Timed CSP and Object-Z

John Derrick

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 1227 764000, Email: J.Derrick@ukc.ac.uk.)

Abstract. In this paper we discuss a simple integration of timed CSP and Object-Z. Following existing work, the components in such an integration are written as either Object-Z classes, or timed CSP processes, and are combined together using CSP parallel composition.
Here we discuss the approach in general, and describe how the semantics of timed CSP can be used as the semantics of the integrated notation. We briefly discuss verification and analysis for integrated descriptions, before providing a more in-depth discussion of refinement in this approach. We describe both refinement of individual components, as well as a two-event model which distinguishes between start and end events. The latter allows operation duration to be specified and we show how refinement in this model integrates into traditional state-based simulation rules.

Keywords: Timed CSP; Object-Z; Refinement.

1 Introduction

In this paper we discuss an integration of timed CSP [15] and Object-Z [18]. The motivation for such a combination is a recognition that some of the hardest implementation issues in computer science involve the correct real-time scheduling of complex interactions. This has led to considerable work on notations which support the specification of timing of events. These include timed process algebras, such as timed CSP, as well as alternatives, such as timed automata [12], which offer related but distinct approaches to concurrency, interaction and time.

There has also been relevant work on integrating notations in order to provide a suitably expressive medium supporting a range of concerns such as timing, temporal ordering as well as data handling. Examples of these include integrations of CSP [10] and Object-Z, such as those described by Smith [17], Fischer [6, 8] and Mahony and Dong [13].

In this paper we extend work in this area by investigating combinations of timed CSP and Object-Z which builds upon the integration of untimed CSP and Object-Z due to Smith [17] and Smith and Derrick [19, 20].

In [17, 19, 20] Object-Z is used to specify the component processes of a system, and CSP is used to describe how the components synchronise and interact. The failures-divergences semantics of CSP is used to provide the integration with a sound semantic basis. This approach can be extended [4] by using CSP processes to describe the desired temporal ordering, in addition to describing the component integration. Here we extend this in the obvious fashion by using *timed* CSP to describe timing requirements on the events in the component processes, which are again specified in Object-Z. We consider a number of ways in which this can be achieved as well as discussing verification and refinement strategies.

Section 2 describes this integration via a simple example, and in Section 3 we discuss the obvious semantic basis for such an integration, i.e., the timed failures semantics of timed CSP. This is achieved by embedding the failures-divergences of an Object-Z component (as derived in [17, 19, 20]) into the timed failures model consisting of timed traces and timed refusals.

Subsequently, in Section 4, we consider to what extent verification and analysis techniques from timed CSP can be used on the integrated notation as well as discussing the compositional refinement of components.

Thus far the integrated language will have been described in terms of a simple and obvious correlation between CSP events and Object-Z operations. This so called one-event model, takes an atomic view of operations and is satisfactory for a simple specification and analysis of a specification. However, occasionally a more complex model is necessary and Section 5 discusses a two-event model whereby we are explicit about the beginning and termination of operations, and thus each operation is mapped to two events. We discuss in some detail refinement in this model and its relation with refinement in the one event model. Finally, we conclude in Section 6.

2 Combining timed CSP and Object-Z

Object-Z is an extension of Z designed to support an object-oriented specification style. Throughout this paper we assume the reader is familiar with the Object-Z notation, introductions are provided by [5] and [18]. CSP on the other hand is a process algebra which models a system as a collection of processes which run concurrently, communicate over unbuffered channels and synchronise on particular events. Both [14] and [15] provide comprehensive introductions to the language.

As indicated in the introduction, we can combine these two languages by using Object-Z to describe the components and CSP to describe their interaction. The integration is described in some depth in [17, 19, 20], and we illustrate its essence with an example based around the classic lift case study [9].

We have a number of floors defined by the following:

$$
\begin{array}{|l}
minFloor, maxFloor : \mathbb{N} \\
\hline
minFloor < maxFloor
\end{array}
$$

$FLOOR == minFloor..maxFloor$

Two of the components are given as Object-Z classes: *Lift* and *Con*. *Con* represents the central controller, operations *Close* and *Open* work the doors, and users external to the lift can *Request* a floor to go to. To communicate which floor the lift should be sent to a *SetTarget* operation is used, and the lift communicates which floor it has arrived at when it stops by using the *Arrive* operation.

The lift on the other hand, also opens and closes its own doors, and can accept requests from those inside it. The movement of the lift is modelled by two operations: *MoveOneFloor* moves the lift one floor at a time, and *Arrive* halts the lift at the desired target. This target is chosen from either the requests inside the lift or from the requests made to the controller. This is achieved by using two operations: *InternalTarget* and *SetTarget*. Only one of these will be invoked per lift movement cycle, and in the case of the latter it will synchronise with the same named operation in the controller.

This gives two components, the final one is described directly as a CSP process, which we use simply to specify the correct cycle of door opening and lift movement. CSP serves much better for this purpose than using, for example, Boolean variables in the Object-Z classes. Our top level description will thus be $LiftSys = Order \| Lift \| Con$, where:

$STATUS ::= open \mid closed$

Lift

$pos, target : FLOOR$
$req : \mathbb{P}\, FLOOR$
$door : STATUS$

INIT

$pos = minFloor \wedge door = open \wedge req = \varnothing \wedge target = minFloor$

RequestD

$\Delta(req)$
$f? : FLOOR$

$req' = req \cup \{f?\}$

Close

$\Delta(door)$

$door = open \wedge door' = closed$

Open

$\Delta(door)$

$door = closed \wedge door' = open$

$$\begin{array}{|l}
\hline
\textit{SetTarget} \\
\Delta(target) \\
f? : FLOOR \\
\hline
target' = f? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{InternalTarget} \\
\Delta(target) \\
\hline
target' \in req \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{MoveOneFloor} \\
\Delta(pos) \\
\hline
pos \neq target \\
pos > target \Rightarrow pos' = pos - 1 \\
pos < target \Rightarrow pos' = pos + 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{Arrive} \\
\Delta(req) \\
f! : FLOOR \\
\hline
pos = target \wedge f! = pos \\
req' = req \setminus \{pos\} \\
\hline
\end{array}$$

