XML-Based Static Type Checking and Dynamic Visualization for TCOZ

Jin Song Dong, Yuan Fang Li, Jing Sun, Jun Sun, and Hai Wang

School of Computing,
National University of Singapore
{dongjs,liyuanfa,sunjing,sunjun,wangh}@comp.nus.edu.sg

Abstract. Timed Communicating Object Z(TCOZ) combines Object-Z's strengths in modelling complex data and state with TCSP's strengths in modeling real-time concurrency. Based on our previous work on the XML environment for TCOZ, this paper firstly demonstrates the development of a type checker for detecting static semantic errors of the TCOZ specification, then illustrates a transformation tool to automatically project TCOZ models into UML statechart diagrams for visualising the dynamic system behaviour.

Keywords: TCOZ tool support, XML/XSL, UML/XMI.

1 Introduction

The main stimulus for the inception of formal specification techniques is to precisely describe software and system requirements so that tools can be applied to perform checking and analysis on the formal requirement models. Various formal notations are often combined for modelling large and complex systems which may have intricate system states and complex concurrent and real-time behavior. Timed Communicating Object Z (TCOZ) [8] builds on the strengths of Object-Z [4, 12] in modeling complex data and state with the strengths of TCSP [10] in modeling process control and real-time interactions. Our previous works on ZML (Z [15] family on the Web through XML) [14, 13] have been focusing on displaying formal specifications on the web and projecting TCOZ models to UML class diagrams. This paper reports on the developments of a type checking and UML statechart visualization tools for TCOZ.

There have been previous works on type checking Z and Object-Z. For example, *Wizard* [5] is a LaTeX-based type checker for Object-Z. Our type checking tool aims to check TCOZ (including Z and Object-Z) specifications with XML as an input format.

UML can be used for visualizing formal specification models. For the purpose of visualizing the static properties of TCOZ specification, we have previously developed a transformation tool from TCOZ to UML class diagram [13]. The second part of this paper aims to develop the techniques and tools for visualizing TCOZ behaviour specifications (mainly TCSP) by transforming TCOZ models into UML statechart diagrams. Brooke and Paige [3] have recently developed a tool-supported graphical notation for TCSP. The difference between Brooke and Paige's approach and ours is that we use existing graphical notations instead of creating new ones.

C. George and H. Miao (Eds.): ICFEM 2002, LNCS 2495, pp. 311–322, 2002.

The remainder of the paper is organized as follows. Section 2 briefly introduces the technical background: the TCOZ notation and XML/XMI. Section 3 provides the overall design of the type checker and outlines type hierarchy and typing rules. Section 4 develops the proper projection rules for transforming TCOZ models to UML statecharts and illustrates the development of the automatic projection tools using JAVA. Section 5 presents a case study of the project, showing the working and output of the type checker and visualization in UML statechart diagrams. Section 6 concludes the paper and comments on possible future work directions.

2 Technical Background

2.1 TCOZ

Timed Communicating Object Z (TCOZ) builds on the strengths of Object-Z and TCSP. The syntactic structure of TCOZ is similar to Object-Z. A TCOZ 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 process definitions.
In the remainder of this subsection, some important features of TCOZ are briefly introduced. A detailed introduction can be found elsewhere [8]. The formal semantics of TCOZ is also documented in [6].

Active Object
In TCOZ, active objects have their own thread of control, while passive objects are controlled by other objects in a system. An identifier *MAIN* is used to represent the behavior of active objects of a given class. 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.

Interface: Channels, sensors and actuators
Channel is one of the most important concepts in CSP and it is given an independent, first class role in TCOZ. The class state-schema convention (mechanism in Object-Z) 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 the type of **chan**, for example in, out in the above example. One thing special about channel is that channels are type heterogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state entities, channels are viewed as shared rather than as encapsulated entities, that is, channels are commonly used to carry information between TCOZ classes.
Complementary to the synchronizing CSP channel mechanism, TCOZ also adopts a non-synchronizing shared variable mechanism [7]. A declaration of the form $s : X$ *sensor* provides a channel-like interface for using the shared variable s as an input. A declaration of the form $s : X$ *actuator* provides a local-variable-like interface for using the shared variable s as an output.

