Refinement and verification of concurrent systems specified in Object-Z and CSP

Graeme Smith* and John Derrick†
* Technische Universität Berlin, FB Informatik, FG Softwaretechnik,
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin, Germany.
† Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: graeme@cs.tu-berlin.de and J.Derrick@ukc.ac.uk.)

Abstract

The formal development of large or complex systems can often be facilitated by the use of more then one formal specification language. Such a combination of languages is particularly suited to the specification of concurrent or distributed systems, where both the modelling of processes and state is necessary. This paper presents an approach to refinement and verification of specifications written using a combination of Object-Z and CSP.

A common semantic basis for the two languages enables a unified method of refinement to be used, based upon CSP refinement. To enable state-based techniques to be used for the Object-Z components of a specification we develop state-based refinement relations which are sound and complete with respect to CSP refinement.

In addition, a verification method for static and dynamic properties is presented. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes by using the laws of the CSP operators together with the logic for Object-Z.

Keywords: Object-Z; CSP; Refinement; Verification; Concurrency.

1 Introduction

The formal development of particularly large, or complex, systems can often be facilitated by the use of more then one formal specification language. While most specification languages can be used to specify entire systems, few, if any, are particularly suited to modelling all aspects of such systems. This realisation has lead to the development of new specification languages which combine features of one or more existing languages[1, 8] and, more recently, approaches for formally integrating existing languages[4, 24, 11, 22, 9].

Such a combination of languages is particularly suited to the specification of concurrent or distributed systems, where both the modelling of processes and state is necessary. Process algebras such as CCS[16] and CSP[12] are suitable vehicles for modelling the interactions between processes or their temporal ordering. State-based languages such as Z[23] or VDM[14], however, offer better facilities for the specification of the complex data structures which may be needed to describe the processes themselves. Indeed, the Open Distributed Processing reference model[13] recognises that different languages are likely to be used in the different viewpoint specifications of a large distributed system.

A method of formally specifying concurrent systems using Object-Z[7], an object-oriented extension of Z, together with CSP is described in [22]. The rationale is that Object-Z provides a convenient method of modelling the complex data structures needed to define component processes, and CSP enables the concise specification of process interaction. The advantage of Object-Z over more traditional state-based languages such as Z is that its class structure provides a construct easily identifiable with CSP processes. The basis of the integration is a semantics of Object-Z classes identical to that of CSP processes. This enables classes specified in Object-Z to be used directly within the CSP part of the specification.

However, in addition to specification, a notation needs to be able to support incremental development of specifications through a well-defined method of refinement. It is also desirable to be able to verify both static and dynamic, i.e. behavioural, properties of these specifications. The work described here presents a method of refining specifications written in the integrated Object-Z / CSP notation, and a method for verifying such properties of those specifications.

The common semantic basis for the two languages enables a unified method of refinement to be developed for the integrated notation: because we give Object-Z classes a CSP semantics, we can use CSP refinement as the refinement relation for the integrated notation. However, as a means for verifying a refinement it is more convenient to be able to use a state-based refinement relation for the Object-Z components, rather than having to calculate their semantics. In order to do so, we adapt the work of Josephs[15], who has developed refinement relations for state-based systems which are sound and complete with respect to CSP refinement.

In order to be able to verify static and dynamic properties, we present a method of verification for the integrated notation. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes by using the laws of the CSP operators presented in [12] together with the logic for Object-Z in [19]. CSP and Object-Z properties are related via auxiliary variables introduced into the Object-Z classes using inheritance.

The paper is structured as follows. Section 2 presents the integration of Object-Z and CSP based on the common semantics. Section 3 then discusses refinement in the integrated notation, and defines the state-based refinement relations that we will use for the Object-Z components of a specification. Section 4 explains how properties of specifications can be verified, and we conclude in Section 5. Throughout the paper we illustrate these techniques with the specification and refinement of a cinema booking system.

2 Integrating Object-Z and CSP

This section presents the integration of Object-Z and CSP. The basis of this integration is a semantics of Object-Z classes identical to that of CSP processes. This allows classes specified in Object-Z to be used directly within the CSP part of the specification. The approach to specification comprises three phases.

