Computer-aided dispatch system family architecture and verification: an integrated formal approach

J. Sun, J.S. Dong, S. Jarzabek and H. Wang

Abstract: Software architecture is an important level of description for software systems. Formal modelling techniques can be used to define and verify software architectures precisely. An integrated formal approach to the architecture modelling and verification of a computer-aided dispatch (CAD) system family, is presented. An incremental three-layer model, that is, architecture style layer, generic system layer and customised system layer, is presented to capture the design of the CAD system family. Critical CAD system properties in the architecture models are formally verified by using the state and event-based proof techniques of the underlying specification language. In summary, it is demonstrated that integrated formal techniques could be a good candidate for modelling and verifying various levels of descriptions of software architectures.

1 Introduction

Software architecture is an important level of description for software systems [1]. It represents the high-level structure of a system, which comprises the definitions of software components involved, the external visible properties of those components and the communications (relationships and constraints) among the components [2]. The current practice of software architecture mainly relies on diagrammatic and textural descriptions. Several architectural description languages (ADL) have been proposed, such as Darwin [3] and Rapide [4]. These ADLs offer approaches to describe software architectures explicitly as hierarchical structures. Formal modelling techniques have also been applied to the software architecture descriptions. The well-defined semantics and syntax make them suitable for precisely specifying and formally verifying software architecture designs. Some researchers [5, 6] have used Z to formalise the computational data/state aspects of software architectures. Allen and Garlan [7] have also applied a CSP-like notation (Wright) [8] to formalise the interactive communication aspects of software architectures. Both approaches are beneficial and provide some formal foundations to software architecture modelling. We believe that the recent advances in the integrated formal methods [9, 10] may provide more promising solutions to the problem.

In this paper, we demonstrate the approach of using the integrated formal notation, that is, Timed Communicating Object-Z (TCOZ) [11], to capture the software architecture modelling and verification of a computer-aided dispatch (CAD) system family. TCOZ builds on the strengths of object-Z [12, 13] in modelling complex data and state with the strengths of TCSP [14, 15] in modelling process control and real-time interactions. It is capable of capturing both the data and computation states of the components, as well as the interactive communication aspects among the components. The class construct in TCOZ is an ideal encapsulation mechanism for composing and extending architecture components. The synchronous and asynchronous communication interfaces in TCOZ are well suited for capturing various interactions between the components. The network topology of TCOZ is a good mechanism to depict the architectural configurations of a system. Furthermore, TCOZ preserves a large part of both the syntax and semantics of the two blending notations, Object-Z and TCSP, hence it can potentially benefit from existing reasoning systems of the two notations. With new additional proof rules for the TCOZ constructs, critical system properties specified in TCOZ architecture models can be formally verified by using state and event-based proof techniques [16]. In this paper, we also demonstrate the formal reasonings towards the verification of a CAD system architecture model.

CAD system is a generic family system that can provide automatic dispatching of the requested tasks within their critical timing requirements. In our research project, 'Software Reuse Framework for Reliable Mission-Critical Systems' one goal was to develop the reuse-based design and development methods of reliable CAD systems (Supported by Singapore–Ontario Joint Research Programme.). We have found that high-level reuse can be best achieved through software architecture models. An effective approach to reuse requires a generic CAD architecture that defines the overall structure and a common base of customisable software assets to be reused across CAD systems. In this paper, we apply TCOZ to represent an incremental three-layered architecture model of the CAD system family [17]. These three layers include the following:

- Style: an architectural style for the CAD system family describes the basic elements and communication patterns in the system.

IEE Proceedings online no. 20050014

doi:10.1049/ip-sen:20050014

Paper first received 8th March 2005 and in revised form 11th January 2006

J. Sun is with the Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand

J.S. Dong and S. Jarzabek are with the Department of Computer Science, School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543, Republic of Singapore

H. Wang is with the Department of Computer Science, University of Manchester M13 9PL, UK

E-mail: j.sun@cs.auckland.ac.nz

• Generalisation: a generic CAD system architecture model is built as a refinement of the style model and specifies a complete architecture description of the system. Critical system properties of the generic layer can be formulated and proved from the architecture specifications.
• Customisation: several specific CAD system architecture models can be derived from the generic model.

The main benefits of having a three-layered approach are reusability, separation of concerns and reliability [18]. The upper layers describe the common patterns among the family systems, that is, generic patterns of components and their interactions, so that high-level relationships among the components could be understood. The lower layers refine the specific requirements within the new domain, that is, specific topology of components and communications, so that new systems can be built as variations and extensions on the existing models. This allows us to describe a system architecture as a collection of reusable architectural elements and their communications. Formal specifications of architecture models permit us to reason about important properties at each desired level, which further leads to reliable system implementations based on the architecture designs.

2 TCOZ features

TCOZ [11] is an integration and extension of the Object-Z [12, 13] and the TCSP [15] formal modelling notations, 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. The primary specification structuring device in TCOZ is the Object-Z class mechanism. In this section, we briefly consider some architecture description aspects of the TCOZ language. A detailed introduction to TCOZ and its TCSP/Object-Z features can be found elsewhere [11]. The formal semantics of TCOZ is also documented in the work of Mahony and Dong [19].

2.1 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 synchronising CSP channel mechanism, TCOZ also adopts a non-synchronising 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 analogue quantities are sampled from, or generated by the digital subsystem) or else within the system (providing a convenient mechanism for describing local communications that do not require synchronisation). The shift from closed to open systems necessitates close attention to issues of control, an area that both Z and CSP are weak [20]. We believe that TCOZ with the actuator and sensor can be a good candidate for specifying open control systems. Mahony and Dong [21] presented a detailed discussion on TCOZ sensor and actuators.

2.2 Network topologies

The syntactic structure of the CSP synchronisation 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 [11]. 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 channel/sensor-actuator df_1 and df_2, then

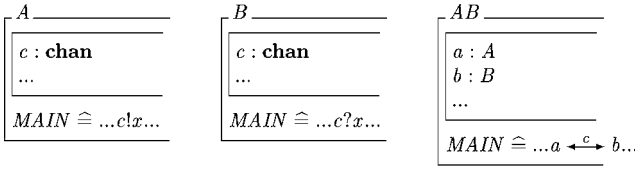
$$\|(D \overset{df_1, df_2}{\longleftrightarrow} F)$$

2.3 TCOZ inference rules

TCOZ preserves a large part of both the syntax and semantics of Object-Z and TCSP, hence can potentially benefit from existing reasoning systems of the individual notations. With new additional inference rules for the TCOZ constructs, a proof system for TCOZ [16] can be established to reason about both state and event properties of a TCOZ specification.

