

Integrating Object-Z with Timed Automata

J.S. Dong¹ R. Duke² P. Hao¹

¹National University of Singapore

²University of Queensland, Australia

Abstract

When designing a complex system, Object-Z is a powerful logic-based language for modeling the system state aspects, and Timed Automata is an excellent graph-based notation for capturing timed control behaviour of the system. This paper presents an effective combination of the two techniques with novel composition and communication mechanisms. The combined notation enhances Object-Z with real-time modeling capability and also extends Timed Automata with enhanced structure and state modeling features.

Keywords: Specification, Object-Z, Timed Automata

1 Introduction

Software system specification is an important activity in software engineering. Formalisms for specifying computer systems have been well researched. Logic-based formalisms, e.g. Z, have been popular in Europe. Graph-based formalisms, e.g. Automata, have been prevalent in North America. The design of complex systems requires techniques for capturing system functionalities and control behaviours. The system functionalities can be best captured in terms of operations and constraints — the ideal application for Z. The system control behaviours can be best captured in terms of visual flows between system functionalities — the ideal application for Automata. Furthermore, complex systems often have intricate system structures and real-time requirements. Object-Z [11, 26] is a structured extension to Z and can be supported by verification tools (e.g. [24, 20, 29, 34]). Timed Automata (TA) [1, 36] is a real-time extension to Automata and can be effectively verified by model checkers (e.g. [3, 7, 30, 33]).

In our previous work [10], we investigated the projection techniques from the TCOZ [22] (extension to Object-Z) to TA and discussed the notion of timed patterns. In this paper, rather than taking the transformation point of view we propose to develop a novel integrated formal language which combines Object-Z with TA. An effective combination of Object-Z and TA can not only help Object-Z with real-time

modeling capability but also help TA with enhanced structure and state modeling features. The result of such a combination can be a powerful unified method for designing complex computer systems. The challenge of achieving an effective combination of Object-Z and TA is to

- semantically and syntactically link the key language constructs so that the two notations can be used in a cohesive way;
- clearly separate system functionality aspects from time control behaviour patterns so that separate tools can later be applied to check the related system properties;
- consistently unify the composition techniques from both Object-Z (class instantiation) and TA (automaton product) so that subsystem models can be easily and meaningfully composed;
- systematically develop the communication mechanisms so that various concurrent interactions between system components can be precisely captured.

In the remaining sections of this paper we will demonstrate how Object-Z and TA can be effectively combined.

2 Object-Z and Timed Automata

In this section, brief introductions to Object-Z and TA are presented together with motivating examples.

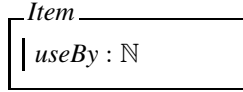
Object-Z

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring.

The essential extension to Z given by Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of

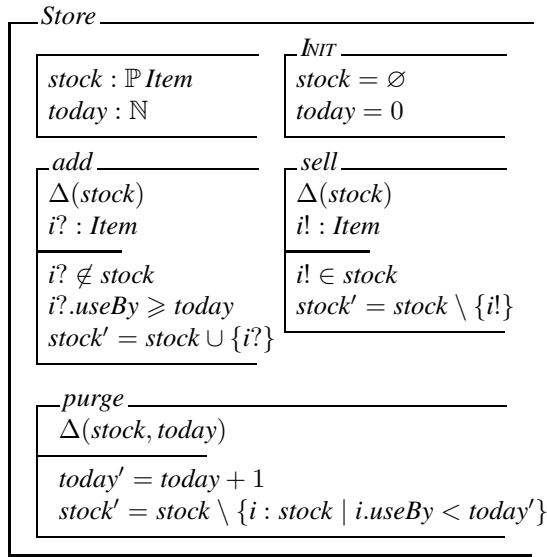
the class. An object is said to be an instance of a class and to evolve according to the definitions of its class. Syntactically, a class definition is a named box. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas. To illustrate Object-Z, consider a simple stock-control system for a store. The store stocks items which each have a fixed use-by date. An item can be added to the store's stock, but only if the use-by date of the added item is today's date or later. Any item can be sold by the store. At the beginning of each day those items whose use-by date is less than the current date are removed (i.e. purged) from the store.

To specify this system in Object-Z, first we specify an item as an object of the class *Item*:



Effectively, at this level of abstraction the only important thing about an item is its (fixed) use-by date.

The stock control system is specified by the class *Store*:



The semantics of Object-Z can be seen as a state transition system. For example, given a particular *Store* object state $\sigma = \{(store, \{item_a, item_b\}), (today, 20)\}$, if operation *add* is then performed with a new input item *item_c*, the new object state would be $\sigma' = \{(store, \{item_a, item_b, item_c\}), (today, 20)\}$.

Notice that although there is an attribute *today* in this class and this attribute is incremented whenever the *purge* operation takes place, no notion of the progressive passing of time is captured by this specification. Conceptually, we think of the purge operation as taking place once a day, but this is not captured explicitly. Furthermore, in standard Object-Z

the operations are assumed to be atomic, so there is no direct way of capturing the idea that an operation may take a specific time to complete.

Timed Automata

Timed Automata (TA) are finite state machines with clocks. It was introduced as a formal notation to model the behavior of real-time systems. Its definition provides a general way to annotate state-transition graphs with timing constraints using finitely many real-valued clock variables. Another interesting aspect of TA is that there exist model checking methods for temporal logics with quantitative temporal operators which are directly applied to TA. Thus a variety of tools are available for specification and verification of real-time system modeled in TA.

In this paper, we follow the definitions given in [1]. Formally, for a set X of clock variables, the set $\Phi(X)$ of clock constraints φ is defined by the following grammar:

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2$$

where x is a clock in X and c is a constant in \mathbb{R} .

A clock interpretation ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to the set of nonnegative reals. We say that a clock interpretation ν for X satisfies a clock constraint φ over X iff φ evaluates to true according to the values given by ν . For $\delta \in \mathbb{R}$, $\nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$, and agrees with ν over the rest of the clocks.

