A XML/XSL Approach to Visualize and Animate TCOZ

Jing Sun, Jin Song Dong, Jing Liu and Hai Wang
Department of Computer Science
School of Computing
National University of Singapore
{sunjing,dongjs,liujing1,wanghai}@comp.nus.edu.sg

ABSTRACT

The challenge for system specification is how to visually and precisely capture static, dynamic and real-time system properties in a highly structured way. Timed Communicating Object-Z (TCOZ) is an integrated formal notation that build on Object-Z's strengths in modeling complex data and state, and on Timed CSP's strengths in modeling process control and real-time interactions. In this paper, we demonstrate approaches of using XML/XSL as a transformation tool to visualize TCOZ models into various UML diagrams and to animate TCOZ specifications with a multi-paradigm programming language - Oz.

Keywords

formal methods, TCOZ, XML/XSL, UML/XMI, specification animation, Oz

1. INTRODUCTION

Requirements capture is a key activity in software and system engineering. The challenge for complex system requirement specification is how to precisely capture static and dynamic system properties in a highly structured way. RAISE[10] as a new language has been developed and it integrates various formalisms, such as VDM, CSP, ML with algebraic modeling languages. This trend has been continued but more conservative and focused. The recent research focus is on combining of Z with event-based formalisms (many approaches reported at recent formal methods conferences, i.e. IFM'99 [1] and FM'99 [19]). Timed Communicating Object Z (TCOZ) [9] is one of these combinations. TCOZ builds on the strengths of Object-Z [2, 5] in modeling complex data and state with the strengths of Timed CSP [12] in modeling real-time concurrency.

In addition to the investigation of the integrated formal methods, it is also important to develop transformation tools (from those integrated formal models) to industry popular graphical notations and animation tools for validating the formal models. Unified Modeling Language (UML) [11] is commonly regarded as one of the dominate graphical notations for industrial software system modeling. It's important to develop links and tools from formal models to UML. Animation plays an important role of validating the consistency between the formal model and the real world requirements. If the formal specification does not reflect the real requirements it is useless to further pursue verification process. 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 requirements. Many approaches have been explored on animating Z using logic and functional programming languages, i.e., Prolog [18], Haskell [13] and so on. Obviously, the best candidates for animating IFM such as TCOZ are those multi-paradigm programming languages, such as Oz [7].

In this paper, we plan firstly to develop a projection tool for visualizing TCOZ specifications in UML diagrams. 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 from TCOZ specifications (in eXtensible Markup Language - XML [15]) to UML (in XMI) via eXtensible Stylesheet Language (XSL) [16] technology. Secondly, we will demonstrate the approach of animating TCOZ models in a multi-paradigm programming language - Oz. Oz is based on a concurrent constraint model and merges several directions of programming language designs 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. XSL is again used as a transformation tool for the code generation from TCOZ (in XML) to Oz.

The remainder of the paper is organized as follows. Section 2 briefly introduces the TCOZ notations. Section 3 presents the TCOZ to UML projection rules and the implementation of the transformation via XML/XSL. Section 4 presents the approach of animating TCOZ specification in Oz. Section 5 concludes the paper.

2. TCOZ FEATURES

Timed Communicating Object Z (TCOZ) [9] is essentially a blending of Object-Z [5] with Timed CSP [12], 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 [9]. The formal semantics of TCOZ is also documented [8].

2.1 Channels

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.

2.2 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 (non-terminating process) is used to determine the behaviour of active objects of a given class [4]. 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 behaviour of the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\mathrm{MAIN} \mid\mid\mid ob_2.\mathrm{MAIN}$

2.3 Semantics of TCOZ

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [8]. 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 hypothesised. 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 what and when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is if a ✓ event occurs), then the state at the time of termination is called the *final* state.

The process model of an operation schema consists of all initial states and update traces (terminated with a ✓) such that the initial state and the final state satisfy the relation described by the schema. If no legal final state exists for a given initial state, the operation diverges immediately. An advantage of this semantics is that it allows CSP process refinement to agree with Z operation refinement.