2.3.1 State-oriented reasoning: TCOZ extends Object-Z state aspects in two ways. First, the state schema convention is extended to allow the declaration of object communication interfaces that is, channels, sensors and actuators. The second extension is that as well as operations (terminating processes), non-terminating processes named MAIN are introduced to represent the behaviour of active classes. On the basis of Smith's [22] extended W logic of Object-Z, additional inference rules can be introduced for the reasoning of the new TCOZ state constructs such as MAIN, chan, sensor and actuator [16]. For example, TCOZ channel-state reasoning rule can be

defined as follows:



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

The above defines that if classes A and B are communicating through channel c, synchronisation will be enforced on the input and outputs, that is, outputs from A through c will lead to the same typed inputs to B. The proviso q defines the substitution in the form of $q \equiv b = (|X_1 \rightsquigarrow t_1, \ldots, X_n \rightsquigarrow t_n|)$.

2.3.2 Event-oriented reasoning: 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 [15] primitives are adopted in the class constructs with satisfaction of the timed failure model that is 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, which regard to TCSP properties are extended to be restricted inside the local environment of a class context as follows.

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

It states that in a local class context, a process Q meets a specification $S(s, \aleph)$ if S holds for every timed failure $(s, \aleph)$ associated with Q. On the basis of the Davies/Schneider's proof system for TCSP [15], additional inference rules can be defined for the new TCOZ communication constructs such as DEADLINE, WAITUNTIL and Network Topology [16]. For example, TCOZ DEADLINE reasoning rule can be presented as follows.

$$\frac{A :: Q \textbf{ sat } S(s, \aleph)}{A :: Q \bullet \text{DEADLINE } d \textbf{ sat } (end(s) \leqslant d \wedge \checkmark \in \sigma(s) \wedge S(s, \aleph \upharpoonright d)) \vee (end(s) > d \wedge S(\langle\rangle, (d, \infty) \times \Sigma^{\checkmark}))}$$

The above states that the DEADLINE process behaviours are the same as the original process Q, but is constrained to terminate no later than d. If it fails to terminate by time d, it deadlocks. In this subsection, we briefly introduce the proof system of TCOZ. For a detailed view of TCOZ inference rules, please refer to the TCOZ reasoning paper [16].

3 CAD system family and architecture style

3.1 Overview of CAD system family

CAD systems represent a family system that can provide automatic dispatching of the requested tasks according to their critical timing constraints. CAD systems are used by the Police, Fire & Rescue, Health Service and in many other contexts. Fig. 1 depicts a basic operational scenario as well as the roles and elements of a CAD system for the Police. An Operator receives information about an incident and informs a task Dispatcher. The Dispatcher examines the 'Situation Display' that shows a map of the area where the incident happened. Then, the Dispatcher assigns a task of handling the incident to a Police Unit, for example, a Police Car that is closest to the place of incident. The Police Unit approaches the place of incident and handles the problem. The information about current and past incidents is stored in the database.

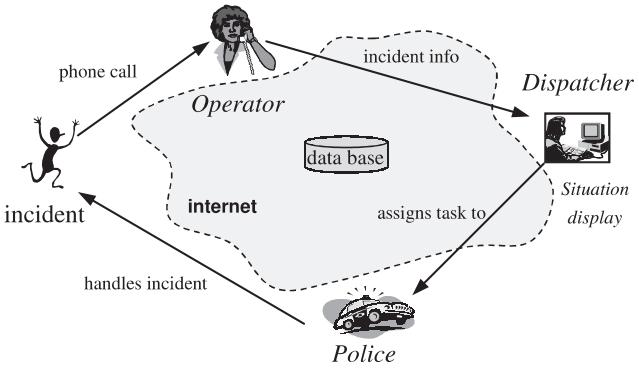


Fig. 1 *Operational scenario in CAD system for Police*

At the basic operational level, CAD systems for Fire & Rescue or Health Services are similar to CAD for Police – basically, all these systems support the dispatch of units to incidents. However, there are also differences across those CAD systems. The specific context of the operation (such as Police or Fire & Rescue) results in many variations on the basic operational scheme. For example, CAD systems differ in rules of how resources are assigned to tasks, monitoring, reporting and timing requirements, specific information to be stored in a database, system component deployment strategies, reliability and availability requirements and so on. If we ignore commonalities, each CAD system must be developed from scratch and maintained as a separate product – an expensive and inefficient solution. However, a reuse-based approach may radically cut development and maintenance cost. An effective approach to reuse requires a generic CAD architecture that defines the overall structure and a common base of customisable software assets to be reused across the CAD systems. CAD systems mentioned above form an important Product Line developed by our industrial partner Singapore Engineering Software Pte Ltd. However, we can further extend the domain analysis [23] and view CAD systems as instances of a general task-resource allocation problem. Then we can observe a similar pattern in the CAD systems mentioned above and the Teleservice and Remote Medical Care System (TRMCS) [24] that supports transition patients from hospital care to home care. In fact, in our examples of illustrating CAD architecture specifications, we shall show how a CAD system for TRMCS can be derived from a common generic CAD architecture model.

3.2 CAD system components and connectors

In architectural descriptions, the three basic elements are components, connectors and configuration (structure) of the system [5]. An architectural style defines the properties that are shared by a family of systems. A style concentrates on the commonalities of communication interfaces, interaction mechanisms and architectural configurations of a family of systems but ignores the details of component functionalities and communications.

We have encountered many CAD systems in our project, 'Software Reuse Framework for Reliable Mission-Critical Systems'. From a high-level architectural view, the core components and communication of these CAD systems are listed as follows:

- Report unit: A group of reporting units serve as information collectors for the central control unit.
- Control unit: 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.
- Execute unit: A group of executing units execute the tasks assigned by the control unit. All of them communicate directly with the central control unit while working independently from each other.
- Auxiliary unit: A group of auxiliary units assist the central control unit or other main units by taking some less important tasks such as collecting and storing auxiliary information.
- CAD system style: A system level configuration acts as a collection of related units that perform the desired functionalities. An UML class representation of the components is depicted in Fig. 2.

As pointed out by Garlan and Perry [25], components are the primary elements for computation in an architecture description. Each component has an interface specification that defines its properties, which include the signature and functionality of its resources together with global relation, performance properties and so on [26]. TCOZ views components in terms of internal computations and interactions with the rest of the system. The internal computations are context-independent, encapsulated behaviours of the components, whereas the context-setting interaction patterns are accomplished by the communication interfaces.

Connector plays a very important role in describing the communication patterns between the interactive components in an architecture description [27]. In our approach, we use implicit connectors to model and encapsulate the corresponding communication patterns inside a component. TCOZ provides a fixed set of connector types to describe component interactions, that is, chans for handling synchronous communication and sensor/actuators for handling asynchronous communication. At the configuration level, system components are attached (connected) together through the TCOZ Network Topology construct, which establishes the overall architecture structure. We can formally specify the above components and style in TCOZ as follows.

