# Reasoning About TCOZ

Jin Song Dong Jing Sun
School of Computing
National University of Singapore
{dongjs,sunjing}@comp.nus.edu.sg

## ABSTRACT

Timed Communicating Object Z (TCOZ) semantically and syntactically blends Object-Z with TCSP. TCOZ can capture the data/state and real-time interactions of system components. TCOZ preserves in large part both the syntax and semantics of the individual notations and hence can potentially benefit from existing reasoning systems of the individual notations. This paper extends and links the Object-Z and TCSP proof systems for reasoning about TCOZ models.

## 1. INTRODUCTION

The rapid increase in terms of size and complexity of software systems has led to rising demands for precise and well structured high level design techniques for modelling complex system data, state, concurrency, and real-time behavior. Recently there has been active investigation of the integration [6, 15, 23] of formal object-oriented methods (e.g. Object-Z [5, 22]) with process description languages (e.g. CSP [11]). One such approach, the Timed Communicating Object Z (TCOZ) [15] combines Object-Z's strengths in modelling complex data and state with TCSP's strengths in modelling real-time concurrency. TCOZ extended the inherited CSP's channel based communication mechanism, in which messages represent discrete synchronization between processes, to continuous function interface mechanisms inspired by process control theory: the sensor/actuator [14]. The introduction of a novel network topology operator allows the communications interfaces of complex processes to be visualized through simple network-topology graphs. This improves decoupling of class definitions by simplifying the interfaces between objects, thereby enhancing the modularity of system specifications.

Based on TCSP semantics [20, 19], the denotational semantics of TCOZ has been developed [13]. However, in order to formally verify system properties, a proof system for TCOZ is needed. TCOZ preserves a large part of both the syntax and semantics of the base notations and hence can potentially benefit from existing reasoning systems of the individual notations. In this paper we extend and link Smith's proof system of Object-Z [21][1] and Davies/Schneider's proof system of TCSP [19] for reasoning about TCOZ models. The new proof rules for the TCOZ novel constructs, i.e., sensor/actuators, network topology, deadline and wait-until commands, etc, are developed in this paper. We demonstrate the verification process of a Railroad Crossing System (RCS) as a case study.

The remainder of the paper is organized as follows. Section 2 briefly introduces the TCOZ notation. Section 3 presents the extended TCOZ inference rules from Object-Z and TCSP with new proof rules. Section 4 presents the formal model of the RCS in TCOZ. Section 5 presents the verifications of critical properties of the RCS. Section 6 concludes the paper.

[1]The proof system of Object-Z [21] extends the logic $\mathcal{W}$[24] of Z.

## 2. TCOZ FEATURES

Timed Communicating Object Z (TCOZ) is essentially a blending of Object-Z[2] with TCSP, 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 can appear wherever processes appear in CSP and CSP process definitions can appear wherever operation definitions 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 TCSP and Object-Z features may be found elsewhere [15]. The formal semantics of TCOZ is also documented [13].

### 2.1 A model of time

In TCOZ, all timing information is represented as real valued measurements in *seconds*, the SI standard unit of time.We believe that a mature approach to measurement and measurement standards is essential to the application of formal techniques to system engineering problems. In order to support the use of standard units of measurement, extensions to the Z typing system suggested by Hayes and Mahony [8] are adopted. Under this convention, time quantities are represented by the type

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

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

In order to describe the timing requirements of operations and sequences of operations, a deadline command inspired by [7, 9] is introduced. If $OP$ is an operation specification (defined through any combination of CSP process primitives and Object-Z operation schemas) then $OP \bullet \textsc{Deadline}\, t$ describes the process which has the same effect as $OP$, but is constrained to terminate no later than $t$. If it cannot terminate by time $t$, it deadlocks.

The $\textsc{WaitUntil}$ operator is a dual to the deadline operator. The process $OP \bullet \textsc{WaitUntil}\, t$ performs $OP$, but will not terminate until at least time $t$. If it would normally terminate before time $t$, it idles.

[2]Object-Z [2] is an OO extension to Z. In the OOP context, Object-Z shares some common features with predicate classes [3].

## 2.2 Channels, sensors and actuators

CSP channels are given an independent, first class role in TCOZ. In order to support the role of CSP channels, the state schema convention is extended to allow the declaration of communication channels. If $c$ is to be used as a communication channel by any of the operations of a class, then it must be declared in the state schema to be of type **chan**. Channels are type heterogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state attributes, channels are viewed as shared (global) rather than as encapsulated entities. This is an essential consequence of their role as communication 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 and preventing aliasing problems [12, 17]. We believe that TCOZ can be a suitable language for specifying component-based systems.

Complementary to the CSP channel mechanism, TCOZ also adopts a non-synchronizing shared variable mechanism. A declaration of the form $s : X$ **sensor** provides a channel-like interface for using the shared variable $s$ as an input. A declaration of the form $s : X$ **actuator** provides a local-variable-like interface for using the shared variable $s$ as an output. Sensors and actuators may appear either at the system boundary (usually describing how global analog quantities are sampled from, or generated by the digital subsystem) or else within the system (providing a convenient mechanism for describing local communications which do not require synchronization). The shift from closed to open systems necessitates close attention to issues of control, an area that both Z and CSP are weak in. We believe that TCOZ with the **actuator** and **sensor** can be a good candidate for specifying open control systems. Mahony and Dong [14] presented a detailed discussion on TCOZ sensors and actuators.

