999] technology. We call this Z family XML web environment and UML projection facilities, *ZML*.

Since we believe that FM can improve software reliability for web 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 paper demonstrates a formal approach to modeling XML applications. Consequently, *we eat our own medicine*.

The remainder of the paper is organised as follows. Section 2 briefly introduces the Z family (Z/Object-Z/TCOZ) 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 paper, discusses related work. Section 5 presents the implementation issues of the web environment and browsing facilities for Z family. Section 6 presents the implementation issues of the projection tools from Object-Z (in XML) to UML (in XMI). Section 7 concludes the paper.

2. Z family languages overview

In this section, we will use a simple message queue system to illustrate the common and difference among Z, Object-Z and TCOZ notations. Z schema calculus and Object-Z/TCOZ inheritance expansions (which is the challenge of the ZML development) are explained. Requirements of building ZML are highlighted (in *italic* fonts).

2.1. Z and schema calculus

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 [Hayes 1993]). 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 total elements in the queue cannot be more than *max* (say, a number larger than 100). The global constant *max* can be defined using Z axiomatic definition as

$$max : \mathbb{N}$$
$$max > 100$$

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

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

ΔQueue
$$items : \operatorname{seq} MSG$$
$$items' : \operatorname{seq} MSG$$
$$\# items \leqslant max$$
$$\# items' \leqslant max$$

QueueInit
$$Queue$$
$$items = \langle\,\rangle$$

QueueInit$_e$
$$items : \operatorname{seq} MSG$$
$$\# items \leqslant max$$
$$items = \langle\,\rangle$$

where *QueueInit*$_e$ is the schema inclusion expanded form of *QueueInit*.

The operations to add message to, and delete message from, the queue can be modeled as

Add
$$\Delta Queue$$
$$item? : MSG$$
$$items' = items^\frown\langle item?\rangle$$

Delete
$$\Delta Queue$$
$$item! : MSG$$
$$items \neq \langle\,\rangle \wedge items = \langle item!\rangle^\frown items'$$

Complex operations can be constructed by using schema calculus, e.g., a new message pushes out an old message, say *Penguin*, can be specified by using the sequential composition schema operator ; as

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

which is an (atomic) operation with the effect of an *Add* followed by a *Delete*. The expanded form of *Penguin* is

Penguin$_e$
$$\Delta Queue$$
$$item?, item! : MSG$$
$$\exists\, items'' : \operatorname{seq} MSG \bullet items'' = items^\frown\langle item?\rangle \wedge items'' \neq \langle\,\rangle$$
$$\wedge items'' = \langle item!\rangle^\frown items'$$

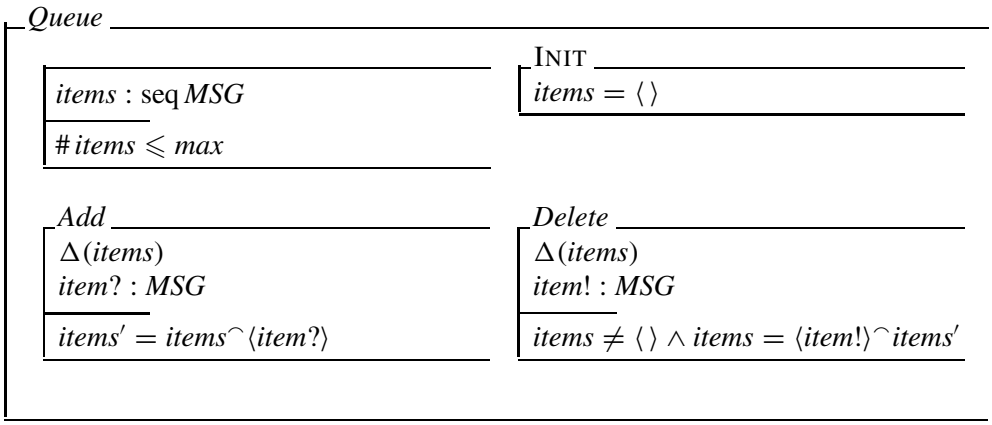
Other forms of schema calculus include schema conjunction '$\wedge$', disjunction '$\vee$', implication '$\Rightarrow$', negation '$\neg$' and pipe '$\gg$', which has been discussed in many Z text books.

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

2.2. Object-Z

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. The essential extension to Z in Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

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



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 behavioural interpretation of Object-Z objects is as transition systems [Smith 1995]. A behaviour 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.

2.3. TCOZ and inheritance

Timed CSP [Schneider and Davies 1995] has strong process control modeling capabilities. The multi-threading and synchronization primitives of CSP are extended with timing primitives. The approach taken in the Timed Communicating Object Z (TCOZ) [Mahony and Dong 1998] is to identify Object-Z operation schemas (semantically) with

(terminating) CSP processes that perform only state update events; to identify active classes[1] [Dong and Mahony 1998] with non-terminating CSP processes (MAIN); and to allow arbitrary (channel-based) communications interfaces between objects.

The syntactic implications of this approach are 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. Since operation schemas take on the syntactic role of CSP processes, they may be combined with other schemas and even CSP processes using the standard CSP process operators. Thus, it becomes possible to represent multi-threaded computation. TCOZ is a superset of Object-Z and all Object-Z classes are treated as passive classes (without MAIN operation) in TCOZ.