2.4 Network topologies

The syntactic structure of the CSP synchronisation operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology. For example, consider that processes A and B communicate privately through the interface ab, processes A and C communicate privately through the interface ac, and processes B and C communicate privately through the interface bc. This network topology of A, B and C may be described by

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

2.5 Two Communicating Buffers example

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

Buffer

$$\begin{array}{l} items : \mathrm{seq}\, MSG \\ left, right : \mathbf{chan} \end{array}$$

INIT

$$items = \langle\ \rangle$$

Add

$$\begin{array}{l} \Delta(items) \\ i? : MSG \\ \hline items' = \langle i? \rangle \frown items \end{array}$$

Remove

$$\begin{array}{l} \Delta(items) \\ \hline items = items' \frown \langle last(items) \rangle \end{array}$$

$$\begin{array}{l} Join \mathrel{\widehat=} [i : MSG \mid \#items < max] \bullet left?i \\ \qquad \to Add \\ Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet right!last(items) \\ \qquad \to Remove \\ \mathrm{MAIN} \mathrel{\widehat=} \mu\, B \bullet Join \,\Box\, Leave;\ B \end{array}$$

Two communicating buffers can be composed in TCOZ respectively as:

TwoBuffers

$$\begin{array}{l} l : Buffer[middle/right] \\ r : Buffer[middle/left] \end{array}$$

$$\mathrm{MAIN} \mathrel{\widehat=} \|(l \xleftrightarrow{middle} r)$$

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

3. TCOZ 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 relative 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's 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 TCOZ communication interface type – **chan**. Upon that, TCOZ sub-expressions can be used (as the same role as OCL) in the statechart diagrams and collaboration diagrams.
- Semantically, UML class diagrams are identified with the signatures of the TCOZ classes. The states of the UML statechart diagram are identified with the TCOZ processes (operations) and the state transition links are identified with TCOZ events/guards. The classifier roles and communications are identified with TCOZ classes and their interactions respectively.
- Effectively, UML diagrams can be seen as the view-point visual projections from a unified formal TCOZ model.

In this section, we will define a set of translation rules between TCOZ and UML and develop a transformation tool via XML/XSL. The *Buffer* example will be used to illustrate the approach.

3.1 Translation Rules

A TCOZ model and a UML model are translated to each other from three views: static view, interaction view and state machine view. Class diagrams are used to present the static view; collaboration diagrams are used to present the interaction view; statechart diagrams are used to present the behaviour view.

3.1.1 Static View

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

- *Class* Each class in TCOZ is translated to a class in UML class diagrams and vice versa. In 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, which 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 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 class.
- *Cardinality* In the aggregation relationship, the cardinality of variables of a certain type will be illustrated in UML class diagram as the multiplicity of the two roles in an aggregation. We can either use the cardinality number as the multiplicity of the corresponding role, or map it to UML default set $1..*$.

3.1.2 Interaction View

In a system, objects of different classes interact with each other. The general arrangement of these interactions are captured with 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 communication interface (**chan** for synchronized communications). The dynamic interactions of classifier roles in UML are illustrated as messages between them, and their property 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 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 TCOZ model communicate through synchronous interface – **chan**, the corresponding data flow direction is set according to the event definitions (from ! to ?).

3.1.3 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 has 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 TCOZ model are viewed as triggers which cause transition of states in the statechart. They match the definition of trigger and guard in UML statechart diagram.
- 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 of one operation consists of only one other operation and events/guards leading to it, instead of using the substate technique, the two operations are modeled as two separate states with events/guards (if any) as the triggers between them.

- If a transition is triggered by multi-events, such as $e1 \rightarrow e2 \rightarrow P$, we define a pseudostate as the junction state between events (as in Figure 1). The stereotype $\{junction\}$ is introduced into this psuedo-state, which does not have any value changes or operations inside. It is just a temporary state during the transitions.
- In the case that an operations calls other operations, the called operations serve as the substates of the calling one, and they together compose a composite state in the statechart.