## 2.3 Active objects

Active objects [16] 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 behavior 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 behavior of the two objects can be represented as $ob_1 \ ||| \ ob_2$, which means $ob_1.\mathrm{MAIN} \ ||| \ ob_2.\mathrm{MAIN}$

## 2.4 Semantics of TCOZ

The details of the blended state/event process model forms the basis for the TCOZ denotational semantics [13]. In brief, the semantic approach identifies the notions of operation and process by providing a process interpretation of the Z operation schema construct. TCOZ differs from many other approaches to blending Object-Z with a process algebra in that it does not identify operations with events. Instead an unspecified, fine-grained collection of state-update events is hypothesized. Operation schemas are modelled by the collection of those sequences of update events that achieve the state change described by the schema. This means that there is no semantic difference between a Z operation schema and a CSP process. It therefore makes sense to also identify their syntactic classes.

The process model used by TCOZ consists of sets of tuples consisting of: an *initial* state, a *trace* (a sequence of time stamped events, including update-events), a *refusal* (a record of what and when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is if a $\checkmark$ 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 $\checkmark$) 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.

## 2.5 Network topologies

The syntactic structure of the CSP synchronization operator is suitable for the case of pipeline like communication topologies. When expressing more complex communication topologies it generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology [15]. 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

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

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

$$\|(A \overset{ab}{\longleftrightarrow} B \overset{bc}{\longleftrightarrow} C \overset{ca}{\longleftrightarrow} A),$$

and multiple channels above the arrow, for example if processes $D$ and $F$ communicate privately through the channels/sensors/actuators $df_1$ and $df_2$, then

$$\|(D \overset{df_1, df_2}{\longleftrightarrow} F).$$

# 3. TCOZ INFERENCE RULES

The proof systems of Object-Z and TCSP can be adopted and extended to facilitate the reasoning of both state and event oriented properties of a TCOZ specification.

## 3.1 State oriented reasoning

### *3.1.1 Adopted Object-Z rules*

The essential extension to Z in Object-Z is the class construct which groups the definition of a state schema and the definitions of its associated operations. From a system point of view, it also enables modular verification. Smith extents the $W$ logic of Z to Object-Z for reasoning about object models and type systems [18]. The fundamental logic in Object-Z is presented in the form of the sequent defined as follows:

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

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

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

The upper part is called a premiss which contains zero or more sequents; the middle part is called a proviso which is a predicate that

makes the rule applicable; and the lower part is called a conclusion which is a single sequent which must be valid when the proviso and the premisses are true.
Some of the inference rules adopted from Object-Z are listed as follows. Detailed information of Object-Z inference rules can be found in Smith's logic for Object-Z [21].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Inheritance related rules

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

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

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

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

### *3.1.2 TCOZ extension rules*

TCOZ [15] extends Object-Z class definitions in two aspects. Firstly, the state schema convention is extended to allow the declaration of object communication interfaces, i.e., channels, sensors and actuators. If $c$ is to be used as a communication interface by any of the operations of a class, then it must be explicitly declared in the state schema. Channels are type heterogeneous and may carry communications of any type, while sensors/actuators are type specific. These communication interfaces are connected by the network topologies in TCOZ. The second extension is that as well as operations (terminating processes), non-terminating processes named $MAIN$ are introduced to represent the behavior of active classes. The inheritance mechanism of active classes differs from the normal passive classes as the $MAIN$ operation must always be redefined explicitly. For a complete TCOZ semantics refer to paper [13].
Based on the logic of Object-Z above some extension rules in TCOZ are defined below:

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

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

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

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

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

$A$

$c : \textbf{chan}$
$...$

$MAIN \mathrel{\widehat=} ...c!x...$

$B$

$c : \textbf{chan}$
$...$

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

$AB$

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

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

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

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

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

$A$

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

$...$

$B$

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

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

$AB$

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

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

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

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

## 3.2 Event oriented reasoning

TCSP [19] is an extension of Hoare's Communicating Sequential Process (CSP) to accommodate the description of time-sensitive behaviors. A requirements specification $S(s, \aleph)$ of TCSP processes in TCOZ is the possible observations that can be made for their executions. These are described in terms of the timed failure model $(s, \aleph)$, which consists of timed traces and timed refusals. Timed trace $s$ is the sequence of events occurring during the execution according to their timing aspects, while the timed refusal set $\aleph$ is the timed events which are refused by the execution. A process $Q$ meets a specification $S(s, \aleph)$ if $S$ holds for every timed failure associated with $Q$.

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

### *3.2.1 TCSP rules in TCOZ*

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

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

Some of the TCSP inference rules adopted in the TCOZ context are listed below. A detailed view of TCSP inference rules can refer to the TCSP proof system [19].