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. Active class can be defined by inheriting passive classes. For instance, an active Queue can be derived from the previous (Object-Z) queue model as

ActiveQueue

Queue

$T_i,\ T_j : \mathbb{T}$ [durations for Join/Leave operations]
in, *out* : **chan** [channels for input and output]

$Join \mathrel{\widehat=} [item : MSG \mid \#items < max] \bullet in?item \rightarrow Add \bullet \text{DEADLINE } T_j$
$Leave \mathrel{\widehat=} [items \neq \langle\rangle] \bullet out!head(items) \rightarrow Delete \bullet \text{DEADLINE } T_l$
$\text{MAIN} \mathrel{\widehat=} \mu Q \bullet Join \,\Box\, Leave;\ Q$

where the expanded form of this active queue model is

ActiveQueue$_e$

$items : \text{seq}\, MSG$
$T_i,\ T_j : \mathbb{T};\ in,\ out$: **chan**
$\#items \leqslant max$

INIT
$items = \langle\rangle$

Add
$\Delta(items)$
$item? : MSG$
$items' = items^\frown\langle item?\rangle$

Delete
$\Delta(items)$
$item! : MSG$
$items \neq \langle\rangle \wedge items = \langle item!\rangle^\frown items'$

[1] Objects of active classes have their own thread of control, while passive objects (of Object-Z classes) are controlled by others.

$$Join \mathrel{\widehat=} [item : MSG \mid \#items < max] \bullet in?item \rightarrow Add \bullet \text{DEADLINE } T_j$$
$$Leave \mathrel{\widehat=} [items \neq \langle\rangle] \bullet out!head(items) \rightarrow Delete \bullet \text{DEADLINE } T_l$$
$$\text{MAIN} \mathrel{\widehat=} \mu Q \bullet Join \,\Box\, Leave;\ Q$$

where real-time type $\mathbb{T}$ can be modelled by $\mathbb{N}$ if discrete or by $\mathcal{R}$ if continuous.

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 facility is 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/reviewing the class in isolation. It is desirable for ZML to automatically support the inheritance zoom-in/out features.

2.4. Instantiation and composition

Let C be the name of a class. It denotes semantically 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 can be composed 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

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

$\text{MAIN} \mathrel{\widehat=} q_1 \,|[talk]|\, q_2$

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 *TwoActive Queues*) captures the concurrent and synchronization behaviour of the two communicating active processes q_1.MAIN and q_2.MAIN.

The models of *TwoQueues* and *TwoActiveQueues* appear to have similar behaviour. However, the behaviour of *TwoQueues* is purely sequential. For example, *Join* (q_1.*Add*) and *Leave* (q_2.*Delete*) cannot concurrently operate or partially overlapping in durations (even assuming 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 *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 ZML tool.

3. Formal design model of ZML

The construction of a formal design model for ZML must start with formalizing the related syntax definitions of Z family languages. The typing and dynamic semantics issues are not related since ZML only concerns the 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 ZML. 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 in ZML. TCOZ is more suited for modeling timed/concurrent interactive systems, and perhaps it is an overkill for designing ZML even though ZML is computationally complex (when dealing with schema calculus and inheritance expansions) but the interactive behaviour of ZML is simple.

3.1. Formal models of the Z family web environment

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

$$Char ::= \text{`a'} \mid \text{`b'} \mid \ldots \mid \text{`1'} \mid \text{`2'} \mid \ldots \mid \text{`:'} \mid \text{`/'} \mid \text{`\#'} \mid \ldots$$

The string type is defined as a sequence of characters:

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

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

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

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:

$$Name == \{s : String \mid \operatorname{ran} s \subseteq \{\text{`a'}, \ldots, \text{`A'}, \ldots, \text{`\_'}\}\}$$

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. ' $\rightarrow$ ', ' $\pfun$ ' and unary constructors i.e. '$\mathbb{P}$', '$\mathbb{F}$'.

$$TypeConst == \{s : String \mid \#s = 1 \land \operatorname{ran} s \subseteq \{\text{'}\mathbb{P}\text{'}, \text{'}\mathbb{F}\text{'}, \text{'}\mathbb{P}_1\text{'}, \text{' } \times \text{ '}, \text{' } \rightarrow \text{ '}, \text{' } \pfun \text{ '}, \ldots\}\}$$

Syntactically, a type constructor, a type and a predicate constructor are similar, which are modelled as

TypeConstructor

content : *TypeConst*

Type

name : *Name*

PredConstructor

content : *String*

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

$$Dtype == \operatorname{seq}(TypeConstructor \cup \ \downarrow Type)$$
$$Predicate == \operatorname{seq}(PredConstructor \cup \ \downarrow Type)$$

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

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

Typedef

Type

defs : *Dtype*

Axiomdef

Type

decpart : *Name* $\rightarrow$ *Dtype*
axpart : $\mathbb{P}$ *Predicate*

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:

$$Inclusion == \{\text{'inc'}, \text{'xi'}, \text{'del'}\} \ \pfun \ \mathbb{P}\, Name$$

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

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

The schema box format is defined as

Schemadef $_1$

Type

$$\begin{array}{l} incl : Inclusion \\ decpart : Name \rightarrow Dtype \\ axpart : \mathbb{P}\ Predicate \end{array}$$

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*:

$$CalcOp == \{s : String \mid \#s = 1 \land \operatorname{ran} s \subseteq \{\text{`}\land\text{'}, \text{`}\lor\text{'}, \text{`}\fatsemi\text{'}, \text{`}\gg\text{'}, \text{`}\neg\text{'}, \ldots\}\}$$

$$PredCalc == CalcConstructor \cup \ \downarrow Type$$

CalcConstructor

$$\begin{array}{l} op : CalcOp \\ items : \mathbb{P}\ PredCalc \end{array}$$

Schemadef $_2$

Type

$$calc : PredCalc$$

Object-Z language is mainly composed of class definitions. Firstly, we model state, initial and operation schemas as follows:

Statedef

$$\begin{array}{l} decpart : Name \rightarrow Dtype \\ axpart : \mathbb{P}\ Predicate \end{array}$$