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

ReportUnit$[X]$

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

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

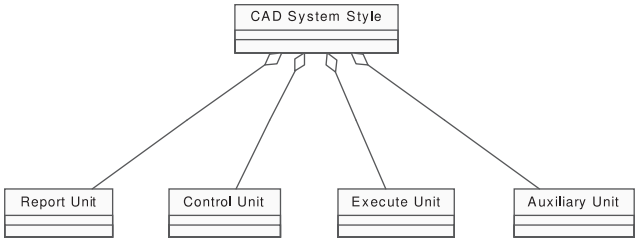


Fig. 2 *CAD system style components*

Note that in the architecture style level the focus is on the identification of the commonalities of components and their communication interfaces. As from the above, the interaction (communication) behaviour of the ReportUnit is captured by the implicit connectors and the non-terminating process Main in the active object [28] (The μ operator indicates a recursive definition of a non-terminating process.). The ReportUnit collects the device information from the synchronous input channel listenport (e.g. phones, monitors, alarms etc. for reporting the incidents) and some additional information from both auxiliary synchronous input channel synauxport and asynchronous input sensor asynauxport (e.g. locations, time, etc. determined from the reports); generates reporting information and pass it through the synchronous output channel reportport to the Control Unit for the purpose of dispatch.

ControlUnit

$reportport$: **chan**
$dispatchport$: **chan**

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

The above describes the basic communication pattern related to the ControlUnit component. The ControlUnit receives the reporting information from the synchronous input channel reportport; generates proper tasks and dispatches them through the synchronous output channel dispatchport to the ExecuteUnit for the purpose of execution.

ExecuteUnit

$dispatchport$: **chan**

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

The ExecuteUnit receives the dispatched task information from the synchronous input channel dispatchport and performs the actual task execution.

AuxiliaryUnit$[X]$

$synauxport$: **chan**
$asynauxport$: X **actuator**

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

The AuxiliaryUnit provides the addition information to other components through the synchronous output channel synauxport and asynchronous output actuator asynauxport. Note that the communications in the AuxUnit may be synchronous or asynchronous, so we give two options in the style.

Each component has its own implicit connectors for communication with the rest of the system. The details of

encapsulated behaviours of the components are deliberately suppressed here in the architectural style, as each component of the same type may have different computation behaviours. In the MAIN operation of each component, we only define the communication patterns.

3.3 CAD system configuration and style

A configuration is a collection of interacting component instances and their attachments in the system. The instances of components are distinguished from the component types. An architectural style defines the common properties of a family of systems, which are shared by any configuration in the style. In our TCOZ approach, configurations are specified by the Network Topology construct in a system component and act as explicit attachments of the interactive components that contain matching implicit connectors in the communication.

$$
\begin{array}{l}
CADStyle[X] \\
\quad c :\downarrow ControlUnit \\
\quad rs : \mathbb{F}_1 \downarrow ReportUnit[X] \\
\quad es : \mathbb{F}_1 \downarrow ExecuteUnit \\
\quad as : \mathbb{F} \downarrow AuxiliaryUnit[X] \\
\text{MAIN} \mathrel{\widehat=} \|_{(a,r,e):as\times rs\times es} (a \overset{synauxport,asynauxport}{\longleftrightarrow} r; \\
\qquad\qquad r \overset{reportport}{\longleftrightarrow} c \overset{dispatchport}{\longleftrightarrow} e)
\end{array}
$$

In the example above, all components comprise a CAD system style. The Network Topology construct in the MAIN operation clearly identifies the interactions among the various components in the system, where the lines connecting the components indicate the interactive communication relationships between the components and the labels on the lines correspond to the implicit connectors (communication interfaces) used. For example, the auxiliary units communicate with the report units through the synauxport channel and asynauxport sensor/actuator; the report units communicate with the control unit through the reportport channel; the control unit communicates with the execution unit through the dispatchport channel. These object interactions through the communication interfaces can also be visualised using the UML diagram as in Fig. 3 (UML collaboration diagrams are used to give a better and visual understanding of the communication patterns specified by the TCOZ network topology in the formal models.).

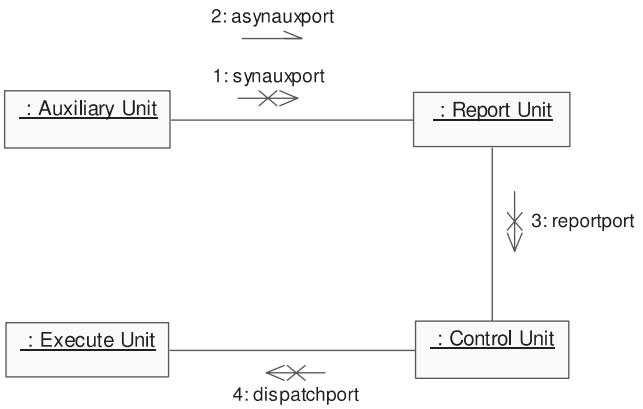


Fig. 3 *CAD system style communication*

4 Generic architecture for CAD system family

In this section, we present a CAD system generic architecture specified in TCOZ, which extends the architectural style presented in the previous section. Unlike the style, a generic model defines crucial computation and communication details of the components in the CAD system family. On the basis of the ReportUnit, ControlUnit and ExecuteUnit in the architectural style, we further refine a generic CAD system into three main types of components:

- The emergency report receivers: obtain emergency information, create detailed tasks and send the tasks to the central dispatcher. This component extends the ReportUnit from the CAD style.
- The central dispatcher: stores, updates and dispatches tasks to related task executers according to the business logic, such as timing constraints. This component extends the ControlUnit from the CAD style.
- The task executers: execute the tasks that are dispatched to them. The role of executers may vary in different CAD systems, such as Police offices in Police system, hospitals in medical system and so on. This component extends the ExecuteUnit from the CAD style.

The hierarchical structure can be illustrated in Fig. 4. Note that the Clock and Log are two auxiliary components extended from the AuxiliaryUnit in the style model. They offer time information and logging of important system actions, respectively.

The subscriber's roles (potential users of specific CAD systems) also vary in different systems, from patients in the medical system to case locations in the Police system. As most CAD 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 customisation into specific CAD systems. The computation behaviours of components are self-encapsulated and the implicit connectors are specified inside corresponding components. As mentioned previously, a system can be viewed as any one of its components interacting with the rest of the system through the Network Topology attachments. Therefore it is natural for us to study the overall system by analysing the components individually first.

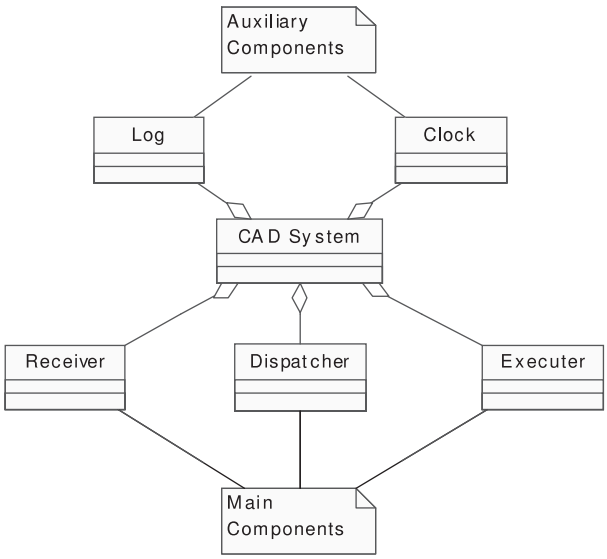


Fig. 4 *Overall structure of a generic CAD system*

4.1 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

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

The clock stores the total elapsed seconds as 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

$\Delta(total, time)$

$total' = total + 1\,\mathsf{s} \land time = cal(total)$

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

The Clock component extends the AuxiliaryUnit in the CAD style, where its asynchronous actuator asynauxport is renamed to time and generic type X is substituted by the calendar-time type Cal$T$. Note that the time value increases every second and the display screen updates in less than 50 ms.