Network Topology
The syntactic structure of the CSP synchronization operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication

topologies generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology. For example, consider that processes *A* and *B* communicate privately through the interface *ab*, processes *A* and *C* communicate privately through the interface *ac*, and processes *B* and *C* communicate privately through the interface *bc*.
This network topology of *A*, *B* and *C* may be described by

$$\Big|\Big|(A \overset{ab}{\longleftrightarrow} B;\ B \overset{bc}{\longleftrightarrow} C;\ C \overset{ca}{\longleftrightarrow} A).$$

2.2 XML and XMI

Our previous work [14] has used XML and XML schema to define a standard exchange format for Z-family languages (Z, Object-Z and TCOZ). An XML Schema file was created for describing the structure of the Z-family languages in XML. It defines the contents of all elements, the order and cardinality of sub-elements, and data types of some of the elements.
XMI (XML Metadata Interchange) is an industry standard for storing and sharing object programming and design information, allowing developers of distributed systems to share object models and other metadata over the Internet. Three key industry standards, XML (eXtensible Markup Language), UML (Unified Modelling Language) and MOF (Meta Object Facility), are integrated in XMI. XMI marries the OMG and W3C metadata and modelling technologies [1]. Rational Rose 2001 from OMG which supports XMI can generate UML diagrams once it imports XMI documents, and it can also export XMI documents for any existing UML diagrams as well. This is very useful for our work since we only need to generate the proper XMI from a TCOZ specification in XML format. The syntax definition of XMI for UML is specified in XMI 1.1 RTF UML DTD [1]. This DTD file defines all entities and XMI syntax signatures for UML. An XMI file validated by UML.DTD version 1.3 has the following structure:

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version='1.1' xmlns:UML='//org.omg/UML/1.3'>
    <XMI.header> ... </XMI.header>
    <XMI.content>
        <UML.Model>
           <UML.StateMachine> ... </UML.StateMachine>
        </UML.Model>
        <UML:Diagram> ... </UML:Diagram>
    </XMI.content\>
</XMI>
```

XMI.header contains general information like the UML.DTD version. UML StateMachine is the most important part of *UML.content*, which contains information about Statechart. *UML* : *Diagram* is used to display the UML diagrams. It contains the exact position of every displayable unit in the UML diagram.

3 Type Checker Design and Typing Rules

3.1 High-Level Design

In order to parse the TCOZ languages in XML format, we need to parse the entire XML file. A compiler approach was taken. A handcrafted[1] front end of a complete compiler was written, which includes modules like the scanner, the symbol table, the expression parser, the predicate parser and other miscellaneous utility modules. This type checker has a handcrafted front end of a compiler. A top-down, or recursive descent approach is taken for two reasons. The first reason is that both DOM and SAX parsers parse an XML file from the root element, which is the top element, down to the bottom elements. The other reason is simplicity; the recursive descent approach is easier to understand and to build. The class diagram of the project is illustrated in Figure 1.

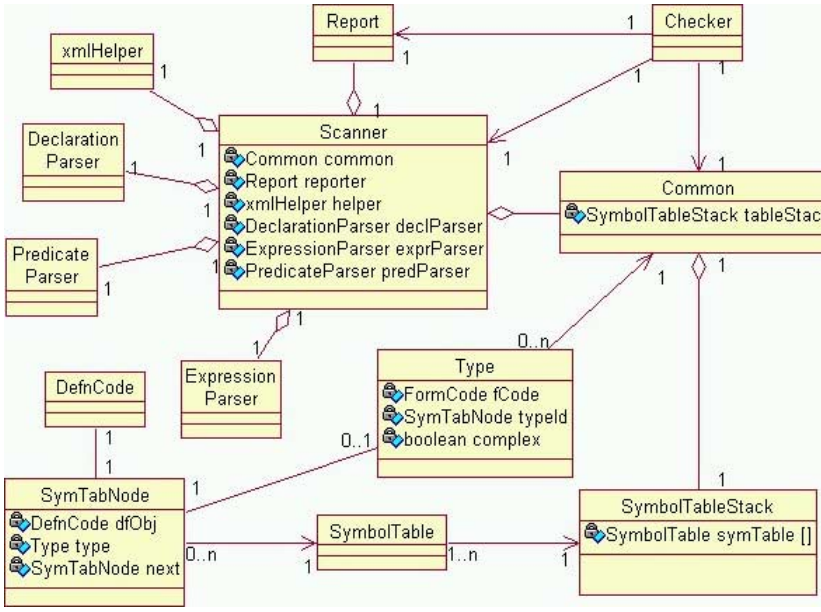


Fig. 1. Class diagram of the project

The two major functionalities of the type checker are to check syntax errors and to check static semantic errors in the TCOZ specifications in XML format. In the normal case, the program works as a 2-pass parser. When a forward declaration is present and recognized by the type checker, the XML document will be parsed again. At startup, the program takes a list of XML files as parameters. For each file, it calls the Xerces XML parser to

[1] Except for the Xerces XML Parser, no other parsing packages, utility functions such as Lex, YACC or JavaCC are used.

parse it and look for syntax errors. If there is any syntax error, the parser flags it then skips the file. If there is no syntax error, the program parses the file again, traversing through the structure to check for type errors. Upon encountering a type error, the program flags it, re-synchronizes itself and continues, until it finishes parsing all the files in the list. The TCOZ type checker has been designed in a way to support modularity and reusability. It is organized into Java packages.

3.2 Type Hierarchy and Rules

Figure 2 represents the hierarchy of types defined in the project. A utility class **FormCode** is also constructed. It has a set of constant definitions, each with a unique integer value identifying one of the types. The hierarchy consists of 10 types and their relationships as discussed below.

Type is the super class in the hierarchy and all other types inherite from **Type**, such as **EnumType**, **RecordType**, **RelationType**, and so on. **ClassType** defines a class as a type with inheritances.

Smith has developed type rules for Object-Z classes [11]. For example, given a generic state definition of class $A[X_1, ..., X_n]$, the state schema rule can be defined as follows:

$$[d_1, \Delta d_2 \mid p]$$

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

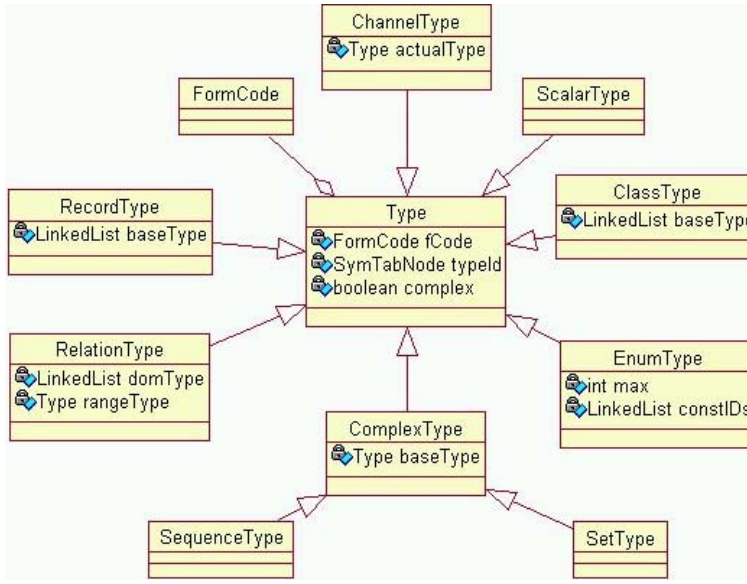
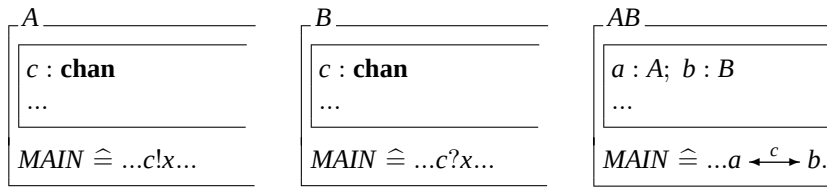


Fig. 2. Class hierarchy of the package TYPING

