# Specifying and Reasoning Generic Architecture in TCOZ

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

## Abstract

*Formal modeling techniques can be used to define and verify software architectures precisely. This paper applies the recently developed integrated formal specification techniques, Timed Communicating Object Z (TCOZ), to the generic software architecture modeling and verification.*

**Keywords***: integrated formal specification and verification*

## 1 Introduction

Software architecture modeling is an important level of description for software systems. Z has been used to formalize the computational data/state aspects of software architectures [8]. CSP-like notation, Wright [1] has also been applied to formalize the interactive communication aspects of software architectures. Both approaches are beneficial and provide some formal foundations to software architecture description, however, the formal link and consistency issues between the models represented in different formalisms remain as a challenge. In this paper, we apply the integrated formal notation, Timed Communicating Object Z (TCOZ) [6], as an architecture description language (ADL) to model and verify a generic dispatch software system architecture. TCOZ builds on the strengths of Object-Z [2, 10] in modeling complex data and state with the strengths of TCSP [7] in modeling process control and real-time interactions. The class construct in TCOZ is an ideal encapsulation mechanism for composing and extending architecture models. The synchronized and asynchronized communication interfaces in TCOZ are well suited for capturing various interactions between the components. The network topology of TCOZ is a good mechanism to depicting the architectural configurations. Furthermore, TCOZ preserves a large part of both the syntax and semantics of the individual notations and hence can potentially benefit from existing reasoning systems of the individual notations. With some additional proof rules for the new TCOZ constructs, Smith's logic for Object-Z [9] (extension to the *W* logic of Z) and Davies/Schneider's proof system for TCSP [7] can be used for the reasoning of TCOZ properties, in this case the architecture model.

The dispatch system used in the paper (as a demonstrating example) is a generic system that provides automatic dispatching of the requested tasks. In this paper, we apply TCOZ to represent two layered architecture model of a generic dispatch system. These two layers include: an top level architectural style of the dispatch system and a generic timed-dispatch system architecture. The generic dispatch system assign the tasks within their critical timing requirements Critical system timing properties of the generic layer can be formulated and verified in TCOZ.

The main benefit of having a layered approach is reusability. The upper layers represent high abstraction of system architecture, i.e., generic patterns of components and connectors, so that high level relationships among system can be understood. The lower layer characterizes the specific domain, i.e., generic timing requirement and specific topology of components and connectors, This allows us to describe a system architecture as an open-ended collection of reusable architectural elements. Formal specifications of architecture models permit us to reason about important properties at each desired level.

The remainder of the paper is organized as follows. In section 2, we briefly introduce the TCOZ notation and its inference rules. Section 3 presents the architecture models of the generic dispatch system . Section 4 presents the verification of critical properties in the system. Section 5 concludes the paper.

## 2 TCOZ

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 TCOZ, all timing information is represented as real valued measurements in e.g. *seconds*. 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 [3] 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.

### 2.1 Interface – 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.

Complementary to the synchronizing 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). We believe that TCOZ with the **actuator** and **sensor** can be a good candidate for specifying open control systems. Mahony and Dong [5] presented a detailed discussion on TCOZ sensor and actuators.

### 2.2 Active objects, semantics and network topologies

Active objects have their own thread of control, while passive objects are controlled by other objects in a system. In TCOZ, an identifier MAIN (non-terminating process) is used to determine the behavior of active objects of a given class.The MAIN operation is optional in a class definition. It only appears in a class definition when the objects of that class are active objects. 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 $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\text{MAIN} \mid\mid\mid ob_2.\text{MAIN}$

The details of the blended state/event process model forms the basis for the TCOZ semantics [4]. 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 modeled 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 syntactic structure of the CSP synchronization operator is more suitable in 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 [6]. 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

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

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

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

and multiple channels above the arrow, for example if processes $D$ and $F$ communicate privately through the channel/sensor-actuator $df_1$ and $df_2$, then

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

## 2.3 TCOZ inference rules

The proof facilities of Object-Z and TCSP can be used to the reasoning of both state and event oriented properties of a TCOZ specification.

### 2.3.1 State aspects reasoning

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 [9] extents the *W* logic of Z to Object-Z in reasoning about object orientation, i.e., the class constructs. The fundamental logic in Object-Z is 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 list 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. A detailed information of Object-Z inference rules can be found in Smith's logic for Object-Z [9].

### 2.3.2 TCOZ extension rules

TCOZ [6] extends Object-Z class definition 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 sensor/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. Based on the logic of Object-Z, additional rules are introduced for the manipulating of the new TCOZ type constructs such as **chan**, **sensor** and **actuator**.

