Generating MSCs from an Integrated Formal Specification Language

Jin Song Dong[1], Shengchao Qin[2], and Jun Sun[1]*

[1] School of Computing
National U. of Singapore
{dongjs,sunj}@comp.nus.eud.sg
[2] Singapore-MIT Alliance
National U. of Singapore
qinsc@comp.nus.edu.sg

Abstract. The requirements capture of complex systems requires powerful mechanisms for specifying system state, structure and interactive behaviors. Integrated formal specification languages are well suited for presenting more complete and coherent requirement models for complex systems. Given an integrated model, one can project it into multiple views for specialized analysis. Message Sequence Charts (MSCs) is a popular graphical notation for presenting interactive viewpoints of a system. In this paper, we investigate the semantic based transformation from an integrated formal specification language TCOZ to MSCs. An automated tool has also been developed for generating MSCs from TCOZ models. Furthermore, by inserting operation constraints (as assertions) into the generated MSCs, system testing requirements can be obtained.

Keywords: Requirement Engineering, TCOZ, MSC

1 Introduction

Multi-viewpoints [3,21] are effective techniques for capturing complex system requirements. Various formal notations are often used for presenting different viewpoints for large and complex systems which may have intricate system states and complex concurrent and interactive behaviors. The formal link and consistency issues between the viewpoint models represented in different formalisms remain as a challenging research topic. Recent investigations on linkings between different formalisms [5,13,17,26,29,30,32] may provide some meta support to the issues of viewpoints consistency. One such linked formalism is Timed Communicating Object Z (TCOZ) [17] which builds on the strengths of Object-Z [11,25] in modeling complex data and state with the strengths of TCSP [9] in modeling process control and real-time interactions. TCOZ can be well suited for presenting more complete and coherent requirement models that comprehend various

* Author for correspondence, fax: +65 6779 4580, phone: +65 6874 4353

E. Boiten, J. Derrick, G. Smith (Eds.): IFM 2004, LNCS 2999, pp. 168–186, 2004.

viewpoints for complex systems. Given an integrated model, one can project it into consistent multiple views for specialized analysis. In this paper, we are interested in one particular viewpoint projection – the communication and interaction perspective. Message Sequence Charts (MSCs) [15] is a popular graphical notation for presenting interactive viewpoints of a system. In this paper, we investigate the semantic based transformation from TCOZ (trace models) to MSCs (process models). By identifying a set of traces with MSCs, the cause and effect relations between partial ordering events in concurrent systems [10] can be captured. An automated tool has also been developed in Java for generating MSCs from TCOZ models. Furthermore, by inserting class invariants and operation constraints (as assertions) into the generated MSCs (execution scenarios), system testing requirements can be obtained.

Various attempts to combine formal specifications with graphical notations have been explored [2,6,7,8,16,20,22]. Bolton and Davies [2] has given a process semantics in CSP for UML activity diagrams. They use the process semantics to demonstrate the consistency of the object model. Our approach is to automatically generate MSCs based on its standard process semantics. Brooke and Paige [4] 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 formal graphical notations instead of creating new ones. Ng and Butler [20] have developed a tool for visualizing CSP in UML for both the static architecture and the dynamic behaviors. In our approach, we are particularly interested in capturing dynamic interactions between objects.

The remainder of the paper is organized as follows. Section 2 briefly introduces the technical background, the TCOZ notation and MSCs, both basic MSCs (BMSCs) and high-level MSCs (HMSCs). Section 3 presents the link between TCOZ and MSCs, both BMSCs and HMSCs, explains how to generate system test requirements from TCOZ specifications. Section 4 presents an XML-based automatic transformation tool built using Java. Section 5 concludes the paper.

2 Overview of TCOZ and MSCs

2.1 Overview of TCOZ

TCOZ is a blending of Object-Z and TCSP. The basic structure of a TCOZ document is the same as for Object-Z, consisting 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. Channels in TCOZ are defined as communication interfaces between objects. All dynamic interactions between objects must take place through channel communication mechanism. The true power of TCOZ comes from the ability to make use of TCSP primitives in describing the process aspects of an operation's behavior. All operation definitions in TCOZ are TCSP process definitions, with operation schemas identified with Timed CSP processes. The data-related aspects of TCOZ are modeled using state bindings and the process-related aspects are modeled using event traces and refusals [18].

We take a simplified version of the Light Control System (LCS) [12] to illustrate the features of TCOZ and as an example to demonstrate the projection from TCOZ to MSCs. LCS is an intelligent control system. It can detect the occupation of a building, then turn on or turn off the lights automatically. It is able to tune illumination (in percentage) in the building according to the outside light level. A typical system behavior is that when a user enters a room: a motion detector senses the presence of the person, the room controller reacts by receiving the current daylight level and turning on the light group with appropriate illumination setting (let $satisfy$ represent the relationship between daylight level and required illumination). When a user leaves a room (leaving it empty): the detector senses no movement, the room controller waits for *absent* time units and then turns off the light group. The occupant can directly turn on/off the light by pushing the button.

