Sensors and Actuators in TCOZ

Brendan Mahony[1]* Jin Song Dong[2]

[1] Information Technology Division
Defence Science and Technology Organisation (DSTO)
Brendan.Mahony@dsto.defence.gov.au
[2] School of Computing,
National University of Singapore,
dongjs@comp.nus.edu.sg

Abstract. Timed Communicating Object Z (TCOZ) combines Object-Z's strengths in modeling complex data and algorithms with Timed CSP's strengths in modeling real-time concurrency. TCOZ inherits CSP's channel-based communication mechanism, in which messages represent discrete synchronisations between processes. The purpose of most control systems is to observe and control analog components. In such cases, the interface between the control system and the controlled systems cannot be satisfactorily described using the channel mechanism. In order to address this problem, TCOZ is extended with continuous-function interface mechanisms inspired by process control theory, the **sensor** and the **actuator**. The utility of these new mechanisms is demonstrated through their application to the design of an automobile cruise control system.

1 Introduction

The design of complex systems requires powerful mechanisms for modeling data, algorithms, concurrency, and real-time behaviour; as well as for structuring and decomposing systems in order to control local complexity. In recognition of this, much recent work in the development of specification and design notations has concentrated on the blending of existing notations with strong mechanisms in one or the other of these areas. An early examples of this trend are the LOTOS language [9], which blends process algebras with algebraic modeling languages, and RAISE [16], which blends VDM, CSP, ML with algebraic modeling languages. More recently there has been active investigation of the integration of object-oriented data-structuring techniques with process description languages. The blending of Z/Object-Z [3] with either CSP [21, 5, 20, 23] or CCS [6, 24] has been a popular approach. TCOZ lies in this last category. It is a blending of Object-Z and Timed CSP [18] that is aimed at providing a powerful design notation for real-time and concurrent systems with digital components.

Many classes of complex digital systems are identified in the literature: concurrent, real-time, hybrid, embedded to name a few. In fact, many of these systems are better characterised as *control* systems [17]. Following Shaw [19], we contend that the architecture of control systems is an important structuring mechanism for the efficient

* This paper was written during the first author's visit to the CSIRO Mathematical and Information Sciences Division as part of the DSTO/CSIRO Fellowship Programme.

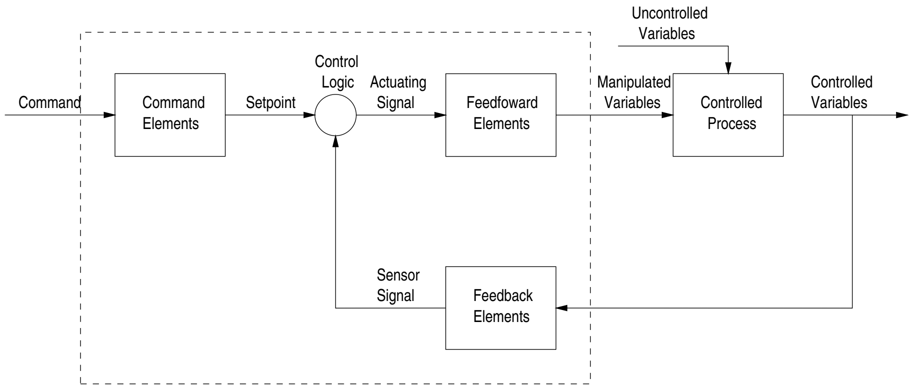

Fig. 1. Abstract control system architecture

design of complex digital systems. The (closed-loop) control architecture is depicted in Figure 1 (which is borrowed with minor modifications from Raven [17, Fig. 1.3]).

Generally the controlled process is described by a differential or integral equation involving the controlled, uncontrolled, and manipulated variables, which are best modeled as continuous real-valued functions of real-valued time. The development of precise models for analog components is essentially beyond the scope of TCOZ, which is restricted by its nature to the description of discrete models. (Even though TCOZ adopts real-valued time, all of the events and quantities described in a TCOZ class remain essentially discrete in nature.) Higher level design languages such as the Timed Refinement Calculus [15] or Duration Calculus [25] should instead be employed to describe the behaviour of the analog components.

However, in a modern digital control system, the subsystem of Figure 1 enclosed within the dashed rectangle is composed solely of digital components. The feedforward, feedback, and command elements are generally digital-to-analog and analog-to-digital converters as appropriate. The actuating signal is used to generate an analog quantity and the sensor and setpoint signals are sampled from analog quantities. The control logic is a non-terminating reactive process executing on a digital processing unit. All of these elements should in theory be amenable to description within a discrete modeling language such as TCOZ.

In drawing the digital subsystem boundary to encompass the conversion elements of the control system, we present some challenges to the CSP channel-based communications mechanism used in TCOZ. The primary challenge lies in the analog nature of the quantities which make up the interfaces. The discrete modeling mechanisms of CSP and Object-Z cannot describe a continuously varying quantity. Another point is that the digital system thus described becomes an open system. CSP channels are better suited to describing closed systems because of CSP's view of communications as representing synchronisations between systems. A closed system is one in which all aspects of system behaviour are fully described, with no need to refer to, nor interface to, other systems. An open system is one which operates in the context of an environment which is determined solely by the interface it presents. Since CSP communications require synchronisation between processes, any system specified in CSP is subject to arbitrary delay by an uncooperative environment. Such a system cannot usually be completely decoupled from its environment because it relies on the co-operation of the environment to make progress.

In order to address these shortcomings of the basic CSP communications mechanism, we propose the introduction to TCOZ of two continuous-function interface mechanisms inspired by the control system architecture. The **sensor** provides a sampling channel linked to a global continuous variable. The **actuator** provides a local-variable linked to a global continuous variable. Sensors and actuators may appear either at the system boundary (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 synchronisations).

Outline of paper

It is assumed that the reader has some familiarity with both Object-Z [3] and CSP, since the mechanics of blending the two notations is considered only briefly in Section 2. The continuous-function interface mechanisms, **sensor** and **actuator**, are introduced in Section 3. Section 4 informally describes the high-level functionality of a standard case study in automatic control, the automobile cruise control. The TCOZ specification of the cruise control is presented and evaluated in Section 5.