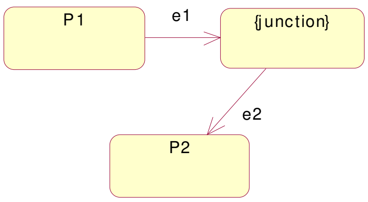

Figure 1: State transition

3.2 Implementation and Examples

XML Metadata Interchange (XMI) 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 first defining a customized XML syntax for TCOZ; then via XSL Transformations (XSLT) [14] technology, define an XSL file to capture all translation rules from TCOZ (in XML) to UML (in XMI). XT [3] 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 main process and techniques for visualizing TCOZ are depicted by the upper part of Figure 2. In the following sections, the *TwoBuffers* example will be used to facilitate the detailed discussion of the XML/XSL implementation approaches.

3.2.1 TCOZ in XML and transformation via XSL

Firstly, a customized XML document for TCOZ is defined according to the TCOZ syntax definitions. We use the recommendation from World Wide Web Consortium (W3C) - XML Schema [17] to define a structure syntax for the TCOZ 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="*"/>
  ...
</ElementType>
<ElementType name="classdef" content="eltOnly">
  <element type="name" minOccurs="1"
    maxOccurs="1" />
  ...
  <element type="inherit" minOccurs="0"
    maxOccurs="*" />
  ...
  <element type="state" minOccurs="0"
    maxOccurs="1" />
  <element type="init" minOccurs="0"
    maxOccurs="1" />
  <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* list, a number of declarations *decl* and some *predicate* definitions. Similarly, a *classdef* is mainly composed of an inheritance list *inherit*, a state schema *state*, an initializing schema *init* and a number of operation schemas *op* according to the TCOZ syntax.

The syntax definition of XMI for UML is specified as XMI 1.1 RTF UML DTD [6]. This DTD file defines all entities and XMI syntax signatures for UML. The XMI file for UML diagrams and the XML file for TCOZ have similar structures. 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 template technology plays a key role in implementing the translation rules. Let's consider the projection from TCOZ to the static view in UML, the class diagrams for instance. Classes and their relationships in TCOZ specifications (basically inheritance and aggregation) are captured according to the translation rules defined previously, which are more specified in detail for the implementation purpose as following:

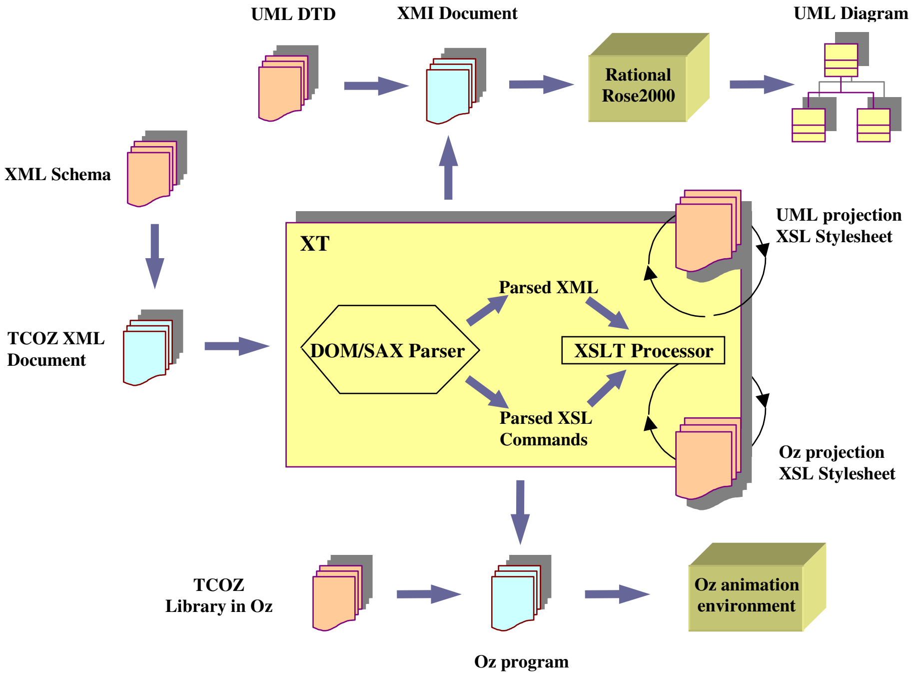

Figure 2: TCOZ - UML/Oz projection

- If a type value in the *Inherit* part of a class matches the name of any other class in current XML file, we regard that former class inherits the second one and illustrate the inheritance relationship between these two classes in our class diagram.
- If a type value in the *state/decl* part, that is , the type of an attribute, matches the name of any class in current XML 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.