A timed automaton A is a tuple $(S, S_0, \Sigma, X, I, E)$, where S is a finite set of states/locations; S_0 , a subset of S , is a set of initial states; Σ is a set of actions/events; X is a finite set of clocks; I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$; E , a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, \lambda, \delta \rangle$ represents a transition from state s to state s' on input symbol a . The set λ gives the clocks to be reset with this transition, and δ is a clock constraint over X that specifies when the switch is enabled.

An example of a timed automaton is shown in Figure 1, a gate in a security system. The gate has three states: *closed*, *open* and *opened* (we assume the gate slams shut instantly when directed, so there is no *close* state). State *closed* is the initial state, as indicated by there being an action into *closed* that emanates from no state. Through action *open-s*, (i.e. open start) the gate starts the operation of opening which it completes within 2 time units. When the opening operation is completed, the action *open-e* (i.e. open end) occurs and the gate becomes opened. Through action *close* the gate closes instantly when exactly 10 time units have elapsed since the gate was opened.

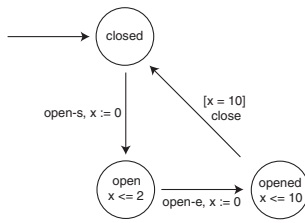


Figure 1. a gate

3 Combining Object-Z and TA

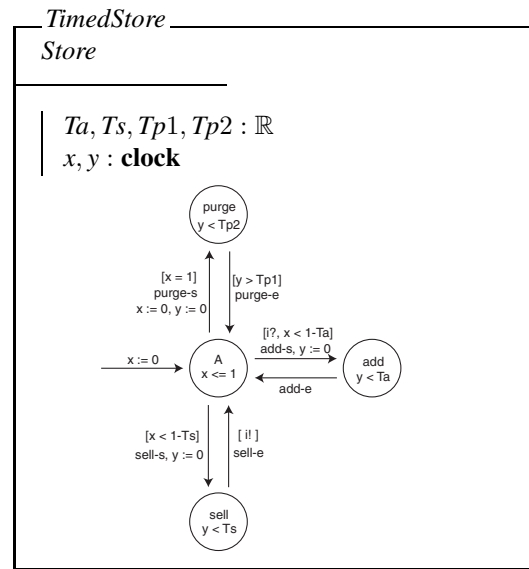
In this section, the semantic and syntactic issues on integrating Object-Z and TA are discussed and a combined notation is proposed.

To illustrate how Object-Z and TA can be effectively integrated, consider the simple stock-control system we met in the last Section. What we want is to integrate into this specification a notion of the sequential passing of time.

Suppose time is a positive real number measured in days starting at 0 (so, for example, 1.5 is halfway through the second day). The use-by date associated with an item is a positive integer denoting the day by which the item must be sold or else purged from the store (e.g. a use-by date of 3 means that if the item is not sold on or before day 3, at the start of day 4 it is purged).

We shall suppose it takes at most Ta time units to add an item to the stock, at most Ts time units to sell an item in stock, and more than $Tp1$ but less than $Tp2$ time units ($Tp1 < Tp2$) to purge the stock at the beginning of the day, where each of Ta , Ts and $Tp2$ is much less than 1. Furthermore, the addition of any item to the stock or the selling of any item in stock must be started and completed within the same day. In our model, operations of the store will be disjoint, i.e. time-wise they do not overlap.

The store with this timing information incorporated is specified by adding a Timed Automaton to the class *Store* to get the class *TimedStore* as shown. The top part of the class box is the standard Object-Z specification we met in the last section and contains no timing information. The bottom part of the class box contains a declaration of the timing constants and the names of the clocks (in this case there are two clocks, x and y) as well as the associated automaton. Declaration $x : \mathbf{clock}$ means that x plays a dual role: it identifies (i.e. names) a clock and also records the time showing on the clock, i.e. it is a variable that takes positive real number values. In fact, as we shall see, the value of the clock x in this specification always lies between 0 and 1 inclusive and denotes the time that has passed in the current day. The clock y is used to ensure that the operations are completed within the specified time. It is assumed both clocks progress at the same rate, i.e. the passage of time is universally uniform.



The locations of the automata represent the various situations in which the store can find itself. A location, together with the switches to and from that location, specifies the timing limits (if any) for the corresponding situation. For each of the three operations specified in the Object-Z part there is a similarly-labeled location to capture the situation when the store is undergoing this operation; the store can undergo this operation only when in the corresponding location. The other location, *A*, represents the situation when the store is idle and no operation is being performed.

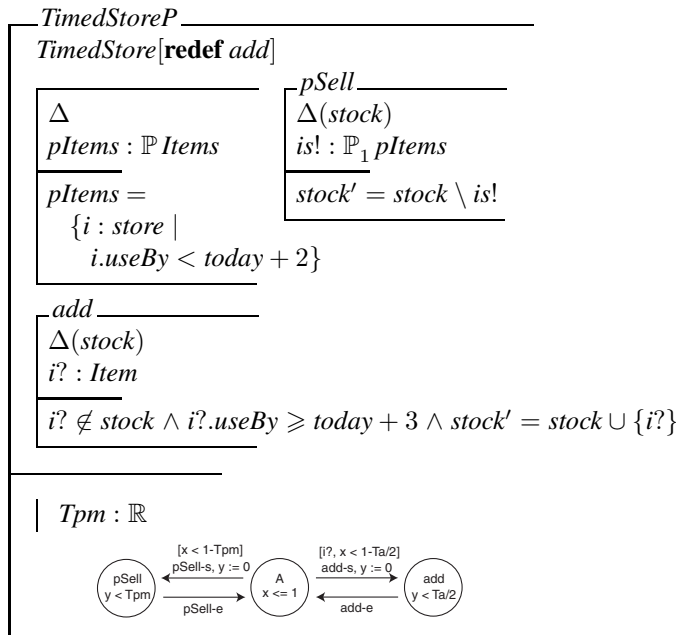
To illustrate the switches, consider those between locations *A* and *add*. The switch from *A* to *add* is labeled *add-s* (i.e. add start), while the switch from *add* to *A* is labeled *add-e* (i.e. add end). The expression in square brackets, i.e. $[i?, x < 1 - Ta]$ in the case of the switch labeled *add-s*, captures the requirements that must be met if the switch is to take place, i.e. the input item $i?$ (as defined in the Object-Z operation *add*) must be supplied, and in addition the time as recorded by the clock x must be less than $1 - Ta$ (so that the operation, which can take up to Ta time units to occur, can be completed within the same day). In addition, the precondition of the Object-Z operation *add* must hold for the *add-s* switch to occur. As the precondition of an operation must always hold before the switch to the place labeled by that operation's name can occur, this precondition is always implicitly conjoined with any specific additional requirements within the square brackets. When the switch from *A* to *add* occurs, the clock y is reset to 0; annotating the location *add* with the condition $y < Ta$ ensures that this location is exited within time duration Ta , as required.