STATE refers to the state definition of a class, $\uparrow$ *STATE* stands for the inherited state definitions from its super classes, and the proviso q is in the form of $q \equiv b = (\!| X_1 \rightsquigarrow t_1, ..., X_n \rightsquigarrow t_n |\!)$ where t_i is the actual parameter substituted to X_i through substitution operator $\odot$. Other type rules for operation schema and inheritance can be found in [11]. We extend Smith's work with extra type rules for additional TCOZ constructs. For example, the **ChannelType** inherits **Type**. Variables declared of **ChannelType** are used for inter-process communications. There are three kinds of channels: channel, sensor and actuator. A simple synchronized communication (Channel) typing rule for a generic network topology definition of classes *A*, *B* and *AB*, can be defined as follows:



$$\frac{A[t_1, ..., t_n] :: STATE \vdash c \in \textbf{chan} \wedge MAIN \vdash c.x \in X \quad B[t_1, ..., t_n] :: STATE \vdash c \in \textbf{chan} \quad AB[t_1, ..., t_n] :: STATE \vdash a \in A \wedge b \in B \wedge MAIN \vdash a \overset{c}{\longleftrightarrow} b}{B[t_1, ..., t_n] :: MAIN \vdash c.x \in X} [\, q \,]$$

The above states that if class *A* and *B* are communicating through channel *c*, synchronization will be enforced on the input and outputs, i.e., outputs from *A* through *c* will lead to inputs to *B*. The typing rules for the asynchronous communication (sensor/actuators) can be similarly developed.

RecordType, SetType and **SequenceType** can have no names associated with them; in other words, they can be anonymous types. The special scalar type **dummyType** is used in two ways. Firstly, it is used to signal type errors when parsing predicates, expressions and declarations. If the typing is correct, **boolType** *boolean* (for predicate) or respective data type (for expression and declaration) is returned; or else **dummyType** is returned. Secondly, it is used as the base type for empty sets or sequences, since an empty set can be subset of a set of any type.

Now we have finished the discussion on the development of a type checker for detecting static semantic errors for TCOZ specifications in XML format. Next we will discuss the development of an automatic tool for transforming TCOZ dynamic behaviour models to UML statecharts via XMI.

4 TCOZ to UML Statechart Projection

As a requirement specification of software systems, TCOZ models are precise and elegant but difficult to read and interpret by software engineers without relevant mathematical background. In comparison, the most popular graphical notation, UML, is much easier to understand and widely accepted by the industry. Our key idea for using UML statechart to visualize TCOZ is:

– States of UML Statechart are identified with TCOZ processes (operations) and the state transition links are identified with TCOZ events /guards.

In TCOZ, behavior of a class is specified by the operations as processes. Figure 3 shows the detailed transformation rules from TCOZ behaviour models to UML statecharts.

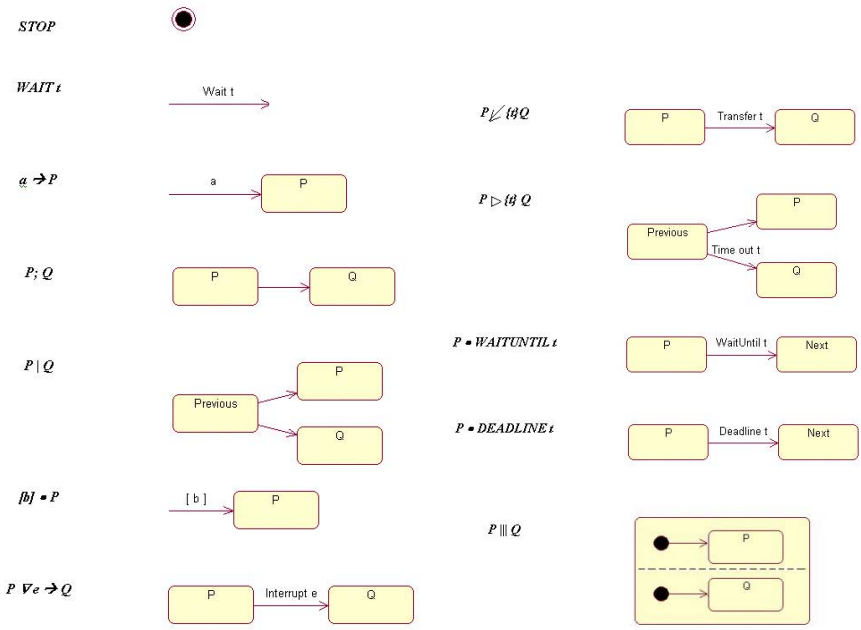


Fig. 3. UML projection rules

The projection rules for automatically translating TCOZ models (in XML) to UML Statecharts (in XMI) is implemented by a JAVA application. The following discusses the algorithm and implementation steps.

Step One: Preparation