2 Aspects of TCOZ

TCOZ is a blending of Object-Z [3] and Timed CSP [18], for the most part preserving them as proper sub-languages of the blended notation. The essential elements of this blending are the unification of the concepts of type, class, and process and the unification of Object-Z operation specification schemas with terminating CSP processes. Thus instances of process may be declared normally and occupy the same syntactic class as objects. Operation schemas and CSP processes also occupy the same syntactic category, operation schema expressions may appear wherever processes may appear in CSP and CSP process definitions may appear wherever operation definitions may appear in Object-Z. In this section we briefly consider the aspects of TCOZ which help to bring the two notations together. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [12]. The semantics of TCOZ can be found in [14].

2.1 Declaring channels

CSP channels are given an independent, first class role in TCOZ. This allows the communications and control topology of a network of objects to be designed orthogonally to their class structure.

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 polymorphic and may carry communications of any type. Being based on ZF set theory, Z is not technically speaking a typed logic [22], so this presents no semantic challenge and prevents unnecessary proliferation of channel names. Channel variables act in the role of `event constructors'. A channel c may either appear alone in the role of an event (applied to the null-value) or else be applied to a Z-value v like so $c.v$. The conventional usages $c?v$ and $c!v$ serve solely as visual feedback to document the intention that an event act in the role of an input or output respectively. They have no semantic implications.

Contrary to the conventions adopted for internal state variables, channels are viewed as shared rather than as encapsulated entities. This is a consequence of their role as communications 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.

2.2 A model of time and quantity

In TCOZ, all timing information is represented as real valued measurements in *seconds*. Describing time and other physical quantities in terms of standard units of measurement is an important aspect of ensuring the completeness and soundness of specifications of real-time, reactive, and hybrid systems. In order to support the use of standard units of measurement, extensions to the Z typing system suggested by Hayes and Mahony [7] are adopted. Under this convention, time quantities are represented by the type $\mathbb{R}\,\mathsf{s}$, where $\mathbb{R}$ represents the real numbers and s is the SI symbol for the standard unit of time. Time literals consist of a real number literal annotated with a symbol representing a unit of time. For example, $3\,\mu\mathsf{s}$ is a literal representing a period of three microseconds, that is three millionths of the standard time unit, the second. All the SI standard units symbols are supported and all the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

2.3 Deadlines and delays

In order to describe the timing requirements of operations and sequences of operations, a deadline command along the lines described by Hayes and Utting [8] is introduced. If *OP* is an operation specification (defined through any combination of CSP process primitives and Object-Z operation schemas) then $OP \bullet \mathrm{DEADLINE}\ t$ describes the process which has the same effect as *OP*, but is constrained to terminate no later than t.

The WAITUNTIL operator is a dual to the deadline operator. The process

$$OP \bullet \mathrm{WAITUNTIL}\ t$$

performs *OP*, but will not terminate until at least time t.

2.4 Guards and preconditions

A novel CSP operator, the state-guard, is used to *block* or *enable* execution of an operation on the basis of an object's local state. For example, the operation $[a \geq 0] \bullet [\Delta(a) \mid$

$a \geq 0 \wedge a' = \sqrt{a}]$ will replace the state variable a with its square root if a is positive otherwise it will *deadlock*, that is be blocked from executing. The blocking or enabling of this operation is achieved by the state guard $[a \geq 0] \bullet \_$ and not by the precondition $a \geq 0$ within the operation schema. If the operation schema alone is invoked with a negative, it will *diverge* rather than block. The difference between deadlock and divergence is that a divergence may be refined away by making an operation more robust, while a deadlock can never be refined away.

An additional function of state guards is as a substitute for CSP's indexed external choice operator. The process $[n : \mathbb{N} \mid 0 \leq n \leq 5] \bullet c?n \rightarrow P(n)$ may input any value of n between 0 and 5 (from channel c) as chosen by its environment. CSP's indexed internal choice is replaced by the operation schema and sequential composition. The process $[n! : \mathbb{N} \mid 0 \leq n! \leq 5];\ c!n \rightarrow P(n)$ may output any value of n between 0 and 5 according to its own designs.

2.5 Active and passive objects

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 behaviour of active objects of a given class [2]. The MAIN process is required to have neither input nor output parameters. If ob_1 and ob_2 are active objects of the class C, then the independent parallel composition behaviour of the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\mathrm{MAIN} \mid\mid\mid ob_2.\mathrm{MAIN}$

2.6 Complex network topologies

In TCOZ, a graph-based approach is adopted to describing network topologies [13]. For example, consider that processes A and B communicate privately through the channel ab, processes A and C communicate privately through the channel ca, and processes B and C communicate privately through the channel bc. This network topology may be described in TCOZ by the *network topology* expression

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

Network topology expressions are a notation intended to mimic the graphical communicating structure. They consist of interface specifications of the form

$$P_1, \ldots, P_n \xleftrightarrow{c_1, \ldots, c_n} Q_1, \ldots, Q_n,$$

indicating that the channels $c_1, \ldots, c_n$ are used to communicate from the processes $P_1, \ldots, P_n$ to the processes $Q_1, \ldots, Q_n$.

3 Adding continuous-function interfaces to TCOZ

Integrating TCOZ specifications with traditional control theory system models presents something of a challenge. The standard CSP communications interface is the channel,

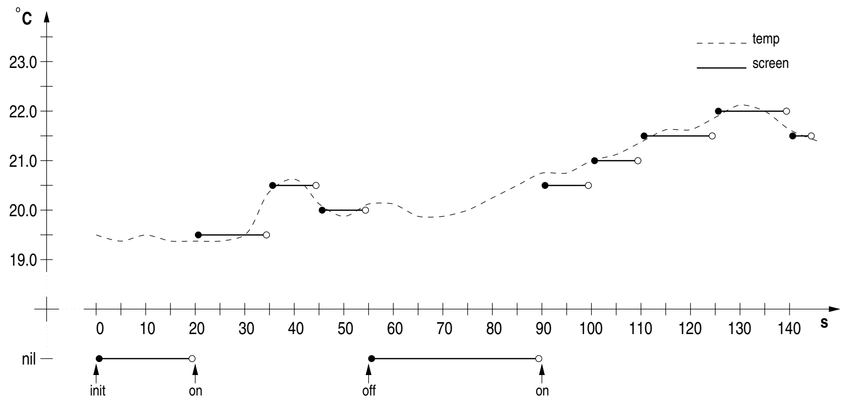

Fig. 2. The office communication scenario.

which represents a sequence of discrete synchronisations between system and environment. The standard model for system interfaces in control theory is the continuous, differentiable function. One approach to this problem is to require the system designer to resolve the mismatch at a higher-level of abstraction, handcrafting a translation from the continuous-function world to TCOZ's discrete world. We reject this approach on the grounds that, though it is very flexible, it constitutes a barrier to the ready acceptance of TCOZ as design tool for digital components of control systems. Instead we adopt an approach by which TCOZ takes it upon itself to become a `good corporate citizen' in the world of control engineering by providing standardised mechanisms for converting from the discrete to the continuous and vice versa, thus allowing TCOZ process classes to present a continuous-function interface to their environments. This allows subsystems specified in TCOZ to fit seamlessly into the overall design of a complex control system.