Light

$dim : 0 \,..\, 100$
$on : \mathbb{B}$

INIT
$dim = 0 \land on = false$

TurningOn
$\Delta(dim, on)$
$dim' = 100 \land on' = true$

TurningOff
$\Delta(dim, on)$
$dim' = 0 \land on' = false$

ControlledLight

Light

$button, dimmer : \textbf{chan}$

$ButtonPushing \mathrel{\widehat=} button?1 \rightarrow ([dim > 0] \bullet$
$\quad TurningOff \;\Box\; [dim = 0] \bullet TurningOn)$
$DimChange \mathrel{\widehat=} [n : 0 \,..\, 100] \bullet dimmer?n \rightarrow ([on] \bullet$
$\quad dim := n \;\Box\; [\neg\, on] \bullet \textsc{Skip})$
$\textsc{Main} \mathrel{\widehat=} \mu\, N \bullet (ButtonPushing \;\Box\; DimChange);\; N$

MotionDetector

$motion : \textbf{chan}$
$md : (Move \mid NoMove)$

$NoUser \mathrel{\widehat=} md?Move \rightarrow motion!1 \rightarrow User \;\Box$
$\quad md?NoMove \rightarrow \textsc{Wait}\ 1;\; NoUser$
$User \mathrel{\widehat=} md?NoMove \rightarrow motion!0 \rightarrow NoUser \;\Box$
$\quad md?Move \rightarrow \textsc{Wait}\ 1;\; User$
$\textsc{Main} \mathrel{\widehat=} NoUser$

Class *Light* is essentially an Object-Z class (passive class). Class *ControlledLight*, a subclass of *Light*, extends *Light* class with process definitions, *ButtonPushing*, *DimChange* and MAIN. A MAIN process indicates that *ControlledLight* defines an active object, which has its own thread of control. It is used to determine the behavior of objects of an active class after initialization. *button* and *dimmer* are defined as channels connecting the light to the environment and room controller.

A motion detector detects any movement in the room so to tell whether some one is in and send proper signal on channel *motion*. □ denotes a choice made by the environment.

RoomController

$dimmer, motion : \mathbf{chan}$
$odsensor : Illumination$
$absent : \mathbb{T}$
$olight : Illumination$

Adjust

$dim! : Percent$

$dim!\ \underline{satisfy}\ olight$

$Ready \mathrel{\widehat=} motion?1 \rightarrow On$

$Regular \mathrel{\widehat=} \mu\, R \bullet [n : Illumination] \bullet$
$\quad odsensor?n \rightarrow Adjust;\ dimmer!dim \rightarrow R$
$On \mathrel{\widehat=} Regular \triangledown motion?0 \rightarrow OnAgain$
$OnAgain \mathrel{\widehat=} (motion?1 \rightarrow On) \rhd\{absent\}\ Off$
$Off \mathrel{\widehat=} dimmer!0 \rightarrow Ready$
$\text{MAIN} \mathrel{\widehat=} Off$

LCS

$m : MotionDetector$
$l : ControlledLight$
$r : RoomController$

$\text{MAIN} \mathrel{\widehat=} \|(m \overset{motion}{\longleftrightarrow} r \overset{dimmer}{\longleftrightarrow} l)$

A room controller communicates with the motion detector and light by declaring channels with same names as those in the respective classes. It takes in signals from the motion detector and sends proper signal to the light. Its behavior is captured by interrupt $\triangledown$ and timeout $\rhd$ operators. Finally, a light control system is composed by a room controller, a motion detector and a light.

2.2 Overview of MSCs Language

The language MSCs is standardized by International Telecommunication Union (ITU). It provides a mean for visualization of the interaction of system components, either graphically or textually. The core of MSCs is called Basic Message

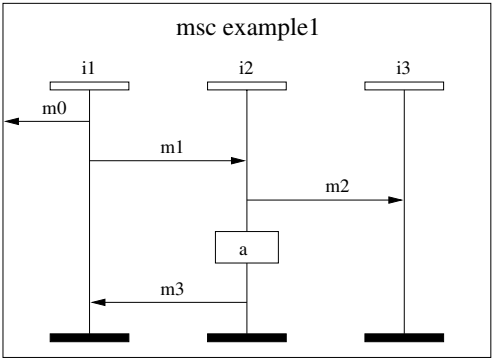

Fig. 1. A BMSC Example

Sequence Charts (BMSCs), which concerns communications and actions only. Then, additional basic concepts like process creation, termination, time handling, incomplete message events and conditions are added. Later, more complicated constructs are introduced. They are inline expressions, MSC reference expressions and High-level Message Sequence Charts (HMSCs), which enrich MSCs with intricate possibilities of describing complex systems.

A simple example of BMSCs is given as Figure 1. Each vertical line represents an active component (Z.120 terminology, an instance) in the system. The frame (Z.120 terminology, parallel frame) represents the environment. Instances can interact with other instances or the environment by sending messages ($m0$,$m1$,$m2$,$m3$). The square labeled with a is an action performed by instance $i2$. The timing information is captured by the two rules below:

- For each message passing, the message output event precedes the corresponding message input event.
- For each vertical line representing an instance, the time progresses from top to bottom.

HMSCs can be constructed incrementally by referencing a MSC using its name. MSCs can be combined vertically, horizontally or alternatively. Various constructors for composing MSCs are: **alt**, **seq**, **par**, **opt**, **exc** and **loop**. Precise semantics are developed for these key words, e.g., **alt** is defined as *delayed choice*, denoted as $\mp$, and **par** are defined as *delayed parallel composition*, denoted as $||$. Figure 2 is a simple example of HMSCs. Its semantics is captured by the process expression $(A \circ C)^{\circledast} \circ (A \circ B)$, where $^{\circledast}$ means unbounded recursion and $\circ$ means sequential composition.

Various semantic models are developed for MSCs. Examples are operational semantics for MSCs based on process algebra [1,15], Petri nets [31], automata, etc. The informal MSC semantics and formal process algebra semantics for MSC [15] are adopted in this paper.