The Aggregation and Inheritance relationships are specified as definition part and reference part in the XSL file. The difficult step here is how to sort out their IDs and locate the references with definitions.

For aggregation references, we should differ the aggregated class(whole) from a constituent part. The following is a simplified structure of the XSL code for it.

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

In the definitions of aggregation relationship, two most important things are to give the multiplicity of aggregation roles and to match the roles with classes. The concept of multiplicity in UML matches that of cardinality in TCOZ in a sense, so we specify it by calculating the cardinality of aggregate attributes, that is, the objects of the constituent class. Differing the role of aggregate class from the role of the constituent class takes the similar approach as we do in specifying the references above.

The approach to deal with the inheritance relationship is quite similar to how we deal with the aggregation relationship. The difference lies in the two aspects: first, there is only one entity of *Generalization* but no 'roles', though we should differ *supertype* from *subtype* for the two related classes; second, the matching pattern is different as:

```
<xsl:for-each select="//inherit/type[node()=
//classdef/name]">
```

3.2.2 Projection Case study – Two Communicating Buffers example

The following is part of the XML format for the *TwoBuffers* example in the previous section.

```
<classdef layout="simpl" align="left">
  <name>Buffer</name>
  <state>
    <decl>
      <name>items</name>
      <dtype>&seq;<type>MSG</type></dtype>
```

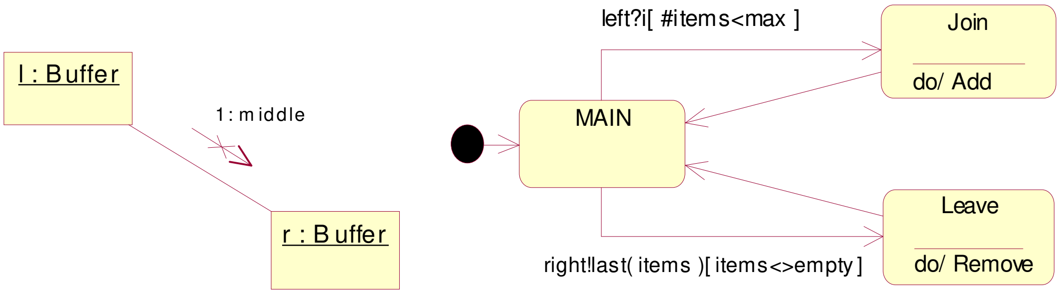

Figure 3: Projection to UML collaboration and statechart diagrams

```
    </decl>
    ...
  </state>
  <init>
    <predicate>items=&emptyseq;</predicate>
  </init>
  <op layout="simpl">
    <name>Add</name>
    ...
  </op>
  ...
</classdef>
<classdef layout="simpl" align="left">
  <name>TwoBuffers</name>
  <state>
    <decl>
      <name>l</name>
      <dtype>
        <type>Buffer</type>[middle/right]
      </dtype>
    </decl>
    ...
  </state>
  ...
</classdef>
```

As in Figure 4, the UML class diagram depicts the static view of the two classes constructed from the *TwoBuffers* example. Note that this diagram was generated automatically from the XML specification via our XSL transformation.