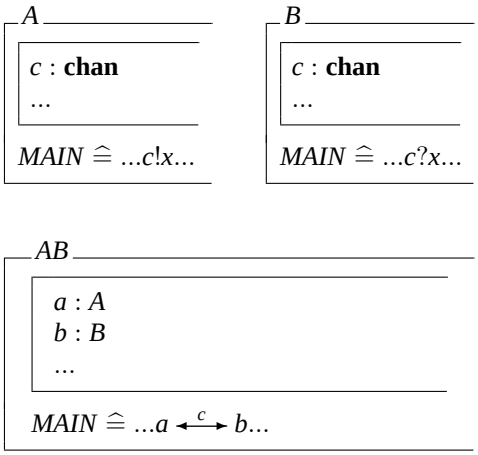
- Non-terminating process (MAIN) – For a generic *MAIN* definition of class $A[X_1, ..., X_n]$, the inference rule is defined as follows:

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

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

  *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 $q \equiv b = (\!|\, X_1 \rightsquigarrow t_1, ..., X_n \rightsquigarrow t_n \,|\!)$.

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

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

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

### 2.3.3 Timed event aspects reasoning

TCSP [7] is an extension to 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 $\aleph$ 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 $Q$.

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

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 operation given the same semantics as process, 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 that regard to TCSP properties are extended to be restricted inside the local environment of a class context as follows:

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

With the above restriction, TCSP inference rules can be directly adopted in the TCOZ class context. A detailed view of TCSP inference rules can be found in Davies/Schneider's proof system for TCSP [7]. Based on the timed failure model of TCSP, additional rules can be defined for the new TCOZ timing constructs such as DEADLINE and WAITUNTIL.

- $Q$ DEADLINE $d$ – The DEADLINE construction allows the successful termination of process $Q$ to be restricted within the $d$ units of time which effect from the beginning of first occurrence in $Q$.

$$\frac{A :: Q \textbf{ sat } S(s, \aleph)}{A :: Q \bullet \text{DEADLINE } d \textbf{ sat } (end(s) \leqslant d \\ \quad \wedge \checkmark \in \sigma(s) \wedge S(s, \aleph \upharpoonright d))}$$

- $Q$ WAITUNTIL $d$ – The WAITUNTIL construction 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 $d$.

$$\frac{A :: Q \textbf{ sat } S(s, \aleph)}{A :: Q \text{ WAITUNTIL } d \textbf{ sat } (end(s) > d \wedge S(s, \aleph) \\ \vee\ end(s) \leqslant d \wedge S(s, \aleph) \wedge \checkmark \textit{ live from d until } \{\checkmark\})}$$

# 3 Dispatch System Architecture Models

## 3.1 Architecture Style

The Concept of the Dispatch System primarily evolved from military community, but now it has been applied to many other areas. Generally, the following should be mentioned: First, the units of the system will be both geographically dispersed and functionally specialized. Second, each of the line units will have critical missions to perform. However, to model an architecture for Dispatch Systems, we need a clearer definition. From a high-level architectural view, the core components and communication of the Dispatch Systems are listed as follows:

- A central control unit manages and dispatches the tasks of the system. This unit makes crucial decisions and assigns tasks to executable resources for engagement against the emergencies. The central controller communicates with all other main units of the system.
- A group of reporting units serve as information collectors for the central controller.
- A group of executing units execute the tasks assigned by the dispatcher. All of them communicate directly with the central dispatcher while working independently from each other.
- A group of auxiliary units assist the central dispatcher or other main units by taking some less important tasks such as collecting and storing auxiliary information.
- A system level configuration acts as a collection of related units which perform the desired functionalities. Note that critical timing requirements are important in the units' computation behavior and their interactive communication.

There are four types of components: report unit, control unit, execute unit and auxiliary unit.

$[ReportInfo]$ [Emergency report type]
$[AuxInfo]$ [Auxiliary information type]
$[Task]$ [Task type]

*ReportUnit*$[X]$

$listenport$ : **chan**
$reportport$ : **chan**
$synauxport$ : **chan**
$asynauxport$ : $X$ **sensor**

$\text{MAIN} \mathrel{\widehat=} \mu R \bullet [r : \mathit{ReportInfo};\ a : \mathit{AuxInfo},\ t : \mathit{Task}] \bullet \mathit{listenport}?(\mathit{self}, r) \rightarrow (\mathit{synauxport}?(\mathit{self}, a) \rightarrow \text{SKIP} \Box \mathit{asynauxport}?(\mathit{self}, a) \rightarrow \text{SKIP});\ \mathit{reportport}!t \rightarrow \text{SKIP};\ R$

*ControlUnit*

$\mathit{reportport} : \textbf{chan}$
$\mathit{dispatchport} : \textbf{chan}$

$\text{MAIN} \mathrel{\widehat=} \mu C \bullet (([t : \mathit{Task}] \bullet \mathit{reportport}?t \rightarrow \text{SKIP}) \Box ([t : \mathit{tasks}, e : \mathit{ExecuteUnit}] \bullet \mathit{dispatchport}!(e, t) \rightarrow \text{SKIP}));\ C$

*ExecuteUnit*

$\mathit{dispachport} : \textbf{chan}$

$\text{MAIN} \mathrel{\widehat=} \mu E \bullet [t : \mathit{Task}] \bullet \mathit{dispatchport}?(\mathit{self}, t) \rightarrow \text{SKIP};\ E$