3.1 The digital temperature display

As a simple demonstration in the use of continuous-function interfaces in TCOZ and their interaction with CSP channels and Object-Z local variables, we consider the communication scenario between a digital temperature display (DTD) and the occupant of an office. The office occupant can turn on/off the DTD by pressing the `on'/`off' buttons on the unit. If the DTD is turned on, then it will monitor the rooms current temperature using its built-in thermometer and update the temperature display every $5\,\mathsf{s}$ to display the current temperature to the nearest half a degree Celsius. If the DTD is turned off, the temperature display goes blank. An example behaviour of the DTD is illustrated by Figure 2.

The communications interfaces to the DTD fall into three distinct classes. The on/off buttons of the unit are best represented using the channel mechanism, because they require explicit co-operation between the user and the DTD unit (that is a synchronisation) and because they are discrete events. A continuous interface could be used, but considerable effort would be required to ensure proper synchronisation. The temperature on the other hand is best modeled as a continuous function of time and is not well suited to being described as a CSP channel. Not only is the continuous function the standard scientific and engineering model for analog quantities, it is also common engineering practice to view digital signals as piece-wise continuous step-functions [15]. The temperature display falls into an area between the truly continuous and the truly discrete, either model may be preferred depending on the application. In this case, because the display falls at the system boundary, the difficulties of using CSP channels to describe open system interfaces mean that modeling the display as a CSP channel is not ideal. For example, the requirement that the display be updated every $5\,\mathsf{s}$ cannot easily be expressed if the display is a CSP channel which may be blocked by an uncooperative environment.

The on/off buttons can be modeled by using CSP channels, one for on-events and one for off-events. In order to describe the thermometer and the display we introduce two new continuous-function interface mechanisms.

The thermometer is introduced by a declaration of the form

$$temp : \mathbb{R}\,°\mathsf{C}\ \mathbf{sensor},$$

which declares *temp* to be a continuous-function interface with public type $\mathbb{R}\,\mathsf{s} \to \mathbb{R}\,°\mathsf{C}$. Internally, *temp* takes the syntactic role of a CSP channel. The relationship between the public continuous-function variable and the internal channel is that whenever a value v is communicated on the internal channel at a time t, that value must be equal to the value of the continuous function at that time, that is $temp(t) = v$.

The temperature display is introduced by a declaration of the form

$$screen : Display\ \mathbf{actuator}, \quad \text{where} \quad Display ::= Temp\langle\langle \mathbb{N} * 0.5°\mathsf{C} \rangle\rangle \mid nil.$$

This declaration also introduces *screen* as a public continuous-function variable, but in this case the internal role is that of the local state variable. Thus *screen* may appear in the delta list of operations and any other place where a local variable may appear.

The TCOZ process class describing the DTD is below.

DTD

$temp : \mathbb{R}\,°\mathsf{C}\ \mathbf{sensor}$ $screen : Display\ \mathbf{actuator}$ $on, off : \mathbf{chan}$	**SetScreen** $\Delta(screen)$ $t? : \mathbb{R}\,°\mathsf{C}$ ——— $\exists\, dt : \mathbb{N} * 0.5°\mathsf{C} \bullet$ $dt = t \pm 0.5°\mathsf{C} \wedge$ $screen' = Temp(dt)$
INIT $screen = nil$	

$$Show \mathrel{\widehat{=}} ([t : \mathbb{R}\,°\mathsf{C}] \bullet temp?t \to SetScreen \bullet \text{DEADLINE}\ 5\,\mathsf{s} \bullet \text{WAITUNTIL}\ 5\,\mathsf{s};\ Show)\ \triangledown\ off \to NoShow$$

$$NoShow \mathrel{\widehat{=}} screen := nil;\ on \to Show$$

$$\text{MAIN} \mathrel{\widehat{=}} on \to Show$$

A DTD object begins with the screen blanked (INIT), then when the on-button is pressed it passes into *Show* mode.

In *Show* mode it polls the temperature sensor and displays the result to the nearest one half degree Celsius. This behaviour is repeated with periodicity $5\,\mathsf{s}$. A repeated activity with period T can be described by the CSP definition of the form

$$PA_0 \mathrel{\widehat=} A;\ \textsc{WaitUntil}\ T;\ PA_0,$$

provided the activity A is guaranteed to terminate before T. In order ensure this a deadline is placed on the activity giving a definition of the form

$$PA \mathrel{\widehat=} A \bullet \textsc{Deadline}\ T \bullet \textsc{WaitUntil}\ T;\ PA.$$

The definition of *Show* is of precisely this form, ensuring that the screen update occurs once every $5\,\mathsf{s}$. The fact that the *temp* channel is a sensor is important in ensuring that the *Show* acts as expected. Since *temp* events do not represent synchronisations with the environment they happen immediately they are offered. A simple CSP channel could be blocked for an arbitrary time, making such a periodic behaviour impossible to guarantee.