Con

$$\begin{array}{|l}
\hline
req : \mathbb{P}\, FLOOR \\
pos : FLOOR \\
doors : FLOOR \rightarrow STATUS \\
\hline
\neg \exists\, i, j : FLOOR \bullet doors(i) = open \wedge doors(j) = open \wedge i \neq j \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textsc{Init} \\
req = \varnothing \wedge pos = minFloor \wedge doors(minFloor) = open \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{RequestE} \\
\Delta(req) \\
f? : FLOOR \\
\hline
req' = req \cup \{f?\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{SetTarget} \\
f! : FLOOR \\
\hline
f! \in req \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Open} \\
\Delta(doors) \\
\hline
doors' = doors \oplus \{pos \mapsto open\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Close} \\
\Delta(doors) \\
\hline
doors' = doors \oplus \{pos \mapsto closed\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{Arrive} \\
\Delta(req, pos) \\
f? : FLOOR \\
\hline
req' = req \setminus \{f?\} \\
pos' = f? \\
\hline
\end{array}$$

$$\begin{array}{ll}
Order = & InternalTarget \rightarrow Q \\
& \Box \\
& SetTarget \rightarrow Q
\end{array}$$

$$Q = Close \rightarrow R$$

$$\begin{aligned} R = {} & MoveOneFloor \rightarrow R \\ & \Box \\ & Arrive \rightarrow Open \rightarrow Order \end{aligned}$$

This produces an elegant and simple style of specification (process instantiation is achieved via parameterised classes, see [17]), with an intuitive semantic basis (as discussed below). The CSP has been used to describe synchronisation between components and the (untimed) temporal ordering. It is an obvious extension to add any necessary timing information by using timed CSP.

Timed CSP is an extension of CSP which contains additional primitives designed to support real-time specification. To the standard CSP primitives a number of new timing constructs are added including, for example, timeout, delay and timed interrupt.

To illustrate these consider adding timing information to the lift case study. The timing requirements we use are taken from [13] (for comparison with another integrated notation). In particular, we require:

- Once the lift door is open, it must remain so for at least t_0 time units before closing.
- Lift travel time between two consecutive floors is a constant (given by time t_1), but there is a constant time delay (given by t_2) for acceleration and braking.
- Any passengers inside the lift are given a period of time, t_3, to make an internal request before the lift accepts any external requests.

These can be expressed within a timed CSP model by the process $TOrder$, defined by[1]:

$$TOrder = P \| CD$$

$$\begin{aligned} P = {} & InternalTarget \rightarrow Q \\ & \Box \\ & WAIT\ t_3;\ SetTarget \rightarrow Q \end{aligned}$$

$$Q = Close \rightarrow R$$

$$\begin{aligned} R = {} & MoveOneFloor \xrightarrow{t_1} R \\ & \Box \\ & Arrive \xrightarrow{t_2} Open \rightarrow P \end{aligned}$$

$$CD = Open \rightarrow WAIT\ t_0;\ Close \rightarrow CD$$

[1] To make the subsequent presentation easier, we have explicitly avoided issues concerned with hiding some of the events such as $MoveOneFloor$.

The process P illustrates the delay operator, and the behaviour of $WAIT\ t_3;\ SetTarget \rightarrow Q$ is such that it delays for precisely t_3 time units before evolving to the behaviour $SetTarget \rightarrow Q$. That is, after t_3 time units (and not before), *SetTarget* will be enabled.

The process $Arrive \xrightarrow{t_2} Open \rightarrow P$ illustrates another sort of delay, i.e. a delay after the occurrence of an event. Here *Arrive* happens at any point in time, but *Open* is not enabled until t_2 time units have elapsed after the occurrence of *Arrive*. A full description of the operators is given in [15].

Having expressed our timing requirements very simply in timed CSP, we can offer the obvious integration with the Object-Z component processes to define a specification complete with timing requirements as follows:

$$TLiftSys = TOrder \| Lift \| Con$$

Postponing for the moment a discussion of the semantics, the informal meaning of this specification should be clear. The temporal *ordering*, communication and effect of this specification is identical to that of *LiftSys*, however, the use of *TOrder* has introduced some precision into the time that events are performed within the lift and the controller. To confirm that our intuition of the meaning of this specification we need to consider the semantics of such an integration, and the next section considers this point.

3 Semantics

The combined Object-Z and CSP notation has been given a semantics as described in more detail in [17, 19]. Combined Object-Z and CSP specifications are given a well-defined meaning by giving the Object-Z classes a failures-divergences semantics identical to that of a CSP process. In a failures-divergences semantics a process is modelled by a triple (A, F, D) where A is its alphabet, F its failures and D its divergences. The failures of a process are pairs (t, X) where t is a finite sequence of events that the process may undergo, and X is a set of events the process may refuse to perform after undergoing t.

To integrate Object-Z components in a failures-divergences semantics, the failures of an Object-Z class are derived from its *histories semantics*, as defined for Object-Z classes in [16]. A history is a non-empty sequence of states together with a sequence of operations. Either both sequences are infinite or the state sequence is one longer than the operation sequence, and the histories of a class with states S and operations O can be represented as a set $H \subseteq S^\omega \times O^\omega$ such that a number of properties hold (e.g., the histories are prefixed closed).