*AuxiliaryUnit*[$X$]

$\mathit{synauxport} : \textbf{chan}$
$\mathit{asynauxport} : X\ \textbf{actuator}$

$\text{MAIN} \mathrel{\widehat=} \mu A \bullet [a : \mathit{AuxInfo}, r : \mathit{ReportUnit}] \bullet ((\mathit{synauxport}!(r, a) \rightarrow \text{SKIP}) \Box (\mathit{synauxport}!(r, a) \rightarrow \text{SKIP}));\ A$

Note that the communications between *ReportUnit* and *AuxUnit* may be synchronous or asynchronous. Each component has its own interfaces for communication with the rest of the system. For example, the component *ControlUnit* has two communication interfaces: *reportport* and *dispatchport*. When data is received by *ControlUnit* through the *reportport*, it is processed and then sent out through the *dispatchport* after processing. The details of encapsulated behaviors of the components are deliberately suppressed here in the architectural style since each component of the same type may have different computation behaviors. In the MAIN operation of each component, we defines the communication patterns. However, according to TCOZ semantics for inheritance, all MAIN operations will be explicitly redefined when this level of style is inherited and extended for a particular system. In the TCOZ approach, system configurations are specified in network topology.

*DispatchSys*[$X$]

$c :\downarrow \mathit{ControlUnit}$
$rs : \mathbb{F}_1 \downarrow \mathit{ReportUnit}[X]$
$es : \mathbb{F}_1 \downarrow \mathit{ExecuteUnit}$
$as : \mathbb{F} \downarrow \mathit{AuxiliaryUnit}[X]$

$\text{MAIN} \mathrel{\widehat=} \Big|\Big|_{(a,r,e):as \times rs \times es} (a \xleftrightarrow{\mathit{synauxport}, \mathit{asynauxport}} r \xleftrightarrow{\mathit{reportport}} c \xleftrightarrow{\mathit{dispatchport}} e)$

The network topology expression clearly identifies the interaction range of each component. For example, the *ControlUnit* communicates with *ReportUnit*s through the *reportport* while it communicates with *ExecuteUnit*s through the *dispatchport*. Note that '$\downarrow$' is the standard polymorphic type constructor in Object-Z language. More importantly, the TCOZ reuse mechanisms (inheritance and instantiation) allows the architecture style models to be extended to build generic and specific system architecture models, which will be shown in later sub-sections.

### 3.2 A Generic Architecture for the Timed Dispatch System

In this section, we will present a generic architecture of the timed dispatch system specified in TCOZ. We inherit, extend and instantiate the architectural style presented in the previous section. Unlike the style, a generic model defines crucial timing computation and communication details of the components in the system. Following the architectural style, we decompose the system into three main types of components (not including auxiliary components):

- The central dispatcher stores the tasks, updates the tasks and dispatches tasks to related task executers according to the business logic.
- The emergency report receivers obtain emergency information, create tasks and send the tasks to the central dispatcher.
- The task executers execute the tasks dispatched to them. The role of executers may vary in different Dispatch Systems, such as police offices in police system, hospitals in medical system, etc.

*Clock* and *Log*, two auxiliary components in the model, offer time information and logging of important system actions respectively. The subscriber's role also vary in different systems, from patients in the medical system to case locations in the police system. Since most Dispatch Systems are time-critical, we make the timing requirement an important feature in our generic model. Furthermore, some

type variants and common functions were introduced for the purpose of easy customization into specific systems.

The computation behaviors of components are self-encapsulated while implicit connectors are also specified inside relative components. As mentioned previously, a system can be viewed as any one of its components interacting with the rest of the system. Therefore, it is natural for us to study the overall system by analyzing the components individually first.

### 3.3 Clock

In order to record the system information at each particular time, a calendar clock is constructed as follows.

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 elapsed seconds since some reference date, and the function

$$cal : \mathbb{N}\,\mathsf{s} \rightarrowtail CalT$$

... [detail of function omitted]

is used to convert the elapsed seconds to a calendar-time.

*Clock*

$AuxiliaryUnit[CalT][time/asynauxport]$

$total : \mathbb{N}\,\mathsf{s}$

*Inc*

$\Delta(total, time)$

$total' = total + 1\,\mathsf{s}$

$time = cal(total)$

$\textsc{Main} \mathrel{\widehat=} \mu C \bullet (Inc \bullet \textsc{Deadline}\,50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\,1\,\mathsf{s};\ C$

Note that the *time* value increases every second and the display screen updates in less than 50 milliseconds.

### 3.4 System logs

Most CAD Systems require strict persistent repository of data and history log. A generic active object of $Log[X]$ is defined as follows, where $X$ is the data structure type of the records in the log.

*Log*[*X*]

$AuxiliaryUnit[X][record/synauxport]$

$log : \mathrm{seq}\,X$

*Add*

$\Delta(log)$

$x? : X$

$log' = log \frown \langle x? \rangle$

$\textsc{Main} \mathrel{\widehat=} \mu L \bullet [x : X] \bullet record?x \rightarrow Add;\ L$

The system logs consist of two types of logs. One is for the logined reports; and the other is for the dispatched tasks. These can also be customized according to various requirements respectively. The content in the log file is modelled as a variant of type $X$, which varies according to each particular system.