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

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

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

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

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

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

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

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

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

### 3.2.2 TCOZ extension rules

In this section, we develop the proof rules for the new TCOZ constructs, i.e., DEADLINE and WAITUNTIL commands and network topology.

#### DEADLINE *command*

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

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

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

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

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

The above states that the process has the same effect as $Q$, but is constrained to terminate no later than $d$. If it fails to terminate by time $d$, it deadlocks. The timed failure representation is as follows:

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

The inference rule for the DEADLINE constructor is presented as follows:

$$\frac{A :: Q \textbf{ sat } S(s,\aleph)}{\begin{aligned}A :: Q \bullet \textsc{Deadline}\, d \textbf{ sat } (end(s) \leqslant d\\ \wedge \checkmark \in \sigma(s) \wedge S(s, \aleph \restriction d))\end{aligned}}$$

#### WAITUNTIL *command*

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

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

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

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

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

The above states that the process has the same effect as $Q$, but it will not terminate until at least time $d$. It is represented in terms of the timed failure model as follows:

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

The inference rule for the WAITUNTIL constructor is presented as follows:

$$\frac{A :: Q \textbf{ sat } S(s,\aleph)}{\begin{aligned}A :: Q \bullet \textsc{WaitUntil}\, d \textbf{ sat } (end(s) > d \wedge S(s,\aleph))\\ \vee (end(s) \leqslant d \wedge S(s,\aleph) \wedge \checkmark\ \mathit{live\ from}\ d\\ \mathit{until}\ \{\checkmark\})\end{aligned}}$$

#### *Network topology*

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

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

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

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

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

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

$$\begin{aligned}&\mathcal{TF}[\![Q_1 \overset{c}{\longleftrightarrow} Q_2]\!] = \{(s,\aleph) \mid \exists\, s_1, s_2, \aleph_1, \aleph_2 \bullet\\ &\quad \aleph = \aleph_1 \cup \aleph_2 \wedge s = s_1 \cup s_2\\ &\quad \wedge (s_1, \aleph_1) \in \mathcal{TF}[\![Q_1]\!]\\ &\quad \wedge (s_2, \aleph_2) \in \mathcal{TF}[\![Q_2]\!]\}\end{aligned}$$

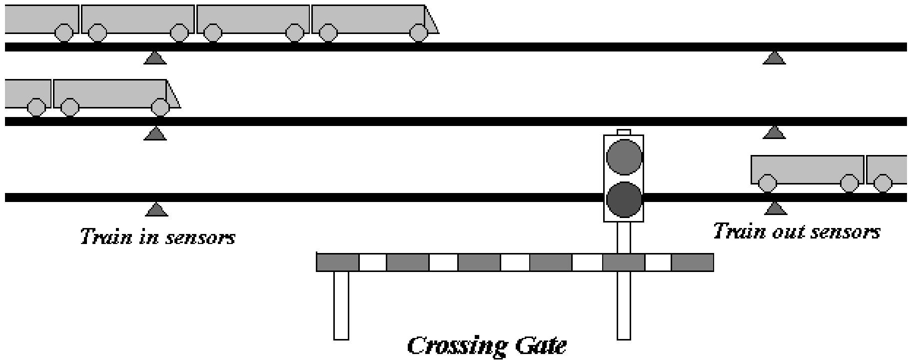


**Figure 1: A general scenario of the Railroad Crossing System.**

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

$$\frac{\begin{array}{l} A :: Q_1 \ \mathbf{sat} \ S_1(s, \aleph) \\ B :: Q_2 \ \mathbf{sat} \ S_2(s, \aleph) \end{array}}{\begin{array}{l} AB :: Q_1 \xleftrightarrow{c} Q_2 \ \mathbf{sat} \ \exists\, s_1, s_2, \aleph_1, \aleph_2 \bullet \\ \quad S_1(s_1, \aleph_1) \wedge S_2(s_2, \aleph_2) \\ \quad \wedge\ s = s_1 \cup s_2 \wedge \aleph = \aleph_1 \cup \aleph_2 \end{array}}$$

# 4. RAILROAD CROSSING SYSTEM

The concept of the Railroad Crossing Problem was primarily evolved by Heitmeyer [10] and used as a case study in many formal systems. The problem is to design a system which operates a gate at a railroad crossing safely. In this section, we will present a Railroad Crossing System (RCS) specified in TCOZ. Two essential properties of RCS are: first, the gate should be down whenever a train is crossing; second, the gate should be up as often as possible when no trains are passing by. A generalized railroad scenario is shown in Figure 1. Based on the above features, we define some assumptions and constraints as follows:

1. The train on each track has a known maximum speed $v_{max}$.
2. The minimum and maximum time for operating the gate (either open or close) are $g_{min}$ s and $g_{max}$ s.
3. The minimum and maximum time for switching the traffic lights are $l_{min}$ s and $l_{max}$ s. And $l_{max}$ satisfies $l_{max} \leqslant g_{min}$.
4. The sensors for detecting an incoming train will be placed at a distance $d$ from the gate, where $d$ satisfies $g_{max} < d/v_{max}$. The sensors for detecting the leaving of a train will be placed at a distance close to the other side of the gate.
5. Multi-track and multi-train situations are considered. We assume the trains are moving in one direction.