In order to derive the failures of a class from its histories, Object-Z operations are mapped to CSP events using the following function which turns an operation op with assignment of values to its parameters p to the appropriate event:

$$event((op, p)) = op.\beta(p)$$

The meta-function β replaces each parameter name in p by its basename, i.e., it removes the ? or !. Thus the event corresponding to an operation (op, p) is a communication event with the operation name op as the channel and an assignment of values to the basenames of the operation's parameters as the value passed on that channel. For example, the event corresponding to an object of class *Lift* requesting a floor l is $RequestD.\{(f, l)\}$.

It is then possible to define the failures directly in terms of the histories, and [19, 20] describes how this is done in detail. Since Object-Z does not allow hiding of operations (hiding is only possible at the CSP level), divergence is not possible within a component. Therefore, a class is represented by its failures together with empty divergences.

For example, we can calculate the failures of the *Lift* class. The traces would contain elements such as (assuming suitable values for l, k, n)

$$\begin{array}{l}\langle Open, RequestD.\{(f, l)\}, RequestD.\{(f, k)\}, SetTarget.\{(f, n)\},\\ \qquad\qquad InternalTarget, Close, MoveOneFloor, Arrive.\{(f, l)\}\rangle\end{array}$$

and failures such as

$$\begin{array}{l}(\langle\,\rangle, \{Open, InternalTarget, MoveOneFloor\})\\ (\langle Close, RequestD.\{(f, l)\}\rangle, \{Close, MoveOneFloor\})\end{array}$$

This approach can be extended in an obvious fashion, and we will give combined timed CSP / Object-Z specifications a similar semantics by embedding the failures derived from the Object-Z components into a timed failures model, which is the semantic basis for timed CSP.

3.1 The timed failures semantic model

Timed CSP adopts a model of time in which events are instantaneous, time is modelled by the reals and passes with respect to a single global clock. The meaning of a timed CSP process is given by its timed failures, where in a manner similar to the untimed failures model, the timed failures model consists of timed traces together with timed refusals.

A timed trace is a sequence of timed events, where a timed event consists of a time together with an event, in which the times are non-decreasing. For example, if $t_3 = 4$, then $\langle(4, SetTarget), (4.5, Close), (4.5, MoveOneFloor)\rangle$ is a timed trace of the process P given above. $ttrace(P)$ denotes the set of timed traces of a timed CSP process P, and $trace(Q)$ denotes the set of traces of a (untimed) CSP process Q.

A timed refusal is a record of which timed events were refused during an execution, and thus a refusal set $\aleph$ is a set of timed events. The refusal of an event a at time t means that a is not possible at the state reached at time t, where at any time the refusal information is subsequent to the events performed at that time. Refusal sets of well-founded processes are conveniently represented by combinations (in particular, finite unions) of refusal tokens, the latter being sets of the form $[t_1, t_2) \times A$, where A is a set of events.

The timed failures of a process can be derived from its executions, and as expected a trace in the timed failures will be a sequence of events occurring in an execution, and a timed refusal will be a set of timed events which could be refused during that execution.

For example, the timed failures of the process P include:

$$
\begin{array}{l}
(\langle\,\rangle, [0, \infty) \times \{Close\}) \\
(\langle(t, InternalTarget)\rangle, ([0, t) \times \{Close\}) \;\; \cup \\
\qquad\qquad\qquad ([t, \infty) \times \{SetTarget, InternalTarget\}))
\end{array}
$$

A number of projection operators are defined on timed traces, including $\uparrow$ and $\upharpoonright$, where $s \uparrow D$ projects the trace onto an interval (i.e., produces a subtrace of s whose timings belong to D), and $s \upharpoonright t = s \uparrow [0, t]$. In addition, for finite traces, $end(s)$ returns the time of the last event in the trace s. $strip(s)$ is the sequence extracted from s by removing the timing information. Projections are also defined on refusals, e.g., $\aleph \upharpoonright A$ restricts the refusal set to events contained in A. $\aleph \upharpoonleft t$ is the refusal set at and after t, and $\aleph \mid\upharpoonright t$ the refusal set strictly before t. $s \downarrow A$ denotes the number of A events in s.

The set of events that appear at some time in $\aleph$ is denoted $\sigma(\aleph)$, and Σ denotes the universal set of events.

The timed failures of a timed CSP process P are denoted $\mathcal{TF}[\![P]\!]$ and the (untimed) failures of a CSP process Q are denoted $\mathcal{F}[\![Q]\!]$.

In order to give a combined timed CSP and Object-Z specification a well-defined meaning, we embed the failures of an Object-Z component into the timed failures semantics of timed CSP. A complete specification such as *TLiftSys* will therefore have a meaning in terms of timed failures.

In order to embed failures into timed failures we can use timed refinement relations which map untimed models into timed models (for an overview see [15]). There are a number of these *timewise refinement relations* that can be used depending upon whether one wants to preserve just trace information or, in addition, include refusal information, but in each case the essence is that a timewise refinement of a process will be consistent with the untimed description.

Trace information. If we wish to consider only trace information we can use trace timewise refinement. In this approach a process Q is a trace timewise refinement of P if, for any trace s of Q, $strip(s)$ is an allowed trace of P. This is defined (for an untimed process P and a timed process Q) as:

$$
P_T \sqsubseteq_{TF} Q \quad = \quad \forall (s, \aleph) \in \mathcal{TF}[\![Q]\!] \bullet \#s < \infty \Rightarrow strip(s) \in traces(P)
$$

The traces of an untimed process do not include infinite traces, so this definition places no restriction upon the infinite timed traces of Q. However, as discussed in

[15] this does not mean that Q has arbitrary infinite traces since the downward closure of timed failures means that all finite prefixes of any infinite trace of Q must be consistent[2] with P.