4.2 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 Log component extends the AuxiliaryUnit in the CAD style, where its synchronous channel synauxport is renamed to record. The system logs consist of two types of logs. One is for the incoming incident reports; and the other for the dispatched tasks. These can also be customised 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.

4.3 Emergency receiving unit

The system receives emergency reports from its environment. When the receiving unit 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 for processing.

$$GenTask : ReportInfo \rightarrow Task$$

... [detail of function omitted]

Receiver

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

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

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

The Receiver component extends the ReportUnit in the CAD style, where its synchronous channel listenport is renamed to listen, synchronous channel synauxport is renamed to record, asynchronous sensor asynauxport is renamed to time, synchronous channel reportport is renamed to login and generic type X is substituted by the calendar-time type Cal$T$. The behaviour of the Receiver is to collect the emergency information from the synchronous input channel listen (e.g. phones, monitors, alarms etc. for reporting the incidents); generate task information and pass it through the synchronous output channel login to the Dispatcher for the dispatch purpose, and at the same it records the login information into the system log by the WriteLog operation. While recording to log file, it obtains the time information from the asynchronous input sensor time and passes the log information through the synchronous output channel record for the repository purpose.

4.4 Central dispatcher

All tasks will be stored and assigned through the Dispatcher. It is the central and crucial unit of the system, actively communicating with other components. Each incoming task has its own severe level, which means it has its own critical timing requirement for dispatching, for example, a Fire & Rescue CAD system requires a fire rescue unit to be sent to the location 5 min after the report of the incident. In a generic way, we define a function Task_T to denote the latest timing constraint before passing a task to an executer.

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

... [detail of function omitted]

A generic function pt is defined to purge the timeout items from the original set into the second set corresponding to the time elapsed and update the time stamps accordingly.

[X]

$pt : (\mathbb{T} \times \mathbb{F}(X \times \mathbb{T})) \rightarrow (\mathbb{F}(X \times \mathbb{T}) \times \mathbb{F}\, X)$

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

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

For example, the above means that after the elapsing of 2 s, the time stamp of b and c would become 1 and 5 s, respectively, and the time out item a is purged into the second set. The central Dispatcher component is

defined as follows:

Dispatcher

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

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

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

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

$\Delta(tasks)$

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

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

The Dispatcher component extends the ControlUnit in the CAD style, where its synchronous channel reportport is renamed to login and synchronous channel dispatchport is renamed to dispatch. The behaviour of the Dispatcher is to receive the task-login information from the synchronous input channel login and dispatch the tasks according to their critical timing requirements through the synchronous output channel dispatch to the execute units for the purpose of execution.

The secondary attribute t records the time value that 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 timeout tasks after each purge operation. The behaviour of the MAIN process of the dispatcher is basically either adding or dispatching tasks. If the task set is empty, only adding is performed; whereas for the non-empty task set, both adding and dispatching are enabled. A Purge process is placed when element(s) of task is timed out. A Dispatch operation is defined (in a flexible way, i.e. any function f) to assign every timeout task to an execution unit in parallel.

Note that the TCSP expression in this general form $a@t \rightarrow P(t)$ is a process primitive, where a denotes the event initially enabled by the process and t denotes the timing relative to the occurrence of event a. The expression $(a \rightarrow P) \rhd \{t\} Q$ describes the timed interrupt primitive, where the process will try to perform $a \rightarrow P$ and would pass control to Q if the event a has not occurred by time t. According to this semantic, when tasks $\neq \varnothing$, if the operation AddTask (when $t_i < t$) is performed, right after the operation, timeup $= \varnothing$ must hold because of the definition of the function pt and class invariant $0 \leq t \leq$ min ran tasks (simplified when tasks $\neq \varnothing$). This is the reason for designing MAIN with Dispatch operation only after Purge, which means that the dispatch will happen exactly at the corresponding timing requirement of each task. It is reasonable to assume that the time durations t_a, t_d and t_p of the operations AddTask, Dispatch and Purge are far less than t or t_i (as t_a, t_d, $t_p \ll t$, t_i). For instance, t could be in the scale of seconds and t_a might be in microseconds. In contrast, if the time durations such as t_a are considered, the AddTask schema can be modified accordingly.

4.5 Executers

Tasks are dispatched to the executers for execution by the central dispatcher. A dispatch log file keep the records of all dispatched tasks. The task Executer component is defined as follows.

Executer

$ExecuteUnit[dispatch/dispatchport]$

$time : CalT$ **sensor**
$record :$ **chan**

$WriteLog \mathrel{\widehat=} [t : CalT;\ task : Task] \bullet time?t \rightarrow$
$record!(t, task, self) \rightarrow \mathrm{SKIP}$
$Driven \mathrel{\widehat=} [task : Task] \bullet dispatch?(self, task) \rightarrow WriteLog$
$\mathrm{MAIN} \mathrel{\widehat=} \mu E \bullet Driven;\ E$

The Executer component extends the ExecuteUnit in the CAD style, where its synchronous channel dispatchport is renamed to dispatch. The behaviour of the Executer is to receive the dispatched task from the synchronous input channel dispatch; execute it and record the dispatched information into the system log by the WriteLog operation. While recording to log file, it obtains the time information from the asynchronous input sensor time and passes the log information through the synchronous output channel record for the repository purpose.

4.6 Generic system architecture configuration

The overall system configuration is a composition of all components that communicate with each other. We organise the interactive relationships through the TCOZ network topologies. This system component CADSystem plays the role of explicitly connecting the implicit connectors inside each corresponding components to establish the configuration topology of the overall architecture structure.

CADSystem

$CADStyle[CalT][d/c]$

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

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

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

The CADSystem component extends the CADStyle component in the CAD style, where its ControlUnit object c is renamed to the Dispatcher object d and generic type X is substituted by the calendar-time type Cal$T$. New instances of auxiliary components such as clock, inlog and dispatchlog are introduced to the system together with the constraints upon them. From a communication point of view, the CADSystem specifies that the receiver communicates with the dispatcher through the login channel; the dispatcher communicates with the executer through the dispatch channel; the receiver communicates with the clock through the time sensor/actuator; the receiver communicates with the input log file through the record channel; the executer communicates with the clock through the

time sensor/actuator; the executer communicates with the dispatch log file through the record channel. Similarly, we could also use UML collaboration diagram to visualises the configuration of the system defined in the formal model as in Fig. 5.