The first and fourth constraints are for the computation of the placement of the sensors to detect the incoming trains. If there is no speed limit, fast trains can always arrive before the gate can be properly closed. For a slow gate this distance will be larger and vice versa. The third constraint is to avoid the case of the gate state changing before the traffic lights change. According to the above discussion, a generic RCS consists of five types of components: a central controller, a group of track components for receiving train's information, light and gate components to control the traffic and a group of auxiliary components for offering timing and logging.
The basic types for the status of the gate, traffic light and train are defined as follows:

$$\begin{array}{l} GateS == UP \mid DOWN \\ LightS == GREEN \mid RED \\ TrainS == IN \mid OUT \end{array}$$

## 4.1 Clock

In order to record the system information at each particular time, a calendar-like clock is constructed as follow.
Calendar time type is defined as:

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

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

$$\begin{array}{|l} cal : \mathbb{N}\,\mathsf{s} \rightarrowtail CalT \\ \hline \ldots \end{array} \qquad \text{[detail of function omitted]}$$

is used to convert the number of elapsed seconds to a calendar-time. The elapsed time counter must be incremented every second and the display screen updated in less than 50 milliseconds.

$$\begin{array}{|l} \textit{Clock} \\ \hline \begin{array}{|l} total : \mathbb{N}\,\mathsf{s} \\ \Delta \\ time : CalT\ \mathbf{actuator} \\ \hline time = cal(total) \end{array} \quad \begin{array}{|l} \textit{Inc} \\ \hline \Delta(total) \\ \hline total' = total + 1\,\mathsf{s} \end{array} \\ \\ \textsc{Main} \mathrel{\widehat=} \mu\, C \bullet (Inc \bullet \textsc{Deadline}\, 50\,\mathsf{ms}) \bullet \\ \qquad\qquad \textsc{WaitUntil}\, 1\,\mathsf{s};\ C \end{array}$$

The attribute $time$ is modelled as a secondary attribute [5] (implicitly included in the delta list of every operation) whose value depends on the value of the primary attribute $total$.

## 4.2 System logs

RCS needs persistent repository of data and history logs. A generic active object $Log[X]$ is defined below, where $X$ is the data structure type of the record in the log.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Log}[X]} \\ \hline
\begin{array}{|l}
\textit{log} : \mathrm{seq}\, X \\
\textit{record} : \textbf{chan} \\ \hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Add}} \\ \hline
\Delta(\textit{log}) \\
x? : X \\ \hline
\textit{log}' = \textit{log} \frown \langle x? \rangle \\ \hline
\end{array} \\
\\
\textit{Write} \mathrel{\widehat=} [x : X] \bullet \textit{record}?x \to \textit{Add} \\
\textsc{Main} \mathrel{\widehat=} \mu\, L \bullet \textit{Write};\ L \\ \hline
\end{array}
$$

The system logs consist two type of logs. One is for the logged train passing information; and the other is for recording of gate status. These can also be customized according to various requirements individually. The content in the log file is modelled as a variant of type $X$, which varies according to each particular system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SystemLogs}} \\ \hline
\begin{array}{|l}
\textit{tracklog} : \textit{Log}[\textit{CalT} \times \textit{Track} \times \textit{Train} \times \textit{TrainS}] \\
\textit{gatelog} : \textit{Log}[\textit{CalT} \times \textit{GateS}] \\ \hline
\end{array} \\
\\
\textsc{Main} \mathrel{\widehat=} \textit{tracklog} \mathbin{|||} \textit{gatelog} \\ \hline
\end{array}
$$

## 4.3 Tracks

The system receives the train passing information from the sensors on each railway track.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Track}} \\ \hline
\begin{array}{|l}
t_{in}, t_{out} : \textit{Train}\ \textbf{sensor} \\
\textit{time} : \textit{CalT}\ \textbf{sensor} \\
\textit{in}, \textit{out}, \textit{record} : \textbf{chan} \\ \hline
\end{array} \\
\\
\textit{Train}_{in} \mathrel{\widehat=} [t : \textit{CalT};\ \textit{train}_i : \textit{Train}] \bullet t_{in}?\textit{train}_i \\
\qquad \to (\textit{in}!(\textit{train}_i, \textit{self}) \to \textit{time}?t \to \\
\qquad \textit{record}!(t, \textit{self}, \textit{train}_i, \textit{IN}) \to \textsc{Skip} \\
\textit{Train}_{out} \mathrel{\widehat=} [t : \textit{CalT};\ \textit{train}_i : \textit{Train}] \bullet t_{out}?\textit{train}_i \\
\qquad \to (\textit{out}!(\textit{train}_i, \textit{self}) \to \textit{time}?t \to \\
\qquad \textit{record}!(t, \textit{self}, \textit{train}_i, \textit{OUT}) \to \textsc{Skip} \\
\textsc{Main} \mathrel{\widehat=} \mu\, T \bullet (\textit{Train}_{in} \mathbin{\Box} \textit{Train}_{out});\ T \\ \hline
\end{array}
$$