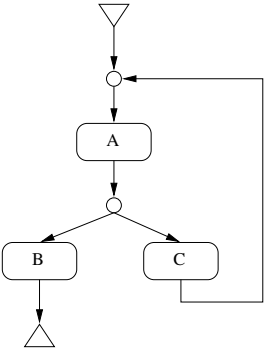

Fig. 2. A HMSC Example

3 Generate MSCs from TCOZ

MSCs is a simple graphical notation for capturing system components interaction. The process semantics MSCs is closely associated with the untimed trace aspects of the TCOZ semantics. Therefore, our first task is to build the untimed trace model for TCOZ that filters out unrelated issues i.e. timed refusals.

3.1 Trace Model for TCOZ

Semantic models for TCOZ is the infinite timed-states model [18] which extends the TCSP's infinite timed-failure model. MSCs, on the other hand, can be referred as 'untimed'. It deliberately abstracts from the precise times when events happen. Instead, MSC uses timers to capture basic timing information (timeout or timer reset). The untimed trace model in this section is mainly based on the trace models for CSP [14].

A TCOZ event may be either an update event, a simple synchronization, a channel communication, or a termination event.

$$\begin{aligned}
&\mathrm{U} == [\mathrm{v}: \textstyle\prod; \mathrm{d}: \mathrm{VAL}] \\
&\mathrm{S} == [\mathrm{c}: \Sigma] \\
&\mathrm{C} == [\mathrm{c}: \Sigma; \mathrm{d}: \mathrm{VAL}] \\
&\Sigma == \mathrm{update}\langle\langle U\rangle\rangle | sync\langle\langle S\rangle\rangle | com\langle\langle C\rangle\rangle | \checkmark
\end{aligned}$$

To make use of the timer constructs in MSC, we extend TCOZ event with a special event $wait(d)$, where d can be any real number. Basically, this event delays a process by time d. To filter out the unnecessary timing information, we simplify semantic functions for each TCOZ process expression. Given $[\Sigma]$ representing events, $[ZE]$ representing Z expressions, $[ZS]$ representing Z schemas, $[NAME]$ representing all valid character strings, a TCOZ process expression is defined as [19]:

Table 1. Simplified Trace Model of TCOZ based on [19]

Predicate	Remarks
$\mathcal{H}(\text{STOP}) = \{\langle\rangle\}$	(Tr-1)
$\mathcal{H}(\text{CHAOS}) = \{(\Sigma \backslash \{\checkmark\})^*\}$	(Tr-2)
$\mathcal{H}(\text{WAIT } ZE) = \{\langle wait(ze)\rangle\}$	(Tr-3)
$\mathcal{H}(ZS \bullet TZE) = \mathcal{H}(TZE) \cup \{\langle\rangle\}$	(Tr-4)
$\mathcal{H}(c.a \rightarrow TZE) = \{\langle c.a\rangle \frown u \mid u \in \mathcal{H}(TZE)\} \cup \{\langle\rangle\}$	(Tr-5)
$\mathcal{H}(TZE_1 \mid TZE_2) = \mathcal{H}(TZE_1) \cup \mathcal{H}(TZE_2)$	(Tr-6)
$\mathcal{H}(TZE_1 \triangledown TZE_2) = \{s \frown t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-7)
$\mathcal{H}(TZE_1 \parallel TZE_2) = \mathcal{H}(TZE_1) \cap \mathcal{H}(TZE_2)$	(Tr-8)
$\mathcal{H}(TZE_1 \mid\mid\mid TZE_2) = \bigcup\{s \mid\mid\mid t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-9)
$\mathcal{H}(TZE_1 \mid[X]\mid TZE_2) = \bigcup\{s \mid[X]\mid t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-10)
$\mathcal{H}(TZE_1;\ TZE_2) = \{\mathcal{H}(TZE_1) \cap (\Sigma \backslash \{\checkmark\})^*\} \cup$ $\{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-11)
$\mathcal{H}(\mu X \bullet TZE) = \bigcup_{n>0} \mathcal{H}(F^n(STOP))$	(Tr-12)

$$
\begin{aligned}
TZE ::= {} & ref\langle\!\langle NAME\rangle\!\rangle \mid \text{STOP} \mid \text{CHAOS} \mid \text{SKIP} \mid \text{WAIT}\langle\!\langle ZE\rangle\!\rangle \mid op\langle\!\langle ZS\rangle\!\rangle \mid \\
& (\_ \bullet \_)\langle\!\langle ZS \times TZE\rangle\!\rangle \mid (\_ \rightarrow \_)\langle\!\langle \Sigma \times TZE\rangle\!\rangle \mid (\_.\_ \rightarrow \_)\langle\!\langle \Sigma \times ZE \times TZE\rangle\!\rangle \mid \\
& (\_ \sqcap \_)\langle\!\langle TZE \times TZE\rangle\!\rangle \mid (\_ \,\Box\, \_)\langle\!\langle TZE \times TZE\rangle\!\rangle \mid (\_ \parallel \_)\langle\!\langle TZE \times TZE\rangle\!\rangle \mid \\
& (\mu \_ \bullet \_)\langle\!\langle NAME \times TZE\rangle\!\rangle \mid (\_[\_/\_])\langle\!\langle TZE \times \Sigma \times \Sigma\rangle\!\rangle \mid (\_\backslash\_)\langle\!\langle TZE \times \mathbb{P}\,\Sigma\rangle\!\rangle \mid \\
& (\_ \triangledown \_)\langle\!\langle TZE_1 \times TZE_2\rangle\!\rangle \mid (\_ \mid\mid\mid \_)\langle\!\langle TZE \times TZE\rangle\!\rangle \mid (\_;\ \_)\langle\!\langle TZE \times TZE\rangle\!\rangle \mid \\
& (\_ \triangleright \{\_\}\ \_)\langle\!\langle TZE \times ZE \times TZE\rangle\!\rangle \mid (\_ \mid[\_]\mid \_)\langle\!\langle TZE \times \mathbb{P}\,\Sigma \times TZE\rangle\!\rangle
\end{aligned}
$$