- The first phase involves specifying the components of the system using Object-Z. Since all interaction of system components is specified in the CSP part of the specification, a restricted subset of Object-Z is used which does not include instantiation of objects of a class (see [7] for details). This restriction greatly simplifies reasoning about the Object-Z part of the specification.
- The components specified in the first phase will generally not be in a form that allows them to be composed using CSP operators. The second phase involves modifying the class interfaces so that they will synchronise and communicate as desired. This may be achieved using Object-Z inheritance.

This optional phase is not required for the simple examples presented in this paper. An example illustrating its use can be found in [22].

- The final phase involves the specification of the system using CSP operators. As detailed in this section, a well-definedness condition is placed on the hiding operator restricting its use.

To illustrate the approach we present a case study of a cinema booking system. This case study is based on the specification of the Apollo box office in [25] but extended to support multiple customers.

2.1 Specifying the components of a system

The Marlowe box office allows customers to book tickets in advance by telephone. When a customer calls, if there is an available ticket then one is allocated and put to one side for the caller. When the customer arrives, they are presented with this ticket.

The components of the booking system are the customers and the Marlowe box office. In our approach, these will be specified by Object-Z classes. A class in Object-Z is represented syntactically by a named box possibly with generic parameters. In this box there may be local type and constant definitions, at most one state schema and associated initial state schema, and zero or more operation schemas. As an example, consider the specification of a customer of the booking system.

Let $Name$ denote the set of all customer names and $Ticket$ the set of all tickets.

Customer

$my\_name : Name$

Book

$name! : Name$

$name! = my\_name$

Arrive

$name! : Name$
$t? : Ticket$

$name! = my\_name$

This class has a single constant my_name denoting the name of the customer and two operations: *Book* and *Arrive*. The operations *Book* and *Arrive* correspond to the customer booking a ticket and arriving to collect a ticket respectively. They have input parameters (denoted by names ending in ?) and output parameters (denoted by names ending in !) for communication with the box office.

A more substantial example of a class is provided by the specification of the Marlowe box office.

Marlowe

$mpool : \mathbb{P}\, Ticket$
$tkt : Name \rightarrowtail Ticket$

INIT

$tkt = \varnothing$

Book

$\Delta(tkt, mpool)$
$name? : Name$

$name? \notin \mathrm{dom}\, tkt$
$mpool \neq \varnothing$
$\exists\, t : mpool \bullet$
$\quad mpool' = mpool \setminus \{t\}$
$\quad tkt' = tkt \cup \{name? \mapsto t\}$

Arrive

$\Delta(tkt)$
$name? : Name$
$t! : Ticket$

$name? \in \mathrm{dom}\, tkt$
$t! = tkt(name?)$
$tkt' = \{name?\} \lhd tkt$

This class has a state schema with two state variables: *mpool*, denoting the pool of tickets, and *tkt*, a partial injective function from *Name* to *Ticket* recording which tickets have been allocated to which customers. Initially, no tickets have been allocated.

Each operation schema has a Δ-list of the state variables which it may change. State variables not listed remain unchanged. The operation *Book* is feasible whenever there are still tickets available ($mpool \neq \varnothing$) and allocates a ticket to a customer who has not already made a booking ($name? \notin \mathrm{dom}\, tkt$). The operation *Arrive* issues the ticket but does not change the pool of tickets ($mpool = mpool'$ is a consequence of *mpool* not appearing in the Δ-list of the operation *Arrive*).

2.2 Specifying the system

To specify the booking system we use CSP operators to capture the interaction between the customers and box office. This is made possible by giving a semantics to Object-Z classes which is identical to that of CSP processes.

2.2.1 Semantics of CSP processes

There are several semantic models for CSP processes. The most widely accepted of these is the failures-divergences semantics of [3, 12]. In this semantics, a process is modelled by the triple (A, F, D) where A is its alphabet (i.e. the set of events that it can possibly engage in)[1], F is its *failures* and D its *divergences*. The failures of a process are pairs (s, X) where s is a *trace* of the process, i.e. a finite sequence of events that the process may undergo, and X is a set of events the

[1] The alphabet is made implicit in [3] by assuming all processes have the same alphabet.

process may refuse to perform after undergoing s. That is, if the process after undergoing s is in an environment which only allows it to undergo events in X, it may deadlock. The divergences of a process are the sequences of events after which the process may undergo an infinite sequence of internal events, i.e. livelock. Divergences also result from unguarded recursion.