The $t_{in}$ and $t_{out}$ sensors detect the incoming and leaving train signals from the environment, report to the central controller and record the information to the repository. The basic behavior of the track component is collecting the train passing information. The multi-track system is a collection of all the railway tracks:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Tracks}} \\ \hline
\begin{array}{|l}
\textit{tracks} : \mathbb{F}\ \textit{Track} \\ \hline
\end{array} \\
\\
\textsc{Main} \mathrel{\widehat=} \ \mathbin{|||}\ \textit{tr} : \textit{tracks} \\ \hline
\end{array}
$$

## 4.4 Traffic Lights

The traffic lights component indicates crossing movements and can be modelled as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Lights}} \\ \hline
\begin{array}{|l}
\textit{value} : \textit{LightS} \\
\textit{setlight} : \textbf{chan} \\ \hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\ \hline
\textit{value} = \textit{GREEN} \\ \hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Change}} \\ \hline
\Delta(\textit{value}) \\
x? : \textit{LightS} \\ \hline
x? \neq \textit{value} \wedge \textit{value}' = x? \\ \hline
\end{array} \\
\\
\textit{Set} \mathrel{\widehat=} [x : \textit{LightS}] \bullet \textit{setlight}?x \to \textit{Change} \\
\textsc{Main} \mathrel{\widehat=} \mu\, L \bullet (\textit{Set} \bullet \textsc{Deadline}\ l_{max}\ \mathsf{s});\ L \\ \hline
\end{array}
$$

The essential behavior of the *Lights* is switching its indication between red and green. Note that the TCOZ DEADLINE expression is used to constrain the light changing operation which must be finished within its maximum light switching period.

## 4.5 Crossing Gate

The railroad crossing gate can be modeled as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Gate}} \\ \hline
\begin{array}{|l}
\textit{status} : \textit{GateS} \\
\textit{time} : \textit{CalT}\ \textbf{sensor} \\
\textit{setgate}, \textit{record} : \textbf{chan} \\ \hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\ \hline
\textit{status} = \textit{UP} \\ \hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Up}} \\ \hline
\Delta(\textit{status}) \\ \hline
\textit{status} = \textit{DOWN} \\
\textit{status}' = \textit{UP} \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Down}} \\ \hline
\Delta(\textit{status}) \\ \hline
\textit{status} = \textit{UP} \\
\textit{status}' = \textit{DOWN} \\ \hline
\end{array} \\
\\
\textit{Open} \mathrel{\widehat=} [t : \textit{CalT}] \bullet \textit{setgate}?\textit{UP} \to \\
\qquad (\textit{Up} \bullet \textsc{Deadline}\ g_{max}\ \mathsf{s}) \to \textit{time}?t \\
\qquad \to \textit{record}!(t, \textit{UP}) \to \textsc{Skip} \\
\textit{Close} \mathrel{\widehat=} [t : \textit{CalT}] \bullet \textit{setgate}?\textit{DOWN} \to \\
\qquad (\textit{Down} \bullet \textsc{Deadline}\ g_{max}\ \mathsf{s}) \to \textit{time}?t \\
\qquad \to \textit{record}!(t, \textit{DOWN}) \to \textsc{Skip} \\
\textsc{Main} \mathrel{\widehat=} \mu\, G \bullet (\textit{Open} \mathbin{\Box} \textit{Close});\ G \\ \hline
\end{array}
$$

The fundamental behaviors of the gate are open and close. After perform the each operation a record is made in the system log.

## 4.6 Central Controller

All train passing information will be stored and the actions on the lights and gate will be assigned through the controller. It is the central and crucial part of the system, actively communicating with other parts. The *Controller* class is modeled as follows:

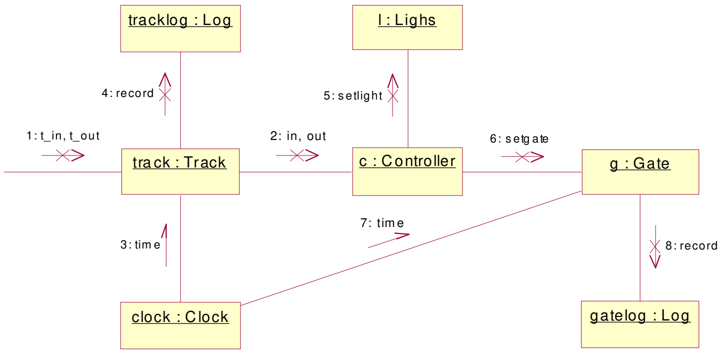


**Figure 2: The communications of the RCS.**

*Controller*

$tracks : \mathbb{F}(Track \times Train)$
$in, out, setlight, setgate : \textbf{chan}$
$g : Gate$