Initdef

$$axpart : \mathbb{P}\ Predicate$$

Opdef

Statedef

$$\begin{array}{l} name : Name \\ delta : \mathbb{P}\ Name \end{array}$$

An Object-Z class is modeled as

Classdef

Type

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

A *ZDefinition* is either a *Typedef*, *Axiomdef*, *Schemadef* or *Classdef*:

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

The Z family web browsing environment is modelled as

WebBE

$$
\begin{array}{l}
zspec : \mathbb{P}\ ZDefinition \qquad \text{[a specification]} \\
mainpage : URL \qquad \text{[the main URL address]} \\
currpage : URL \qquad \text{[the current page URL address]} \\
expandpos : Name \pfun \mathbb{B} \qquad \text{[all expansible positions]}
\end{array}
$$

INIT

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

Clicklink

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

Clickexpand

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

As for the expansion purpose we introduced an attribute *expandpos* which stores the names of inherited classes and schemas defined by inclusion/schema-calculus. 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:

$$expand : (Classdef \cup Schemadef) \rightarrowtail (Classdef \cup Schemadef)$$

$$\begin{aligned}
&\forall\, def : (Classdef \cup Schemadef) \bullet \\
&\quad def \in Classdef \Rightarrow expand(def) = expand_c(def) \\
&\quad def \in Schemadef_1 \Rightarrow expand(def) = expand_{z1}(def) \\
&\quad def \in Schemadef_2 \Rightarrow expand(def) = expand_{z2}(def)
\end{aligned}$$

where $expand_c$, $expand_{z1}$, $expand_{z2}$ and other auxiliary functions are defined in the Appendix.

3.2. *Formal model of the projection facilities*

An UML class consists of a class name, a set of attributes and a set of operation names:

UMLClass

$$\begin{aligned}
&name : String \\
&attris : String \rightarrow Dtype \\
&ops : \mathbb{P}\, String
\end{aligned}$$

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

UMLDiagram

$$\begin{aligned}
&classes : \mathbb{P}\, UMLClass \\
&inh, agg : UMLClass \leftrightarrow UMLClass \\
&\hline
&\mathrm{dom}(inh \cup agg) \cup \mathrm{ran}(inh \cup agg) \subseteq classes \\
&\forall\, h : classes \bullet (h, h) \notin inh^+
\end{aligned}$$

A function *project* models the transformation from an Object-Z specification to a UML class diagram:

$$\begin{aligned}
&project : \mathbb{P}\, Classdef \rightarrow UMLDiagram \\
&\hline
&\forall\, (oz, uml) : project \bullet \\
&\quad \{c : oz \bullet c.name\} = \{c : uml.classes \bullet c.name\} \bullet \\
&\quad \forall\, c_1, c_2 : oz \bullet \\
&\qquad \exists_1\, c' : uml.classes \bullet \\
&\qquad\quad c'.name = c_1.name \\
&\qquad\quad c'.attris = \{cls : oz \bullet cls.name\} \lhd c_1.state.decpart \\
&\qquad\quad c'.ops = \{o : Opdef \mid o \in c_1.ops \bullet o.name\}
\end{aligned}$$

$$
\begin{aligned}
& c_2.name \in \{t : \operatorname{ran} c_1.state.decpart \bullet t.name\} \Rightarrow \\
& \quad \exists_1 (c_1', c_2') : uml.agg \bullet c_1'.name = c_1.name \land c_2'.name = c_2.name \\
& c_2.name \in \{inh : \operatorname{dom} c_1.inherit \bullet inh.name\} \Rightarrow \\
& \quad \exists_1 (c_1', c_2') : uml.inh \bullet c_1'.name = c_1.name \land c_2'.name = c_2.name
\end{aligned}
$$

Note that our projection function from Object-Z/TCOZ specifications to UML diagrams focuses on UML class diagrams at the current stage. The projection to UML behaviour diagrams such as statecharts may not be uniquely determined given an Object-Z/TCOZ specification. We will discuss about the projection to statechart diagrams further in section 6.

4. Main implementation issues and related background

Formal method like CafeOBJ system [Futatsugi and Nakagawa 1997] has included an environment supporting formal specifications over networks. Pure Z notation on the web based on HTML and Java applet has also been investigated by Bowen and Chippington [1998] and Ciancarini, Mascolo and Vitali [1998]. HTML has been successful in presenting information on the Internet, however, the lack of content information and overburdened of all kinds of tags have made the retrieval and exchange of resource to become more and more difficult to perform.

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 (Java Applet) of Ciancarini *et al.* [1998], 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 to UML are built into our system. The conceptual projection techniques are derived from our research on linking UML with Object-Z [Liu *et al.* 2000], which are similar to the translation rules developed by Kim and Carrington [2000]. The difference is that we are working on the projection from Object-Z/TCOZ to UML where Kim and Carrington focus on translating UML to a partial Object-Z specification (a different direction from ours). We share similar goals (of visualizing Object-Z) with the work of [Wafula and Swatman 1995]. Other work (e.g., [Evans and Clark 1998]) on linking Z and UML mainly concentrates on using Z to define the semantics for UML class diagrams.