For the operation *sell*, the supply of the output item $i!$ is a requirement that must be met for the switch *sell-e* to occur after the completion of the operation. Compare this with the *add* operation where the input $i?$ was required for the switch starting the operation to occur.

Looking now at the switch *purge-s*, this switch can occur only when x is 1. Furthermore, it must occur at this time because of the time restriction placed on location A. This ensures that the purge operation occurs precisely once a day (starting at the end of each day and the beginning of the next). When the switch does occur, the clock x is reset to 0 (ensuring that x always lies between 0 and 1 and hence denotes the time that has passed in the current day).

Location A is the automaton's initial location. The understanding is that the initial conditions as specified by the *Inv* schema must hold when the automaton is started in location A, and at the same time the clock x is set to 0 (the initial value of the clock y can be arbitrary and so is not specified). The fact that the 'start' switches associated with each operation emanate from location A and the 'end' switches each return to A, ensures that the operations *add*, *sell* and *purge* do not overlap time-wise.

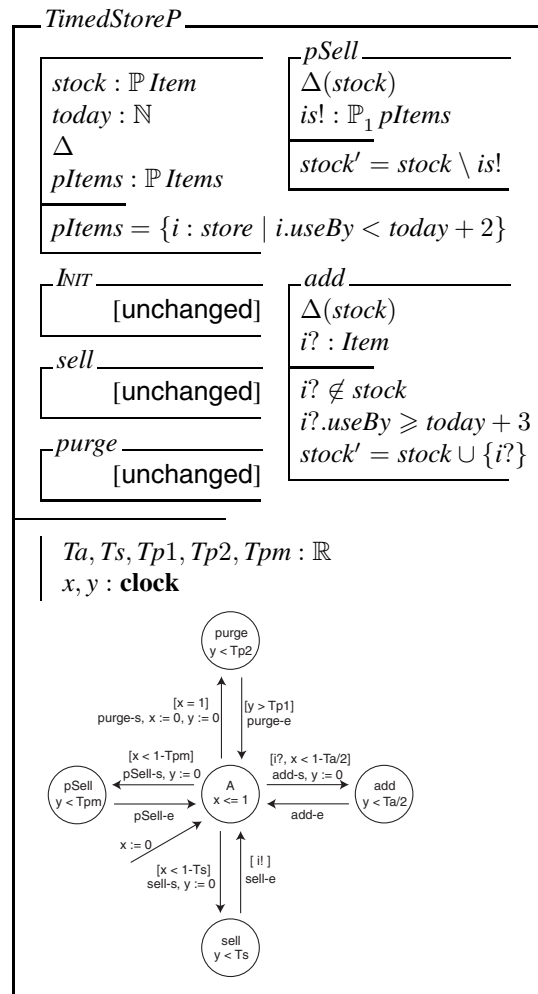
Note that the naming of switches can be systematic, e.g., a switch pointing to an operation state can be labeled with the operation name followed by 's' (for start). If a switch is pointing from an operation state to an idle (control) state, then it can be labeled with the operation name followed by 'e' (for end).



Inheritance

Inheritance is a mechanism for incremental specification and reuse, whereby new classes may be derived from an existing class. Object-Z inheritance has a similar style as the Z schema inclusion. We propose that the control behaviour (expressed by the TA) can also be inherited and extended in a simple way. Consider a system *TimedStoreP* which has the same *sell* and *purge* functionalities with *TimedStore* ex-

cept the *add* operation: only items with their expiration date at least 3 days ahead of the current day can be added into the store and the *add* operation takes less than a half of the Ta time units to finish. In addition, The system is able to identify the set *pItems* (promotion items) of items which have only two days left before their expiration. An extra operation *pSell* (promotion sell), which takes at most Tpm time units to execute, can sell a subset of these promotion items. The class *TimedStoreP* can be defined by inheriting the class *TimedStore*. For the behaviour (automaton) part, the *add* location refers to the redefined *add* operation and its local invariant changes to $y < Ta/2$ and enabling condition changes to $x < 1 - Ta/2$. And a new location *pSell* is introduced and is connected (by new switches) with the control (idle) location A from *TimedStore*. The other locations and their connections remain unchanged as follows: If we expand the inheritance, then *TimedStoreP* becomes:



Note that *pItems* is modelled as a secondary attribute whose value is subject to change with each operation (implicitly it is included in every operation's Δ list). For multiple inheritance cases, the rules are that all simi-

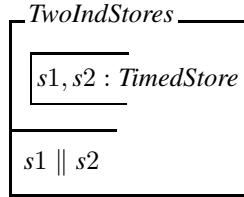
larly named locations (and switches) are merged, with all corresponding invariants and conditions conjoined.