At this stage, the XML file is read in and parsed class-by-class, operation-by-operation. A tag named *processexpr* is associated with each operation, which identifies the computational logic of the operation. The *processexpr* follows the grammar defined for TCOZ operation expressions. *processexpr* is divided into 13 types [2]. The activities preformed by the preprocessor are:

– Build up the operation table for each class.

[2] For detail information about fully annotated operation expressions, please refer to http://nt-appn.comp.nus.edu.sg/fm/zml/zml.xsd.

- Associate each class with its corresponding super class. One class may have more than one super class and it may invoke operations defined in different super classes.
- Build up the variable table for each class.
- For each operation, identify its *processexpr*. Check whether the operation invokes other operations. If not, mark this operation as a simple operation and generate the proper string representation of this operation. Otherwise, identify the type of the *processexpr*. For each type of *processexpr*, gather the important information for that type. For example, if the type is *networktopology*, identify what are the active objects and what are communication channels. For *processexpr* contained in *processexpr*, do the same recursively.

Step Two: Generation

For each active object, a new XMI file is created with the necessary header information. A top level composite state named 'op' is added to the Statemachine. An initial state (pseudostate) is added to the top-level composite state. A MAIN operation matches to a main state in the Statechart, which is the first state besides the initial state. Starting from the main operation, we syntactically analyze the *processexpr* based on the type information and generate proper states for each operation.
One challenge here is that at some point we may not know which projection rules could be used. For example, if some other operation is invoked by MAIN, shall we model the called operation as a simple state or a composite state? (At this point, we may not be able to find out whether the called operation will consequently invoke other operations.) One simple solution is to model all called operations as composite states and later replace those unnecessary composite states by simple states.

Step Three: Simplification

After the a complete Statechart is generated, the simplification process involves:

- Remove unnecessary simple states. "Unnecessary simple states" means state that are temporarily added into the statechart.
- Remove trivial composite states. "trivial composite state" means composite states that have one or even no substates.

Step Four: Layout

At this stage, we need to calculate the exact positions of all the states, transitions and events/guards in a diagram. The following formulas are used to calculate the width and height of a composite state. Given W the width, H the height, M the number of simple states in the composite state, N the number of composite states in the composite state. *WSimple* is the default width of any simple state. *HSimple* is the default height of any simple state. $W_1, \cdots W_N$ are width for each composite state in this composite state. $H_1, \cdots, H_N$ is the height for each composite state in this composite state. S is the default horizontal space between states. K is the default vertical space between states. P is the width (or height) of any pseudostate and Q is the width (or height) of any final state.

$W = max\{(\sqrt{M}+1) * (WSimple + S), W_1, W_2, \cdots, W_N\} + 4S + P + Q$
$H = (\sqrt{M}+1) * (HSimple + K) + (H_1 + H_2 + \cdots + H_N) + N * K$