### 3.5 Emergency receiving part

The system receives emergency reports from its environment. Components in this part inherit *ReportUnit* in the style. When the receiving part of the system receives an emergency report, it generates a *Task* from the reported information *ReportInfo* by the function *GenTask* and sends the task to the central dispatcher.

$GenTask : ReportInfo \rightarrow Task$

*Receiver*

$ReportUnit[CalT][listen/listenport, record/synauxport, time/asynauxport, login/reportport]$

$WriteLog \mathrel{\widehat=} [t : CalT;\ r_i : ReportInfo] \bullet time?t \rightarrow record!(t, GenTask(r_i)) \rightarrow \textsc{Skip}$

$\textsc{Main} \mathrel{\widehat=} \mu R \bullet [r_i : ReportInfo] \bullet listen?r_i \rightarrow (login!(GenTask(r_i)) \rightarrow WriteLog);\ R$

### 3.6 Executers

Tasks are dispatched to the executers for execution via the central dispatcher. An assigned log file keep the records of all dispatched tasks.

*Executer*

$ExecuteUnit[dispatch/dispatchport]$

$time : CalT$ **sensor**

$record$ : **chan**

$WriteLog \mathrel{\widehat=} [t : CalT;\ task : Task] \bullet time?t \rightarrow record!(t, task, self) \rightarrow \textsc{Skip}$

$Driven \mathrel{\widehat=} [task : Task] \bullet dispatch?(self, task) \rightarrow WriteLog$

$\textsc{Main} \mathrel{\widehat=} \mu E \bullet Driven;\ E$

### 3.7 Central dispatcher

All tasks will be stored and assigned through the *Dispatcher*. It is the central and crucial part of the system, actively communicating with other parts.

Each task has its own severe level, which means it has its own critical timing requirement. In a generic way, we define a function $Task_T$ to denote the latest time before passing it to an executer.

$Task_T : Task \to \mathbb{T}$

A generic function *pt* (purge task) is defined to purge the time out items from the original set into the second set corresponding to the time elapsed and update the time stamps accordingly:

$$[X]$$
$$pt : (\mathbb{T} \times \mathbb{F}(X \times \mathbb{T})) \to (\mathbb{F}(X \times \mathbb{T}) \times \mathbb{F}\, X)$$
$$\forall t : \mathbb{T};\ s : \mathbb{F}(X \times \mathbb{T}) \bullet pt(t,s) = (\{(e, t_o) : s \mid t_o > t \bullet (e, t_o - t)\},\ \{(e, t_o) : s \mid t_o \leqslant t \bullet e\})$$

e.g.

$$pt(2\,\mathsf{s}, \{(a, 1\,\mathsf{s}), (b, 3\,\mathsf{s}), (c, 7\,\mathsf{s})\}) = (\{(b, 1\,\mathsf{s}), (c, 5\,\mathsf{s})\}, \{a\})$$

which means that after the elapsing of 2 seconds the time stamp of *b* and *c* would become 1 and 5, and the time out item *a* is purged into the second set.

The most critical system component is the *Dispatcher* class:

*Dispatcher*

$ControlUnit[login/reportport, dispatch/dispatchport]$

$ex : \mathbb{F}_1\ Executer$
$tasks : \mathbb{F}(Task \times \mathbb{T})$
$\Delta$
$t : \mathbb{T};\ timeup : \mathbb{F}\ Task$

$tasks \neq \varnothing \Rightarrow 0 \leqslant t \leqslant min\, \mathrm{ran}\, tasks$

INIT

$tasks = \varnothing$

*Add*

$\Delta(tasks)$
$task? : Task;\ \ t_i? : \mathbb{T}$

$task' = \mathrm{fst}(pt(t_i?, tasks)) \cup (task?, Task_T(task?))$
$timeup' = snd(pt(t_i?, tasks))$

*Purge*

$\Delta(tasks)$

$pt(t, tasks) = (tasks', timeup')$

$AddTask \mathrel{\widehat=} [task : (Task - \mathrm{dom}\, tasks);\ t_i : \mathbb{T}] \bullet$
$\quad login?task@t_i \to Add$
$Dispatch \mathrel{\widehat=} [f : timeup \to ex] \bullet$
$\quad |||_{(task,e):f}\ dispatch!(e, task) \to \textsc{Skip}$
$\textsc{Main} \mathrel{\widehat=} \mu D \bullet ([tasks = \varnothing] \bullet AddTask \,\Box\, [tasks \neq \varnothing] \bullet (AddTask \triangleright \{t\}\ (Purge;\ Dispatch)));\ D$

Secondary attribute *t* records the time value which is less than or equal to the minimum time stamp in the task set. This constraint is captured by the class invariant, which must be preserved by all operations. Attribute *timeup* stores all the time-out tasks after each purge operation.