We then define the timed interpretation (which we denote $[\![P]\!]_t$) of an untimed process or Object-Z component (since this generates untimed failures) to be any timed failures produced from valid timed trace refinement:

$$[\![P]\!]_t = \bigcup_i \mathcal{TF}[\![Q_i]\!] \qquad \text{where } Q_i \text{ is any timed process such that } P_T \sqsubseteq_{TF} Q_i$$

Since $\bigcup_i \mathcal{TF}[\![Q_i]\!] = \mathcal{TF}[\![\sqcap_i Q_i]\!]$, this is equivalent to viewing the embedding of an untimed process P as being given by the choice of all possible timed processes consistent with P.

Thus this definition offers an interpretation of an (untimed) Object-Z component in the timed failures model such that the timed traces are consistent with the temporal ordering defined within the Object-Z component.

For example, consider the *Lift* component. Its *timed* interpretation according to this model will consist of elements $(s, \aleph)$ such that s is consistent with the untimed traces. Since one such untimed trace is

$$\langle \mathit{Open}, \mathit{RequestD}.\{(f,l)\}, \mathit{RequestD}.\{(f,k)\}, \mathit{SetTarget}.\{(f,n)\}, \\ \mathit{InternalTarget}, \mathit{Close} \rangle$$

a valid timed trace consistent with this is

$$\langle (0, \mathit{Open}), (1.2, \mathit{RequestD}.\{(f,l)\}), (1.3, \mathit{RequestD}.\{(f,k)\}), \\ (1.3, \mathit{SetTarget}.\{(f,n)\}), (2, \mathit{InternalTarget}), (7, \mathit{Close}) \rangle$$

The overall behaviour of *TLiftSys* is given by combining the behaviours of *Lift*, *Con* and *TOrder* according to the standard timed CSP definition of $\|$.

Refusal information. More interesting and useful, though, is an interpretation which also considers refusal information. In order to interpret an untimed process P in a timed failures model we use failures timewise refinement. This is based on the observation that a timed refusal $(s, \aleph)$ will be consistent with P if $\mathit{strip}(s)$ is the untimed trace, and that the timed refusal will contain refusal tokens of the form $[t, \infty) \times X \subseteq \aleph$. That is, an event being in X means, in the context of time, that it could eventually be refused for ever after the performance of the trace $\mathit{strip}(s)$.

Failures timewise refinement is defined as:

$$\begin{array}{l} P_{SF} \sqsubseteq_{TF} Q \quad = \\ \quad \forall (s, \aleph) \in \mathcal{TF}[\![Q]\!] \bullet \#s < \infty \Rightarrow \\ \qquad \mathit{strip}(s) \in \mathit{traces}(P) \land \\ \qquad \forall X \subseteq \Sigma \bullet ((\exists t : \mathbb{R}^+ \bullet [t, \infty) \times X \subseteq \aleph) \Rightarrow (\mathit{strip}(s), X) \in \mathcal{F}[\![P]\!]) \end{array}$$

[2] Strictly speaking this holds for processes which do not contain infinite non-determinism, however, for the purposes of this paper we restrict ourselves to processes which only contain finite non-determinism.

As before this leads to the following embedding of an untimed process P into a timed failures model:

$$[\![P]\!]_f = \bigcup_i \mathcal{TF}[\![Q_i]\!] \quad \text{where } Q_i \text{ is any timed process such that } P_{SF} \sqsubseteq_{TF} Q_i$$

Thus in terms of the *Lift* component, since we know that

$$(\langle\,\rangle, \{Open, InternalTarget, MoveOneFloor\}) \quad \text{and}$$
$$(\langle Close, RequestD.\{(f,l)\}\rangle, \{Close, MoveOneFloor\})$$

are valid untimed failures, it follows that

$$(\langle\,\rangle, [0,\infty) \times \{Open, InternalTarget, MoveOneFloor\})$$
$$(\langle(1, Close), (3, RequestD.\{(f,l)\})\rangle, [3,\infty) \times \{Close, MoveOneFloor\})$$

are valid timed failures of *Lift*. Note that in this interpretation an untimed event being enabled corresponds in the timed framework to an eventual offer. That is, it might not be offered straight away, however, it cannot be refused forever. An illustration of this is the fact that $WAIT\ t_3;\ SetTarget \rightarrow STOP$ is a timed process that is consistent with the untimed process $SetTarget \rightarrow STOP$.

Thus $[\![P]\!]_f$ provides a mechanism by which we can interpret Object-Z components in the timed failures model, and hence calculate the overall timed failures of a specification such as $TLiftSys = TOrder \| Lift \| Con$ described above.

4 Analysis and Refinement

In order to analyse a combined Object-Z/timed CSP specification, we can verify properties by expressing them as constraints on timed failures. In general, there are at least three types of properties that one might consider trying to verify for an integrated specification (an extensive overview of the range of analysis available in timed CSP is contained in [15]):

- untimed properties,
- properties on timed traces, and
- properties on timed failures.

These properties are expressed as predicates on timed failures using the notation familiar from CSP: P **sat** $S(s, \aleph)$, meaning $\forall (s, \aleph) \in \mathcal{TF}[\![P]\!] \bullet S(s, \aleph)$. For an integrated specification of the form $P \| C$ we can verify properties componentwise by using the following result [15]:

$$\frac{P \ \mathbf{sat}\ S_1(s,\aleph) \qquad C \ \mathbf{sat}\ S_2(s,\aleph)}{P \| C \ \mathbf{sat}\ \exists \aleph_1, \aleph_2 \bullet S_1(s,\aleph_1) \wedge S_2(s,\aleph_2) \wedge \aleph = \aleph_1 \cup \aleph_2}$$

Verification of the timed CSP components, e.g., P, proceeds as normal, and with the embeddings of untimed processes into timed models, properties can be translated from an untimed context to a timed context. For example, with the trace interpretation if an Object-Z component C satisfies a property expressed in terms of its traces, i.e., C **sat** $S(tr)$, then this property can be translated to a timed interpretation: $[\![C]\!]_t$ **sat** $\#s < \infty \Rightarrow S(strip(s))$.