We adopt, however, a variant of the simpler failures semantics of [2]. This semantics doesn't include a component corresponding to the divergences of a process. The reason for adopting this simpler semantics is because Object-Z is capable of modelling *unbounded nondeterminism*, i.e. where a choice is made from an infinite set, which cannot be modelled in standard CSP. As shown in [17] and [22], this can lead to problems when calculating divergences.

Given a class with alphabet A and failures $F \subseteq A^* \times \mathbb{P}\, A$, the properties of the semantics we adopt are as follows.

$$(\langle\rangle, \varnothing) \in F \tag{F1}$$
$$(s \frown t, \varnothing) \in F \Rightarrow (s, \varnothing) \in F \tag{F2}$$
$$(s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F \tag{F3}$$
$$(s, X) \in F \land (\forall\, x \in Y \bullet (s \frown \langle x \rangle, \varnothing) \notin F) \Rightarrow (s, X \cup Y) \in F \tag{F4}$$

That is, we have dropped the restriction in [2] that the set of refused events is finite as is also done in [3][2] and [15].

For the failures semantics to be adequate, however, we must ensure that our specifications are divergence free. This is true of processes corresponding to Object-Z classes since Object-Z has no notion of internal operations nor recursive definitions of operations[3]. It can be ensured for other processes in our approach by placing a well-definedness condition on the hiding operator of CSP as is done in [15]. That is, given a process P with failures F, $P \setminus C$ is well-defined only if

$$\forall\, s \in \operatorname{dom} F \bullet \neg\, (\forall\, n \in \mathbb{N} \bullet \exists\, t \in C^* \bullet \#t > n \land s \frown t \in \operatorname{dom} F)$$

This prevents infinite sequences of events being hidden.

An alternative solution to the problem of unbounded nondeterminism would be to add to the failures-divergences semantics a component corresponding to the infinite traces of a process as is done in [18]. In this case, no restriction would be required on hiding. Whether the benefits of adopting this more complicated semantics are worthwhile, however, needs to be investigated.

2.2.2 Semantics of Object-Z classes

A semantics of Object-Z classes is presented in [21] where, following the work of [6], a class is modelled by its set of *histories*, i.e. the sequences of states it can pass through together with the corresponding sequences of operations it can undergo.

Given the set of all possible identifiers Id and the set of all possible values $Value$, the states of a class can be represented by a set

$$S \subseteq (Id \pfun Value)$$

and the operations by a set

[2] The additional property stating that a set is refusable if all its finite subsets are refusable in [3] was shown to be unnecessary in [17].

[3] Although recursive definitions of operations have been suggested for Object-Z (e.g. [5]), we have adopted a more conservative view of Object-Z in this paper.

$$O \subseteq Id \times (Id \nrightarrow Value).$$

The operations are instances of the class' operation schemas. They comprise the name of the operation schema together with an assignment of values to its parameters. For example, $(Book, \{(name?, n)\})$ where $n \in Name$ is a possible operation of the class *Marlowe*.

A history is a non-empty sequence of states together with a sequence of operations. Either both sequences are infinite[4] or the state sequence is one longer than the operation sequence. The histories of a class with states S and operations O can be represented by a set

$$H \subseteq S^\omega \times O^\omega$$

such that the following properties hold.

$$(s, o) \in H \Rightarrow s \neq \langle\rangle \qquad (H1)$$
$$(s, o) \in H \wedge s \notin S^* \Rightarrow o \notin O^* \qquad (H2)$$
$$(s, o) \in H \wedge s \in S^* \Rightarrow \#s = \#o + 1 \qquad (H3)$$
$$(s_1 \frown s_2, o_1 \frown o_2) \in H \wedge \#s_1 = \#o_1 + 1 \Rightarrow (s_1, o_1) \in H \qquad (H4)$$

The first three properties capture the requirements on an individual history detailed above. The final property is a condition on the set of histories representing a class. This set must be *prefix-closed*. This is necessary since any prefix of a class' history also represents a possible evolution of the class.