The behavior of the MAIN process of the dispatcher is basically either adding or dispatching tasks. If the task set is empty, only adding is performed; while for the non-empty task set, both adding and dispatching are enabled.

The overall system is a composition of all components that communicate with each other. We organize the interactive components through network topologies.

*TimedDispatchSys*

$DispatchSys[CalT][d/c]$

$clock : Clock$
$inlog : Log[CalT \times Task]$
$dispatchlog : Log[CalT \times Task \times Executer]$

$d \in Dispatcher \wedge d.ex = es$
$\forall r : rs \bullet r \in Receiver$
$\forall e : es \bullet e \in Executer$
$\{clock, inlog, dispatchlog\} \subseteq as$

$$\textsc{Main} \mathrel{\widehat=} \Big|\Big|_{(r,e):rs\times es} (r \overset{login}{\longleftrightarrow} d \overset{dispatch}{\longleftrightarrow} e;\ inlog \overset{record}{\longleftrightarrow} r \overset{time}{\longleftrightarrow} clock \overset{time}{\longleftrightarrow} e \overset{record}{\longleftrightarrow} dispatchlog)$$

## 4 Analysis and Reasoning

From a safety critical perspective, the key point of the Dispatch system is to provide guaranteed time critical service to all the valid tasks. This critical property can be formally interpreted from our model as:

**Thm:** $TimedDispatchSys \bullet \forall task_o : Task;\ ct_1 : CalT \bullet (ct_1, task_o) \in \mathrm{ran}\, inlog.log \Rightarrow \exists ct_2 : CalT;\ e : es \bullet (ct_2, task_o, e) \in \mathrm{ran}\, dispatchlog.log \wedge (cal^{\sim}(ct_2) - cal^{\sim}(ct_1)) \leqslant Task_T(task_o)$ [*P*]

The above simply states that any task which logged into the system will be dispatched within its critical time requirement. To prove the validity of the theorem *P*, 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 into the following timed specification in terms of the timed failure model.

**Lemma:** $L_0(s, \aleph) = Clock :: \forall total : \mathbb{N}\,\mathsf{s};\ t_0 : \mathbb{T} \bullet time!cal(total)\ at\ t_0 \Rightarrow time!cal(total')\ at\ (t_0 + 1\,\mathsf{s})$

**Proof:**

Base case: The specification is trivially satisfied by *STOP*.

Assuming the $C$ **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:

$$L_1(s, \aleph) = Clock :: \forall total : \mathbb{N}\,\mathsf{s};\ t_0 : \mathbb{T} \bullet time!cal(total)\ at\ t_0 \Rightarrow time!cal(total')\ at\ [t_0, \infty)$$

$$L_2(s, \aleph) = Clock :: \forall total : \mathbb{N}\,\mathsf{s};\ t_0 : \mathbb{T} \bullet time!cal(total)\ at\ t_0 \Rightarrow time!cal(total')\ at\ [t_0, t_0 + 50\,\mathsf{ms}]$$

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

$$\frac{Clock :: Inc\ \textbf{sat}\ L_1(s, \aleph)}{Clock :: Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}\ \textbf{sat}\ (end(s) \leqslant 50\,\mathsf{ms} \land \checkmark \in \sigma(s) \land L_1(s, \aleph \upharpoonright 50\,\mathsf{ms}))}\ [\ \text{DEADLINE}\ ]$$

$$\frac{Clock :: Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}\ \textbf{sat}\ (end(s) \leqslant 50\,\mathsf{ms} \land \checkmark \in \sigma(s) \land L_1(s, \aleph \upharpoonright 50\,\mathsf{ms}))}{Clock :: Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}\ \textbf{sat}\ L_2(s, \aleph)}\ [\ \textit{Weaken}\ ]$$

$$\frac{Clock :: Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}\ \textbf{sat}\ L_2(s, \aleph)}{Clock :: (Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}) \bullet \text{WAITUNTIL}\ 1\,\mathsf{s}\ \textbf{sat}\ ((end(s) > 1\,\mathsf{s} \land L_2(s, \aleph)) \lor (end(s) \leqslant 1\,\mathsf{s} \land L_2(s, \aleph) \land \checkmark\ live\ from\ 1\,\mathsf{s}\ until\ \{\checkmark\}))}\ [\ \text{WAITUNTIL}\ ]$$

$$\frac{Clock :: (Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}) \bullet \text{WAITUNTIL}\ 1\,\mathsf{s}\ \textbf{sat}\ L_0(s, \aleph)}{Clock :: (Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}) \bullet \text{WAITUNTIL}\ 1\,\mathsf{s}\ \textbf{sat}\ ((\checkmark \notin \sigma(s) \land L_0(s, \aleph \cup [0, \infty) \times \{\checkmark\})) \lor (\exists s_1, s_2 \bullet s = s_1 \frown s_2 \land \checkmark \notin \sigma(s_1) \land begin(s_1) \geqslant t \land L_0(s_1 \frown \langle (t, \checkmark) \rangle, \aleph \upharpoonright t \cup [0, t) \times \{\checkmark\}) \land L_0((s_2, \aleph) - t)))}\ [\ \textit{Weaken}\ ]$$

$$\frac{Clock :: C\ \textbf{sat}\ L_0(s, \aleph)}{Clock :: ((Inc \bullet \text{DEADLINE}\ 50\,\mathsf{ms}) \bullet \text{WAITUNTIL}\ 1\,\mathsf{s});\ C\ \textbf{sat}\ L_0(s, \aleph)}\ [\ \textit{Composition}\ ]$$