In a similar fashion properties expressed on both traces and failures can be translated from an untimed context to a timed context. So, if C satisfies a property expressed in terms of its traces and failures, i.e., C **sat** $S = (S_t(tr), S_f(tr, X))$, then this property can be translated to a timed interpretation: $[\![C]\!]_f$ **sat** T where $T = (\#s < \infty \Rightarrow S_t(strip(s))) \wedge \forall\, t : \mathbb{R}^+;\ A \subseteq \Sigma \bullet ([t, \infty) \times A \subseteq \aleph \Rightarrow S_f(strip(s), A))$.

As an example of this, deadlock freedom is preserved by such a translation. Thus we can calculate deadlock freedom on the individual components in an integrated Object-Z / timed CSP specification, knowing that their timed interpretations will also be deadlock free (although obviously not necessarily their parallel composition).

There are further properties that we might wish to verify that are independent of time. For example, the property that the lift doors alternate between being opened and closed can be expressed as

$$TLiftSys \ \mathbf{sat}\ s \downarrow \{Close\} \leq s \downarrow \{Open\} \leq s \downarrow \{Close\} + 1$$

and this is verified by showing that $Lift$ **sat** $s \downarrow \{Close\} \leq s \downarrow \{Open\} \leq s \downarrow \{Close\} + 1$. This amounts to showing the corresponding untimed version is true, and this can be verified for an Object-Z component using methods explained in [20].

Not all properties hold for each component. For example, the timed requirement that external targets cannot be set initially until after t_3 time units have elapsed is the following:

$$s = \langle\,\rangle \Rightarrow SetTarget \in \sigma(\aleph \Vert\!\restriction t_3)$$

The $Lift$ and Con components do not satisfy this property, since they place no constraints upon the timings of events. However, it is sufficient to show that this property is satisfied by the timed CSP process $TOrder$, and since this component refuses $SetTarget$, the synchronisation must also.

As another example, consider the requirement that $Close$ must be available after a certain time, t_5 say, after a target has been set. If $got\_target(s)$ represents a new target having been set and $door\_open(s)$ the lift door being open, this property can be expressed as:

$$\begin{aligned} & door\_open(s) \wedge got\_target(s) \Rightarrow \\ & \qquad (Close \notin \sigma(\aleph \upharpoonleft (end(s \restriction InternalTarget) + t_5)) \vee \\ & \qquad\qquad Close \notin \sigma(\aleph \upharpoonleft (end(s \restriction SetTarget) + t_5))) \end{aligned}$$

However, this property doesn't necessarily hold for all the components. In particular, since the timed interpretation of *Lift* is to allow all possible timings, some of these will indeed refuse *Close* after t_5 of having received a target.

The property still holds for the complete specification however. In effect the timed interpretation of *Lift* offers a choice of all possible timings, and the synchronisation with *TOrder* chooses those which are compatible with the timed requirements in that timed CSP process. Verification of this property, therefore, is still possible for the complete specification. To do so what we need to note is that in *Lift* and *Con*

$$door\_open(s) \wedge got\_target(s) \Rightarrow \text{pre } Close$$

That is, *Close* will be available with unspecified timing, and since the timed property holds for *TOrder*, we can easily conclude that the property holds for the complete specification.

4.1 Refinement

Compositionality results can also be obtained for refinement in the integrated notation. In CSP, refinement is defined in terms of failures and divergences [2]: C is a refinement of A if

$$failures\ C \subseteq failures\ A \text{ and } divergences\ C \subseteq divergences\ A$$

In an integration of (untimed) CSP and Object-Z, this serves as a basis for refinement in the integrated notation [20] (since the divergences of an Object-Z component are empty, we in fact need only consider the failures).

Because state-based simulations are sound and jointly complete with respect to CSP failures-divergences refinement [11], they can be adapted for use on the Object-Z components, enabling component-wise refinements to be undertaken (e.g., see the use of downward simulations in [20]).

We can follow a similar approach when combining timed CSP with Object-Z. In the timed CSP semantics the natural refinement relation is sub-setting of timed failures. However, in addition to the timed CSP components, some of our components are untimed Object-Z classes. Thus the natural question to ask is whether state-based refinement for the Object-Z components implies timed failures refinement (i.e., sub-setting of timed failures). This we show now.

Because simulations are sound with respect to CSP failures-divergences refinement [11], the question reduces to whether failures-divergences refinement implies timed failures refinement for their timed interpretation. Suppose, therefore, that the process P_2 is a failures-divergences refinement of P_1. We wish to show that, interpreted as timed processes, the timed failures of P_2 are included in those of P_1, i.e., $[\![P_2]\!]_f \subseteq [\![P_1]\!]_f$.

Suppose $(s, \aleph) \in [\![P_2]\!]_f$. Since the timed interpretations are constructed to be the closure of finite failures, it suffices to consider just the finite ones. For example, consider a failure $(s, \aleph)$ of the form $(s, [t, \infty) \times X)$ for some t and $X \in \mathcal{F}[\![P_2]\!]$. Then since $\mathcal{F}[\![P_2]\!] \subseteq \mathcal{F}[\![P_1]\!]$, we have $X \in \mathcal{F}[\![P_1]\!]$. Thus $strip(s) \in traces(P_2) \subseteq traces(P_1)$, and hence $(s, \aleph) \in [\![P_1]\!]_f$.

The consequences of this is that we can refine the Object-Z components in the standard fashion, i.e., using state-based simulations, knowing that their timed interpretations are timed refinements of each other. The next section considers more consequences of refinement, this time in the context of a two-event model of operations.