5 CAD system architecture verification

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

Theorem:

$$\begin{aligned} &CADSystem :: \forall task_o : Task; ct_1 : CalT \bullet \\ &(ct_1, task_o) \in \text{ran } inlog.log \Longrightarrow \exists ct_2 : CalT; e : es; \bullet \\ &(ct_2, task_o, e) \in \text{ran } dispatchlog.log \\ &\wedge (cal^{\sim}(ct_2) - cal^{\sim}(ct_1)) = Task_T(task_o) \end{aligned} \qquad [P]$$

The theorem simply states that any task which logged into the system will be dispatched at its critical time requirement. In order to prove the validity of the Theorem P, the first thing is to show that the Clock component in the system correctly models the behaviour of a physical timing device – the global clock. This property can be interpreted into the following timed failure specification.

Lemma:

$$\begin{aligned} L_0(s, \aleph) = Clock :: \forall total : \mathbb{N}\,\mathsf{s}; t_0, t_1 : \mathbb{T} \bullet \\ time!cal(total)\ live[t_0, t_1) \Longrightarrow (t_1 - t_0 = 1\mathsf{s}) \end{aligned}$$

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

$$\begin{aligned} a\ live[t_1, t_2) = \forall t \in [t_1, t_2) \bullet a\ at\ t \wedge \forall t_i : \mathbb{T} \bullet \\ (t_i < t_1 \Longrightarrow \neg(a\ at\ t_i) \wedge t_i \geq t_2 \\ \Longrightarrow \neg\,(a\ at\ t_i)) \end{aligned}$$

This macro simply expresses that the event a is continuously recorded in the trace as having occurred at every point on a maximal interval I, where I is in the form of $[t_1, t_2)$.

Proof: Base case: The specification is trivially satisfied by *STOP*. Assuming the C sat $L_0(s, \aleph)$, it is sufficient to show that

$$(Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\ 1\,\mathsf{s} \fatsemi^{\circ} C\ \mathbf{sat}\ L_0(s, \aleph)$$

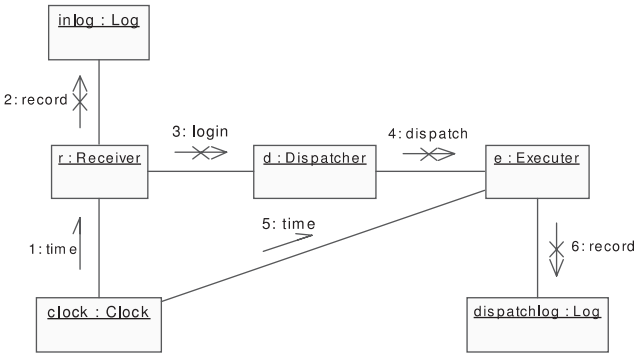


Fig. 5 *Configuration of a generic CAD system*

Let

$$\begin{aligned} L_1(s, \aleph) = Clock :: \forall total : \mathbb{N}\,\mathsf{s}; t_0, t_1 : \mathbb{T} \bullet \\ time!cal(total)\ live\ [t_0, t_1) \Longrightarrow (t_1 - t_0 \in [0, \infty)) \end{aligned}$$

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Clock :: Inc\ \mathbf{sat}\ L_1(s, \aleph)}{Clock :: Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}\ \mathbf{sat}\ (end(s) \leqslant 50\,\mathsf{ms} \wedge \checkmark \in \sigma(s) \wedge L_1(s, \aleph \restriction 50\,\mathsf{ms})) \vee (end(s) > 50\,\mathsf{ms} \wedge L_1(\langle\rangle, (50\,\mathsf{ms}, \infty) \times \Sigma^{\checkmark}))}\ [Deadline]}{Clock :: Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}\ \mathbf{sat}\ L_2(s, \aleph)}\ [Weaken]}{Clock :: (Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\ 1\,\mathsf{s}\ \mathbf{sat}\ ((end(s) > 1\,\mathsf{s} \wedge L_2(s, \aleph)) \vee (end(s) \leqslant 1\,\mathsf{s} \wedge L_2(s \frown \langle 1\,\mathsf{s}, \checkmark\rangle, \aleph \cup [end(s), 1\,\mathsf{s}) \times \Sigma^{\checkmark})))}\ [WaitUntil]}{Clock :: (Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\ 1\,\mathsf{s}\ \mathbf{sat}\ L_0(s, \aleph)}\ [Weaken]}{\dfrac{Clock :: C\ \mathbf{sat}\ L_0(s, \aleph)}{Clock :: ((Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\ 1\,\mathsf{s}) \fatsemi C\ \mathbf{sat}\ \checkmark \notin \sigma(s) \wedge L_0(s, \aleph \cup [0, \infty) \times \{\checkmark\}) \vee \exists s_1, s_2, t \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) \wedge L_0(s_1 \frown \langle(t, \checkmark)\rangle, \aleph \restriction t \cup [0, t) \times \{\checkmark\}) \wedge L_0((s_2, \aleph) - t)}\ [Sequential]}\ [Weaken]}{Clock :: ((Inc \bullet \textsc{Deadline}\ 50\,\mathsf{ms}) \bullet \textsc{WaitUntil}\ 1\,\mathsf{s}) \fatsemi C\ \mathbf{sat}\ L_0(s, \aleph)}\ [Weaken]$$

Note that we construct the proofs by applying TCOZ inference rules at each step. For a detailed view of TCOZ inference rules, please refer to the TCOZ reasoning paper [16]. From the above, according to the recursion induction rule, the behaviour specification $L_0(s, \aleph)$ is satisfied, therefore Lemma L_0 has been proved. □

After showing that the Clock component is consistent with the global clock, we are now ready to prove the correctness of Theorem P. First, Theorem P can be rewritten into state-based 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

Theorem 1:

$$\begin{aligned} &CADSystem :: \forall\ task : Task \bullet \\ &task \in \text{ran ran } inlog.log \\ &\Longrightarrow task \in \text{ran ran } dispatchlog.log \end{aligned} \qquad [P_1]$$

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

Theorem 2:

$$\begin{aligned} &CADSystem :: \forall\ task : Task; t_0 : \mathbb{T}; e : es \bullet \\ &login?task\ at\ t_0 \\ &\Longrightarrow dispatch!(e, task)\ at\ (t_0 + Task_T(task)) \end{aligned} \qquad [P_2]$$

As from above, Theorem P can be formally translated into a data (state-based) property P_1 and a timing

(event-based) property P_2, which later can be proved by the TCOZ inference rules.