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 now can decompose the theorem *P* into state-based properties and event/time-based properties as follows:

- No message lost – This property claims that no tasks will be lost once they are in the system. It can be translated into the statement that any task in the login log would be eventually in the dispatched log:

  **Thm 1:** $TimedDispatchSys :: \forall task : Task \bullet task \in \operatorname{ran}\operatorname{ran} inlog.log \Rightarrow task \in \operatorname{ran}\operatorname{ran} dispatchlog.log$ $[P_1]$

- Dispatching within critical time range – This property claims that all tasks in the system will be dispatched to a execution unit at their required critical time range. It can be translated into the statement that the duration from login the system to its dispatch of each task should be exactly equal to its time requirement $Task_T(task)$:

  **Thm 2:** $TimedDispatchSys :: \forall task : Task;\ t_0 : \mathbb{T};\ e : es \bullet login?task\ at\ t_0 \Rightarrow dispatch!(e, task)\ at\ (t_0, t_0 + Task_T(task))$ $[P_2]$

As from above, theorem *P* can be formally decomposed into data (state-based) property $P_1$ and timing (event-based) property $P_2$, which later can be proved by the combination of Object-Z and TCSP's inference systems respectively.

### 4.1 Proof of $P_1$

First, we use induction to prove the following property holds by the *Dispatcher* class.

**Theorem 3:** $Dispatcher :: \forall task : Task \bullet (task, Task_T(task)) \in tasks \Rightarrow dispatch.(e, task) \in (Executer \times Task)$ $[P_3]$

**Proof:**

Initially: $Dispatcher :: INIT \vdash tasks = \varnothing$, therefore predicate $[P_3]$ holds (trivial).

Assume the pre-state of the operations in class *Dispatcher* is true, which is $[\forall task : Task \bullet (task, Task_T(task)) \in tasks \Rightarrow dispatch.(e, task) \in (Executer \times Task)]$. The post-state of *Dispatcher* is depicted by two kinds of behaviors, *AddTask* and (*Purge*; *Dispatch*), which associated with the timeout constraint as follows:

- If no new task is added after the minimum time stamp of all tasks – *t*, the (*Purge*; *Dispatch*) operation will perform, which will reduce the number of tasks in the *tasks* set. According to the assumption, $[P_3]$ holds for the post-state.

- If a new task is added to the *tasks* set before *t*, by the definition of the *pt* function, the time stamp of this particular task will monotonously decrease as either the

*AddTask* or (*Purge*; *Dispatch*) operation would perform. Thus the task will eventually be purged from the *tasks* set and dispatched to the *Executor*s. Therefore, $[P_3]$ holds for the post-state.

According to the structured induction, **Theorem** $P_3$ is proved.

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

$$\frac{\begin{array}{l} \mathit{TimedDispatchSys} :: \mathit{STATE} \vdash d \in \mathit{Dispatcher} \land \\ \quad rs \in \mathbb{F}_1\, \mathit{Receiver} \land \mathit{inlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task}] \\ \mathit{Receiver} :: \mathit{STATE} \vdash \mathit{listen}, \mathit{login}, \mathit{record} \in \mathbf{chan} \land \\ \quad \mathit{MAIN} \vdash \mathit{listen.task} \in \mathit{Task} \Rightarrow \mathit{login.task} \in \mathit{Task} \\ \qquad \land\ \mathit{record}.(t, \mathit{task}) \in \mathit{CalT} \times \mathit{Task} \\ \mathit{Dispatcher} :: \mathit{STATE} \vdash \mathit{login} \in \mathbf{chan} \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan} \\ \mathit{TimedDispatchSys} :: \mathit{MAIN} \vdash r \in rs \land \\ \quad d \xleftarrow{\mathit{login}} r \xleftarrow{\mathit{record}} \mathit{inlog} \end{array}}{\begin{array}{l} \mathit{Dispatcher} :: \mathit{MAIN} \vdash \mathit{login.task} \in \mathit{Task} \Rightarrow \\ \quad (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks} \ \land \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{MAIN} \vdash \\ \quad \mathit{record}.(t, \mathit{task}) \in \mathit{CalT} \times \mathit{Task} \Rightarrow \\ \qquad (t, \mathit{task}) \in \operatorname{ran} \mathit{log} \end{array}}$$