A trace is defined as a sequence of events. Given a set of events (Σ), Σ^* denotes all possible traces can be composed by events in Σ. Function $\upharpoonright$ filters out a set of events from a trace.

$$Trace \mathrel{\widehat{=}} \text{seq}\,\Sigma$$

$$\_^* : \mathbb{P}\,\Sigma \rightarrow \mathbb{P}\,Trace$$

$$\forall e : \mathbb{P}\,\Sigma;\ tr : Trace \bullet \quad tr \in e^* \Leftrightarrow \text{ran}\,tr \subseteq e$$

$$\_ \upharpoonright \_ : Trace \times \mathbb{P}\,\Sigma \rightarrow Trace$$

$$
\begin{aligned}
& \forall t_i, t_e : Trace, e : \mathbb{P}\,\Sigma \bullet \\
& \quad t_i \upharpoonright e = t_e \Leftrightarrow \\
& \quad \text{ran}\,t_e \cap e = \varnothing \wedge \\
& \quad (\text{head}(t_i) = \text{head}(t_e) \wedge \\
& \qquad \text{tail}(t_i) \upharpoonright e = \text{tail}(t_e)) \vee \\
& \quad (\text{head}(t_i) \neq \text{head}(t_e) \wedge \\
& \qquad \text{tail}(t_i) \upharpoonright e = t_e)
\end{aligned}
$$

We define a function $\mathcal{H} : TZE \rightarrow \mathbb{P}\,Trace$, which returns the set of all possible traces given a TCOZ process expression. Table 1 illustrates the detailed definition for the function $\mathcal{H}$, which defines how to compute the set of all possible traces for a TCOZ process expression inductively. The definition of the function $\mathcal{H}$ is based on the denotational semantics of CSP [14,24].

STOP means deadlock and performs no events (Tr-1), while CHAOS can perform any event (Tr-2). WAIT ZE delays a process by ze time unit (Tr-3). The state-guard, is used to *block* or *enable* execution of an operation on the basis

of an object's local state (the instance's state) (Tr-4). In general, state-guard can be complex. In our trace model, it is treated as non-deterministic choice, which may introduce unexpected traces. This is not a problem for our work. Tr-5 captures the case that a process expression is guarded by some channel communication event. The only way to proceed is to perform the communication. Tr-6 covers both internal choice ($P \sqcap Q$) and external choice ($P \Box Q$). For internal choice, the choice is made upon the internal state of the system. While for external choice, the choice is made by the environment. From the system interactive (MSC) viewpoint, this distinction is irrelevant. Tr-7 captures that Q interrupts P. Tr-8 expresses that the two processes synchronize on all events. Tr-9 expresses that two processes run completely independently. Tr-10 expresses a general case of Tr-8 and Tr-9, i.e., instead of synchronizing all (or none) of the events, only events in the set X are synchronized.

$$
\begin{array}{|l}
\_ \,|||\, \_ : \mathit{Trace} \times \mathit{Trace} \to \mathbb{P}\, \mathit{Trace} \\
\hline
\forall\, a, b, c : \mathit{Trace} \bullet \\
\quad c \in a \,|||\, b \Leftrightarrow \\
\qquad \#c = \#a + \#b \land \\
\qquad c \upharpoonright \mathrm{ran}\, a = a \land \\
\qquad c \upharpoonright \mathrm{ran}\, b = b
\end{array}
\qquad
\begin{array}{|l}
\_ [\![\_]\!] \_ : \mathit{Trace} \times \mathbb{P}\, \Sigma \times \mathit{Trace} \to \mathbb{P}\, \mathit{Trace} \\
\hline
\forall\, a, b, c : \mathit{Trace};\ e : \mathbb{P}\, \Sigma \bullet \\
\quad c \in a \,[\![\, e \,]\!]\, b \Leftrightarrow \\
\qquad c \upharpoonright e = a \upharpoonright e = b \upharpoonright e \land \\
\qquad c \upharpoonright \mathrm{ran}\, a = a \land \\
\qquad c \upharpoonright \mathrm{ran}\, b = b
\end{array}
$$

Tr-11 expresses sequential composition of process expressions. TZE_2 can only take control after TZE_1 successfully terminates. This definition of sequential composition is known as *strong sequential composition* [1]. The trace model for recursion (Tr-12) is a fixed point definition. The trace function for SKIP, $\triangleright\{t\}$, $\swarrow\{t\}$ can be derived using the following laws.

$$
\begin{array}{l}
\mathsf{Skip} \mathrel{\widehat{=}} \checkmark \to \textsc{Stop} \\
TZE_1 \triangleright\{t\} TZE_2 = TZE_1 \Box (\textsc{Wait}\ t; TZE_2) \\
TZE_1 \swarrow \{t\} TZE_2 = TZE_1 \triangledown (\textsc{Wait}\ t; TZE_2)
\end{array}
$$

Example: The set of traces for a process expression can be efficiently identified by applying function $\mathcal{H}$ recursively. We take the process *ButtonPushing* as an example.