INIT
$tracks = \varnothing$

*Arrive*
$\Delta(tracks)$
$train? : Train$
$track? : Track$

---

$train? \notin \operatorname{ran} tracks$
$tracks' = tracks \cup \{(track?, train?)\}$

*Leave*
$\Delta(tracks)$
$train? : Train$
$track? : Track$

---

$(track?, train?) \in tracks$
$tracks = tracks' \cup \{(track?, train?)\}$

$Train_{in} \mathrel{\widehat{=}} [train_i : Train;\ track_i : Track] \bullet in?(train_i, track_i) \rightarrow Arrive$
$Train_{out} \mathrel{\widehat{=}} [train_i : Train;\ track_i : Track] \bullet out?(train_i, track_i) \rightarrow Leave$
$GateOpen \mathrel{\widehat{=}} [tracks = \varnothing \wedge g.state = DOWN] \bullet setgate!UP \rightarrow setlight!GREEN \rightarrow \textsc{Skip}$
$GateClose \mathrel{\widehat{=}} [tracks \neq \varnothing \wedge g.state = UP] \bullet setlight!RED \rightarrow setgate!DOWN \rightarrow \textsc{Skip}$
$\textsc{Main} \mathrel{\widehat{=}} \mu\, C \bullet (Train_{in} \,\Box\, Train_{out} \,\Box\, GateOpen \,\Box\, GateClose);\ C$

The attribute *tracks* keeps the records of the requested train passings in the system. When a train reaches the sensor $t_{in}$ a new record is added into the request set. When a train passes the sensor $t_{out}$ the corresponding train record will be deleted from the set. The main processes of the controller are adding or deleting track requests and manipulating the gate operations at the same time. If the request set is not empty and the gate is open then instructions on turning on the red light and closing the gate will be sent to the *Lights* and *Gate* correspondingly. On the other hand, when the request set is empty and the gate is still closed, the controller will open the gate and change the light to green.

## 4.7 RCS Configuration

After specifying individual components, the next step is to compose them into a whole system. The overall system is a composition of all components, which communicate with each other. We organize the interactive components through network topologies.

*RCSystem*

$clock : Clock$
$logs : SystemLogs$
$trs : Tracks$
$g : Gate$
$l : Light$
$c : Controller$

---

$c.g = g$

$\textsc{Main} \mathrel{\widehat{=}} \|(trs \xleftrightarrow{in,out} c;\ l \xleftrightarrow{setlight} c \xleftrightarrow{setgate} g;\ clock \xleftrightarrow{time} trs,\ g \xleftrightarrow{record} logs)$

The UML collaboration diagram in Figure 2 visualizes the configuration of the system. The lines connecting components depict the interactive communication relationships, on which are the names of corresponding implicit connectors (communication interfaces, in other words). For example, the *Controller* and the *Gate* communicate through the *setgate* channel, which is written above the line between the two components.

# 5. SYSTEM ANALYSIS AND REASONING

From a safety critical perspective, the key point of the RCS is to provide guaranteed safety and efficient service to the customers. The safety property can be formally interpreted from our model as:

- Gate down status – This property states that if there are train passing requests and the gate is still open, a gate state changing will be recorded in the log. It can be translated into a theorem as follows:

  **Theorem 1:** $RCSystem$ :: $[P_1]$
  $c.tracks \neq \varnothing \wedge g.status = UP \Rightarrow snd(gatelog.log(\#gatelog.log)) = DOWN$

- Gate closing within critical time range – This property ensures that a gate closing operation will be performed within a required critical time range. It can be translated into a theorem as follows:

  **Theorem 2:** $RCSystem$ :: $[P_2]$
  $\forall\, train_i : Train;\ track_i : trs;\ t_i : CalT;\ t_0 : \mathbb{T}$
  $\bullet\ (c.tracks \neq \varnothing \wedge g.status = UP)\ at\ t_0$
  $\Rightarrow record!(t_i, DOWN)\ at\ (t_0, t_0 + g_{max})$

Before we formally verify the safety property of the system, the first thing is to show that the $Clock$ component in our system correctly models the behavior of a physical timing device – the global clock. This property can be interpreted as the following timed specification in terms of the timed failure model.

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

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

$$a\ live\ I = \forall\, t \in I \bullet a\ at\ t \wedge \forall\, t_1, t_2 : \mathbb{T} \bullet (t_1 < min\ I \\ \Rightarrow \neg(a\ at\ t_1) \wedge t_2 > max\ I \Rightarrow \neg(a\ at\ t_2))$$

This macro simply expresses that the event $a$ is continuously recorded in the trace as having occurred at every point on an maximal interval $I$.