The reason that we chose XML rather than MathML 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 be constructed in MathML. Furthermore, MathML usually consists of heavy load of defined tags, which is unbearable for the 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

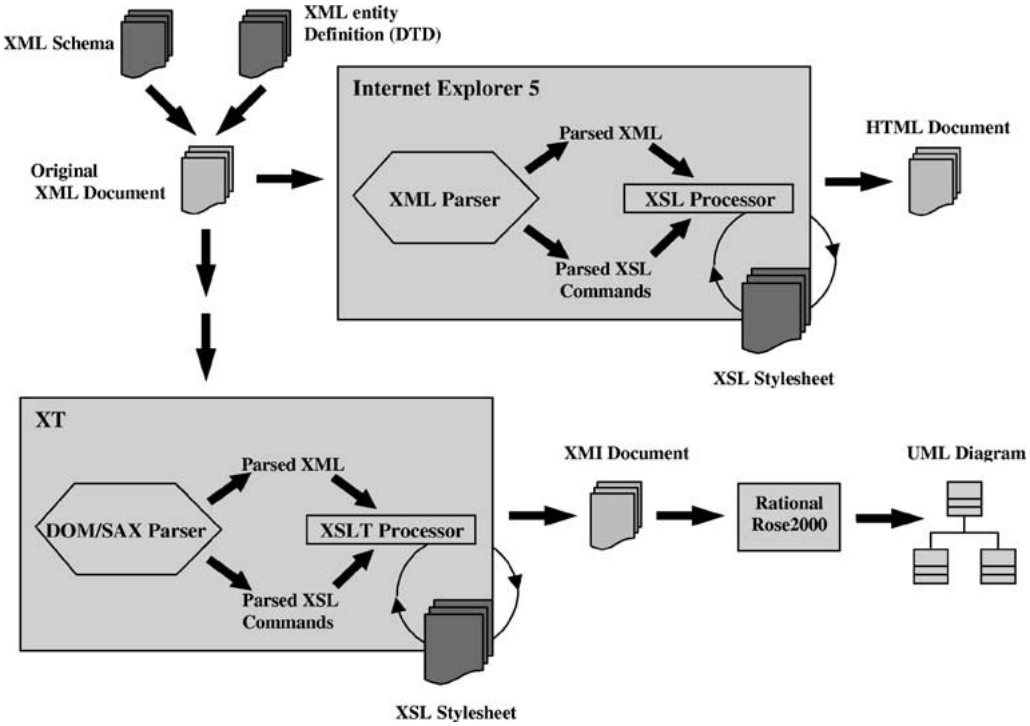


Figure 1. ZML overview diagram.

possible to the LaTeX style files for Z/Object-Z (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.

The main process and techniques for ZML are depicted by figure 1. 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 is acted as a precise design reference document and provides clear guidelines to 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 5 is based on the expand function defined in section 3.1 and the appendix; the XSLT codes for projecting Object-Z to UML in section 6 is based on the project function defined in section 3.2.

5. Web environment for Z family

5.1. Syntax definition and usage

Firstly, a customized XML document for Object-Z is defined according to its syntax formal definitions. This document is used for checking the syntax validity of the user input specifications in XML. The World Wide Web Consortium (W3C) has provided

two mechanisms for describing XML structures: Document Type Definition (DTD) and XML Schema. The former is originated from SGML Recommendation and used a totally different syntax. XML Schema is a kind of XML file itself and is going to play the role of DTD in defining customized XML structure in the future. It is consistent with XML syntax and easy to write over DTD. We use XML Schema to define our ZML structure syntax for the Z family notations. Part of the XML Schema (for defining a class and its operation schema) is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
...
<ElementType name="op" content="eltOnly" order="seq">
  <element type="name" minOccurs="1" maxOccurs="1"/>
  <element type="delta" minOccurs="0" maxOccurs="1"/>
  <element type="decl" minOccurs="0" maxOccurs="*"/>
  <element type="st" minOccurs="0" maxOccurs="1"/>
  <element type="predicate" minOccurs="0" maxOccurs="*"/>
  <AttributeType name="layout" dt:type="enumeration" dt:values="simpl calc"
default="simpl"/>
  <attribute type="layout"/>
</ElementType>
<ElementType name="classdef" content="eltOnly">
  ...
  <element type="op" minOccurs="0" maxOccurs="*"/>
  ...
</ElementType>
...
</Schema>
```

It states that the *op* tag is an element of *classdef* and consists of one *name*, a $\Delta -$ *delta* list, a number of declarations *decl*, an horizontal line *st* and some *predicate* definitions. An attribute *layout* is defined to distinguish between vertical layout schemas *simpl* and horizontal layout schemas *calc*.

Z family languages consist of a rich set of mathematical symbols. Those symbols can be presented directly in Unicode 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. Part of the entity declaration DTD is defined as follows:

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

As most existing Z specifications were constructed in LaTeX, translating them to our format can be a trivial task due to that each entity is given a Z LaTeX compatible name. DTD is chosen to define our entity declaration because XML Schema does not support entity declaration at the moment. When authoring ZML files, the user simply declares the name space of the XML schema and Entity DTD file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<!DOCTYPE unicode SYSTEM
"http://nt-appn.comp.nus.edu.sg/fm/zml/unicode.dtd">
<objectZnotation xmlns="x-schema:
  http://nt-appn.comp.nus.edu.sg/fm/zml/objectZschema.xml"
  xmlns:HTML="http://www.w3.org/Profiles/XHTML-transitional">
...
</objectZnotation>
```

With the above namespace links, the XML editing tools can check the validity of the file via XML Schema definition and the DTD entity declarations. Any unspecified structures and entity symbols would be reported as a syntax error. The following is the Web browsing environment for the *Queue* class (of the queue specification example) in our ZML format:

```
<classdef layout="simpl" align="left">
  <name>Queue</name>
  <state>
    <decl>
      <name>items</name>
      <dtype>&seq; <type>MSG</type></dtype>
    </decl>
    <st/>
    <predicate># items &leq; max</predicate>
  </state>
  <init>
    <predicate>items=&emptyseq;</predicate>
  </init>
  <op layout="simpl">
    <name>Add</name>
    <delta>items</delta>
    <decl>
      <name>item?</name>
      <dtype><type>MSG</type></dtype>
    </decl>
    <st/>
    <predicate>items'=items &cat; &lseq;item?&rseq;</predicate>
  </op>
  <op layout="simpl">
    <name>Delete</name>
    ...
  </op>
</classdef>
```