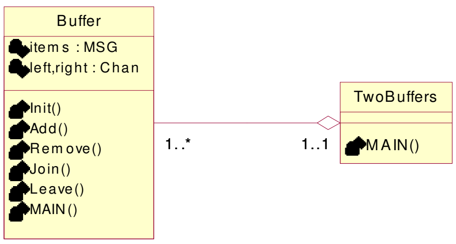

Figure 4: Generated UML class diagram

The relationship between *TwoBuffers* and *Buffer* is aggregation. The generated 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.10010'/>
        <!--  TwoBuffers   -->
      </type>
    </AssociationEnd>
    <AssociationEnd xmi.id='G.4'>
      <name />
      <multiplicity>1..*</multiplicity>
      <type>
        <xmi.idref='S.10001' />
        <!--  Buffer   -->
      </type>
    </AssociationEnd>
  </connection>
</Association>
```

From above, we demonstrated a XML/XSL approach for visualizing TCOZ models in UML diagrams. For the explanation purpose we mainly focused the process on UML class diagrams. The projections to other UML diagrams such as collaboration and statechart diagrams can be achieved in a similar manner (see Figure 3) according to the translation rules defined earlier.

The documentation about TCOZ to UML transformation and downloadable codes are available at:

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

4. ANIMATING TCOZ IN OZ

4.1 Specification validation

Animation plays an important role of validating the consistency between the formal model and the real world informal requirements. Besides the correctness of formal specification itself there still lie 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 process of verifying the consistency between the formal model and real world model could never be formalized. Animation is an engineering process that brings one step closer to this goal. It allows the system analysts to explore the behavior of the formal model 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. Animation acts a vital part in the early stage of formal modeling.

4.2 Nature of animation

Animation is focused on the abstraction of logic relationships within the required system. Programs are collections of detailed instructions to a computer. Implementation is a transformation of taking a specification to produce a program (perhaps using 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 non-functional requirement eased. Animation is a mapping version of the specification that 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 apparent to the specification. The key difference between animation and implementation lies in two aspects as below.

- Data types – Data type provides information about the possible values that a variable could take. Variable declaration such as '$var\ x : N$' is in equivalence to '$var\ x \ \wedge\ x \in N$'. Thus type information is really a membership relation between the variable and its type set. Data types inside an animation need not to be actual data sets that same as those within an implementation since the primary purpose of an animation is to explore the consequences of a specification, rather than produce a final implementation of the system or even a full scale prototype that is capable of handling realistically-sized data sets. It could be a virtual data set or even a subset as long as the type information would be demonstrated. In this way the focus is concentrated on the logic relationships and the behaviors of the specification.

- Logical equivalence – Animation should ensure that each animated operation is equivalent to its corresponding specification, rather than a refinement. 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 keen to the implementation process but not adoptable in the animation stage. The translation from formal model into animation language should be kept as equivalent as possible to its original specification.

4.3 Animation language - Oz a candidate for TCOZ

Generally speaking, any programming language could be used for animation. However, every 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 (search strategies), PowerBuilder is good at database applications and so on. 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 the first criteria of selection, i.e., animating Z using Prolog [18]. Since most animation languages have differences to the formal specification notations. One solution is to provide an equivalent library which handle all those specification constructs. The completeness of the exiting library compare to the formal notation could be the second measure for the selection. The running properties of the evaluator such as efficiency, termination and so on would be another criterion of choosing a desired animation language. Thus select a programming language that has a high logic abstraction level, contains most of the features that common to the specification notation, and along with the help of properly designed library functions animation could be fulfilled more easily and directly.

The programming language Oz [7] is a multi-paradigm language based on the concurrent constraint model. It merges several directions of programming language designs such as Object Orientation, constraint and logic programming, functional programming and concurrent programming into a single coherent design. 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. Object orientation in Object-Z, currency in CSP and the mixture of the two in TCOZ all could find a majority of their corresponding features in Oz. With a help of proper library functions and logic programming figures in Oz, integrated formal notations such as TCOZ could be well animated in this kind of multi-paradigm language.

4.4 Translation rules – An executable interpretation for TCOZ in Oz

To provide a translation guideline from TCOZ to Oz is the same as offering a runnable semantics of TCOZ in Oz. Some rules are defined as follow.

- Data types are referred to given sets ('List') since Oz is a dynamic typed language. This is because each data type is basically a set of possible values the variable can have, for the purpose of animation these sets could be much smaller;

- Sequence is referred to 'List' data type in Oz, set and its corresponding functions are referred to the library functions;

- TCOZ class is referred to Oz class with inheritance expanded since TCOZ class has different inheritance rules;

- Type and function definitions local to a class is referred to local declarations to an Oz class;

- The type declaration of the state schema in TCOZ class is referred to as 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 instance of the class, declaration in the feature via an anonymous variable '_', which all instances of the class will share the same variable, in our case the common referred object;
- Operations that are not in the visible list of the TCOZ class are referred to methods labelled by variables instead of literals in Oz class, which are private to the class;
- Generic class definition is referred to function definition with type information as its parameter and returns a Oz class declaration;
- Channel is treated as features of cell type in Oz class, which later can be assigned in the system specification according to the network topology;
- Active object class is referred to an Oz class that inherit the Oz 'Time.repeat' class, which is capable of setting up an action method (main) for repeatedly running.

4.5 Implementation and case study

4.5.1 TCOZ Oz library

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