2.2.3 Modelling classes as processes

In order to relate classes and processes, we need to relate operations and events. This needs to be done in such a way that appropriate input and output parameters of synchronising operations can be identified. We therefore define a meta-function β which returns the basename of a parameter name, i.e. $\beta(x?) = \beta(x!) = x$, and allow it be applied to the assignment of values to an operation's parameters as follows.

$$\beta(\{(x_1, v_1), \ldots, (x_n, v_n)\}) = \{(\beta(x_1), v_1), \ldots, (\beta(x_n), v_n)\}$$
$$\text{where } \{x_1, \ldots, x_n\} \subseteq Id \text{ and } \{v_1, \ldots, v_n\} \subseteq Value$$

The function relating operations and events is then defined as follows.

$$event((n, p)) = n.\beta(p) \quad \text{where } n \in Id \text{ and } p \in (Id \nrightarrow Value)$$

For example, the event corresponding to a customer with name n making a booking is $Book.\{(name, n)\}$. This event is identical to that corresponding to the box office accepting a booking from a customer with name n. Hence, these two operations will be able to synchronise when their classes are combined using the CSP parallel composition operator $||$. Similarly, the events corresponding to a customer with name n arriving and collecting a ticket s and the box office allocating ticket s to that customer will be the event $Arrive.\{(name, n), (t, s)\}$.

We let a class C be modelled by a parameterised process C_i. The parameter i is an assignment of values to a subset of the state of C satisfying a possible initial state of C. That is, $i \in \{j \mid$

[4] Infinite histories enable liveness properties of classes to be modelled. Such properties have been ignored in the description of Object-Z in this paper.

$\exists (s, o) \in H \bullet j \subseteq s(1)\}$[5]. This allows us to refer to the class' constants when it is used as a process. For example, we can define a process $Customer_n$ corresponding to the customer with name n as follows.

$$Customer_n = Customer_{\{(my\_name, n)\}}$$

For notational convenience, we introduce the convention that $C = C_{\varnothing}$ allowing us to write, for example, *Marlowe* rather than $Marlowe_{\varnothing}$ for the process corresponding to the class *Marlowe* without any restriction on the initial state.

Given a class C with states S, operations O and histories H, the alphabet of process C_i comprises the events corresponding to the operations in O.

$$alphabet(C_i) = \{event(op) \mid op \in O\}$$

To define the failures of a class we use the following function which maps a sequence of operations to a sequence of events.

$$events(\langle\rangle) = \langle\rangle$$
$$events(\langle op\rangle \frown o) = \langle event(op)\rangle \frown events(o)$$

The failures of C_i are derived from the histories in H as follows: (t, X) is a failure of C_i if

- there exists a finite history of C whose initial state is satisfied by i,
- the sequence of operations of the history corresponds to the sequence of events in t and
- for each event in X, there does not exist a history which extends the original history by an operation corresponding to that event.

$$\begin{aligned} failures(C_i) = \{(t, X) \mid \exists (s, o) \in H \bullet \\ & s \in S^* \wedge \\ & i \subseteq s(1) \wedge \\ & t = events(o) \wedge \\ & \forall e \in X \bullet \nexists st \in S, op \in O \bullet \\ & \qquad e = event(op) \wedge (s \frown \langle st\rangle, o \frown \langle op\rangle) \in H\} \end{aligned}$$

As shown in [22], the failures of C_i defined in this way satisfy the properties $F1$ to $F4$ of the failures semantics.

2.2.4 The booking system specification

The processes $Customer_n$ and *Marlowe* can now be composed to specify the booking system.

$$BookingSystem = (|||_{n:Name}\ Customer_n) \parallel Marlowe$$

That is, the booking system consists of the box office running concurrently with a collection of customers – one for each name in *Name*. Since this part of the specification is a CSP specification,

[5] An Object-Z class with unsatisfiable initial constraints is not given a semantics in this approach. Such degenerate classes are, however, unimplementable and of no practical interest to the specifier.

we can state properties we wish to prove about it in the same way as they are stated in CSP (see [12]). That is, in the form P **sat** S where P is a process and S is a predicate in terms of tr, the traces, and ref, the refusal sets, of the failures of process P. For example, the property that the number of bookings made is greater than or equal to the number of tickets allocated to arriving customers can be stated as follows[6].

$$BookingSystem \textbf{ sat } \#tr \downarrow Book \geqslant \#tr \downarrow Arrive$$

An approach to proving such properties in terms of the component Object-Z classes is presented in Section 4.