If the off-button is pressed with the DTD in *Show* mode, it immediately passes to the *NoShow* mode by blanking the screen. This is expressed using the Timed CSP interrupt operator ($\_ \ \nabla\ \_$), which shifts control to the interrupt routine as soon as an interrupt event (in this case *off*) is enabled. The DTD remains in *NoShow* mode until the on-button is once again pressed. Note that the expression $screen := nil$ is a short form of the schema $[\Delta(screen) \mid screen' = nil]$

3.2 The local virtues

The experienced CSP practitioner is probably not entirely convinced by the preceding argument. After all the so-called \`continuous-function interface' is really just an asynchronous communications medium and it is well known how to model such things in CSP. To a degree this criticism is valid, at least in a closed system. A local continuous-function interface a of type A may be modeled by the following TCOZ process, provided that it appears in a context in which the channels la and ra are hidden from the environment and therefore cannot be blocked.

loc_a

la, ra : **chan**
$a : A$

$$\textsc{Main} \mathrel{\widehat=} \mu\, LA \bullet ([i : A] \bullet la?i \to a := i;\ LA) \,\Box\, (ra!a \to LA)$$

If B is a process which uses a as an actuator and C is a process that uses a as a sensor, then

$$P \mathrel{\widehat=} \big\|\, B \overset{a}{\longleftrightarrow} C$$

has the same behaviour as

$$P^* \mathrel{\widehat=} \| \; B^* \xleftrightarrow{la} A; \; A \xleftrightarrow{ra} C^*,$$

where B^* is B with a replaced by la and updates to a replaced by outputs to la; and C^* is C with a replaced by ra. This interfacing model is inferior in two ways.

Firstly it is only effective when used in closed systems. This means that processes designed to interface with such a channel cannot be understood in isolation, leading to a highly coupled system design. In the above examples, the processes B^* and C^* have the correct timing behaviour only when placed in conjunction with A as in P^*. Unless la and ra are hidden, a hostile environment may interfere with their behaviour. In contrast the **sensor** and **actuator** mechanism provide a truly localised model of an asynchronous interface. Systems designed using these mechanisms are decomposed more easily because the individual components are more easily understood in isolation. In the DTD specification there is no need to describe the behaviour of system environment at all, only the interface it presents.

Secondly in the **sensor/actuator** mechanism the associated continuous function becomes a public interface to the TCOZ process. This means that TCOZ processes may be treated transparently as normal components in a formal approach to analog systems design. One such approach is being developed by Fidge and Hayes *et al* [4, 8], based on Mahony's timed refinement calculus [11, 15]. In any case, describing digital components as truly open subsystems seems preferable to requiring the designer to artificially close a system design by providing unsatisfactory digital approximations to analog system components.

3.3 Generating a real-time clock timer

As another example of the utility of continuous-function interfaces, consider the specification of a real-time system clock for a digital system. A real-time clock provides a synchronisation signal for the various components in a system, in the form of a simple square-wave which oscillates with a set frequency. Such clocks are generally rated in terms of the number of cycles per second (hz) which they generate.

In TCOZ the signal from the real-time clock can be modeled by using a boolean actuator and the square-wave generated by a periodic process.

RT_Clock

$freq : \mathbb{R}\,\mathsf{hz}$
$per, gain : \mathbb{R}\,\mathsf{s}$

$per * freq = 1 \land gain < per/10$

$rtc : \mathbb{B}\,$**actuator**

$\textsc{Main} \mathrel{\widehat=} \mu\, C \bullet rtc := \neg\, rtc \bullet \textsc{Deadline}\, gain \bullet \textsc{WaitUntil}\, per/10;\; C$

The parameters of the *RT_Clock* represent the frequency *freq* and period *per* of the clock; and the time taken to change state *gain*, which is much smaller than the period.

3.4 Monitoring input signals

A number of the continuous-function interfaces in the cruise control specification in Section 4 represent digital signals, that is signals of type boolean. In such cases it is often of great interest to detect transitions in the signal, either from high to low or vice versa. The following TCOZ class describes a process which monitors a digital signal and raises events whenever it encounters a leading or a trailing edge.

Edges

$signal : \mathbb{B}\,\mathbf{sensor}$
$up, dn : \mathbf{chan}$

$High \mathrel{\widehat=} signal.\mathsf{false} \rightarrow dn \rightarrow Low$
$Low \mathrel{\widehat=} signal.\mathsf{true} \rightarrow up \rightarrow High$
$\textsc{Main} \mathrel{\widehat=} signal.\mathsf{false} \rightarrow Low \;\Box\; signal.\mathsf{true} \rightarrow High$

It may seem strange to introduce a signal interface and then re-interpret it as events, but this mechanism gives a more accurate local model of the situation than if the events themselves were the interface. With a channel based interface, a failure of the system to process the signal rapidly enough may simply result in the environment waiting till it is finished. With the continuous-function interface the missing of a processing deadline definitely results in the missing of an edge.

Sometimes we will be interested only in leading edges or else only in trailing edges.

$Lead \mathrel{\widehat=} (Edges \setminus dn)$
$Trail \mathrel{\widehat=} (Edges \setminus up)$

Both the *RT_Clock* and the *Edges* classes will be reused extensively in the cruise control specification.

4 Cruise control overview

The aim of a cruise control system for a car is to maintain the speed of the car even over varying terrain. The high-level system structure of a cruise control system of a car is illustrated in Figure 3. The *Car* is an analog system capable of moving forward, changing direction, producing heat, and many other observable behaviours which we represent as an abstract variable *perf*. The variable *perf* is a function of three inputs. The driver provides control inputs by by turning the steering-wheel, applying the brake, etc the aggregate of which we represent by the abstract variable *driver_inp*. The throttle setting, represented by *throttle*, controls the forward speed of the car. Finally, various environmental factors including wind drag, road incline, etc (represented by *env*) can affect the response of the system to the controlling variables.

The purpose of the *Cruise* class is to monitor the linear speed of the car and to modulate the throttle setting so as to maintain the speed of the car at a point determined by the driver inputs.

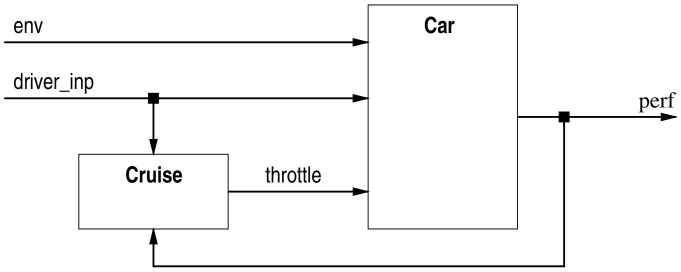

Fig. 3. Block diagram of cruise system and car.