Note that the calculation is done in a bottom-up manner since the size of the outer composite state depends on the size of the inner one. Once we know the width and height, we place simple states at the top ($\sqrt{M}$ simple states per row) and composite states at the bottom (one per row).

5 Case Study: Light Control System

In this section, we firstly present a TCOZ LCS (Light Control System)[3] model. Then we use this model to test our type checker and transformation tool to UML statechart. The LCS system composes of *RoomController* and *RoomDevices*. *RoomController* controls the whole system. *RoomDevices* consists of lights and motion detectors. The TCOZ specification for LCS is given as follows. *Illmination* is an abstract type, *Percent* is defined as $Percent == \{0\} \cup 10..100$

Light

$dim : Percent$ **actuator**; $on : \mathbb{B}$

$TurningOn \mathrel{\widehat=} dim := 100;\ on := true$
$TurningOff \mathrel{\widehat=} dim := 0;\ on := false$

The *Light* class has two operations, *TurningOn* and *TurningOff*.

ControlledLight
Light

$button, dimmer$: **chan**

$ButtonPushing \mathrel{\widehat=} button?1 \rightarrow ([dim > 0] \bullet TurningOff \ \Box$
$[dim = 0] \bullet TurningOn)$
$DimChange \mathrel{\widehat=} [n : Percent] \bullet dimmer?n \rightarrow ([on] \bullet dim := n \ \Box$
$[\neg\, on] \bullet \textsc{Skip})$
$\textsc{Main} \mathrel{\widehat=} \mu N \bullet (ButtonPushing \ \Box \ DimChange);\ N$

The *ControlledLight* is a subclass of *Light*. Two extra operations are defined: *ButtonPushing* and *DimChange*. Any occupant can manually turn on or turn off the light using *ButtonPushing* or the system will automatically adjust the illumination using *DimChange*. For each light, a CSP channel *button* is defined to capture the status of the button. The other channel *dimmer* is used to communicate with the system controller.

[3] LCS is an intelligent embedded control system. It can detect the occupation of the building, then turn on or turn off the lights automatically. It is able to adjust illumination in the building according to the outside light level. The full specification model can be found at: http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/light.xml

The *RoomContrller* controls both *MotionDetector* (for detecting any movement in the room) and *ControlledLight* by sending proper signals on channel *motion* and *dimmer*. (The *MotionDetector* definition is omitted due to space limitation.)

RoomController

$dimmer, motion : \textbf{chan}$
$odsensor : Illumination\ \textbf{sensor}$
$absenT : \mathbb{T}$
$olight : Illumination$

… [behaviour part is omitted]

5.1 Static Type Checking

Object-Z related type errors can be detected in a similar way as Wizard. The following example illustrates that the type checker spots TCOZ channel-related error and reports it. In the MAIN operation of class *LCS*, processes *m* and *r* (belonging to classes *MotionController* and *RoomController* respectively) communicate via a common channel called *motion*; *r* and *l* (belonging to *ControlledLight*) also communicate via another common channel called *dimmer*. Assuming that in class *RoomController*, the channel *motion* is renamed to *movement* and there is no other change to the specification:

RoomController

$dimmer : \textbf{chan}$
$movement : \textbf{chan}$ [Changed!]
…

…

As a result, the processes *m* and *r* cannot communicate any more since they no longer share a common channel. This is captured by the type checker as follows.