$$
\begin{aligned}
\mathcal{H}(\mathit{ButtonPushing}) &= \mathcal{H}(\mathit{button}?1 \to ([\mathit{dim} > 0] \bullet \mathit{TurningOff} \ \Box \\
&\qquad [\mathit{dim} = 0] \bullet \mathit{TurningOn})) \\
&= \{\langle \mathit{button}?1 \rangle \frown u \mid u \in \mathcal{H}([\mathit{dim} > 0] \bullet \mathit{TurningOff} \ \Box \\
&\qquad [\mathit{dim} = 0] \bullet \mathit{TurningOn})\} \cup \{\langle\rangle\} \\
&= \{\langle \mathit{button}?1 \rangle \frown u \mid u \in \mathcal{H}([\mathit{dim} > 0] \bullet \mathit{TurningOff}) \cup \\
&\qquad \mathcal{H}([\mathit{dim} = 0] \bullet \mathit{TurningOn})\} \cup \{\langle\rangle\} \\
&= \{\langle \mathit{button}?1, \mathit{TurningOff} \rangle, \langle \mathit{button}?1, \mathit{TurningOn} \rangle, \langle \mathit{button}?1 \rangle, \langle\rangle\}
\end{aligned}
$$

3.2 Link Traces with BMSCs

Given one active object, we can identify the set of possible traces by applying the $\mathcal{H}$ function to the MAIN process. A trace can be transformed to a BMSC by identifying update events in TCOZ with MSC atomic local actions and identifying channel communications in TCOZ with message passing in MSCs.

In the previous subsection, a TCOZ event is defined as either an update event, a simple synchronization, a channel communication, or a termination event, or a WAIT(d) event.

- Update events are distinguished from the others in the way that they do not require the cooperation of the environment or other processes. They perform on a single instance. A MSC local action is defined as an orderable single instance event requiring no cooperation from environment. Update events are identified with local actions in MSCs.
- Synchronization and channel communication do require cooperation either from environment or other processes. Channel communications in TCOZ are identified with message passings in MSCs.
  MSCs support both synchronous and asynchronous message passing, channel communication in TCOZ is identified with synchronous message passing (message passing with a 0-capacity buffer).
- The special *wait* event in TCOZ is identified with the timer event in MSCs. In particular, it is identified with a timer set event in MSCs and consequently associated with a timeout or reset event.

Example: Figure 3 (the generated BMSC from TCOZ by applying $\mathcal{H}$ function) represents a possible scenario of the process *ControlledLight*. Initially the light is off. Starting with MAIN, the process *DimChange* is executed. A message input event *dimmer*?n takes place. At that moment *on* is false, no action is taken. Process *ButtonPushing* is then activated by a message input event from channel *button*. Action *TurningOn* is invoked. After that, no event occurs.

3.3 Project TCOZ Specifications to HMSCs

Due to unbounded recursion (iteration) and non-determinism, possible traces (and generated BMSCs) for some systems could be numerous or even infinite[1]. HMSCs offer various constructive operators to compose MSCs in a hierarchical, iterating and nondeterministic way. In this section, we link TCOZ specification with HMSCs by identifying various operators in TCOZ with constructs in HMSCs.

The body of a TCOZ class is essentially a system of simultaneous equations defining a collection of operations (processes). Each equation consists of a name [$NAME$] and a TCOZ process expression.

[1] For LCS, 600+ traces are generated if we unfold recursions 5 times.

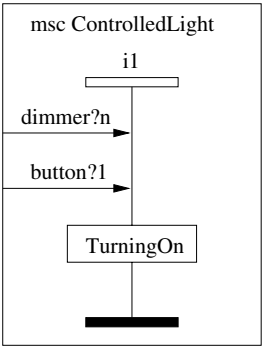

Fig. 3. BMSC: ControlledLight

A TCOZ class is identified with a MSC document[2]. A TCOZ process expression is identified with a MSC. If a TCOZ process invokes other process expressions (by name), the process expression name will be identified with a MSC reference. A MSC reference expression is defined as the following,

$$MRE ::= ref\langle\langle NAME\rangle\rangle \mid \epsilon \mid \delta \mid (\_ \circ \_)\langle\langle MRE \times MRE\rangle\rangle \mid$$
$$(\_ \mp \_)\langle\langle MRE \times MRE\rangle\rangle \mid (\_ \parallel \_)\langle\langle MRE \times MRE\rangle\rangle \mid$$
$$(\_)^{\circledast}\langle\langle MRE\rangle\rangle \mid (\_)^{\infty}\langle\langle MRE\rangle\rangle$$

Given a MSC reference expression, Function $\mathcal{G} : MRE \rightarrow \mathbb{P}\, Trace$ returns a set of traces capturing all possible behaviors of the process. In [15], semantics of various constructs of MSCs are specified by sets of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and conclusion are of the form $s \xrightarrow{a} s'$ or $s \downarrow$ for arbitrary $s, s' \in MRE$ and $a \in A$, where A denotes all events represented by atomic actions in MSCs, for example, message input, message output, local action and timer events. Following those deduction rules, the trace models $\mathcal{G}$ can be constructed.

$$\frac{\mathcal{P} : TZE \rightarrow MRE}{\forall\, s : TZE;\ t : MRE \bullet \mathcal{P}(s) = t \Rightarrow \mathcal{H}(s) = \mathcal{G}(t)}$$

A projection function $\mathcal{P}$ from TCOZ process expression to MSC reference expression can be established when the set of possible traces for the TCOZ process expressions is identical with those for the MSC reference expressions.

STOP and SKIP. In TCOZ, Stop means deadlock and no communications. Skip performs no action except for successful termination. The two basic constants, denoted as δ and ε, also play the same role in the process semantics for

[2] A MSC document consists a set of MSCs.