**Proof of $L_0$:**
Base case: The specification is trivially satisfied by $STOP$.
Assuming that $C$ **sat** $L_0(s,\aleph)$, it is sufficient to show that $(Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms}) \bullet \text{WAITUNTIL}\,1\,\mathsf{s};\; C$ **sat** $L_0(s,\aleph)$.
Let:

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

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

$$\begin{array}{ll}
Clock :: Inc \textbf{ sat } L_1(s,\aleph) & \\ \hline
Clock :: Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms} \textbf{ sat } & [\,\text{DEADLINE}\,] \\
\quad (end(s) \leqslant 50\,\mathsf{ms} \wedge \checkmark \in \sigma(s) \wedge & \\
\quad L_1(s, \aleph \upharpoonright 50\,\mathsf{ms})) & \\ \hline
Clock :: Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms} \textbf{ sat } & [\,Weaken\,] \\
\quad L_2(s,\aleph) & \\ \hline
Clock :: (Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms}) \bullet & [\,\text{WAITUNTIL}\,] \\
\quad \text{WAITUNTIL}\,1\,\mathsf{s} \textbf{ sat } & \\
\quad ((end(s) > 1\,\mathsf{s} \wedge & \\
\quad L_2(s,\aleph)) \vee & \\
\quad (end(s) \leqslant 1\,\mathsf{s} \wedge L_2(s,\aleph) & \\
\quad \wedge \checkmark\ live\ from\ 1\,\mathsf{s}\ \ until\ \{\checkmark\})) & \\ \hline
Clock :: (Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms}) \bullet & [\,Weaken\,] \\
\quad \text{WAITUNTIL}\,1\,\mathsf{s} \textbf{ sat } L_0(s,\aleph) & \\
Clock :: C \textbf{ sat } L_0(s,\aleph) & \\ \hline
Clock :: ((Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms}) \bullet & [\,Composition\,] \\
\quad \text{WAITUNTIL}\,1\,\mathsf{s});\; C \textbf{ sat } & \\
\quad ((\checkmark \notin \sigma(s) \wedge & \\
\quad L_0(s, \aleph \cup [0,\infty) \times \{\checkmark\})) & \\
\quad \vee\ (\exists\, s_1, s_2 \bullet s = s_1 \frown s_2 & \\
\quad \wedge \checkmark \notin \sigma(s_1) \wedge & \\
\quad begin(s_1) \geqslant t \wedge & \\
\quad L_0(s_1 \frown \langle (t, \checkmark) \rangle, & \\
\quad \aleph \upharpoonright t \cup [0, t) \times \{\checkmark\}) \wedge & \\
\quad L_0((s_2, \aleph) - t))) & \\ \hline
Clock :: ((Inc \bullet \text{DEADLINE}\,50\,\mathsf{ms}) \bullet & [\,Weaken\,] \\
\quad \text{WAITUNTIL}\,1\,\mathsf{s});\; C \textbf{ sat } L_0(s,\aleph) &
\end{array}$$

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

After showing that the $Clock$ component is consistent to the global clock, we can now prove $P_1$ and $P_2$ with respect to their state and event based aspects.

## 5.1 Proof of $P_1$

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

$$\begin{array}{ll}
RCSystem :: STATE \vdash c \in controller & \\
\quad \wedge\ g \in Gate \wedge gatelog \in Log[CalT & \\
\quad \times GateS] & \\
Controller :: STATE \vdash setgate \in \textbf{chan} & \\
\quad \wedge\ MAIN \vdash tracks \neq \varnothing \wedge g.status & \\
\quad = UP \Rightarrow setgate!DOWN & \\
Gate :: STATE \vdash setgate \in \textbf{chan} & \\
RCSystem :: STATE \vdash c.g = g & \\
\quad \wedge\ MAIN \vdash c \overset{setgate}{\longleftrightarrow} g & \\ \hline
Gate :: MAIN \vdash setgate.DOWN & [\,CHANNEL\,] \\
Gate :: STATE \vdash record \in \textbf{chan} & \\
\quad \wedge\ MAIN \vdash setgate.DOWN & \\
\quad \Rightarrow record!(t, DOWN) & \\
Log[CalT \times GateS] :: & \\
\quad STATE \vdash record \in \textbf{chan} & \\
RCSystem :: MAIN \vdash g \overset{record}{\longleftrightarrow} gatelog & \\ \hline
Log[CalT \times GateS] :: MAIN \vdash & [\,CHANNEL\,] \\
\quad record.(t, DOWN) \in (CalT \times \{UP, & \\
\quad DOWN\}) &
\end{array}$$

As shown above, the train passing request which causes the changing of the gate status from $UP$ to $DOWN$ will add a new record of $(t, DOWN)$ to the gate repository. Hence $P_1$ holds.

## 5.2 Proof of $P_2$

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

$$P_2(s,\aleph) = RCSystem :: \forall\, train_i : Train; \\ track_i : trs;\; t_i : CalT;\; t_0 : \mathbb{T} \bullet (c.tracks \neq \varnothing \\ \wedge\ g.status = UP)\ at\ t_0 \Rightarrow record!(t_i, DOWN) \\ at\ (t_0, t_0 + g_{max}]$$