4 Composition and Communication

In this section, various composition and communication aspects of the combined language are discussed and synchronized communication links are systematically introduced.

Independent stores

Consider now a system consisting of two stores operating independently. This system is specified by the class *TwoIndStores*:



The timed automaton of this class is simply the product [1] of the automata for the two stores *s1* and *s2*. The timed automaton for *s1* is just the automaton of the *TimedStore* class but with the label of each location, the label of each switch, and the names of the clocks distinguished by an ‘*s1.*’ prefix, as illustrated in Figure 2. Notice that the input/output variables are not prefixed.

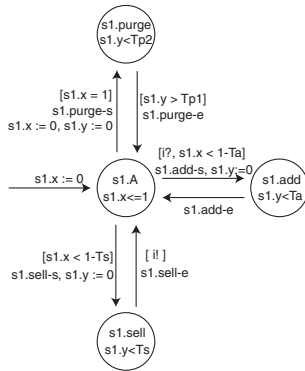
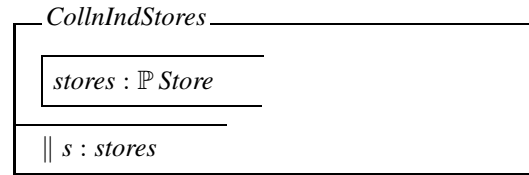


Figure 2. the automaton *s1*

The timed automaton for *s2* is labeled similarly. The *s1 || s2* notation in the class *TwoIndStores* denotes the product of the associated automata. The implication here is that the two stores are not only completely independent, but operations in different stores can be executed concurrently. Indeed, when an object of the class *TwoIndStores* is instantiated, the two store objects start at the same time in their *A* position with *s1.x* and *s2.x* set to 0 synchronously. As time passes at the same rate for all clocks, both stores will always

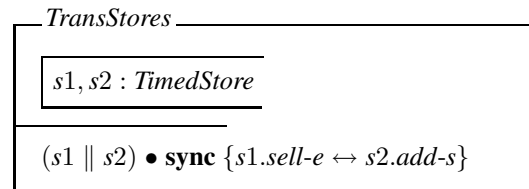
synchronise on the start of their respective *purge* operations, namely, at the start of the next day, but apart from that they run completely independently.

The two-stores example can be generalised to a collection of independent stores, as specified by the class *CollnIndStores*. In this class the expression $\parallel s : stores$ denotes the timed automata product ($s_1 \parallel s_2 \parallel \dots$) where the set *stores* is $\{s_1, s_2, \dots\}$.



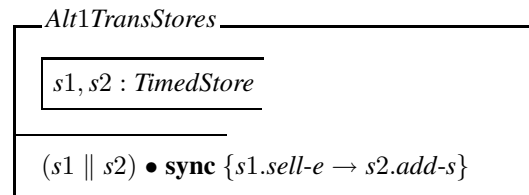
Transferring between stores

Consider now a system consisting of two stores, where each item sold by the first store is added (i.e. transferred) to the second. Effectively, the first store sells items only to the second store. A specification of this system is given by the class *TransStores*:



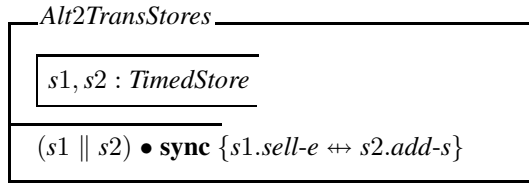
The **sync** clause indicates that the two switches labeled *s1.sell-e* and *s2.add-s* are to be treated as if these labels were identical, i.e. the automata must synchronize on these switches. As part of this synchronization, as the output *i!* and the input *i?* have the same base-name they are identified and hidden (just as is the case for the Object-Z parallel operator, i.e. they specify internal communication rather than communication with the environment). Apart from this synchronization, the product of the two timed automata effectively ensures that the two automata operate independently and concurrently.

Now consider a system like *TransStores* where again each item sold by the first store is added (i.e. transferred) to the second. However, an item from the environment may also be added to the second store, i.e. not all items added to the second store are necessarily transferring from the first. This system is specified in the class *Alt1TransStores*:



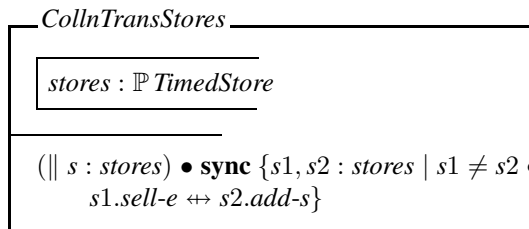
The implication here is that whenever the switch $s1.sell-e$ is taken then there must be synchronization with the switch $s2.add-s$. However, the switch $s2.add-s$ can occur independent of (i.e. without synchronizing with) the switch $s1.sell-e$. With this notation, notice that the synchronization **sync** $\{s1.sell-e \leftrightarrow s2.add-s\}$ in the *TransStores* class could have been alternatively (but less elegantly) expressed as **sync** $\{s1.sell-e \rightarrow s2.add-s, s2.add-s \rightarrow s1.sell-e\}$.

Now consider the situation as before where an item sold by the first store can be transferred to the second, but in addition not only can an item from the environment be directly added to the second store, (i.e. not all items added to the second store are necessarily transferring from the first) but also an item sold by the first store can be passed to the environment (i.e. not all items sold by the first store are necessarily transferred to the second). This system is specified in the class *Alt2TransStores*:



The implication here is that when any of the switches $s1.sell-e$ or $s2.add-s$ is taken there may or may not (the choice is non-deterministic) be synchronization with the switch $s2.add-s$ or $s1.sell-e$ respectively.