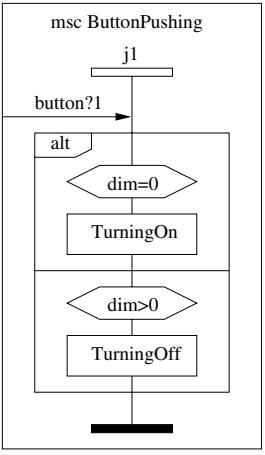

Fig. 4. Transformation: Choice

MSCs. No deduction rule is associated with δ. The rule associated with ε is $\varepsilon\downarrow$, meaning successful termination.

$\mathcal{G}(\delta) = \{\langle\rangle\}$
$\mathcal{G}(\varepsilon) = \{\langle\surd\rangle\} \cup \{\langle\rangle\}$

Graphically, SKIP is mapped to a MSC containing no event.

Choice. In [15], a structural operation *delayed choice* is denoted by $\mp$. Its semantics is expressed in the following rules:

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

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

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

We can verify that the set of possible traces of $x \mp y$ is the union of traces of x and y, i.e., $\mathcal{G}(x \mp y) = \mathcal{G}(x) \cup \mathcal{G}(y)$. Assume $s \in \mathcal{G}(x \mp y)$, if $head(s)$ is an event that can be performed by process x and not by process y, $DC3$ applies, the set of all such traces is $\{\langle head(s)\rangle \frown u | u \in \mathcal{G}(x')\}$, which is a subset of $\mathcal{G}(x)$. If $head(s)$ is an event that can be performed by y and not by x, rule $DC4$ applies, following

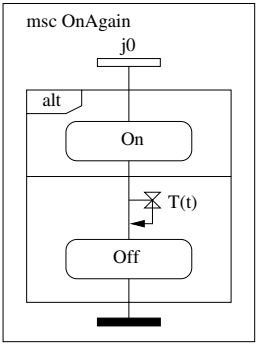

Fig. 5. Transformation: Timeout

the same argument, we can verify that the set of such traces is a subset of $\mathcal{G}(y)$. If $head(s)$ can be performed by both x and y, $DC5$ applies, the set of possible traces is $\{\langle head(s)\rangle^\frown u | u \in \mathcal{G}(x' \mp y')\}$, which is a subset of $\mathcal{G}(x) \cup \mathcal{G}(y)$. Since s is an arbitrary trace in the set $\mathcal{G}(x \mp y)$, we conclude that $\mathcal{G}(x \mp y)$ is a subset of $\mathcal{G}(x) \cup \mathcal{G}(y)$. By the similar construction, we can conclude that for any trace in $\mathcal{G}(x) \cup \mathcal{G}(y)$, it is also in $\mathcal{G}(x \mp y)$. This completes the construction of the choice operator. In term, other mappings can be formulated in a similar way.

Figure 4 illustrates how the transformation is done graphically for the process *ButtonPushing* in *ControlledLight* class. Keyword **alt** (short for alternative) is used to denote *delayed choice* graphically.

Timeout. In TCOZ, $P \triangleright \{t\}Q = P\Box$ (WAIT $t; Q$). By identifying external choice with MSC delayed choice and WAIT t with timer events, timeout can be identified with MSCs constructed by a delayed choice between P and Q with a timeout event as the initial event of Q. In the room controller model of the LCS example, given process $OnAgain \widehat{=} (motion?1 \rightarrow On) \triangleright \{t\}$ *Off*, it is transformed to the MSC in Figure 5.

Interleaving. Delayed parallel composition, denoted by $||$ in MSC, is rewritten as $||_m$ to avoid confusion. Delayed parallel composition defines the interleaving operator, i.e., no synchronization is required and processes can interleave freely[3]. Interleaving in TCOZ ($|||$) is identified with delayed parallel composition in MSCs.

$$\mathcal{G}(P||_m Q) = \bigcup\{s|||t \,|\, s \in \mathcal{G}(P) \wedge t \in \mathcal{G}(Q)\}$$

Synchronization. In TCOZ, all dynamic interactions between active objects must take place through the CSP channel communication mechanism. All synchronization is done by message passing through channels.

[3] Refer to [15] for detailed definition of delayed parallel composition.

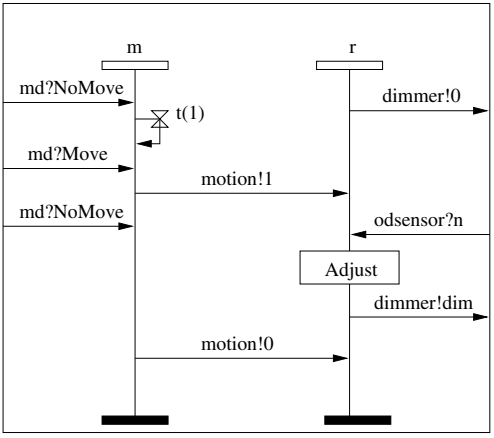

Fig. 6. Transformation: Synchronization

No synchronization construct is defined in MSCs. Graphically, given two synchronizing MSCs (P and Q), the composed MSC is constructed by putting the MSCs in the same parallel frame and connecting corresponding message output and message input events. By adopting the view that message passing are synchronized, the newly constructed MSC represents the set of traces as

$$\bigcup\{s \parallel [X]t | s \in \mathcal{G}(P) \wedge t \in \mathcal{G}(Q)\}$$

where X denotes all events on the sharing channel.

In the LCS class, active object m (motion detector) shares the channel *motion* with the active object r (room controller). A possible trace for m would be

$$\langle md?NoMove, wait\ 1, md?Move, motion!1, md?NoMove, motion!0, \cdots \rangle$$

A matching trace for r must contain the same events on channel *motion*. For example, a matching sequence would be

$$\langle dimmer!0, motion?1, odsensor?n, olight := n, dimmer!dim, motion?0, \cdots \rangle$$

The interaction can be visualized as Figure 6.