5 The two-event model

The standard assumption in CSP and Object-Z is that events and operations are instantaneous, i.e., they are atomic and take no duration. In a timed context, however, one sometimes wishes to reason about the duration of operations, e.g., in the *Lift* the *RequestD* operation can probably be assumed to be atomic, however, *MoveOneFloor* in reality certainly does take time to complete. There are a number of ways of dealing with operation and event duration, and one of the simplest is to adopt the, so-called, two-event model.

The two-event model [7] considers an operation to be defined by its beginning and end, and thus two events are used to model each operation, allowing us to reason about start and end times, and therefore duration. Thus in the integration described here, we will distinguish between start and finish events explicitly in the timed CSP part, and map each operation in the Object-Z components to two events. This will allow us to be explicit about start and end times in timed CSP, whilst specifying the effects of the operation in terms of state changes in a single description.

Subscripts s and f denote the start and finish events associated with a single operation, the absence of a subscript denoting an atomic event. Thus one might write the following in the timed CSP component:

$$\begin{aligned} R = {} & MoveOneFloor_s \xrightarrow{d_1} MoveOneFloor_f \xrightarrow{t_1} R \\ & \Box \\ & Arrive \xrightarrow{t_2} Open_s \xrightarrow{d_2} Open_f \rightarrow P \end{aligned}$$

where here *Arrive* is instantaneous, but *MoveOneFloor* and *Open* have durations as specified.

To integrate Object-Z into this framework we need a model whereby operations are embedded into the semantics as two, instead of one, events. Since timings are absent from the Object-Z components, we work at the untimed level, and thus consider how to embed a two-event model into the Object-Z history semantics, and hence into the failures semantics.

There are a number of ways to split a single operation into two events. The choice we take here is that the start event determines when the operation is possible (and determines what inputs are used), and the end event determines the after state and the outputs, where the state space the end event works with is that identified by the start event.

We will thus split an operation Op into two events embedded in the history semantics. The two events are effectively an enabled and an effect part. The enabled part, called Op_s, is enabled on an instance of the precondition and does not change the state, i.e., $Op_s = Y \wedge \Xi State$ where Y is an instance of the precondition (this can be encoded as a schema in the schema calculus but we omit the details here).

The effect part, called Op_f, is the effect of the operation if we have chosen this particular instance (i.e., a particular set of bindings that satisfies the precondition) of the precondition, and this can be expressed as $\exists\, outputsof\, \overline{Y} \bullet \overline{Y}' \fatsemi Op$. Here $\overline{Y}$ is an IO decoration [4] that turns all inputs in Y to outputs with the same basename. These outputs are then consumed by the operation Op (and hidden so that they are no longer visible), this ensures that Op is being applied using before states and inputs consistent with those chosen by the particular instance Y.

So, for example, for the operation

$$\begin{array}{|l}
\multicolumn{1}{l}{Op} \\
\hline
\Delta(x) \\
n?, m! : \mathbb{N} \\
\hline
n? \geq 2 \wedge x = 1 \wedge x' = 2 \\
m! = n? + 1 \\
\hline
\end{array}$$

we have pre $Op \mathrel{\widehat=} [n? : \mathbb{N} \mid x = 1 \wedge n? \geq 2]$, and this will give rise to pairs of events in the history model. For example, one instance of the precondition is $[n? : \mathbb{N} \mid x = 1 \wedge n? = 3]$. If we call this Y, then $\overline{Y}$ will be $[n! : \mathbb{N} \mid x = 1 \wedge n! = 3]$, and thus $\overline{Y}'$ will be $[n! : \mathbb{N} \mid x' = 1 \wedge n! = 3]$. We can then form the composition $\exists\, outputsof\, \overline{Y} \bullet \overline{Y}' \fatsemi Op$ which will then be the end event associated with the start event given by Y:

$$\exists\, n! \bullet [n! : \mathbb{N} \mid x' = 1 \wedge n! = 3] \fatsemi Op = [\Delta(x)\ \ m! : \mathbb{N} \mid x' = 2 \wedge m! = 4]$$

Notice that we are splitting the operation into two events, rather than two operations, i.e., we have a single operation which gives rise to two events in the history semantics, and these events correspond to all instances $Y \wedge \Xi State$ and $\exists\, outputsof\, \overline{Y} \bullet \overline{Y}' \fatsemi Op$.

In embedding these events into the history semantics, we need to make sure that the histories of each component correspond to ones consistent with the interpretation of starting and finishing an operation. In particular, we require:

- operations must start before they finish;

- the termination of operations cannot be refused after they have started;
- operations must eventually terminate after they have started.

The histories derived from an Object-Z component using this two-event model will therefore be subject to a number of healthiness conditions that represent the points highlighted. The histories, remember, are sequences of states and operations which represent the evolution of an object. A history h is a tuple $(s, o) \in S^\omega \times O^\omega$, where O^ω is the set of sequences of operations (that an object could evolve through). The healthiness conditions are restrictions on the allowable histories, and reflect the structure of the operation sequence o. The first and third conditions can be written as:

- $(\neg \exists i \bullet o(i) = Op_f \land \forall j \leq i \bullet o(j) \neq Op_s) \land \#\{i \mid o(i) = Op_s\} = \#\{i \mid o(i) = Op_f\}$
- If $o \notin O^*$ then $\#\{i \mid o(i) = Op_s\} = \#\{i \mid o(i) = Op_f\}$

In the latter condition, O^* is the set of finite sequences of operations, and this condition thus formalises the requirement that operations must terminate if started. Note that an explicit condition for the second point is not necessary since by construction of the two events corresponding to an operation, Op_f can never be refused given Op_s has occurred.