5.1 Proof of Theorem P_1

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

Lemma:

$$Dispatcher :: \forall\ task : Task \bullet (task, Task_T(task)) \in tasks \implies dispatch.(e, task) \in (Executer \times Task) \qquad [P_{1.1}]$$

Proof: Initially: Dispatcher :: INIT $\vdash$ tasks $= \varnothing$, therefore predicate $[P_{1.1}]$ 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 types of behaviours, that is, AddTask and (Purge $\overset{\circ}{9}$ Dispatch), which are associated with the timeout constraint as follows:

- If no new task is added after the minimum time stamp of all tasks t, the (Purge $\overset{\circ}{9}$ Dispatch) operation will perform, which will reduce the number of tasks in the tasks set. According to the assumption, $[P_{1.1}]$ 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 decrease in a monotonic manner as either the AddTask or (Purge $\overset{\circ}{9}$ Dispatch) operation would perform. Thus the task will eventually be purged from the tasks set and dispatched to the Executers. Therefore $[P_{1.1}]$ holds for the post-state.

According to the induction rule, Lemma $P_{1.1}$ is proved. Thus the proof of $[P_1]$ can be constructed via state reasoning rules as follows.

$$\frac{\begin{array}{l} CADSystem :: STATE \vdash d \in Dispatcher \wedge rs \in \mathbb{F}_1\, Receiver \\ \quad \wedge\ inlog \in Log[CalT \times Task] \\ Receiver :: STATE \vdash listen, login, record \in \mathbf{chan} \wedge \\ \quad MAIN \vdash listen.task \in Task \Rightarrow login.task \in Task \\ \quad\quad \wedge\ record.(t, task) \in (CalT \times Task) \\ Dispatcher :: STATE \vdash login \in \mathbf{chan} \\ Log[CalT \times Task] :: STATE \vdash record \in \mathbf{chan} \\ CADSystem :: MAIN \vdash r \in rs \wedge d \xleftarrow{login} r \xleftarrow{record} inlog \end{array}}{\begin{array}{l} Dispatcher :: MAIN \vdash login.task \in Task \Rightarrow \\ \quad (task, Task_T(task)) \in tasks \ \wedge \\ Log[CalT \times Task] :: MAIN \vdash record.(t, task) \in \\ \quad (CalT \times Task) \Rightarrow (t, task) \in \operatorname{ran} log \end{array}} [\ Channel\ ] \qquad [P_{1.2}]$$

$$\frac{\begin{array}{l} CADSystem :: STATE \vdash d \in Dispatcher \wedge es \in \mathbb{F}_1\, Executer \\ \quad \wedge\ dispatchlog \in Log[CalT \times Task \times Executer] \\ Dispatcher :: MAIN \vdash login.task \in Task \Rightarrow \\ \quad (task, Task_T(task)) \in tasks \\ Dispatcher :: STATE \vdash dispatch \in \mathbf{chan} \\ Dispatcher :: \ \vdash (task, Task_T(task)) \in tasks \Rightarrow \\ \quad dispatch.(e, task) \in (Executer \times Task) \qquad [P_{1.1}] \\ Executer :: STATE \vdash dispatch \in \mathbf{chan} \\ CADSystem :: MAIN \vdash e \in es \wedge d \xleftarrow{dispatch} e \end{array}}{\begin{array}{l} Executer :: MAIN \vdash dispatch.(e, task) \in (Executer \times Task) \\ Executer :: STATE \vdash record \in \mathbf{chan} \wedge \\ \quad MAIN \vdash dispatch.(e, task) \in (Executer \times Task) \Rightarrow \\ \quad\quad record.(t, task, self) \in (CalT \times Task \times Executer) \\ Log[CalT \times Task \times Executer] :: STATE \vdash record \in \mathbf{chan} \\ CADSystem :: MAIN \vdash e \in es \wedge e \xleftarrow{record} dispatchlog \end{array}} [\ Channel\ ]$$

$$\frac{}{\begin{array}{l} Log[CalT \times Task \times Executer] :: MAIN \vdash \\ \quad record.(t, task, e) \in (CalT \times Task \times Executer) \\ \quad\quad \Rightarrow (t, task, e) \in \operatorname{ran} log \end{array}} \qquad [P_{1.3}]$$

Therefore Theorem P_1 can be clearly derived from $P_{1.2}$ and $P_{1.3}$ above as follows.

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

□

5.2 Proof of Theorem P_2

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

$$\begin{aligned} &P_2(s, \aleph) \\ &\quad = Dispatcher :: \forall\ task : Task; t_0 : \mathbb{T}; e : es \bullet \\ &\quad login?task at\ t_0 \implies dispatch!(e, task) \\ &\quad at\ (t_0 + Task_T(task)) \end{aligned}$$

Proof: Base case: The specification is trivially satisfied by *STOP*. Assuming the D **sat** $P_2(s, \aleph)$, it is sufficient to show that ([tasks $= \varnothing$] $\bullet$ AddTask $\Box$ [tasks $\neq \varnothing$] $\bullet$ AddTask $\rhd \{t\}$ (Purge $\overset{\circ}{9}$ Dispatch)) $\overset{\circ}{9}$ D **sat** $P_2(s, \aleph)$.

Let $P_{2.1}$, $P_{2.2}$ be two time failure expressions represented as follows.

$$\begin{aligned} P_{2.1}(s, \aleph) = {} & Dispatcher :: \forall\ task : Task; t_0 : \mathbb{T}; e : es \bullet \\ & login?task\ at\ t_0 \implies (t_0 < t \wedge timeup \\ & = \varnothing \wedge \neg(dispatch!(e, task)\ at\ t_0)) \end{aligned}$$

$$\begin{aligned} P_{2.2}(s, \aleph) = {} & Dispatcher :: \forall\ task : Task; t_0 : \mathbb{T} e : es \bullet \\ & (dispatch!(e, task)\ at\ t_0 \\ & \implies (t_0 = t \wedge timeup \neq \varnothing \wedge \\ & \exists ts \subseteq tasks \bullet \forall (task_1, t_1), (task_2, t_2) \in ts \bullet \\ & (t_1 = t_2 = t \wedge Task_T(task_1) = Task_T(task_2)))) \end{aligned}$$

In our model, the behaviour of adding and assigning valid tasks is determined by the functions *pt*, Add and Purge operations in the non-terminating process MAIN of the class Dispatcher. Considering each non-recursive transaction trace of the MAIN process as one execution cycle, the possible actions of the Dispatcher within a cycle are as follows.

$$\begin{aligned} &A_1 : \text{AddTask when tasks} = \varnothing \\ &A_2 : \text{AddTask when tasks} \neq \varnothing \wedge t_i < t \\ &A_3 : \text{Purge } \overset{\circ}{9} \text{ Dispatch when tasks} \neq \varnothing \wedge t_i = t \end{aligned}$$

Therefore it is trivial to show that $P_{2.1}$ and $P_{2.2}$ are satisfied by AddTask and (Purge $\overset{\circ}{9}$ Dispatch), respectively. Therefore the proof of Theorem P_2 can be constructed via

event reasoning rules as follows.