3 Refining Object-Z and CSP specifications

This section presents a method of refinement for systems specified using the integrated Object-Z / CSP notation. The use of a CSP semantics for Object-Z classes enables us to use CSP refinement as the refinement relation for the integrated notation. To verify such a refinement there are two different approaches that can be employed:

- The first is based on the approach used in CSP. The refinement is verified directly by calculating and comparing the failures of the specifications or, in the case where the specifications have identical structure, the failures of the components of the specifications.
- The second involves using state-based methods to verify the refinement of the component Object-Z classes of a specification. This is achieved by adapting the work of Josephs[15], which provides refinement relations for state-based systems that are sound and complete with respect to CSP refinement. This approach is only possible when the specifications have identical structure.

In this section we illustrate both approaches by refining the cinema booking system of Section 2.

3.1 Failures Approach

Refinement in CSP is defined in terms of failures and divergences[3]. A process Q is a refinement of a process P if

$$failures\ Q \subseteq failures\ P \text{ and } divergences\ Q \subseteq divergences\ P$$

or when using the simpler failures semantics if

$$failures\ Q \subseteq failures\ P.$$

We write $P \sqsubseteq Q$ to denote the latter. Because we have modelled Object-Z classes semantically as processes, CSP refinement can be used as the basis for refining specifications written in the integrated Object-Z / CSP notation. As an example, consider an alternative booking system to the *BookingSystem* specification given in Section 2.

Like the Marlowe box office, the Kurbel box office allows customers to book tickets in advance by telephone. However, the procedure is different from that used at the Marlowe. When a customer calls, if there is an available ticket then the customer's name is simply recorded. When a customer whose name has been recorded arrives at the box office, a ticket is allocated.

[6] $s \downarrow c$ denotes the sequence of values v of events of the form $c.v$ in s, e.g. $\langle c.1, a.4, c.3, d.1\rangle \downarrow c = \langle 1, 3\rangle$.

The contrast between the Marlowe and the Kurbel box offices is the point of allocation of tickets (at booking time *vs* at collection time). However, at this level of abstraction the customer cannot tell that the Kurbel is behaving differently to the Marlowe. We will prove this property by showing that the Kurbel booking system is a CSP refinement of the Marlowe booking system.

The components of the Kurbel booking system are the customers and the Kurbel box office. The specification of a customer is identical to that given in the Marlowe booking system. The Kurbel box office is represented by the following Object-Z class.

Kurbel

$kpool : \mathbb{P}\ Ticket$
$bkd : \mathbb{P}\ Name$

INIT

$bkd = \varnothing$

Book

$\Delta(bkd)$
$name? : Name$

$name? \notin bkd$
$\#bkd < \#kpool$
$bkd' = bkd \cup \{name?\}$

Arrive

$\Delta(bkd, kpool)$
$name? : Name$
$t! : Ticket$

$name? \in bkd$
$bkd' = bkd \setminus \{name?\}$
$t! \in kpool$
$kpool' = kpool \setminus \{t!\}$

The state variable *kpool* denotes the pool of tickets and *bkd* denotes the set of names of customers who have booked a ticket. Initially, *bkd* is empty. The operation *Book* records a booking provided that there are currently less bookings than tickets and, hence, still tickets available. The operation *Arrive* allocates a ticket to a customer who has a booking.

The complete system again consists of the box office running concurrently with a collection of customers.

$$BookingSystem_K = (|||_{n:Name}\ Customer_n) \,||\, Kurbel$$

To show that $BookingSystem_K$ is a refinement of *BookingSystem*, we will compare their failures. Since the structure of the booking system specifications are identical and the components $Customer_n$ are identical, we need only show that $failures(Kurbel) \subseteq failures(Marlowe)$.

Consider first the class *Kurbel*. The failures of *Kurbel* can be given in terms of the failures of the processes $Kurbel_{\{(kpool,p)\}}$ for each possible set of tickets p.

$$failures(Kurbel) = \bigcup_{p \in \mathbb{P}\ Ticket} failures(Kurbel_{\{(kpool,p)\}})$$

The traces of $Kurbel_{\{(kpool,p)\}}$ comprise the empty trace and any trace formed by extending a trace of $Kurbel_{\{(kpool,p)\}}$ by