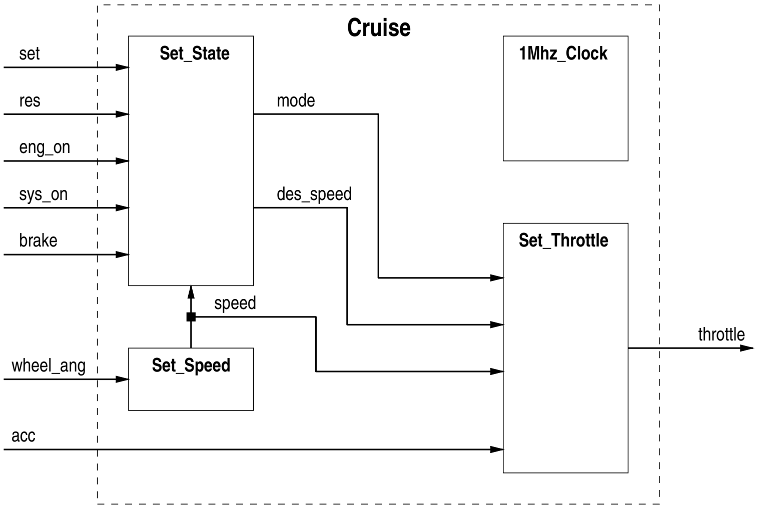

Fig. 4. Block diagram of throttle control.

Notice that the structure of the cruise control system is precisely that of the abstract control system presented in Figure 1. The controlled variable is *perf*, the uncontrolled variables are *env* and some components of *driver_inp*, the manipulated variable is *throttle*, and the command variables are the components of *driver_inp* which determine the mode of behaviour of the *Cruise* class. This gives *Cruise* class itself the same external interface as the subsystem enclosed by the dashed rectangle in Figure 1, suggesting the structure of that subsystem as a suitable architecture for the *Cruise* class. Section 5 is devoted to a formal TCOZ specification of the *Cruise* subsystem following this control system architecture.

5 TCOZ model of the cruise control system

The top-level design of the cruise control system, illustrated in Figure 4, follows the basic structure of the digital subsystem from Figure 1. The command variables are the components of *driver_inp* (driver input) that are devoted to operating the cruise control. These consist of the cruise-on/off button *sys_on*, the set-cruise button *set*, the resume-cruise button *res*, the accelerator pedal *acc*, and the brake pedal *brake*. The exact function of these command variables is described in Section 5.3, but for the moment it is sufficient to note that they are interpreted by the *Cruise* class command element *Set_State*. The setpoint signals are *des_speed*, *mode*, and *acc* which is passed through directly. The monitored components of the car are *eng_on* which indicates when the engine is running and *wheel_ang* which measures the angular position of a reference point on one of the wheels. The feedback element of the *Cruise* class is the *Set_Speed* class, which uses *wheel_ang* to calculate the current speed. The control function is performed by the *Set_Throttle* class which determines the correct *throttle* setting according to the current *speed*, *mode*, *des_speed*, and accelerator heel *pulses* and (acceleration) *acc*. The output from the cruise control is the *throttle*, which constitutes both the actuating signal and the manipulated variable. The feedforward element is the trivial process which propagates *throttle* through to the *Car* system. The final class is the real-time clock which is used to drive the activities of the other classes.

5.1 The clock

In order to drive the cruise control circuitry we introduce a $1\,\mathsf{Mhz}$ clock, that is $1\,\mathsf{cu} = 1\,\mu\mathsf{s}$.

$1\,\mathsf{Mhz}\ \mathit{Clock}$

$\mathit{RT\_Clock}$

$|\ \mathit{per} = 1\,\mathsf{cu} \wedge \mathit{gain} < 0.1\,\mathsf{cu}$

5.2 Car speed

The speed of the car is determined by counting clock signals between pulses from a wheel sensor. This process consists of four components as shown in Figure 5: one to generate the wheel pulses, one to detect wheel pulses, one to detect clock signals, and one to actually count the clock signals in each period of revolution and calculate the speed. The speed is calculated with a precision of $0.1\,\mathsf{km\,hr}^{-1}$, that is in units of $\mathsf{su} == 0.1\,\mathsf{km\,hr}^{-1}$, and also to an accuracy of $0.1\,\mathsf{km\,hr}^{-1}$, under the assumptions that the maximum speed of the car is $max_s == 300\,\mathsf{km\,hr}^{-1}$ and the wheel circumference is $C_w == 3\,\mathsf{m}$. The period of revolution is calculated with a precision is ± 1 clock cycle, the wheel pulse timings may be so as to allow a whole extra clock signal when the time elapsed is a small fraction of a clock cycle or vice versa. To ensure that the overall accuracy of the speed calculation is better than $\pm 1\,\mathsf{su}$, the clock unit cu must satisfy the condition

$$1\,\mathsf{cu} \leq \frac{C_w}{max_s} * \frac{0.5\,\mathsf{su}}{max_s - 0.5\,\mathsf{su}}.$$

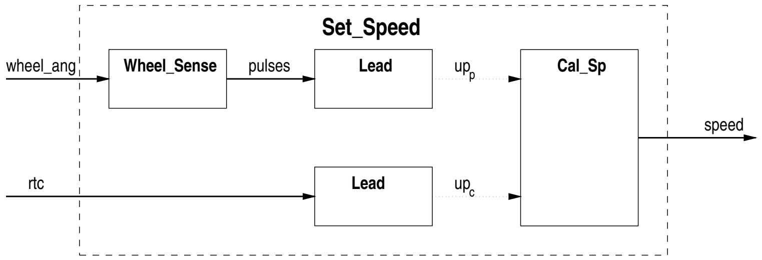

Fig. 5. Block diagram of speed calculation.

The suggested clock rate of 1 Mhz is adequate to ensure this condition.

The *Wheel_Sense* process monitors the *wheel_ang* variable and raises the pulse signal while the angle is between 0 rad and $\epsilon_a : \mathbb{R}$ rad. In order to ensure the precision of ± 1 cu, we require that raising and dropping the signal take no longer than $\delta_a == 0.1$ cu.