$$\frac{Dispatcher :: ([tasks \neq \varnothing] \bullet AddTask)\ \mathbf{sat}\ P_{2.1}(s, \aleph) \quad Dispatcher :: ([tasks \neq \varnothing] \bullet (Purge \fatsemi Dispatch))\ \mathbf{sat}\ P_{2.2}(s, \aleph)}{Dispatcher :: ([tasks \neq \varnothing] \bullet AddTask \triangleright \{t\}\ (Purge \fatsemi Dispatch))\ \mathbf{sat}\ (begin(s) \leqslant t \wedge P_{2.1}(s, \aleph)) \vee (begin(s) \geqslant t \wedge P_{2.1}(\langle\,\rangle, \aleph \upharpoonright t) \wedge P_{2.2}((s, \aleph) - t))}\ [\ Timeout\ ]$$

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

$$\frac{Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing] \bullet AddTask \triangleright \{t\}\ (Purge \fatsemi Dispatch))\ \mathbf{sat}\ P_2(s, \aleph) \quad Dispatcher :: D\ \mathbf{sat}\ P_2(s, \aleph)}{\begin{array}{l} Dispatcher :: ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing] \bullet AddTask \triangleright \{t\}\ (Purge \fatsemi Dispatch)) \fatsemi D\ \mathbf{sat}\ \checkmark \notin \sigma(s) \\ \quad \wedge P_2(s, \aleph \cup [0, \infty) \times \{\checkmark\}) \\ \quad \vee \exists s_1, s_2, t_i \bullet s = s_1 \frown s_2 \wedge \checkmark \notin \sigma(s_1) \wedge P_2(s_1 \frown \langle (t_i, \checkmark) \rangle, \\ \quad\quad \aleph \upharpoonright t_i \cup [0, t_i) \times \{\checkmark\}) \wedge P_2((s_2, \aleph) - t_i) \end{array}}\ [\ Sequential\ ]$$

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

According to the recursion induction rule, the behaviour specification $P_2(s, \aleph)$ is satisfied, therefore Theorem P_2 has been proved. Thus from the proofs of P_1 and P_2, we can see that the critical timing requirement of the generic CAD system architecture Theorem P is formally verified. □

6 CAD system architecture customisation

A generic system architecture must be easily customisable to meet the requirements of specific systems. The customisation includes customising computation behaviours of components and customising architectural configuration in terms of connectors. There are two common approaches in achieving such customisations. One is to model the generic architecture in as compact a manner as possible, which includes only the intersection parts among all system family members. In this way, specific system architectures can be derived from the generic model through inheriting and expanding the components. The other approach is to cover most common functionalities of the system family in the generic model, and then model specific system architectures through cutting down and modifying relevant components.

The first approach is suitable for system families in which most systems share not only the main structure but also many component behaviour and communication details. The second one, in a sense, is better for the system family in which, among systems, there are only minor differences in architectural configuration while the component inner behaviours are not very interactive. Real-world systems are usually complex and cannot be simply classified into any one of the above two approaches. Therefore the customisation approach might be a blend of the above two approaches. Most CAD systems share common architecture features on a large scale. However, the types and functionalities differ from system to system and need to be specifically redefined into particular systems. We demonstrate the customisation of the generic architecture into specific systems through a TRMCS example.

6.1 CAD system for teleservices and remote medical care

TRMCS [24] provides services for the transition of patients from hospital care to home care. In the TRMC system, the ReportInfo includes the patient's symptoms and the place of the incident.

$$\text{ReportInfo} == \text{Symptom} \times \text{Location}$$

The TRMCS consists of a number of help centres for performing the emergency job execution. For the sake of urgency, a task might be put up for open bid, and the help centres compete to answer it. At the same time, the system must guarantee that at least one help centre responds. Therefore we offer two mechanisms for help centres to be assigned tasks. First, the help centres are aware of what tasks are available at the current time and they can actively select tasks from the dispatcher. Secondly, tasks are passively dispatched to the help centers for execution in the case that some tasks are not selected by any help centre within a certain deadline. Thus the HelpCentre and $\text{Dispatcher}_{\text{TRMCS}}$ components that extend the Executer and Dispatcher components from the generic CAD architecture model can be defined as follows.

HelpCentre
Executer

$d : Dispatcher_{TRMCS}$
$select, choose : \mathbf{chan}$

$Select \mathrel{\widehat=} [task : \mathrm{dom}\, d.tasks] \bullet select?task \rightarrow choose!task \rightarrow dispatch?(self, task) \rightarrow WriteLog$
$\textsc{Main} \mathrel{\widehat=} \mu H \bullet (Select \Box Driven) \fatsemi H$

$Dispatcher_{TRMCS}$
Dispatcher

$choose : \mathbf{chan}$

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

$tasks \neq \varnothing$
$pt(t_i?, task? \lhd tasks) = (tasks', timeup')$

$Assign \mathrel{\widehat=} [task : tasks;\ e : ex;\ t_i : \mathbb{T}] \bullet choose?(e, task)@t_i \rightarrow dispatch!(e, task) \rightarrow Delete$
$\textsc{Main} \mathrel{\widehat=} \mu D \bullet ([tasks = \varnothing] \bullet AddTask \Box [tasks \neq \varnothing] \bullet ((AddTask \Box Assign) \triangleright \{t\}\ (Purge \fatsemi Dispatch))) \fatsemi D$

By customising a task selection function into the system, the TRMCS configuration is modelled with additional communication and extended components as follows.

TRMCSystem
CADSystem

$d \in Dispatcher_{TRMCS}$
$\forall h : es \bullet h \in HelpCentre \wedge h.d = d$

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

Note that the $\text{Dispatcher}_{\text{TRMCS}}$ and HelpCentre components are also communicating through the synchronous channel choose. From the above system architecture, by means of active selection and passive assignment, the tasks are dispatched within their critical timing requirement. Thus, Theorem P is redefined into P' as follows

Theorem 3:

$$TRMCSystem :: \forall\, task_o : Task; ct_1 : CalT \bullet$$
$$(ct_1, task_o) \in \text{ran}\ inlog.log \Longrightarrow \exists ct_2 : CalT; e : es; \bullet$$
$$(ct_2, task_o, e) \in \text{ran}\ dispatchlog.log$$
$$\wedge\, (cal^{\sim}(ct_2) - cal^{\sim}(ct_1)) \leq Task_T(task_o) \qquad [P']$$

The above states that the dispatching of a task should be performed within its timing requirement $\text{Task}_T(\text{task})$ because of active selections, whereas in the generalised CAD system this should perform exactly at $\text{Task}_\text{T}(\text{task})$. Note that Theorem P' can also be proved similarly as demonstrated in Section 5. Hence, the TRMCS architecture is customised from the generic CAD system model into its own specific requirements.

7 Conclusion

In this paper, we have applied the integrated formal notation TCOZ to the design and verification of an incremental three-layer architecture model for the CAD system family, that is, the style, the generalisation and the customisation. The CAD style captures the most common interaction patterns among the CAD family. The generalisation layer models the essential computational functionalities and communication of the CAD systems. The customisation characterises the additional specific requirements within each particular system. Thus new systems are built as variations and customisations of the up-level designs, and the whole family architecture is depicted as an open-ended design for reuse.