```
% set
fun {SubSet A B}
   case A
   of nil then true
   [] H|T then {And {Member H B} {SubSet T B}}
   end
end
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. Note that these functions are implemented using the logic programming aspects in Oz, which will preserve the same abstraction level with the specification notation. We have completed the entire TCOZ set operations in Oz, and only a few was demonstrated in the paper due to the space limitation.

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. Here we programmed a signaling mechanism for producers and consumers. 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 notifying 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.

4.5.2 TCOZ Oz projection

To animate TCOZ specifications in Oz, we first use XSL Transformation to project TCOZ model into Oz code frames, together with test cases and auxiliary library to perform the validation. Note that customized codes segments are needed during the process. The main procedure for animating TCOZ are depicted by the lower part of Figure 2. The following is part of the XSL stylesheet for Oz projection.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<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:text>
    </xsl:text>
    <xsl:apply-templates select="state"/>
    <xsl:apply-templates select="init"/>
    <xsl:apply-templates select="op"/>
    <xsl:text>
end </xsl:text>
</xsl:template>
```

From the above, it states that a projection will be made on each defined TCOZ class in XML to construct their 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.

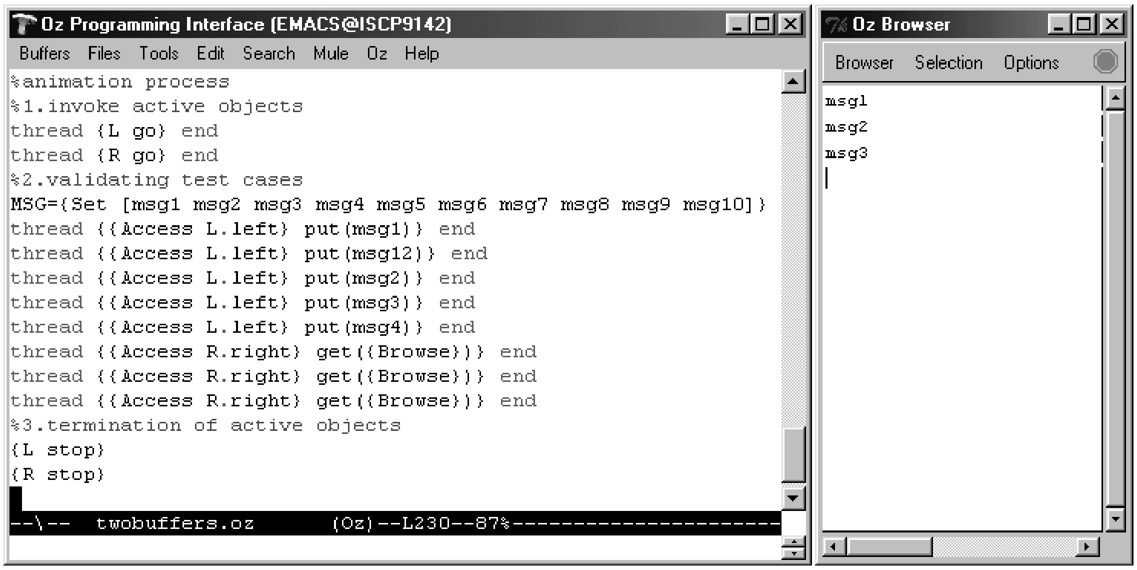

Figure 5: Animation of the Two Communicating Buffers example

4.5.3 Two Communicating Buffers example

Consider the *Buffer* example in the previous section, its translated specifications in Oz is as follow.