Wheel_Sense

$wheel\_ang : \mathbb{R}$ rad **sensor**
$pulse : \mathbb{B}$ **actuator**

INIT
$pulse =$ false

$\text{MAIN} \mathrel{\widehat=} \mu\, WS \bullet$
$\quad [a : \mathbb{R}\ \text{rad} \mid 0\ \text{rad} \leq a \leq \epsilon_a] \bullet wheel\_ang?a \rightarrow$
$\qquad pulse := \text{true} \bullet \text{DEADLINE}\ \delta_a;$
$\quad [a : \mathbb{R}\ \text{rad} \mid \epsilon_a \leq a] \bullet wheel\_ang?a \rightarrow$
$\qquad pulse := \text{false} \bullet \text{DEADLINE}\ \delta_a;\ WS$

Between each wheel pulse the number of clock pulses is counted to determine the period of rotation to the nearest clock unit. The speed is then calculated in speed units by dividing the circumference (C_w) of the wheel by the period between pulses.

Cal_Sp

up_c, up_p : **chan**
$speed : \mathbb{N}$ su **actuator**
$per : \mathbb{N}$ cu

INIT
$speed = 0\ \text{cu} \wedge per = 0\ \text{cu}$

NewSpeed
$\Delta(speed)$

$per > C_w / max_s \wedge$
$\quad speed = \frac{C_w}{per} \pm 0.5\ \text{su} \vee$
$per \leq C_w / max_s \wedge speed = max_s$

$$
\begin{array}{|l}
\mathit{Count} \mathrel{\widehat{=}} \\
\quad [\mathit{per} < C_w/1\,\mathsf{su}] \bullet (\\
\qquad (\mathit{up}_c \rightarrow \mathit{per} := \mathit{per} + 1\,\mathsf{cu} \bullet \text{DEADLINE}\ 0.1\,\mathsf{cu};\ \mathit{Count}) \ \Box \\
\qquad (\mathit{up}_p \rightarrow (\mathit{NewSpeed};\ \mathit{per} := 0\,\mathsf{cu}) \bullet \text{DEADLINE}\ 0.1\,\mathsf{cu};\ \mathit{Count})) \\
\quad \Box \\
\quad [\mathit{per} \geq C_w/1\,\mathsf{su}] \bullet \\
\qquad (\mathit{speed} := 0\,\mathsf{su};\ \mathit{per} := 0\,\mathsf{cu}) \bullet \text{DEADLINE}\ 0.1\,\mathsf{cu}; \\
\qquad\quad \mathit{up}_p \rightarrow \mathit{Count} \\
\text{MAIN} \mathrel{\widehat{=}} \mathit{up}_p \rightarrow \mathit{Count} \\
\hline
\end{array}
$$

Two exceptional behaviours are considered. If the wheel is rotating very slowly, the period calculation times out when the count exceeds C_w/max_s, the speed is set to $0\,\mathsf{su}$ and the count is not restarted until the next wheel pulse is encountered. If the wheel is rotating very fast then the speed is set to max_s.

Set_Speed

$$
\begin{array}{|l}
\quad pe : Lead[\frac{pulses}{signal}, \frac{up_p}{up}] \\
\quad ce : Lead[\frac{rtc}{signal}, \frac{up_c}{up}] \\
\quad cs : Cal\_Sp \\
\quad ws : Wheel\_Sense \\
\text{MAIN} \mathrel{\widehat{=}} \Big\|(ws \xleftrightarrow{pulses} pe;\ pe \xleftrightarrow{up_p} cs;\ ce \xleftrightarrow{up_c} cs) \\
\hline
\end{array}
$$

5.3 Cruise modes

When operating, the cruise control can be in any of four modes of operation.

$$CM \mathrel{\widehat{=}} setpoint \mid accel \mid decel \mid rest$$

setpoint The speed of the car is maintained at the desired speed by manipulating the throttle setting.
accel The speed of the car is increased by opening the throttle.
decel The speed of the car is decreased by closing the throttle.
rest No throttle manipulation is performed, but the desired speed is remembered.

The mode of operation of the cruise control is determined by the following input signals.

eng_on The cruise control cannot operate if the engine is off.
sys_on The cruise control is switched on and off by this signal.
set While the cruise control is in any operating mode, briefly raising the *set*-signal sets the desired speed to the current speed and initiates *setpoint*-mode. Holding the *set*-signal high for longer than

$$t_h == 1\,\mathsf{s}$$

causes the car to enter the *decel*-mode. When the *set*-signal falls the desired speed is set to the current speed, then control returns to *setpoint*-mode.

res While the cruise control is in any operating mode, briefly raising the *res*-signal initiates *setpoint*-mode, but does not alter the desired speed. Holding the *res*-signal high for longer than t_h causes the car to enter the *accel*-mode. When the *res*-signal falls the desired speed is set to the current speed, then control enters the *setpoint*-mode.

brake While the cruise control is in any operating mode, touching the brake causes the control to enter *rest*-mode, but does not alter the desired speed.

speed If operating, the cruise control cannot be in *setpoint*-mode if the desired speed is less than $min_d == 50.0\,\mathsf{km\,hr^{-1}}$.

The purpose of the *Set_State* process is to determine the correct operating mode and to maintain the value of the desired speed.

In order to correctly interpret the control signals from the driver, especially in light of the dual purpose nature of the *set* and *res* signals, monitors are placed on these signals to convert them to a sequence of driver events as depicted in Figure 6. The possible events on the *set* signal are *sp* for enaging cruise control and *dc* for decelerating. The possible events on the *res* signal are *rs* for resuming cruise control and *ac* for accelerating.

Set

$se : Edges[\frac{set}{signal}]$
$sp, dc : \mathbf{chan}$

$SetPt \mathrel{\widehat=} dn \rightarrow sp \rightarrow Interpret$
$Interpret \mathrel{\widehat=} up \rightarrow$
$\quad SetPt \rhd\{t_h\}\ (dc \rightarrow SetPt)$
$\textsc{Main} \mathrel{\widehat=} \|(Interpret \xleftrightarrow{up,dn} se)$

Resume

$re : Edges[\frac{res}{signal}]$
$sp, rs : \mathbf{chan}$

$Res \mathrel{\widehat=} dn \rightarrow rs \rightarrow Interpret$
$SetPt \mathrel{\widehat=} dn \rightarrow sp \rightarrow Interpret$
$Interpret \mathrel{\widehat=} up \rightarrow$
$\quad Res \rhd\{t_h\}\ (ac \rightarrow SetPt)$
$\textsc{Main} \mathrel{\widehat=} \|(Interpret \xleftrightarrow{up,dn} re)$

A simple edge monitor is used on the brake signal.