The examples involving two stores given so far in this section can be generalised to a collection of stores. Consider a system consisting of a collection of stores where an item from the environment can be added to any store, an item sold by any store can be passed back to the environment, and given any two stores in the collection, an item sold by the first store can be added (transferred) to the second. Such a system is specified in the class *CollnTransStores*:



More on synchronization

To further illustrate synchronization in timed automata, consider the three timed automata U , V and W illustrated in Figure 3. The timed automaton $(U \parallel V \parallel W) \bullet \text{sync } \{a \leftrightarrow b\}$ is behaviorally equivalent to the product $U1 \parallel V1 \parallel W1$ of the timed automata $U1$, $V1$ and $W1$ illustrated in Figure 4. In this case the switches labeled a and b have been

re-named to a common label d . As these labels are the same, the product automaton will synchronize on these switches. Consequently, the switch from location $u1$ to location $u2$ in $U1$ is always synchronized with the switch from location $v1$ to location $v2$ in $V1$, and conversely.

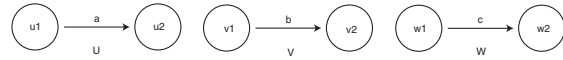


Figure 3. timed automata U , V and W

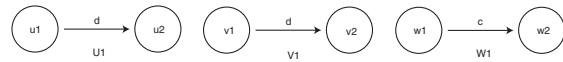


Figure 4. timed automata $U1$, $V1$ and $W1$

The timed automaton

$$(U \parallel V \parallel W) \bullet \text{sync } \{a \leftrightarrow b, a \leftrightarrow c, b \leftrightarrow c\}$$

is behaviorally equivalent to the product $U2 \parallel V2 \parallel W2$ of the 3 automata $U2$, $V2$ and $W2$ illustrated in Figure 5. In this case the switch from location $u1$ to $u2$ in $U2$ must synchronize with either the switch from location $v1$ to $v2$ in $V2$ or from $w1$ to $w2$ in $W2$; the switch from location $v1$ to $v2$ in $V2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $w1$ to $w2$ in $W2$; and the switch from location $w1$ to $w2$ in $W2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $v1$ to $v2$ in $V2$.

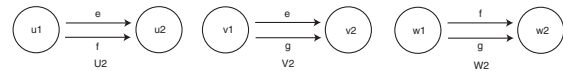


Figure 5. timed automata $U2$, $V2$ and $W2$

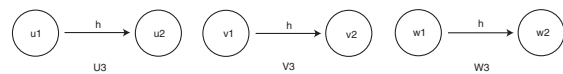


Figure 6. timed automata $U3$, $V3$ and $W3$

Compare this to the automaton

$$(U \parallel V \parallel W) \bullet \text{sync } \{a \leftrightarrow b \leftrightarrow c\}.$$

This automaton is behaviorally equivalent to the product $U3 \parallel V3 \parallel W3$ of the three automata $U3$, $V3$ and $W3$ illustrated in Figure 6. In this case the three switches from location $u1$ to $u2$ in $U3$, from location $v1$ to $v2$ in $V3$ and from location $w1$ to $w2$ in $W3$ must synchronize.

The timed automaton $(U \parallel V) \bullet \text{sync } \{a \rightarrow b\}$ is behaviorally equivalent to the product $U4 \parallel V4$ of the timed automata $U4$ and $V4$ illustrated in Figure 7. In $V4$ a switch

labeled a is added to duplicate the switch labeled b . As the switch in automaton $U4$ is also labeled a , this ensures that the product automaton will synchronize on these two switches. Consequently, the switch from location $u1$ to location $u2$ in $U4$ is always synchronized with a switch from location $v1$ to location $v2$ in $V4$, but not conversely. The transition from location $v1$ to location $v2$ can use the switch labeled b in which case no synchronization takes place.



Figure 7. timed automata $U4$ and $V4$



Figure 8. timed automata $U5$ and $V5$

The timed automaton $(U \parallel V) \bullet \text{sync } \{a \leftrightarrow b\}$ is behaviorally equivalent to the product $U5 \parallel V5$ of the timed automata $U5$ and $V5$ illustrated in Figure 8. In this case both of the switches labeled a and b are duplicated and a common name, d , is assigned to these new switches. This ensures that the product automaton will synchronize on these two switches. Consequently, a transition from location $u1$ to location $u2$ in $U5$ can synchronize with a transition from location $v1$ to location $v2$ in $V5$ if the switch labeled d is used. However, a transition from location $u1$ to location $u2$ in $U5$ could use switch a , or a transition from location $v1$ to location $v2$ in $V5$ could use switch b ; in either case no synchronization takes place.

5 Semantics

In this section we present a formal description of the operational behavior of this integrated language. The fundamental semantic links between Object-Z and TA are:

- Object-Z operations are identified as states in Timed Automata.
- Pre/Post-conditions of an Object-Z operation are identified to TA transition conditions.

The key novel idea of integrating the Object-Z semantics and TA semantics is to embed object state updates (of Object-Z) into the action transition semantics of TA. To facilitate the description of dynamic behaviors of a system, we introduce a set of locations A , called control locations, to coordinate the location switches from one Object-Z operation to another. Each location of a timed automaton specified in a class must be either a control location or an

Object-Z operation location. A class has an Object-Z part $OZDefinition$ which obeys the conventional definition [26]. $OZop$ represents the set of Object-Z operations defined in the class. The original Object-Z operation operators: parallel composition, nondeterministic choice, sequential composition are replaced by $TADefinition$, which is a timed automaton:

\mathbb{S}_{OZTA} is a tuple $(S, S_0, \Sigma, X, I, E)$, where

- S is a union of A and Op , in which A is a finite set of control (idle) states and Op is a finite set of operation states correspond to the Object-Z operations
- S_0 , a subset of S , is a set of initial locations
- Σ is a set of labels
- X is a finite set of clocks
- I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$
- E , a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, r, \varphi \rangle$ represents a transition from location s to location s' on input symbol a . The set r gives the clocks to be reset with this transition, and φ is a clock constraint over X that specifies when the switch is enabled.