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. A XSL file is a XML document itself and it can perform the transformation between 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 Internet Explorer 5 (IE5) has already supported 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 *op* and class definition *classdef* are defined as below:

```
<xsl:template match="op[@layout='simpl']">
<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 | decl">
    <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>
```

XSL stylesheet defines *match* method for each customized tag in the XML structure and describes the corresponding HTML codes. From the example above, in matching the '`op`' 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 a 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 make a function call in programming language, and each template will perform its own transformation. When authoring Z family specifications in our ZML format, the users only need to construct their ZML files and add an 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"?>
```

With this link, the browser (IE5) will automatically transform ZML document into desired HTML outputs. This process is totally user transparent and much faster than

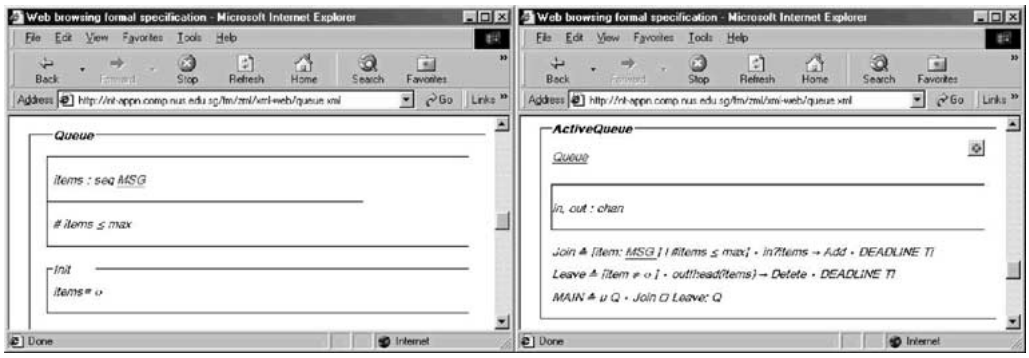


Figure 2. Queue specification on web.

the Java Applet approaches [Bowen and Chippington 1998; Ciancarini *et al.* 1998]. For example, the `Queue` and `ActiveQueue` classes in ZML format specified previously is transformed into HTML as in figure 2.

A full demonstration of the *Queue* specification example is available at `http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/queue.xml`

5.3. Extensive browsing facilities

This section discusses the extensive browsing facilities for type reference, class inheritance expansion and schema calculus expansion.

5.3.1. Type referencing

When building a large formal model, which could include large numbers of type definitions, users often want to recall the definition of a particular type. Type referencing allows the user to browse back to actual type definition.

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

```
<xsl:template match="type">
  <xsl:choose>
    <xsl:when test="//classdef[$any$ name=context(-1)] |
      //tydef[$any$ 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>
```

5.3.2. Class inheritance and schema calculus expansions

The aim of class inheritance expansion is to allow user to view the full definition of a derived class. In the *ActiveQueue* class case (in the right-hand side of figure 2), when a user clicks the button '+', the full definition of class of "ActiveQueue" will be shown. This implementation is based on the inheritance expansion rules defined in the $expand_c$ function of appendix. Clicking button '−' is for going back to the unexpanded version.

The core part of the expansion techniques uses the match facilities provided by XSL to find the corresponding definitions in the parent class and merge them in 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)/inherit/type]/state/decl">
...
</xsl:for-each>
<xsl:for-each select="state/decl">
...
</xsl:for-each>
...
```

As we can see from above, the *select* constraint will restrict a search through the entire ZML document for a match of same named class definition corresponding to the name in the *inherit* list. And then the state declaration of super class is merged with the current class. Thus, the whole definition of *state* declarations in the derived class is constructed. 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.

5.4. Server side transformation

As mentioned in section 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 our ZML data available to all kinds of browsers, we can perform the transform on the server side and sent back pure HTML to the browsers. 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.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 the code creates an instance of the Microsoft XML parser (XMLDOM), and loads the XML file into memory. 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 document, and returns the result HTML to the browser.

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

6. UML photos

UML can be used to visualize the Object-Z/TCOZ models. The textual specifications of UML models are in XMI format. Based on XSL Transformations (XSLT) [(W3C) 1999] technology, we define an XSL file to capture all translation rules from Object-Z/TCOZ (in XML) to UML (in XMI). XT [Clark 1999] is chosen as the XSLT processor and Rational Rose 2000 is used as the UML tool. By now we have fully implemented the visualization of UML class diagrams (including reverse transformation) and are looking into other dynamic UML diagrams, i.e. statecharts. In our approach, all elements from the static view, such as attributes, operations, classes and their relationships (inheritance and aggregation) can be successfully captured through the transformation process.

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 3). An XMI file has the structure as follows:

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

The *XMI.header* section includes some optional information about 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 3.2) and the file is consistent with *UML.DTD*. The template technology plays a key role in implementing the translation rules. Consider the implementation issues and the translation rules based on the formal model, the following guidelines are formed:

- Each class in Object-Z/TCOZ XML models corresponds to a class in UML XMI models. They have the same name, attributes and operations.

- If a type value in the *Inherit* part of a class matches the name of any other class in the current ZML file, we regard that 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 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 current ZML file, this is regarded as aggregation relationship between these two classes. The cardinality of the aggregation will be calculated and classified into UML aggregation ranges.