The normal behaviour of the cruise state is to set the *mode* and *des_speed* signals in accordance with driver cruise events and any brake events. However, this behaviour is suppressed if the *eng_on* or *sys_on* signals go low. When both signal go high again the *mode* is set to *rest* and the *des_speed* to $0\,\mathsf{su}$.

Cruise_State

$ac, sp, rs, dc, up_b, dn_b : \mathbf{chan}$
$mode : CM\ \mathbf{actuator}$
$des\_speed : \mathbb{N}\,\mathsf{su}\,\mathbf{actuator}$
$speed : \mathbb{N}\,\mathsf{su}\,\mathbf{sensor}$
$eng\_on, sys\_on : \mathbb{B}\,\mathbf{sensor}$

$\textsc{Init}$

$mode = rest$
$des\_speed = 0\,\mathsf{su}$

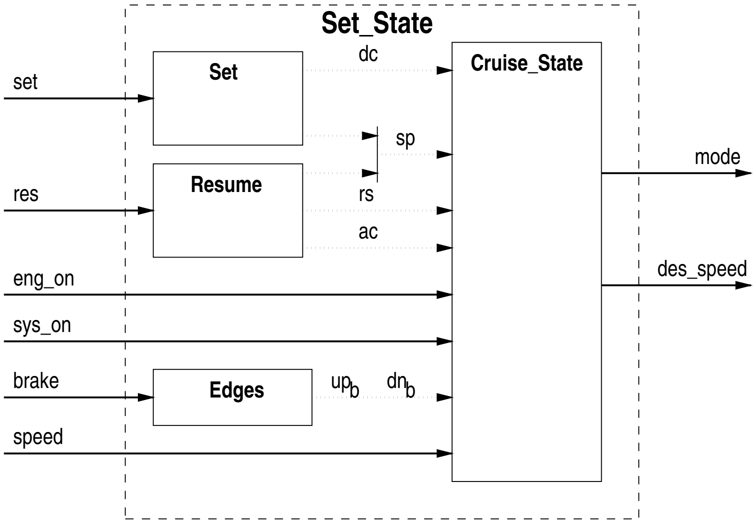

Fig. 6. Block diagram of cruise state determination.

$$
\begin{array}{|l}
SP \mathrel{\widehat=} sp \rightarrow [s : \mathbb{N}\,\mathsf{su}] \bullet speed?s \rightarrow (\\
\qquad\qquad [s \geq min_d] \bullet des\_speed := s;\ mode := setpoint \ \Box \\
\qquad\qquad [s < min_d] \bullet \textsc{Skip}) \\
RS \mathrel{\widehat=} rs \rightarrow (\\
\qquad\qquad [des\_speed \geq min_d] \bullet mode := setpoint \ \Box \\
\qquad\qquad [des\_speed < min_d] \bullet \textsc{Skip}) \\
AC \mathrel{\widehat=} ac \rightarrow mode := accel \\
DC \mathrel{\widehat=} dc \rightarrow mode := decel \\
Normal \mathrel{\widehat=} (SP \ \Box\ RS \ \Box\ AC \ \Box\ DC);\ Normal \\
Active \mathrel{\widehat=} Normal \ \triangledown\ up_b \rightarrow mode := rest;\ dn_b \rightarrow Active \\
Sys\_Off \mathrel{\widehat=} sys\_on?\mathsf{true} \rightarrow Active \\
\qquad \triangledown\ sys\_on?\mathsf{false} \rightarrow mode := rest;\ des\_speed := 0\,\mathsf{su};\ Sys\_Off \\
Eng\_Off \mathrel{\widehat=} eng\_on?\mathsf{true} \rightarrow Sys\_Off \\
\qquad \triangledown\ eng\_on?\mathsf{false} \rightarrow mode := rest;\ des\_speed := 0\,\mathsf{su};\ Eng\_Off \\
\textsc{Main} \mathrel{\widehat=} Eng\_Off \\
\hline
\end{array}
$$

A *Set_State* class then consists of monitors on the *set*, *res*, and *brake* signals; and a *Cruise_State* class communicating with each other as described in Figure 6.

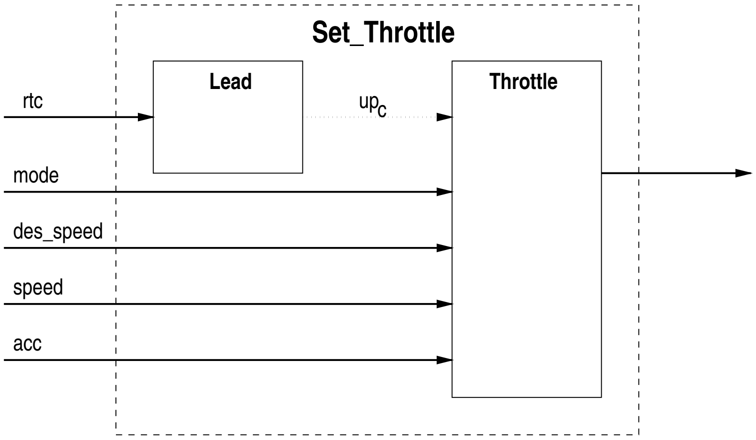

Fig. 7. Block diagram of throttle determination.

Set_State

$sm : Set$
$rm : Resume$
$br : Edges[\frac{brake}{signal}, \frac{up_b}{up}, \frac{dn_b}{dn}]$
$cs : Cruise\_State$

$\textsc{Main} \mathrel{\widehat{=}} \|(sm \xleftrightarrow{sp,dc} cs;\ rm \xleftrightarrow{sp,rs,ac} cs;\ br \xleftrightarrow{up_b,dn_b} cs)$

5.4 Throttle

The final component of the cruise control determines the appropriate throttle setting for all cruise modes. A block diagram of the *Set_Throttle* component is depicted in Figure 7. It is a clocked component which calculates a new throttle setting each clock cycle, based on the current speed, the cruise mode, the accelerator pedal, and the desired speed (in *setpoint*-mode). The throttle and accelerator pedal quantities are represented abstractly by a unit symbol au.