Operational Semantics

In this subsection, we present a timed transition system \mathbb{S}_{OZTA} to represent operational semantic models for this integrated language. Before we start to define the operational semantics, we need some definitions for the validity of Object-Z and TA expressions.

The fact that a state guard G is valid under the semantic function $\sigma : Var \leftrightarrow Value$ is denoted as $\sigma \models G$, which reads that G is valid under the semantic function σ . The fact that an operation Op is valid under the semantic functions σ_1, σ_2 is denoted as $\sigma_1, \sigma_2 \models Op$. For example, in the context of the Store system,

$$\begin{aligned} & \{(store, \{item_a, item_b\}), (today, 20)\}, \\ & \{(store, \{item_a, item_b, item_c\}), (today, 20)\} \models add[i? \mapsto item_c] \end{aligned}$$

To keep track of the changes of clock values, we use functions known as clock assignments mapping X to the non-negative reals R_+ . Let u, v denotes such functions, and use $u \models \varphi$ to mean that the clock values denoted by u satisfy the guard φ . For $d \in R_+$, let $u + d$ denote the clock assignment that maps all $x \in X$ to $u(x) + d$, and for $r \subseteq X$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and agree with u for the other clocks in $X \setminus r$.

To facilitate the description of operational semantics, let

$$\mid OP : Location \leftrightarrow OZop$$

denote the association between TA locations to Object-Z operations.

The operational semantics of this integrated language is an extension of TA transition semantics coupled with object states. The timed state transition system \mathbb{S}_{OZTA} consists of states which are tuples $\langle l, u, \sigma, \sigma_1 \rangle$ and state transitions are defined by the rules:

$$R_1 : \frac{l \stackrel{a, \varphi, \sigma_1}{\longrightarrow} l' \quad \sigma_1, \sigma_2 \models OP(l) \quad \sigma_1, \sigma_2 \models OP(l') \quad u \models \varphi \quad u' = [r \rightarrow 0]u \quad u' \models I(l') \quad l, l' \in Op}{\langle l, u, \sigma, \sigma_1 \rangle \xrightarrow{a} \langle l', u', \sigma_1, \sigma_2 \rangle}$$

R_1 is an action transition from one operation (location) state l to another operation state l' where the post object state of l must be the same as the pre object state of l' , the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_2 : \frac{\sigma_1, \sigma_2 \models OP(l) \quad u \models I(l) \quad u + d \models I(l) \quad d \in R_+ \quad l \in Op}{\langle l, u, \sigma_1, \sigma_2 \rangle \xrightarrow{d} \langle l, u + d, \sigma_1, \sigma_2 \rangle}$$

R_2 is a delay transition in a certain operation state where only time is progressed.

$$R_3 : \frac{l \stackrel{a, \varphi, \sigma_1}{\longrightarrow} l' \quad u \models \varphi \quad u' = [r \rightarrow 0]u \quad u' \models I(l') \quad l \in A \quad l' \in A}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{a} \langle l', u', \sigma, \sigma \rangle}$$

R_3 is a transition from one control (location) state l to another control state l' where object states remain the same.

$$R_4 : \frac{u \models I(l) \quad u + d \models I(l) \quad d \in R_+ \quad l \in A}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{d} \langle l, u + d, \sigma, \sigma \rangle}$$

R_4 is a delay transition in a control state where time is progressed.

$$R_5 : \frac{l \stackrel{a, \varphi, \sigma_1}{\longrightarrow} l' \quad \sigma_1, \sigma_2 \models OP(l) \quad u \models \varphi \quad u' = [r \rightarrow 0]u \quad u' \models I(l') \quad l \in Op \quad l' \in A}{\langle l, u, \sigma_1, \sigma_2 \rangle \xrightarrow{a} \langle l', u', \sigma_2, \sigma_2 \rangle}$$

R_5 is an action transition from one operation (location) state l to a control state l' where the post object state of l must be the same as the object state of l' , the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_6 : \frac{l \stackrel{a, \varphi, \sigma_1}{\longrightarrow} l' \quad \sigma_1, \sigma_2 \models OP(l') \quad u \models \varphi \quad u' = [r \rightarrow 0]u \quad u' \models I(l') \quad l \in A \quad l' \in Op}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{a} \langle l', u', \sigma, \sigma_1 \rangle}$$

R_6 is the inverse of R_5 .

These rules define six types of transitions in \mathbb{S}_{OZTA} . These rules are applied to a single timed transition system. A complex system can be described as a product of interacting timed transition systems. The communications between two transition systems are obtained by synchronizing the transition with identical labels.

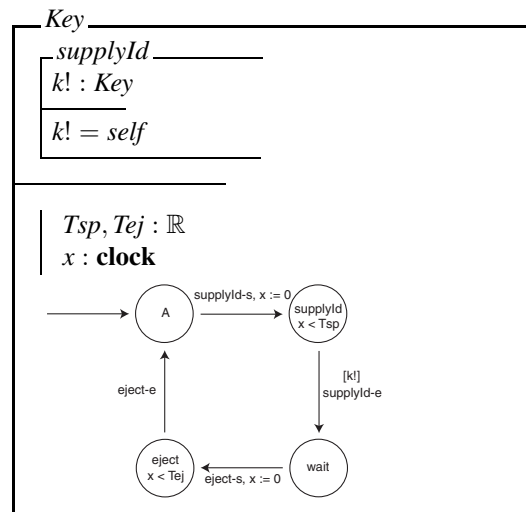
6 A Case Study

As an illustration of how Object-Z and TA can be successfully integrated in practice, we present here a case study of an electronic key system.

A room can be accessed through a sliding door. To open the sliding door, an electronic key is inserted into the door's electronic lock. The identity of the key (as encoded as part of the key) is passed to the lock that then checks to see if the key has permission to access the room. When access permission has been checked the key is ejected from the lock. If the key has access permission the door is opened (or remains open); otherwise the door is closed (or remains closed).