```
Error!  LCS.xml: 305: Identifier not found Symbol: motion
Location:  SymTabNode: RoomController   >>
    Class Definition [Class Type: RoomController]

Error!  LCS.xml: 305: No common channel defined!
Symbol: motion Location: SymTabNode: Main >> Class Operation
    OF  SymTabNode: LCS >> Class Definition [Class Type: LCS]

LCS.xml parsed 2 times.

2 errors.
```

5.2 Dynamic UML Visualizing

In this subsection we will show how to apply projection techniques and the tool to generate a proper UML statechart from the LCS formal model. For *controlledLight*, we start from the MAIN operation since MAIN is the state that starting state leads to. By mapping the TCOZ notations to a Statechart diagram, a rough statechart is generated as shown in the left part of figure 4. After that, we apply projection rules to operation *DimChange* and *ButtonPushing* and get the final statechart as shown in the righthand side of figure 4.

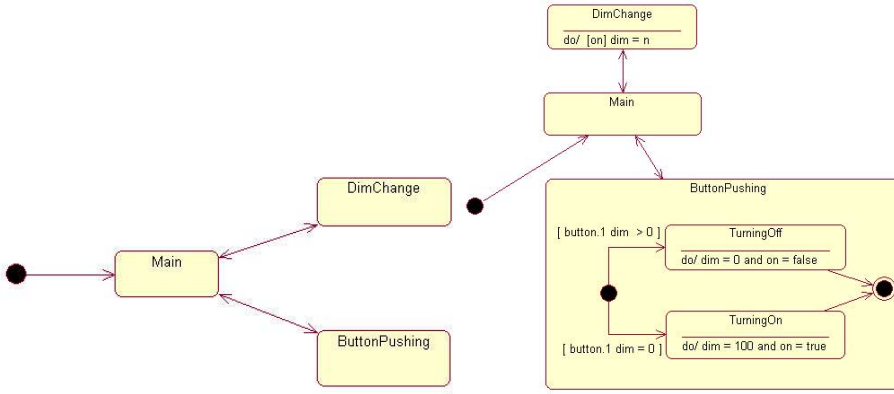


Fig. 4. Statechart for ControlledLight

6 Conclusions

The first contribution of the paper is the development of a syntax and static semantic checker for the TCOZ language in XML format, using a compiler approach. As TCOZ is a super set of Object-Z and Z, the type checker can also be used to type check Object-Z and Z specifications in XML format.

For the purpose of visualizing TCOZ behaviour model, the second part of the paper defined a set of projection rules for transforming a TCOZ model to UML statechart and demonstrated the implementation steps for the tool development.

In summary, this paper presents some 'light-weight' tool support for the TCOZ integrated formal specification technique. One future work on the type checker is to extend its capabilities with the techniques from ESC [9]. Looking at even more 'heave-weight' tools support, i.e. model checking and theorem proving, one further work is to translate TCOZ to timed automata so that tools like UPPAAL [2] can be used to check other TCOZ properties. We are currently also investigating the encoding of TCOZ notation in theorem provers such as HOL/Isabelle for automatic verification.

Acknowledgements

We would like to thank Hugh Anderson for many helpful comments. This work is supported by the Research Grants *Integrated Formal Methods* from National University of Singapore and *Techniques and Tools for Designing Embedded and Hybrid Systems* from Singapore A*STAR Program.

References

1. Xmi: Xml metadata interchange. http://www-4.ibm.com/software/ad/standards/xmi.html, 2000.
2. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.
3. P. J. Brooke and R. F. Paige. The Design of a Tool-Supported Graphical Notation for Timed CSP. In *Proc. Integrated Formal Methods 2002 (IFM'02)*, pages 299–318, Turku, Finland, May 2002.
4. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z.* Cornerstones of Computing. Macmillan, March 2000.
5. W. Johnston. A type checker for Object-Z. Technical report 96-24, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, July 1996.
6. B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*, pages 66–85. Springer-Verlag, June 1999.
7. B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., pages 1166–1185, Toulouse, France, September 1999. Springer-Verlag.
8. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
9. G. Nelson, K. Rustan, M. Leino, J. Saxe, and R. Stata. Extended static checking home page. Available on the Internet from http://www.research.digital.com/SRC/esc/Esc.html., 1996.
10. S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.
11. G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.
12. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
13. J. Sun, J. S. Dong, J. Liu, and H. Wang. A XML/XSL Approach to Visualize and Animate TCOZ. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 453–460. IEEE Press, 2001.
14. J. Sun, J. S. Dong, J. Liu, and H. Wang. Object-Z Web Environment and Projections to UML. In *WWW-10: 10th International World Wide Web Conference*, pages 725–734. ACM Press, May 2001.
15. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.