Throttle

$\delta_{th} : \mathbb{N}$ au
$max_{th}, min_{th} : \mathbb{N}$ au
$\ldots$

$s_p : \mathbb{N}$ su
$speed, des\_speed : \mathbb{N}$ su **sensor**
$mode : CM$ **sensor**
$acc : \mathbb{N}$ au **sensor**
$throttle : \mathbb{N}$ au **actuator**

$\ldots$ [details omitted]

The details of the above class are omitted due to the space limitation. (See the tech-report [10] for the complete class definition.)

The *Set_Throttle* class consists of a monitor on the clock signal and a *Throttle* class for updating the throttle every clock cycle.

Set_Throttle

$$ce : Lead[\tfrac{rtc}{signal}, \tfrac{up_c}{up}]$$
$$th : Throttle$$

$$\text{MAIN} \mathrel{\widehat=} \|(ce \xleftrightarrow{up_c} th)$$

5.5 Cruise system

As stated in the introduction to this section, the *Cruise* class consists of a 1 Mhz clock, a speed monitor, a user input monitor, and a throttle actuator interacting as described in Figure 4.

Cruise

$$c : 1\ \mathsf{Mhz}\ Clock$$
$$ss : Set\_State$$
$$sd : Set\_Speed$$
$$st : Set\_Throttle$$

$$\text{MAIN} \mathrel{\widehat=} \|(c \xleftrightarrow{rtc} sd, st;\ ss \xleftrightarrow{mode, des\_speed} st;\ sd \xleftrightarrow{speed} ss, st)$$

6 Conclusion

In this paper, Timed Communicating Object Z (TCOZ) has been extended with new communications mechanism, the *continuous-function interface*. The basic idea is to use a (usually real-valued) function of real-valued time as communications medium between objects. The **actuator** and **sensor** mechanism differ only in the manner in which the continuous-function interface is utilised by a class. A **actuator** takes on the role of a local variable through which an object `controls' the value of the continuous-function interface. A **sensor** takes on the role of a CSP channel through which the object `monitors' the value of the continuous-function interface.

The standard method of communication between components in an object-oriented architecture is the method invocation by which an object may request a service from another object if it knows the appropriate method name and object identifier. This form of communication leads to a high degree of coupling between object classes because so much information is needed to set up communications. In CSP the standard communications mechanism is the channel which provides a more abstract interface between processes. Each component interacts only with its channels and need have little detailed knowledge about the structure of other components. However, because communications on CSP channels represent explicit synchronisations between processes, each process must obey the correct `protocol' for the channel in order to avoid deadlock situations. Thus there remains a residual amount of coupling between processes linked by CSP channels. This coupling is removed by the continuous-function interface mechanism which does not require a synchronisation between processes for communication to occur. Through judicious use of channels where synchronisation is truly required (as for service requests) and continuous-function interfaces where synchronisation is not required, it is possible to adopt a `open' approach to systems design with a minimum of inter-module coupling. We believe the open systems approach to be essential to the treatment of large-scale formal systems design.

The coupling problem with CSP channels has also been recognised by Davies [1], who suggested the use of *signal* events as a means of addressing the problem. A signal event is simply an event which cannot be blocked by its environment. However, if no process is ready to accept the signal immediately, the information is lost forever. The continuous-function interface is superior to the signal mechanism, because the information transmitted on an **actuator** signal remains available to any other process until overwritten by the controlling process.

The **actuator** and **sensor** metaphors are drawn from the theory of automatic control systems. Following Shaw [19], we advocate the control system as an important architectural framework for the design of real-time, hybrid systems. In this paper we have demonstrated the power of the control systems architecture by applying it to the classic hybrid-system case-study, the automobile cruise control. Applying the control system template of Figure 1 to the cruise control allowed us to identify and describe the high-level components of the cruise control with a minimum of effort. By adopting the `natural' architecture for the problem domain, we were able to produce a design with a low degree of coupling between components; a factor that is likely to make later development phases both cheaper and faster. The case study has also served as a vehicle for demonstrating the power of the continuous-function interface as a means of supporting the description of `open' system components. The formal incarnation of the cruise control design was able to reflect the elegance of the informal architecture because the continuous-function interface does not bias the design toward higher coupling as would the method invocation and channel communications mechanisms.

References

1. J. Davies. *Specification and Proof in Real-Time Systems*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1991.
2. J.S. Dong and B. Mahony. Active Objects in TCOZ. In J. Staples, M. Hinchey, and S. Liu, editors, *the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 16–25. IEEE Computer Society Press, December 1998.
3. R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
4. C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In *Mathematics of Program Construction*, 1998.
5. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

6. A. J. Galloway and W. J. Stoddart. An operational semantics for ZCCS. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 272–282, Hiroshima, Japan, November 1997. IEEE Computer Society Press.
7. I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3), 1995.
8. I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, 1997.
9. ISO 8807. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, 1989.
10. B. Mahony and J.S. Dong. Sensors and Actuators: Process Control in TCOZ. Technical Report 98-38, Mathematical and Information Sciences, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia, 1998.
11. B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, University of Queensland, 1991. Available as `ftp://ftp.it.uq.edu.au/pub/Thesis/brendan_mahony.ps.Z`.
12. B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Computer Society Press.
13. B. P. Mahony and J.S. Dong. Network topology and a case-study in TCOZ. In *ZUM'98 The* 11^{th} *International Conference of Z Users*. Springer-Verlag, September 1998.
14. B. P. Mahony and J.S. Dong. Overview of the semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. Integrated Formal Methods (IFM'99)*, York, UK, June 1999. Springer-Verlag. (accepted).
15. B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.
16. M. Nielsen, K. Havelund, R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1:85–114, 1989.
17. F. H. Raven. *Automatic Control Engineering*. McGraw-Hill, second edition, 1968.
18. S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.
19. M. Shaw. Beyond objects. *ACM Software Engineering Notes*, 20(1), January 1995.
20. A. Simpson, J. Davies, and J. Woodcock. Security management via Z and CSP. In J. Grundy, M. Schwenke, and T. Vickers, editors, *IRW/FMP'98*. Springer-Verlag, 1998.
21. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of FME'97: Industrial Benefit of Formal Methods*, Graz, Austria, September 1997. Springer-Verlag.
22. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.
23. C. Suhl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. Integrated Formal Methods (IFM'99)*, York, UK, June 1999. Springer-Verlag. (accepted).
24. K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Computer Society Press.
25. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.