The particular two-event model we have chosen uses the initial state for computation of the after state, and this has consequences for how operations can be interleaved.

Because the before state and inputs have been fixed by the start event, once the start event has occurred, the end event can never be refused (i.e., $Y \land \Xi State \vdash \mathrm{pre}(\exists\, outputsof\, \overline{Y} \bullet \overline{Y}' \fatsemi Op))$. In addition, it means that in any interleaving of operations, the result of an operation depends only upon when it was invoked, and the final state depends only upon the last operation to terminate. This guarantees that operations, once invoked, cannot be interfered with. Again we note that other choices of two-event model might take a different approach in this respect.

With start and finish events made explicit within the timed CSP part of the specification, a wider range of behaviours can be described. For example, we might wish to specify some fault tolerant behaviour, given by a process FTB, which is invoked if the doors do not close properly. This could be specified by:

$$Q = Close_s \rightarrow ((WAIT\ d_2;\ Close_f \rightarrow R) \rhd^{d_3}\ FTB)$$

saying that it takes at least d_2 time units to close the doors, but if they haven't closed by d_3 then invoke the fault tolerant mechanism. Here, $\rhd^{d_3}$ is the timeout operator. In general, the process $P \rhd^{d} Q$ offers a time-sensitive choice between P and Q. Initially the process P is available, and if P performs some external event before d time has elapsed then the choice is resolved in favour of P. However, if

P does not perform any external event before d time has elapsed then the choice is resolved in favour of Q.

The whole process of using a two-event model thus consists of the following. In an integrated specification $P \| C$, where P is a CSP process and C an Object-Z class, we first specify start and finish events in the CSP process for those events which we wish to have explicit duration. The correct synchronisation between components is found by determining the timed failures semantics of P and C and then calculating the composition. The timed failures semantics of C is often intuitive, but if not can be calculated via the history semantics of Object-Z.

When deriving the history semantics for C, we use a two-event interpretation for those events in P which use a two-event model. Histories which do not satisfy the healthiness conditions are removed. The failures semantics, and hence the timed failures semantics, is then derived from the remaining histories, and synchronisation between events in P and C then yields the complete specification with timing information embedded in it.

5.1 Refinement

By embedding a single operation as two events in the semantics, we have broken the original symmetry between operations and events. Since state-based simulations were used on Object-Z components, as they correspond to failures-divergences refinement at the semantic level, the natural question to ask is whether this result still holds for the two-event model. In fact, the result is still valid as we illustrate now.

To verify the result we need to work in the framework defined by Joseph [11] that provides a link between simulations and failures-divergences refinement. In this framework a process P is defined by a tuple $(A, S, \longrightarrow, R)$, where A is the set of events; S the set of states; $\longrightarrow \subseteq S \times A \times S$ its transition relation, and R its initial states. A transition under event e from state σ_1 to σ_2 is denoted by $\sigma_1 \xrightarrow{e} \sigma_2$. In addition, the set of next possible events that a system P can undergo when in state σ is denoted $next_P(\sigma)$.

Simulations can be defined in this framework, e.g., the definition of downward simulation between two processes $P_i = (A, S_i, \longrightarrow_i, R_i)$, $(i = 1, 2)$ is as follows.

Definition 1 *Downward simulation*
P_2 is a downward simulation of P_1 if there is a relation $D \subseteq S_1 \times S_2$ such that

1. $\forall \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 \, D \, \sigma_2$ *implies* $next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2)$
2. $\forall \sigma_1 \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet$
   $\sigma_1 \, D \, \sigma_2 \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2'$ *implies* $\exists \sigma_1' \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1' \, D \, \sigma_2'$
3. $\forall \sigma_2 \in R_2 \bullet \exists \sigma_1 \in R_1 \bullet \sigma_1 \, D \, \sigma_2$

As commented above, downward and upward simulations are sound and jointly complete with respect to CSP failures-divergences refinement [11]. What we show now is that the downward simulation conditions for a single event interpretation are equivalent to the downward simulation conditions for both start and finish events in the two-event model.

Consider the applicability condition:

$$\forall \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 \, D \, \sigma_2 \text{ implies } next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2)$$

Now for any given event e we know two things. First, if $\sigma \xrightarrow{e}$ then $\sigma \xrightarrow{e_s\, e_f}$, and hence $\sigma \xrightarrow{e_s}$. Second, from $Y \wedge \Xi State \vdash \text{pre} \exists\, outputsof\, \overline{Y} \bullet \overline{Y}' \fatsemi Op$ we know that if e_s has occurred in a trace then e_f cannot be refused, and that if $\sigma \xrightarrow{e_f}$ then $\sigma \xrightarrow{e_s} \sigma \xrightarrow{e_f}$.

Putting these together means that the applicability condition is equivalent to the applicability condition for start events together with the applicability condition for finish events.

Correctness is similar. For example, given correctness for the single event interpretation:

$$\forall \sigma_1 \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_1 \, D \, \sigma_2 \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2' \text{ implies}$$
$$\exists \sigma_1' \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1' \, D \, \sigma_2'$$

If $\sigma_2 \xrightarrow{e}_2 \sigma_2'$ then $\sigma_2 \xrightarrow{e_s}_2 \sigma_2 \xrightarrow{e_f}_2 \sigma_2'$. Correctness allows us to deduce that $\sigma_1 \xrightarrow{e_s}_1 \sigma_1 \xrightarrow{e_f}_1 \sigma_1'$. Correctness for e_s follows immediately from this. Correctness for e_f (and the reverse implication) follows for an arbitrary state if we again note that e_f is in a trace only if e_s precedes it. Thus if $\sigma_2'' \xrightarrow{e_f}_2 \sigma_2$ with $\sigma_1'' \, D \, \sigma_2''$, then $\sigma_2'' \xrightarrow{e_s}_2 \sigma_2'' \xrightarrow{e_f}_2 \sigma_2$, and correctness for e locates the necessary state in S_1 for correctness of e_f.