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 in the paper:

```
<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>
```

The projection rules for translating formal model to UML class diagrams are trivial. As in figure 3, 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 our 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 XMI segment is as follows (simplified):

```
<Foundation.Core.Generalization xmi.id = ' G.13 '>
        <name/>
```

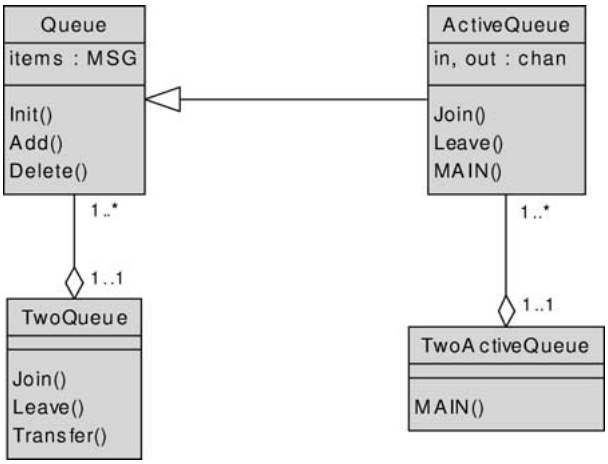


Figure 3. Generated class diagram.

```
        <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 XMI segment (simplified):