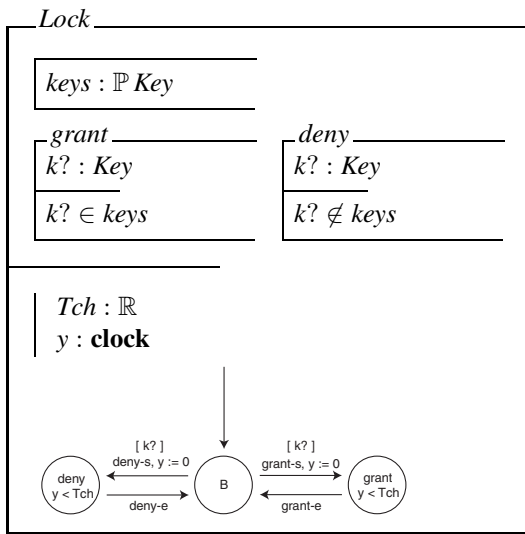
We shall suppose that it takes less than Tsp time units from the time the key is inserted in the door's lock for the key to supply its identity to the lock, less than Tch time units for the lock to check if the key has permission to access the room, less than Tej time units for the key to be ejected from the lock, and less than Top time units for the door to satisfy an 'open' request. Also, if the door has been open Tto time units since the last 'open' request, a time-out occurs and the door is closed. It takes less than Tcl time units for the door to satisfy a 'close' request.

A key is specified by the class *Key*:

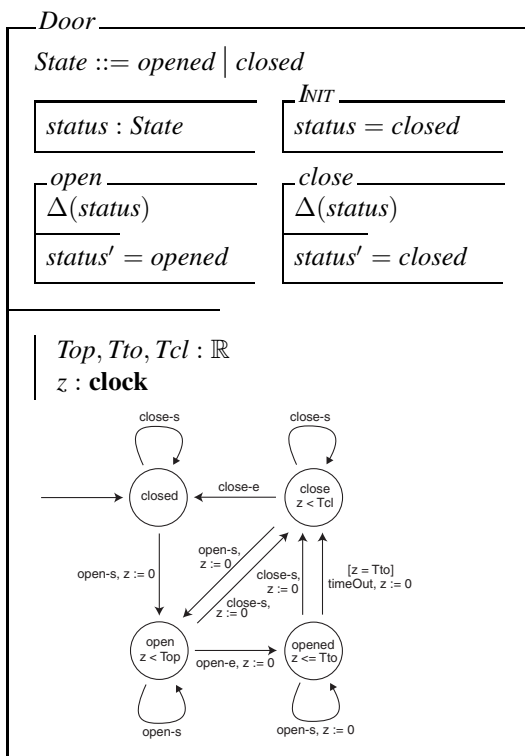


The only operation in the Object-Z section of this class is *supplyId* specifying the situation in which a key supplies its identity (to the lock). When considering time aspects, however, other situations arise. A key will be in location *wait* after it has supplied its identity and is waiting to see whether or not access is granted. A key will be in location *eject* when it is being ejected from the lock once access permission has been decided.

The lock is specified by the class *Lock*. The attribute *keys* in this class denotes the set of keys that have permission to access the room. The operations *grant* and *deny* capture whether or not any supplied key is in this set, and hence whether or not access to the room is granted or denied.



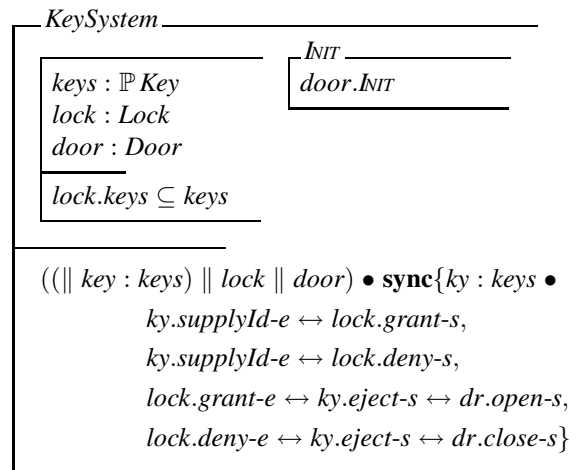
The door is specified by the class *Door*:



A door can be in any of four situations: closed (location *closed*) opening (location *open* where the operation *open* occurs) opened (location *opened*) and closing (location *close* where the operation *close* occurs). In each situation, the door can receive an instruction to open or close the door. In all cases, when an instruction to open the door is received the switch *open-s* is taken, while if an instruction to close the door is received the switch *close-s* is taken. In locations *closed* or *close*, if the instruction to open is received the operation *open* is invoked, while if the instruction

to close is received effectively the door continues as if nothing had happened. In location *open*, if the instruction to open is received effectively the door continues as if nothing had happened, while if the instruction to close is received the operation *close* is invoked. In location *opened*, if the instruction to open is received the door remains open but the timing is reset to 0, while if the instruction to close is received the operation *close* is invoked.

The complete electronic key system can now be specified by the class *KeySystem*. In this class, the attribute *keys* denotes the set of all keys in the system; the set of keys that have permission to access the room will be a subset of *keys*. The synchronization conditions ensure that the key identity output by a key is passed to the lock and used to determine whether or not that key has permission to access the room, and that once the access permission has been decided the key is ejected and the door requested to open or close, depending on whether access was granted or denied.



7 Related Works and Conclusion

This research can be classified as one of the integrated formal methods (IFM). The IFM research area has been active for a number of years (e.g. [2, 14, 6, 4]) with a particular focus on integrating state based and event based formalisms (e.g. [13, 31, 27, 32, 35]).

One closely related area to ours is the research work on integration of Object-Z with various timed calculi. For example, Object-Z is combined with Timed CSP [25] in [22, 8], with timed refinement calculus [21, 12] in [28] and with duration calculus [37] in [18]. Indeed, those combinations have made improvements in comparison to some early conservative framework approaches [9, 23].