To summarise, we have shown that with the two-event embedding defined above, downward simulation for a single event model is equivalent to downward simulation for the two-event model. This is essentially due to the fact that applicability for the finish event and correctness for the start event are always true, thus the downward simulation for a single event splits into two conditions: applicability for the start events and correctness for the finish events.

The consequences of this is that even with this two-event model, state-based simulations are the correct refinement relations to use, since they generate simulations in the two-event model, and thus correspond to failures-divergences refinement in the two-event semantics. Another way to put this is to note that failures sub-setting of the two-event model can be verified by simulations for the single events.

This has important practical consequences. Because the correct state-based simulation rules are invariant with respect to the CSP model chosen, Object-Z components can be specified and developed independently without concern for whether the one or two-event model semantics will be used. This in itself is a good motivation for the use of this particular two-event model.

We can also explore the link with non-atomic refinement [3, 4]. Non-atomic refinement is a generalisation of the standard Z and Object-Z simulation rules, and it deals with a change of granularity between specifications. In particular, non-atomic refinement defines simulation conditions for refinements where a *single* abstract operation is decomposed into a *sequence* of concrete operations. This has obvious links to the two-event model, since there a single event e has been decomposed to the sequence $e_s;\ e_f$.

In fact, it is easy to verify that in a relational setting the refinement between e and $e_s;\ e_f$ corresponds to non-atomic refinement as defined in [3, 4]. The link then to notions of action refinement in process algebras [1], however, remain to be made, and in particular the link to failure-divergences refinement has to be explored further.

6 Conclusions

We are not the first to consider integrations of timed CSP with state-based notations such as Z and Object-Z, nor are we the first to investigate a two-event model.

Integrations of timed CSP and state-based languages include those defined by Mahony and Dong [13] and Sühl [21]. The former produces a language called TCOZ, and is a monolithic integration which combines timed CSP and Object-Z in a rather non-compositional way. That is, the specification style involves writing a specification in a completely new notation rather than by combining components written in the individual languages as we have done here. A specification in [13] provided the basis for the lift example presented above, and comparing the two approaches and two examples it is clear that although TCOZ is very expressive, it perhaps lacks the simplicity of integration that we have achieved here.

The closest approach to our work is the language RT-Z as defined by Sühl in [21]. Indeed, the philosophy behind the two is close, although [21] uses Z as opposed to Object-Z. However, [21] does not provide a full discussion of the semantics as we have done here, nor does Sühl investigate the related notions of refinement.

RT-Z does, however, consider the two-event model, and an overview of the use of such models when combining process algebras with state-based systems is provided by Fischer in [7]. Indeed, [7] contains a useful comparison of single, double, and multi-event approaches to the granularity of operation decomposition.

As discussed by Fischer in [7] there is a potential problem with synchronisation in the two-event model. In particular, input parameters are included in the start event, and output parameters in the finish event. Then synchronisation via parallel composition is a potential problem, because the synchronisation needs to match input values with output values. The solution to this is to ensure that specifications which contain synchronisation on two-event events only have one way communication (indeed causality would suggest that synchronisation in an event with duration has to necessarily be one way). Then the first event of one component can synchronise with the second event of the other. However, the full consequences of this need to be explored further, and in fact we have restricted the use of two-events above to operations which needed no synchronisation.

The alternative approach is to adopt a multi-event model, with three events distinguished in general: start, finish and middle, and to ensure that all communication (which is hidden anyway) is achieved via the middle event which is introduced especially for this purpose. The usefulness of this model needs to be investigated.

References

1. L. Aceto. *Action Refinement in Process Algebras.* CUP, London, 1992.
2. S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *Pittsburgh Symposium on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer-Verlag, 1985.
3. J. Derrick and E. A. Boiten. Non-atomic refinement in Z. In J. M. Wing, J. C. P. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 1477–1496, Berlin, 1999. Springer-Verlag.
4. J. Derrick and E.A. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications.* Springer-Verlag, 2001.
5. R. Duke and G. A. Rose. *Formal Object-Oriented Specification Using Object-Z.* Cornerstones of Computing. Macmillan, 2000.
6. C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Second IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 423–438. Chapman & Hall, July 1997.
7. C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23. Springer-Verlag, 1998.
8. C. Fischer. *Combination and implementation of processes and data: from CSP-OZ to Java.* PhD thesis, University of Oldenburg, January 2000.
9. C. Fischer and H. Wehrheim. Model checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods 1999 (IFM'99)*, pages 315–334, York, July 1999. Springer.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.
12. N. A. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Bakker, W.-P. de Roever, C. Huizing, and G. Rozenberg, editors, *Real-Time: Theory in Practice (REX Workshop, Mook, The Netherlands, June 1991)*, LNCS 600, pages 397–446. Springer-Verlag, 1992.
13. B. Mahony and J.S. Dong. Timed communicating Object-Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
14. A.W. Roscoe. *The Theory and Practice of Concurrency.* International Series in Computer Science. Prentice Hall, 1998.

15. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* Wiley, 2000.
16. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
17. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME '97)*, LNCS 1313, pages 62–81, Graz, Austria, September 1997. Springer-Verlag.
18. G. Smith. *The Object-Z specification language.* Kluwer Academic Publishers, 2000.
19. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. G. Hinchey and S. Liu, editors, *Formal Engineering Methods*, pages 293–302, Hiroshima, Japan, 12–14 November 1997. IEEE Computer Society Press.
20. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, May 2001.
21. C. Sühl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *International conference on Integrated Formal Methods 1999 (IFM'99)*, pages 29–48. Springer, July 1999.