```
<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>
```

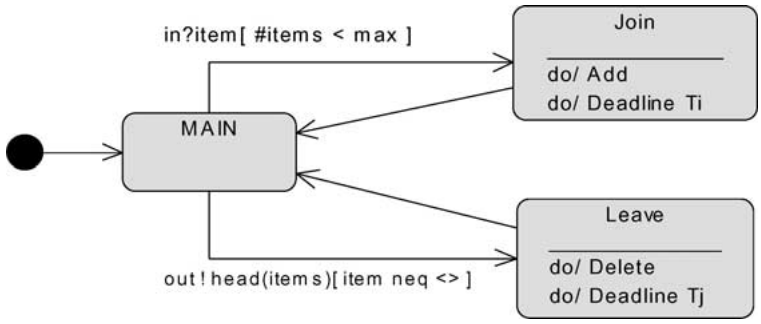


Figure 4. ActiveQueue statechart diagram.

Currently we are investigating the dynamic view transformation. The basic semantic links between TCOZ class and UML statecharts are:

- TCOZ operation names are corresponding to UML statechart states.
- TCOZ events and guards are corresponding to the triggers of the UML state transitions.

Based on these semantic links, a statechart diagram for the class *ActiveQueue* may be constructed as in figure 4.

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>
</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`

7. Conclusions

The first contribution of this paper 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 (Z/Object-Z/TCOZ) 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 [Nielsen *et al.* 1989], SOFL [Liu *et al.* 1998] etc.) – 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). Although we have some ideas on Object-Z behaviour 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. We are currently in contact with various UML tools vendors.

The third contribution of this paper 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 ZML (the Z family web environment and UML projections). We have found that the formal model is acted as a precise design document and has also provided clear guidelines to 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), one immediate future work is to develop a translation tool to map existing Z/Object-Z specifications in LaTeX to the ZML format. Perhaps a reverse tool is also necessary as long as LaTeX is not totally replaced by XML technology.

Acknowledgements

This work is supported by the academic research grants, *Integrated Formal Methods* (R-252-000-050-107) and *Adding Formality to UML* (R-252-000-076-112) from National University of Singapore.

We would also like to thank the numerous anonymous referees who have reviewed the manuscript and whose valuable comments have contributed to the clarification of many of the ideas presented in the paper.

Appendix. Auxiliary functions of ZML formal models

The following auxiliary functions capture 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:

$$
\begin{array}{|l}
expand_c : Classdef \rightarrowtail Classdef \\
\hline
\forall\, c : Classdef \bullet \\
\quad c.inherit = \varnothing \Rightarrow expand_c(c) = c \\
\quad c.inherit \neq \varnothing \Rightarrow \\
\qquad expand_c(c).name = c.name \\
\qquad expand_c(c).inherit = \varnothing \\
\qquad expand_c(c).state.decpart = \bigcup\{c_0 : classdef, t : Type \mid c_0.name = t.name \\
\qquad\quad \wedge\, t \in \mathrm{dom}\ c.inherit \bullet expand_c(rename(c_0, c.inherit(t))).state.decpart\} \\
\qquad\quad \cup\, c.state.decpart \\
\qquad expand_c(c).state.axpart = \bigcup\{c_0 : classdef, t : Type \mid c_0.name = t.name \\
\qquad\quad \wedge\, t \in \mathrm{dom}\ c.inherit \bullet expand_c(rename(c_0, c.inherit(t))).state.axpart\} \\
\qquad\quad \cup\, c.state.axpart \\
\qquad expand_c(c).init.axpart = \bigcup\{c_0 : classdef, t : Type \mid c_0.name = t.name \\
\qquad\quad \wedge\, t \in \mathrm{dom}\ c.inherit \bullet expand_c(rename(c_0, c.inherit(t))).init.axpart\} \\
\qquad\quad \cup\, c.init.axpart \\
\qquad expand_c(c).ops = \{opers : classify(\bigcup\{c_0 : classdef, t : Type \mid c_0.name \\
\qquad\quad = t.name \wedge t \in \mathrm{dom}\, c.inherit \bullet expand_c(rename(c_0, c.inherit(t))).ops\} \\
\qquad\quad \cup\, c.ops) \bullet merge(opers)\}
\end{array}
$$

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

$$
\begin{array}{|l}
rename : (Classdef \times (Name \rightarrow Name)) \rightarrow Classdef \\
\hline
\forall\, c : Classdef; l : Name \rightarrow Name \bullet \\
\quad \mathrm{dom}\, l \in (\mathrm{dom}\, c.state.decpart \cup \{op : c.ops \bullet op.name\}) \Rightarrow \\
\qquad l = \varnothing \Rightarrow rename(c, l) = c \\
\qquad l \neq \varnothing \Rightarrow \\
\qquad\quad rename(c, l).name = c.name \\
\qquad\quad rename(c, l).inherit = \{i : c.inherit \bullet (\mathrm{fst}(i), \\
\qquad\qquad \{(a, b) : snd(i) \bullet (a, match_1(b, l))\})\} \\
\qquad\quad rename(c, l).state.decpart = \{(na, dt) : c.state.decpart \bullet \\
\qquad\qquad (match_1(na, l), dt)\} \\
\qquad\quad rename(c, l).state.axpart = \{p : c.state.axpart \\
\qquad\qquad \bullet \{(n, pred) : p \bullet (n, match_2(pred, l))\}\}
\end{array}
$$

$$
\begin{aligned}
&rename(c, l).init.axpart = \{p : c.init.axpart \\
&\quad \bullet \{(n, pred) : p \bullet (n, match_2(pred, l))\}\} \\
&rename(c, l).ops = \{op_2 : Opdef \mid op_1 : c.ops \bullet \\
&\quad op_2.name = match_1(op_1.name, l) \\
&\quad op_2.detla = \{d : op_1.delta \bullet match_1(d, l)\} \\
&\quad op_2.axpart = \{p : op_1.axpart \bullet \{(n, pred) : p \\
&\quad\quad \bullet (n, match_2(pred, l))\}\}
\end{aligned}
$$

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:

$$
\begin{aligned}
&match_1 : (Name \times (Name \to Name)) \to Name \\
&\hline
&\forall\, old : Name; l : Name \to Name \bullet \\
&\quad old \in \operatorname{dom} l \Rightarrow match_1(old, l) = l(old) \\
&\quad old \notin \operatorname{dom} l \Rightarrow match_1(old, l) = old
\end{aligned}
$$

$$
\begin{aligned}
&match_2 : ((PredConstructor \cup \ \downarrow Type) \times (Name \to Name)) \\
&\quad \to (PredConstructor \cup \ \downarrow Type) \\
&\hline
&\forall\, old : (PredConstructor \cup \ \downarrow Type); l : Name \to Name \bullet \\
&\quad old \in PredConstructor \Rightarrow \\
&\qquad old.content \in \operatorname{dom} l \Rightarrow match_2(old, l).content \\
&\qquad\quad = l(old.content) \\
&\qquad old.content \notin \operatorname{dom} l \Rightarrow match_2(old, l).content \\
&\qquad\quad = old.content \\
&\quad old \in \ \downarrow Type \Rightarrow \\
&\qquad match_2(old, l) = old
\end{aligned}
$$

Function *classify* takes in a set of operation definition and divides them into subsets, which within each subset the name of the operation is the same:

$$
\begin{aligned}
&classify : \mathbb{P}\, Opdef \ \to \mathbb{P}(\mathbb{P}\, Opdef) \\
&\hline
&\forall\, (s, ss) : classify \bullet s = \bigcup ss \wedge \\
&\qquad \forall ops : ss \bullet \forall op_1, op_2 : ops \bullet op_1.name = op_2.name
\end{aligned}
$$

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

$$
\begin{aligned}
&merge : \mathbb{P}\, Opdef \to Opdef \\
&\hline
&\forall\, ops : \mathbb{P}\, Opdef \bullet \\
&\quad merge(ops).name \in \{op : ops \bullet op.name\} \\
&\quad merge(ops).delta = \bigcup \{op : ops \bullet op.delta\} \\
&\quad merge(ops).decpart = \bigcup \{op : ops \bullet op.decpart\} \\