$$\frac{\begin{array}{l} \mathit{TimedDispatchSys} :: \mathit{STATE} \vdash d \in \mathit{Dispatcher} \land \\ \quad es \in \mathbb{F}_1\, \mathit{Executer} \land \mathit{dipatchlog} \in \\ \quad \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executor}] \\ \mathit{Dispatcher} :: \mathit{MAIN} \vdash \mathit{login.task} \in \mathit{Task} \Rightarrow \\ \quad (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks} \\ \mathit{Dispatcher} :: \mathit{STATE} \vdash \mathit{dispatch} \in \mathbf{chan} \\ \mathit{Dispatcher} :: \ \vdash (\mathit{task}, \mathit{Task}_T(\mathit{task})) \in \mathit{tasks} \Rightarrow \\ \quad \mathit{dispatch}.(e, \mathit{task}) \in \mathit{Executer} \times \mathit{Task} \\ \mathit{Executer} :: \mathit{STATE} \vdash \mathit{dispatch} \in \mathbf{chan} \\ \mathit{DispatchSysyem} :: \mathit{MAIN} \vdash e \in es \land d \xleftarrow{\mathit{dispatch}} e \end{array}}{\mathit{Executer} :: \mathit{MAIN} \vdash \mathit{dispatch}.(e, \mathit{task}) \in \mathit{Executer} \times \mathit{Task}}$$

$$\frac{\begin{array}{l} \mathit{Executer} :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan} \land \mathit{MAIN} \vdash \\ \quad \mathit{dispatch}.(e, \mathit{task}) \in \mathit{Executer} \times \mathit{Task} \Rightarrow \\ \quad \mathit{record}.(t, \mathit{task}, \mathit{self}) \in \mathit{CalT} \times \mathit{Task} \times \mathit{Executer} \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{STATE} \vdash \mathit{record} \in \mathbf{chan} \\ \mathit{DispatchSysyem} :: \mathit{MAIN} \vdash e \in es \land e \xleftarrow{\mathit{record}} \mathit{dispatchlog} \end{array}}{\begin{array}{l} \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{MAIN} \vdash \\ \quad \mathit{record}.(t, \mathit{task}, e) \in \mathit{CalT} \times \mathit{Task} \times \mathit{Executer} \Rightarrow \\ \quad (t, \mathit{task}, e) \in \operatorname{ran} \mathit{log} \end{array}}$$

Thus $P_1$ can be clearly derived from $P_{1.1}$ and $P_{1.2}$ as follows:

$$\frac{\begin{array}{l} \mathit{TimedDispatchSys} :: \mathit{STATE} \vdash \mathit{inlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task}] \\ \quad \land\ \mathit{dipatchlog} \in \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executor}] \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task}] :: \mathit{MAIN} \vdash \\ \quad \mathit{record}.(t, \mathit{task}) \in \mathit{CalT} \times \mathit{Task} \Rightarrow (t, \mathit{task}) \in \operatorname{ran} \mathit{log} \\ \mathit{Log}[\mathit{CalT} \times \mathit{Task} \times \mathit{Executer}] :: \mathit{MAIN} \vdash \\ \quad \mathit{record}.(t, \mathit{task}, e) \in \mathit{CalT} \times \mathit{Task} \times \mathit{Executer} \Rightarrow \\ \quad (t, \mathit{task}, e) \in \operatorname{ran} \mathit{log} \end{array}}{\begin{array}{l} \mathit{TimedDispatchSys} :: \ \vdash \forall\, \mathit{task} : \mathit{Task} \bullet \\ \quad \mathit{task} \in \operatorname{ran} \operatorname{ran} \mathit{inlog.log} \Rightarrow \\ \qquad \mathit{task} \in \operatorname{ran} \operatorname{ran} \mathit{dispatchlog.log} \end{array}}$$

## 4.2 Proof of $P_2$

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

$$\begin{array}{l} P_2(s, \aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : es \bullet \\ \quad \mathit{login}?\mathit{task}\ \mathit{at}\ t_0 \Rightarrow \mathit{dispatch}!(e, \mathit{task})\ \mathit{at} \\ \quad (t_0, t_0 + \mathit{Task}_T(\mathit{task})] \end{array}$$

**Proof:**

Base case: The specification is trivially satisfied by *STOP*.

Assuming the $D$ **sat** $P_2(s, \aleph)$, it is sufficient to show that $([\mathit{tasks} = \varnothing] \bullet \mathit{AddTask} \ \Box\ [\mathit{tasks} \neq \varnothing] \bullet \mathit{AddTask} \rhd \{t\}\ (\mathit{Purge};\ \mathit{Dispatch}));\ D$ **sat** $P_2(s, \aleph)$.