```
%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 skip
    end
  end
  ...
  meth main
    ...
  end
end
```

Note that the preconditions in the TCOZ schema is treated as logical conditional statements *cond* in Oz. The '*else skip*' statement is introduced for the executing purpose only. Without the statement, the process will hang when the preconditions are not satisfied. A *cond* statement has the following semantics. Assume a thread is executing the statement in space SP.

- The thread is blocked.
- A space SP_1 is created, with a single thread executing the guard *cond* $X1$ ... XN in $S0$.
- Execution of the father thread remains blocked until SP_1 is either entailed or disentailed. Notice that these conditions may never occur, e.g. when some thread is suspending or running forever in SP_1.
- If SP_1 is disentailed, the father thread continues with $S2$.
- 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 father thread continues with the execution of $S1$.

The *TwoBuffers* example depicted by TCOZ network topology can be translated into the follow 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 corresponding feature variables in the *Buffer* definition according to the network topology of the TCOZ specification. The function *setRepAll* is to set up a repeat action for the TCOZ active objects.

After fulfilled the translation from TCOZ specification into Oz, it's time to build up test cases and carry out the validating process. As see from Figure 5, we firstly invoked the two active objects and let them running concurrently in their own threads. Then, five inputs along the *left* channel of the *TwoBuffers* was 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 effectively. 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 and as well as the user requirements.

5. CONCLUSION

The first contribution of this paper is the investigation of the semantic links between TCOZ (in XML) with UML diagrams (in XMI). In our approach, UML diagrams are visual projections from a formal TCOZ model, therefore they are consistent with the formal model. Although we have some guidelines on TCOZ behaviour projections to statecharts, the development of the environment for systematic transformation from TCOZ to statechart 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 second contribution of this paper 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, i.e., OO, logic and concurrency, we defined TCOZ constructs library so that animating TCOZ model in Oz can be easily and effectively achieved. We also constructed a XSLT stylesheet for the automatic transformation from TCOZ specification into Oz code frames. 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. This will be part of our future works.

Although our project is in the initial stage, the essential ideas and techniques for visualizing and animating TCOZ models through XML as a common medium have been demonstrated in this paper. We also plan to build 'heave' tools support such as model checker (perhaps to build projection tools to FDR and Alloy) and reason tools (perhaps to encode TCOZ semantics into Isabel and PVS).

6. ACKNOWLEDGMENTS

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.

7. REFERENCES

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

[2] 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.

[3] James Clark. Xt version 19991105. http://www.jclark.com/xml/xt.html, 1999.

[4] 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.

[5] 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.

[6] Object Management Group. Uml resource page. http://www.omg.org/uml/.

[7] Mozart Research Groups. The mozart programming system. http://www.mozart-oz.org/, 2000.

[8] B. Mahony and J.S. Dong. Overview of the semantics of TCOZ. In Araki et al. [1], pages 66–85.

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

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

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

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

[13] Mark 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.

[14] World Wide Web Consortium (W3C). Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 1999.

[15] World Wide Web Consortium (W3C). Extensible markup language (xml). http://www.w3.org/XML, 2000.

[16] World Wide Web Consortium (W3C). Extensible stylesheet language (xsl). http://www.w3.org/Style/XSL, 2000.

[17] World Wide Web Consortium (W3C). Xml schema. http://www.w3.org/XML/Schema, 2000.

[18] Margaret M. West and Barry M. Eaglestone. Software development: two approaches to animation of z specifications using prolog. *Software Engineering Journal*, 7(4):264 – 276, July 1992.

[19] J. Wing, J. Woodcock, and J. Davies, editors. *FM'99: World Congress on Formal Methods*, volume 1709 of *Lect. Notes in Comput. Sci.*, Toulouse, France, September 1999. Springer-Verlag.