By constructing the composed MSCs as above, we can make use of the full power of MSC's partial ordering property. That is, to leave the order of single instance events from different instances unspecified. Thus, one MSC is capable of representing a set of scenarios.

Sequential Composition. Sequential composition in TCOZ is best described as *strong sequential composition*. *Strong sequential composition* of two processes x and y behaves like process x and upon termination of x it starts behaving like process y. No action from process y can be executed before x has the option

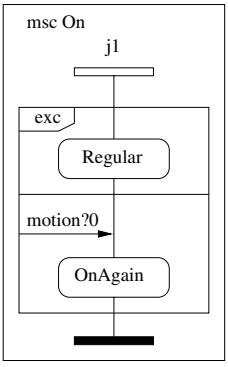

Fig. 7. Transformation: Interrupt

to terminate. In [15], a different approach, named *weak sequential composition*, denoted as $\circ$ is adopted to compose two MSCs vertically. The *weak sequential composition* allows the execution of actions from y before x has the option to terminate.

All synchronizations in TCOZ are taken through channels, the ordering information of local actions from different instances are irrelevant. On the other hand, in case two MSCs only involve events on the same process, The *weak sequential composition* and the *strong sequential composition* are the same. Sequential composition in TCOZ is identified with sequential composition in MSCs.

Graphically, sequential composition of MSCs on the same instances is captured by putting the MSCs one below the other.

Interrupt. MSCs have a key word **exc** for representing exceptions, however there is no formal rules defined in [15]. We define the rules for **exc** (the symbol ∇_m is used instead) as follows.

$$\frac{x\downarrow}{x\ \nabla_m\ y\downarrow} \qquad \frac{x \xrightarrow{a} x',\ y \not\xrightarrow{a} y'}{x\ \nabla_m\ y \xrightarrow{a} x'\ \nabla_m\ y} \qquad \frac{y \xrightarrow{a} y'}{x\ \nabla_m\ y \xrightarrow{a} y'}$$

In the process $X \nabla Y$ (Y has an initial event e), any time e takes place, X is interrupted and control transfers to Y. Interrupt in TCOZ is identified with ∇_m in MSCs with e as the initial event of the interrupting process. For example, in *RoomController*,

$$On \widehat{=} Regular \nabla motion?0 \rightarrow OnAgain$$

can be transformed to MSC as in Figure 7.

Besides the projection links above, the rest constructs in TCOZ can be transformed to constructs in MSCs in obvious ways. TCOZ recursion can be resolved as iteration and interpreted by a sequence of sequential composition, which can

be generalized as the iteration operator $^{\circledast}$. TCOZ $state - guard$ is identified with *condition* in MSCs.

3.4 Generate Test Requirements

Test requirements can be used to develop test cases, test oracles and test drivers in a system development. Specification based testing can play an important role in software engineering [23,27]. TCOZ specification based testing can be based on the generated MSCs (execution scenarios). Our goal is to support automatic generation of test requirements.

Four steps are essential, which are all based on the MSCs generated.

- Starting with a HMSC, one can expand the HMSC into a set of BMSCs. In the recursion case, at least one iteration should be covered by the expanded BMSCs.
- Upon creation of an instance, TCOZ class initial state condition is instrumented as an assertion at the start of the BMSC.
- For each instance in the system, TCOZ class invariants are instrumented as assertions before and after each action on the BMSC instance.
- The pre/post-conditions of TCOZ operations is transformed to assertions at the entry/exit of the corresponding MSC actions.

We will illustrate these steps by taking the *ControlledLight* class as an example. First, we resolve the unbound recursion in MAIN process by performing it once. From Figure 4, we identify two event sequences from both *DimChange* and *ButtonPushing* because of the delayed choice operator. The testing requirements for *ControlledLight* is captured by Figure 8 (an expanded BMSC with Assertions). Assertions are placed in the dash line square-boxes. The event sequence for *DimChange* containing SKIP is dropped since SKIP represents an empty sequence of events.

4 Automation

The translation process can be automated by employing XML/XSL technology. In our previous work [28], a XML interchange format ZML for Z family languages, i.e., Z/Object-Z/TCOZ, has been defined using XML Schema. MSC also offer a standard text representation for the graphical notations. In this work, an automatic transformation tool is developed in Java to project TCOZ models (in ZML) into MSCs (in standard text format).

Building on the strength of ZML, our tool makes use of XML parser Xerces to extract information from TCOZ specifications. For example, the following is a part of the *ControlledLight* class model in ZML.

```
<classDef>
      <name>ControlledLight</name>
      <inheritedClass><name>Light</name></inheritedClass>
      <state>...</state>
      <operation>
```

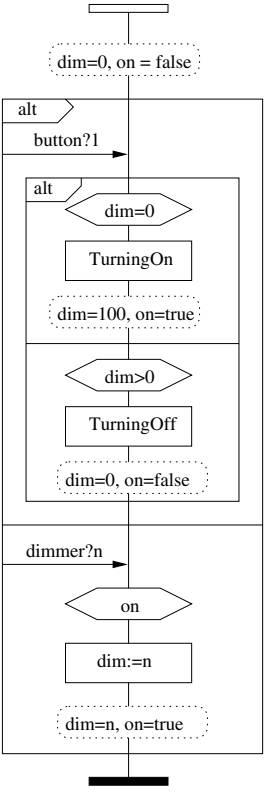

Fig. 8. Test Requirements

```
            <name>Main</name>
            <processExpr>
                  <mu>N</mu>
                  <processExpr>
                        <processExpr>
                              <simpleProExp>ButtonPushing</simpleProExp>
                        </processExpr>
                        <proConnSym>externalChoice</proConnSym>
                        <processExpr>
                              <simpleProExp>DimChange</simpleProExp>
                        </processExpr>
                  </processExpr>
            </processExpr>
      </operation>
</classDef>
```

The automatic transformation is achieved by first implementing a ZML parser, which will take in a specification model in ZML and build a virtual model in the memory. This ZML parser can be reused for other projection tools (e.g. the transformation from TCOZ to Timed Automata for timing analysis).