&\quad merge(ops).axpart = \bigcup \{op : ops \bullet op.axpart\}
\end{aligned}
$$

The $expand_{z1}$ function expands a schema box definition according to the inclusion of other schemas, and outputs the expanded schema:

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

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

$$
\begin{array}{|l}
findlist : Schemadef_1 \rightarrow (Name \rightarrow Name) \\
\hline
\forall\, s : Schemadef_1 \bullet findlist(s) = \{decl : s.decpart \bullet (fst(decl), fst(decl)^\frown\langle\text{`}'\text{'}\rangle)\}
\end{array}
$$

The *predxi* function is used to get the inexplicit predicates for *xi* schema, that is its post-state unchanged:

$$
\begin{array}{|l}
predxi : (Name \rightarrow Name) \rightarrow (\mathbb{P}\, Pred) \\
\hline
\forall\, l : \operatorname{dom} predxi \bullet (\exists\, post, pre, eq : PredConstructor \bullet post.content = snd(l) \wedge \\
\quad eq.content = \langle\text{`}=\text{'}\rangle \wedge pre.content = \operatorname{fst}(l) \wedge predxi(l) = post^\frown eq^\frown pre)
\end{array}
$$

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

$$
\begin{array}{|ll}
expand_{z2} : Schemadef_2 \rightarrowtail Schemadef_1 & \\
\hline
\forall\, s : Schemadef_2 \bullet & \\
\quad expand_2(s).name = s.name & \text{[Name]} \\
\quad expand_2(s).incl = formIncl(s.calc) & \text{[Incl]} \\
\quad expand_2(s).decpart = formDecpart(s.calc) & \text{[Decpart]} \\
\quad expand_2(s).axpart = \{formAxpart(s.calc)\} & \text{[Axpart]}
\end{array}
$$

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

$$
\begin{array}{|l}
formIncl : PredCalc \rightarrow Inclusion \\
\hline
\forall\ p : PredCalc \bullet \\
\quad (p \in\ \downarrow Type) \Rightarrow \\
\qquad formIncl(p) = \{\exists_1\ s_1 : Schemadef_1 \mid s_1.name = p.name \bullet s_1.incl\} \\
\quad (p \in CalcConstructor) \Rightarrow \\
\qquad formIncl(p) = \bigcup\{p_i : p.items \bullet formIncl(p_i)\}
\end{array}
$$

$$
\begin{array}{|l}
formDecpart : PredCalc \rightarrow Decpart \\
\hline
\forall\ p : PredCalc \bullet \\
\quad (p \in\ \downarrow Type) \Rightarrow \\
\qquad formDecpart(p) = \{\exists_1\ s_1 : Schemadef_1 \mid s_1.name = p.name \bullet s_1.decpart\} \\
\quad (p \in CalcConstructor) \Rightarrow \\
\qquad formDecpart(p) = \bigcup\{p_i : p.items \bullet formDecpart(p_i)\}
\end{array}
$$

$$
\begin{array}{|l}
formAxpart : PredCalc \rightarrow Pred \\
\hline
\forall\ p : PredCalc \bullet \\
\quad (p \in\ \downarrow Type \Rightarrow \\
\qquad \exists_1\ s_1 : Schemadef_1 \bullet s_1.name = p.name \wedge \\
\qquad\quad formAxpart(p) = \mathrm{tail}(\frown/\{prd : s_1.axpart; op : PredConstructor \mid \\
\qquad\quad op.content = \langle \text{`} \wedge \text{'} \rangle \bullet op \frown prd\})) \\
\quad (p \in CalcConstructor \Rightarrow \\
\qquad formAxpart(p) = \mathrm{tail}(\frown/\{p_i : p.items; op, op_1, op_2 : PredConstructor \mid \\
\qquad op.content = p.op \wedge op_1.content = \langle \text{`(`} \rangle \wedge op_2.content = \langle \text{`)'} \rangle \bullet \\
\qquad op \frown op_1 \frown formAxpart(p_i) \frown op_2\}))
\end{array}
$$

References

Bowen, J.P. and D. Chippington (1998), "Z on the Web using Java," In *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users*, J.P. Bowen, A. Fett, and M.G. Hinchey, Eds., Lecture Notes in Computer Science, Vol. 1493, Springer-Verlag, Berlin, pp. 66–80.

Ciancarini, P., C. Mascolo, and F. Vitali (1998), "Visualizing Z Notation in HTML Documents," In *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users*, J.P. Bowen, A. Fett, and M.G. Hinchey, Eds., Lecture Notes in Computer Science, Vol. 1493, Springer-Verlag, Berlin, pp. 81–95.

Clark, J. (1999), "XT Version 19991105," `http://www.jclark.com/xml/xt.html`

Dong, J.S. and B. Mahony (1998), "Active Objects in TCOZ," In *The 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, J. Staples, M. Hinchey, and S. Liu, Eds., IEEE Computer Society Press, pp. 16–25.

Duke, R. and G. Rose (2000), *Formal Object Oriented Specification Using Object-Z*, Cornerstones of Computing, Macmillan.

Evans, A.S. and A.N. Clark (1998), "Foundations of the Unified Modeling Language," In *BCS-FACS Northern Formal Methods Workshop*, D.J. Duke and A.S. Evans, Eds., Electronic Workshops in Computing, Springer Verlag.

Futatsugi, K. and A. Nakagawa (1997), "An Overview of CAFE Specification Environment," In *The IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, M. Hinchey and S. Liu, Eds., IEEE Computer Society Press, Hiroshima, Japan, pp. 170–181.

Hayes, I., Ed. (1993), *Specification Case Studies*, International Series in Computer Science, 2nd ed., Prentice-Hall.

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

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

Liu, J., J.S. Dong, B. Mahony, and K. Shi (2000), "Linking UML with Integrated Formal Techniques," In *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, K. Siau and T. Halpin, Eds., Idea Group Publishing, pp. 210–223.

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

Mahony, B. and J.S. Dong (1999), "Sensors and Actuators in TCOZ," In *FM'99: World Congress on Formal Methods*, J. Wing, J. Woodcock, and J. Davies, Eds., Lecture Notes in Computer Science, Vol. 1709, Springer-Verlag, Toulouse, pp. 1166–1185.

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

Mahony, B.P. and J.S. Dong (1998), "Blending Object-Z and Timed CSP: An Introduction to TCOZ," In *The 20th International Conference on Software Engineering (ICSE'98)*, K. Futatsugi, R. Kemmerer, and K. Torii, Eds., IEEE Press, Kyoto, Japan, pp. 95–104.

Nielsen, M., K. Havelund, R. Wagner, and C. George (1989), "The RAISE Language, Method and Tools," *Formal Aspects of Computing 1*, 85–114.

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

Schneider, S. and J. Davies (1995), "A Brief History of Timed CSP," *Theoretical Computer Science 138*.

Smith, G. (1995), "A Fully Abstract Semantics of Classes for Object-Z," *Formal Aspects of Computing 7*, 3, 289–313.

Smith, G. (2000), *The Object-Z Specification Language*, Advances in Formal Methods, Kluwer Academic.

(W3C), W.W.W.C. (1999), "XSL Transformations (XSLT) Version 1.0," `http://www.w3.org/TR/xslt`

(W3C), W.W.W.C. (2000a), "Extensible Markup Language (XML)," `http://www.w3.org/XML`

(W3C), W.W.W.C. (2000b), "Extensible Stylesheet Language (XSL)," `http://www.w3.org/Style/XSL`

Wafula, E.N. and P.A. Swatman (1995), "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.

Woodcock, J. and J. Davies (1996), *Using Z: Specification, Refinement, and Proof*, Prentice-Hall International.