We found that TCOZ could be a potential candidate for the formal modelling of the software architecture design. The class constructs in TCOZ are well suited for component definitions. The communication interfaces, that is, channel, sensor and actuator, act as implicit connectors for modelling the communications between components. The network topology is used as explicit attachment for connecting the interactive components in defining the overall 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 applied both state and event-based proof techniques for the verification of TCOZ architecture specifications. Complex system properties are decomposed into state and event-related properties and proved, respectively. In summary, the paper demonstrates that integrated formal modelling techniques (TCOZ) could provide a promising solution for modelling and verifying various levels of software architecture descriptions. As the verification process presented in the paper is still manual-based, we are currently investigating the embedding of TCOZ proof rules into the theorem provers such as Isabelle/HOL [29] for automatic proof assistance. In the future, we also plan to apply our approach to the specification and verification of software product line architectures [30].

8 Acknowledgments

This work was supported by the academic research grant, Integrated Formal Methods (R-252-000-050-107) from the National University of Singapore. We would also like to thank the numerous anonymous referees who have reviewed the manuscript and whose valuable comments have contributed to the clarification of many of the ideas presented in the paper.

9 References

1 Perry, D., and Wolf, A.: 'Foundations for the study of software architecture', *ACM SIGSOFT Softw. Eng. Notes*, 1992, **17**, pp. 40–52
2 Bass, L., Clements, P., and Kazman, R.: 'Software architecture in practice' (Addison-Wesley, Reading, MA, 1998)
3 Magee, J., Dulay, N., Eisenbach, S., and Kramer, J.: 'Specifying distributed software architectures'. Proc. 5th European Software Engineering Conf., 1994
4 Luckham, D.C., and Vera, J.: 'An event-based architecture definition language', *IEEE Trans. Softw. Eng.*, 1995, **21**, (9), pp. 717–734
5 Abowd, G.D., Allen, R., and Garlan, D.: 'Formalizing style to understand descriptions of software architecture', *ACM Trans. Softw. Eng. Methodol.*, 1995, **4**, (4), pp. 319–364
6 Shaw, M., and Garlan, D.: 'Software Architecture: perspectives on an emerging discipline' (Prentice-Hall, 1996)
7 Allen, R., and Garlan, D.: 'A formal basis for architectural connection', *ACM Trans. Softw. Eng. Methodol.*, 1997, **6**, pp. 213–249
8 Allen, R.: 'A formal approach to software architecture'. PhD Thesis, School of Computer Science, Carnegie Mellon, January, 1997 Issued as CMU Technical Report CMU-CS-97-144
9 Araki, K., Galloway, A., and Taguchi, K. (Eds.): 'IFM'99: Integrated formal methods', York, UK (Springer-Verlag, 1999)
10 Grieskamp, W., Santen, T., and Stoddart, B. (Eds.): 'IFM'00: integrated formal methods', Dagstuhl Castle, Germany, *Lect. Notes Comput. Sci.*, 2000
11 Mahony, B., and Dong, J.S.: 'Timed communicating object Z', *IEEE Trans. Softw. Eng.*, 2000, **26**, (2), pp. 150–177
12 Duke, R., and Rose, G.: 'Formal object oriented specification using object-Z', Cornerstones of Computing Series (Macmillan, 2000)
13 Smith, G.: 'The object-Z specification language' 'Advances in formal methods' (Kluwer Academic Publishers, 2000)
14 Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., and Roscoe, A.W.: 'Timed CSP: theory and practice' in de Bakker, J.W., Huizing, C., de Roever, W.P., and Rozenberg, G. (Eds.): 'Real-time: theory in practice', *Lect. Notes Comput. Sci.*, 1992, **600**, pp. 640–675
15 Schneider, S., and Davies, J.: 'A brief history of timed CSP', *Theor. Comput. Sci.*, 1995, **138**, pp. 243–271
16 Sun, J., and Dong, J.S.: 'Reasoning about TCOZ'. Technical Report., TRA3/02, School of Computing, National University of Singapore, March 2002
17 Garlan, D., and Delisle, N.: 'Formal specification of an architecture for a family of instrumentation systems', in Hinchey, M., and Bowen, J. (Eds.): 'Applications of formal methods' (Prentice-Hall, 1995), pp. 55–72
18 van Zyl, J.: 'Product line architecture and the separation of concerns'. in SPLC 2: Proc. 2nd Int. Conf. on Software Product Lines, (Springer-Verlag, 2002), pp. 90–109
19 Mahony, B., and Dong, J.S.: 'Overview of the semantics of TCOZ' in Araki, K., Galloway, A., and Taguchi, K. (Eds.): 'IFM'99: Integrated formal methods', York, UK (Springer-Verlag, 1999), pp. 66–85
20 Zave, P., and Jackson, M.: 'Four dark corners of requirements engineering', *ACM Trans. Softw. Eng. Methodol.*, 1997, **6**, (1), pp. 1–30
21 Mahony, B., and Dong, J.S.: 'Sensors and actuators in TCOZ' in Wing, J., Woodcock, J., and Davies, J. (Eds.): 'FM'99: World Congress on Formal Methods', Toulose, France, September 1999, *Lect. Notes Comput. Sci.*, pp. 1116–1185
22 Smith, G.: 'Extending W for Object-Z' in Bowen, J.P., and Hinchey, M.G. (Eds.): 'Proc. 9th Annual Z-User Meeting' (Springer-Verlag, 1999), pp. 276–295
23 Lung, C., and Urban, J.: 'An approach to the classification of domain models in support of analogical reuse'. Proc. ACM SIGSOFT 1995 Symp. on Software Reusability, (ACM Press, 1995), pp. 169–178
24 Inverardi, P., Muccini, H., Richardson, D., and Ficks, S.: 'The teleservices and remote medical care system (TRMCS)', 2000
25 Garlan, D., and Perry, D.: 'Software architecture: practice, potential and pitfalls'. Proc. 16th Int. Conf. on Software Engineering, May 1994
26 van den Brand, M., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., and Visser, J.: 'The ASF+SDF meta-environment: a component-based language development environment'. Proc. Compiler Construction 2001 (CC 2001), *Lect. Notes Comput. Sci.*, 2001
27 Bishop, J., and Faria, R.: 'Connectors in configuration programming: are they necessary?'. 3rd Int. Conf. on Configurable Distributed Systems, Annapolis, USA, May 1996, pp. 11–18
28 Dong, J.S., and Mahony, B.: 'Active objects in TCOZ' in Staples, J., Hinchey, M., and Liu, S. (Eds.): 'The 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)' (IEEE Computer Society Press, 1998), pp. 16–25
29 Nipkow, T., Paulson, L.C., and Wenzel, M.: 'Isabelle/HOL – a proof assistant for higher-order logic', *Lect. Notes Comput. Sci.*, 2002, **2283**
30 Zhang, H., Jarzabek, S., and Yang, B.: 'Quality prediction and assessment for product lines'. CAiSE, 2003, pp. 681–695