**Proof:** Let $P_{2.1}$, $P_{2.2}$, $P_{2.3}$ and $P_{2.4}$ be four time failure expressions represented as follows:

$$P_{2.1}(s,\aleph) = Gate :: \forall\, setgate.DOWN\ at\ t_0 \Rightarrow \\ status' = DOWN\ at\ (t_0, \infty)$$
$$P_{2.2}(s,\aleph) = Gate :: setgate.DOWN\ at\ t_0 \Rightarrow \\ status' = DOWN\ at\ (t_0, t_0 + g_{max}]$$
$$P_{2.3}(s,\aleph) = Gate :: setgate.DOWN\ at\ t_0 \Rightarrow \\ record!(t, DOWN)\ at\ (t_0, t_0 + g_{max}]$$
$$P_{2.4}(s,\aleph) = Controller :: (tracks \neq \varnothing \\ \wedge\ g.gate = UP)\ at\ t_0 \Rightarrow setgate!DOWN\ at\ t_0$$

The proof of $[P_2]$ can be constructed via event reasoning rules as follows:

$$\dfrac{Gate :: Down \textbf{ sat } P_{2.1}(s, \aleph)}{Gate :: Down \text{ DEADLINE } g_{max} \textbf{ sat } (end(s) \leqslant g_{max} \wedge \checkmark \in \sigma(s) \wedge P_{2.1}(s, \aleph \upharpoonright g_{max})} [\text{ DEADLINE }]$$

$$\dfrac{Gate :: Down \text{ DEADLINE } g_{max} \textbf{ sat } (end(s) \leqslant g_{max} \wedge \checkmark \in \sigma(s) \wedge P_{2.1}(s, \aleph \upharpoonright g_{max})}{Gate :: Down \text{ DEADLINE } d \textbf{ sat } P_{2.2}(s, \aleph)} [\ Weaken\ ]$$

$$\dfrac{Gate :: Down \text{ DEADLINE } d \textbf{ sat } P_{2.2}(s, \aleph)}{Gate :: Close \textbf{ sat } P_{2.3}(s, \aleph)} [\ Weaken\ ]$$

$$\dfrac{\begin{array}{l} Gate :: Close \textbf{ sat } P_{2.3}(s, \aleph) \\ Controller :: GateClose \textbf{ sat } P_{2.4}(s, \aleph) \end{array}}{\begin{array}{l} RCSytem :: c.GateClose \overset{setgate}{\longleftrightarrow} g.Close \\ \quad \textbf{sat } \exists s_1, s_2, \aleph_1, \aleph_2 \bullet P_{2.3}(s_1, \aleph_1) \\ \quad \wedge P_{2.4}(s_2, \aleph_2) \wedge s = s_1 \cup s_2 \\ \quad \wedge \aleph = \aleph_1 \cup \aleph_2 \end{array}} [\ Network\ ]$$

$$\dfrac{\ldots}{RCSystem :: c \overset{setgate}{\longleftrightarrow} g \textbf{ sat } P_2(s, \aleph)} [\ Weaken\ ]$$

According to the network topology rule, the behavior specification $P_2(s, \aleph)$ is satisfied, therefore **Theorem** $P_2$ has been proved. From the above we can see that if the gate is open and there are passing train requests the gate will be closed within its maximum operating period. Thus the safety requirements of the RCS are formally verified. Verifications on the efficiency of the system, i.e., the gate will be open whenever no trains are passing by, can be constructed via similar techniques.

# 6. CONCLUSIONS

In this paper, we combined and extended both state-based Object-Z and event-based TCSP proof systems for verifying TCOZ specifications. We demonstrated the verification of the RCS properties by applying TCOZ proof rules. Complex system properties can be decomposed into state and event related properties and proved respectively. In summary, the paper demonstrates that integrated formal modelling techniques, TCOZ, can be a good candidate for modelling and verifying real-time composite systems. As the proof process in the paper is still hand-based, we are currently investigating the encoding of TCOZ proof rules in theorem provers such as HOL/Isabelle for automatic verifications.

## Acknowledgements

We would like to thank Hugh Anderson for his helpful comments on this work. This work is supported by the academic research grant *Integrated Formal Methods* (R-252-000-050-107) 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. *Formal Description Techniques, II (FORTE'89)*, pages 281–296. North-Holland, 1990.

[3] C. Chambers. Predicate classes. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 707, pages 268–296. Springer-Verlag, 1993.

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

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

[6] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [1].

[7] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. Technical Report CS-TR-3274, 1994. also appeared in RTSS'94.

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

[9] I. J. Hayes and M. Utting. Deadlines are termination. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98)*. Chapman and Hall, 1998. accepted.

[10] C. L. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In *Proceedings of RTSS'94, Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE Computer Society Press.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[12] J. Hogg, R. Holt D. Lea, A. Wills, and D. de Chanmpeaux. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, April 1992.

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

[14] B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., pages 1166–1185, Toulouse, France, September 1999. Springer-Verlag.

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

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

[17] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. *ECOOP '98 – Object-Oriented Programming, 12th European Conference*, volume 1445, pages 158–185. Springer-Verlag, 1998.

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

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

[20] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. *Real-Time: Theory in Practice*, volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.

[21] G. Smith. Extending W for Object-Z. *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

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

[23] 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.

[24] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. *the Sixth Annual Z User Meeting, York, UK.*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.