Let $P_{2.1}$,$P_{2.2}$ be two parts of the timeout primitive in $P_2$ as follows:

$$\begin{array}{l} P_{2.1}(s, \aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : es \bullet \\ \quad \mathit{login}?\mathit{task}\ \mathit{at}\ t_0 \Rightarrow t_0 < t \land \mathit{timeup} = \varnothing \land \\ \quad \neg\ (\mathit{dispatch}!(e, \mathit{task})\ \mathit{at}\ t_0) \\ P_{2.2}(s, \aleph) = \mathit{Dispatcher} :: \forall\, \mathit{task} : \mathit{Task};\ t_0 : \mathbb{T};\ e : es \bullet \\ \quad (\mathit{dispatch}!(e, \mathit{task})\ \mathit{at}\ t_0 \Rightarrow t_0 = t \land \mathit{timeup} \neq \varnothing \land \\ \quad \exists\, \mathit{ts} \subseteq \mathit{tasks} \bullet \forall (\mathit{task}_1, t_1), (\mathit{task}_2, t_2) \in \mathit{ts} \bullet \\ \quad t_1 = t_2 = t \land \mathit{Task}_T(\mathit{task}_1) = \mathit{Task}_T(\mathit{task}_2) \end{array}$$

From the definition of function *pt*, *Add* and *Purge* operations, it is trivial to show that $P_{2.1}$ and $P_{2.2}$ are satisfied by *AddTask* and (*Purge*; *Dispatch*) respectively. The proof of $[P_2]$ can be constructed via event reasoning rules as follows:

$Dispatcher :: ([tasks \neq \varnothing] \bullet AddTask)$ **sat** $P_{2.1}(s, \aleph)$
$Dispatcher :: ([tasks \neq \varnothing] \bullet (Purge;\ Dispatch))$
**sat** $P_{2.2}(s, \aleph)$

---

$Dispatcher :: ([tasks \neq \varnothing] \bullet AddTask \triangleright \{t\}\ (Purge;$
$Dispatch))$**sat** $(begin(s) < t \wedge P_{2.1}(s, \aleph))$
$\vee\ (begin(s) \geqslant t \wedge P_{2.1}(\langle\,\rangle, \aleph \upharpoonright t) \wedge$
$P_{2.2}((s, \aleph) - t))$

---

$Dispatcher :: ([tasks \neq \varnothing] \bullet AddTask \triangleright \{t\}\ (Purge;$
$Dispatch))$**sat** $P_2(s, \aleph)$
$Dispatcher :: ([tasks = \varnothing] \bullet AddTask)$ **sat** $P_2(s, \aleph)$

---

$Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing]$
$\bullet\ AddTask \triangleright \{t\}\ (Purge;\ Dispatch))$ **sat** $P_2(s, \aleph)$
$\wedge\ P_2(\langle\,\rangle, \aleph \upharpoonright begin(s))$

---

$Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing]$
$\bullet\ AddTask \triangleright \{t\}\ (Purge;\ Dispatch))$ **sat** $P_2(s, \aleph)$
$Dispatcher :: D$ **sat** $P_2(s, \aleph)$

---

$Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing]$
$\bullet\ AddTask \triangleright \{t\}\ (Purge;\ Dispatch));\ D$ **sat**
$\checkmark \notin \sigma(s) \wedge P_2(s, \aleph \cup [0, \infty) \times \{\checkmark\})$
$\vee\ \exists s_1, s_2, d \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) \wedge$
$P_2(s_1 \frown \langle (d, \checkmark) \rangle, \aleph \upharpoonright d \cup [0, d) \times \{\checkmark\}) \wedge$
$P_2((s_2, \aleph) - d)$

---

$Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing]$
$\bullet\ AddTask \triangleright \{t\}\ (Purge;\ Dispatch));\ D$
**sat** $P_2(s, \aleph)$

According to the recursion induction rule, the behavior specification $P_2(s, \aleph)$ is satisfied, therefore **Theorem** $P_2$ has been proved. Thus the critical requirements of the timed dispatch system architecture are formally verified.

## 5 Conclusions

In this paper, we have applied the integrated formal notation TCOZ [6] as an ADL to the design and verification of an incremental two layer architecture model for the dispatch system architecture, i.e., the style and the timed generalization. The dispatch style captures the most common patterns among the dispatch systems. The generalization layer models the essential functionalities of the timed dispatch systems. Thus new systems are built as variations and customizations of the up-level designs, and the whole architecture is depicted as an open-ended design for reuse. In this paper, we also found that TCOZ could be a potential candidate of Architecture Description Language for the formal specifications of software architecture models. The class constructs in TCOZ are well suited for component declaration. The communication interfaces, i.e., channel, sensor and actuator, act as implicit connecters for modeling the communications between components. The network topology is used for defining the configuration of the system. All these features may provide a more consistent and flexible way of specifying software architectures. Furthermore, in this paper we also demonstrated the verification of architecture properties via formal reasoning. We combined and extended both state-based and event-based proof systems of Object-Z and TCSP for verifying TCOZ specifications. Complex system properties are decomposed into state and event related properties and proved respectively. In summary, the paper demonstrates that integrated formal modeling techniques (i.e. TCOZ) can be a good candidate for modeling and verifying various level descriptions of software architectures.

## References


[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering nd Methodology*, 1997.

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

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

[4] B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*, pages 66–85. Springer-Verlag, June 1999.

[5] B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. *FM'99: World Congress on Formal Methods*, LNCS, pages 1166–1185, Sep 1999.

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

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

[8] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

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

[10] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.