The technical difference between our approach to the others has been the way to clearly separate functionalities and timed behaviour and the use of graph based Timed Automata instead of the timed calculi to capture the behaviour.

One clear benefit of our approach is that many existing well developed tools [3, 7, 30, 33] for TA can be used to check the timed behaviour of the design models. In addition to the benefit of bring graphical appeal in capturing the object behaviour of Object-Z classes, our approach also provide a way to structure TA automaton using Object-Z classes so that the scale up problem of TA can be managed. The novel communication mechanism developed in our approach is also more flexible and expressive than CSP channels. For example, arbitrary communication between various objects can be captured at the composite class level with the elegant communication links.

Another related work is the combination of Z with graphical diagrams, i.e. statecharts and petri nets. For example, in [5] a framework is presented to link Z with statecharts and treats Z operation schemas as state transition links in statecharts. Similarly, the language OZS [15] blends Object-Z with statecharts and treats Object-Z operations as state transition links. Combinations of Z and Petri Nets have also been investigated in [17, 16]. All those approaches suffer a common drawback that the states in the graph have no systematic correspondence in Z or Object-Z parts. The issues of object composition and timing have not been addressed. Our approach is different: we treat Object-Z operations as states (TA locations) instead of state transitions (TA switches) and furthermore we have a systematic naming convention for all switches. Object composition and real-time issues are the main focus points in our approach. In our future work, we plan to investigate the refinement techniques for this combination of Object-Z and TA. We also plan to develop various tools to support the combination, e.g. to develop an OZTA editor and various linking programs to the Object-Z tools and TA model checkers. For the OZTA editor, we chose to represent the OZTA specification information in an XML format to enable easy linkage with Object-Z and TA tools. Some of our on-going tool development can be accessed here [19].

References

- [1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [2] K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
- [3] J. Bengtsson, K. G. Larsen, F. Larsson, and P. Pettersson and Y. Wang. UP-PAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control*, pages 232–243. Springer, 1996.
- [4] E. Boiten, J. Derrick, and G. Smith, editors. *IFM'04: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, April 2004.
- [5] R. Bussow and W. Grieskamp. A Modular Framework for the Integration of Heterogeneous Notations and Tools. In Araki et al. [2], pages 211–230.
- [6] M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2002.
- [7] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, pages 208–219. Springer, 1996.
- [8] J. Derrick. Timed csp and object-z. In *3rd International Conference of Z and B Users (ZB'03)*, LNCS. Springer, June 2003.
- [9] J. S. Dong, J. Colton, and L. Zucconi. A Formal Object Approach to Real-Time Specification. In *the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, Seoul, Korea, December 1996. IEEE Press.
- [10] J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi. Timed Patterns: TCOZ to Timed Automata. In Jim Davies and Wolfram Schulte, editors, *The 6th International Conference on Formal Engineering Methods (ICFEM'04)*, Seattle, USA, November 2004.
- [11] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.
- [12] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhurst. A set-theoretic model for real-time specification and reasoning. In *Mathematics of Program Construction*, 1998.
- [13] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [2].
- [14] W. Grieskamp, T. Santen, and B. Stoddart, editors. *IFM'00: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.
- [15] J. P. Gruer, V. Hilaire, A. Koukam, and P. Rovarini. Heterogeneous formal specification based on object-z and startchart: semantics and verification. *J. Systems and Software*, 2004.
- [16] X. He. Pz nets a formal method integrating petri nets with z. *Information & Software Technology*, 43(1):1–18, 2001.
- [17] M. Heiner and M. Heisel. Modeling safety-critical systems with Z and Petri nets. In *International Conference on Computer Safety, Reliability and Security, LNCS*. Springer, pages 361–374, 1999.
- [18] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes, Data and Time. In Butler et al. [6].
- [19] NUS Software Engineering Lab. OZTA tools. <http://nt-appn.comp.nus.edu.sg/fm/ozta/introduction.htm>.
- [20] K. Lüth, E. W. Karlson, Kolyang, S. Westmeier, and B. Wolff. Hol-Z in the UniForm-workbench – a case study in tool integration for Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lect. Notes in Comput. Sci.*, pages 116–134. Springer-Verlag, 1998.
- [21] B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, University of Queensland, 1991.
- [22] 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 Press.
- [23] K. Periyasamy and V.S. Alagar. Adding Real-Time Filters to Object-Oriented Specification of Time Critical Systems. In *the 1998 IEEE Workshop on Industrial-strength Formal specification Techniques*, Boca Raton, Florida, USA, October 1998. IEEE Press.
- [24] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.
- [25] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.
- [26] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
- [27] G. Smith and J. Derrick. Specification, Refinement and Verification of Current Systems — An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.
- [28] G. Smith and I. Hayes. Towards Real-Time Object-Z. In Araki et al. [2].
- [29] G. Smith, F. Kammüller, and T. Santen. Encoding object-z in isabelle/hol. In *2nd International Conference of Z and B Users (ZB'02)*, LNCS. Springer, 2002.
- [30] M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proceedings of Workshop on Real-time Tools*, August 2001.
- [31] C. Suhl. RT-Z: An integration of Z and timed CSP. In Araki et al. [2].
- [32] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Press.
- [33] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *Proceedings of the 7th Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 546–562. Springer, 1996.
- [34] K. Winter and R. Duke. Model Checking Object-Z Using ASM. In Butler et al. [6], pages 165–184.
- [35] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *2nd International Conference on Z and B*, volume 2272 of *Lect. Notes in Comput. Sci.*, pages 184–203. Springer-Verlag, 2002.
- [36] X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-time Specifications into Extended Automata. In *IEEE TSE Special Issue on Real-Time Systems*, volume 18(9), pages 794–804, 1999.
- [37] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.