A trace generation module is built to automatically generate all possible traces for a specification model, each trace can be transformed to a BMSC by syntax rewriting. In the case of unbounded recursion, users may provide the expected number of iterations.

An MSC interface is built according to the MSC document structure, e.g., each MSC document contains multiple MSCs and each MSC contains one or more instances and etc. A transformation module is built to get information from the ZML parser, apply the right transformation rules (specified in Section 3) and feed the outcome of the transformation to the MSC interface. The transformation rules are used as a design document and guide the construction of various transformation algorithms in the implementation.

The outcome of our transformation tool is Z.120 standard text representation of MSCs, which is ready to be taken as inputs for various tool support for MSCs. For example, the above XML representation of MAIN operation in *ControlledLight* is transformed to a HMSC as:

```
msc ControlledLight;
    instance i1;
        loop begin;
        alt begin;
            reference ButtonPushing;
        alt;
            reference DimChange;
        alt end;
        loop end;
   endinstance;
endmsc;
```

Reuse for Timed Viewpoint Projection

The same strategy can be applied for implementing various transformation tools. For example, for the timing analysis purpose, a TCOZ specification can be transformed into Timed Automata (we are currently building this tool), the same ZML parser can be reused and we only need to build a Timed Automata interface and a new transformation module.

5 Conclusion

In this paper, we investigate the semantic based transformation from TCOZ to MSCs and present a tool to automatically generate MSCs from the TCOZ specifications.

An untimed trace model for TCOZ is introduced to focus on the interaction viewpoints. Each possible trace is identified with a BMSC by linking TCOZ update events as MSC local actions and channel communication as synchronous message-passing. The projection from TCOZ to HMSCs is constructed based on linking the trace semantic models of TCOZ constructs with the process semantics of HMSC constructs.

By inserting appropriate TCOZ specification constraints (as assertions) into the generated MSCs, we further explore ways of generating system test requirements from TCOZ.

Acknowledgments. This work is supported by the A*STAR research grant *Formal Design Techniques for Reactive Embedded Systems*

References

1. J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 18(1), 1990.
2. C. Bolton and J. Davies. Activity Graphs and Processes. In W. Grieskamp, T. Santen, and W. Stoddart, editors, *Proceedings of IFM 2000*, pages 77–96. Springer, 2000.
3. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A Formal Framework for Viewpoint Consistency. *Formal Methods in System Design*, 21:111–166, September 2002.
4. P. J. Brooke and R. F. Paige. The Design of a Tool-Supported Graphical Notation for Timed CSP. In M. J. Butler, L. Petre, and K. Sere, editors, *Proc. Integrated Formal Methods 2002 (IFM'02)*.
5. M. Butler. csp2B: A Practical Approach To Combining CSP and B. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., Toulouse, France, September 1999. Springer-Verlag.
6. C-A. Chen, S. Kalvala, and J. Sinclair. Generating b specifications from message sequence charts. In *St.Eve Workshop*, September 2003.
7. A. Coombes and J. A. McDermid. Using Diagrams to Give a Formal Specification of Timing Constraints in Z. In *Z User Workshop*, pages 119–130, 1992.
8. J. Davies and C. Crichton. Using State Diagrams to Describe Concurrent Behaviour. In *ICFEM 2003, LNCS 2885*, pages 105–125, 2003.
9. J. Davies and S. Schneider. A Brief History of Timed CSP. *Theoretical Computer Science*, 138, 1995.
10. Marcio S. Dias and Debra J. Richardson. Identifying Cause and Effect Relations between Events in Concurrent Event-Based Componenents. In J. Richardson, W. Emmerich, and D. Wile, editors, *The 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002.
11. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing Series. Macmillan, March 2000.
12. R. L. Feldmann, J. Munch, S. Queins, S. Vorwieger, and G. Zimmermann. Baselining a Doman-Specific Software Development Process. Tech Report SFB501 TR-02/99, University of Kaiserslautern, 1999.
13. C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
14. C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
15. ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.
16. Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. Sofl: A formal engineering methodology for industrial applications. pages 24–45, 1998.
17. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
18. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

19. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(1):142–160, 2002.
20. M. Y. Ng and M. Butler. Tool Support for Visualizing CSP in UML. In C. George and H. Miao, editors, *International Conference on Formal Engineering Methods (ICFEM'02)*, pages 287–298. LNCS, Springer-Verlag, October 2002.
21. B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirement Specifications. *IEEE Trans. Software Eng.*, 20(10):760–773, October 1994.
22. L. Petre and K. Sere. Developing Control Systems Components. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *IFM'00: Integrated Formal Methods,*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.
23. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *International Conference on Software Engineering*, pages 105–118, 1992.
24. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
25. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
26. G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems - an Integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.
27. P. Stocks and D. Carrington. A Framework for Specification-based Testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.
28. J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13:329–356, 2002.
29. K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Press.
30. H. Treharne and S. Schneider. Using a Process Algebra to Control B Operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
31. K.M. van Hee. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, 1994.
32. J. Woodcock and A. Cavalcanti. The Steam Boiler in a Unified Theory of Z and CSP. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 291–298. IEEE Press, 2001.