- a *Book* event whenever the customer doing the booking has arrived and collected any tickets he or she has previously booked and
- an *Arrive* event whenever
  - the ticket being collected was initially in *kpool*,
  - the ticket being collected has not been previously collected by any customer and
  - the customer arriving has booked once more than he or she has arrived to collect a ticket.

$$
\begin{aligned}
&traces(Kurbel) = \{\langle\rangle\} \\
&\quad \cup \\
&\quad \{s \frown \langle Book.\{(name, n)\}\rangle \mid s \in traces(Kurbel) \wedge n \in Name \wedge \\
&\qquad \#(s \upharpoonright \{Book.\{(name, n)\}\}) = \#(s \upharpoonright \{Arrive.\{(name, n), (t, x)\} \mid x \in Ticket\})\} \\
&\quad \cup \\
&\quad \{s \frown \langle Arrive.\{(name, n), (t, x)\}\rangle \mid s \in traces(Kurbel) \wedge n \in Name \wedge \\
&\qquad x \in p \wedge \#(s \upharpoonright \{Arrive.\{(name, m), (t, x)\} \mid m \in Name\}) = 0 \wedge \\
&\qquad \#(s \upharpoonright \{Book.\{(name, n)\}\}) = \#(s \upharpoonright \{Arrive.\{(name, n), (t, y)\} \mid y \in Ticket\}) + 1\}
\end{aligned}
$$

$Kurbel_{\{(kpool,p)\}}$ can refuse a *Book* event whenever the customer making the booking has booked more times than he or she has arrived, or there are no tickets remaining in *kpool*. It can refuse an *Arrive* event whenever the customer arriving has already arrived as many times as he or she has booked, the ticket of the *Arrive* event has already been allocated to a customer or the ticket of the *Arrive* event was not in *kpool* initially.

Hence, the failures of $Kurbel_{\{(kpool,p)\}}$ are

$$failures(Kurbel_{\{(kpool,p)\}}) = \{(tr, X) \mid tr \in traces(Kurbel_{\{(kpool,p)\}}) \wedge X \subseteq S\}$$

where

$$
\begin{aligned}
S = \{&Book.\{(name, n)\}, Arrive.\{(name, m), (t, x)\} \mid x \in Ticket \wedge n, m \in Name \wedge \\
&(\#(tr \upharpoonright \{Book.\{(name, n)\}\} > \#(tr \upharpoonright \{Arrive.\{(name, n), (t, y) \mid y \in Ticket\}) \\
&\qquad \vee \#(tr \upharpoonright \{Arrive.\{(name, l), (t, y) \mid l \in Name \wedge y \in Ticket\} = \#p) \\
&(\#(tr \upharpoonright \{Book.\{(name, m)\}\}) = \#(tr \upharpoonright \{Arrive.\{(name, m), (t, x)\}\}) \\
&\qquad \vee \#(tr \upharpoonright \{Arrive.\{(name, l), (t, x)\} \mid l \in Name\}) \neq 0 \\
&\qquad \vee x \notin p\}.
\end{aligned}
$$

The failures of *Marlowe* can similarly be given in terms of the failures of the processes $Marlowe_{\{(mpool,p)\}}$ for each possible set of tickets p.

$$failures(Marlowe) = \bigcup_{p \in \mathbb{P}\ Ticket} failures(Marlowe_{\{(mpool,p)\}})$$

It easy to see that the traces of $Marlowe_{\{(mpool,p)\}}$ are identical to those of $Kurbel_{\{(kpool,p)\}}$. Furthermore, $Marlowe_{\{(mpool,p)\}}$ can refuse any events that $Kurbel_{\{(kpool,p)\}}$ can refuse after the same trace. It can, in fact, refuse more events after a given trace because it can refuse an *Arrive* event whenever the ticket of the *Arrive* event is not that previously allocated to the customer. Hence, $failures(Kurbel_{\{(mpool,k)\}}) \subset failures(Marlowe_{\{(mpool,k)\}})$ and, therefore, $failures(Kurbel) \subset failures(Marlowe)$ as desired.

3.2 State-based Approach

Calculating and comparing the failures of classes as illustrated above is feasible, but can be complex for non-trivial specifications. The purpose of this section is to show how we can use state-based refinement techniques for the Object-Z component of a specification. This will enable refinements to be verified at the specification level, rather than working explicitly in terms of failures, traces and refusals at the semantic level.

Work on state-based refinement for concurrent systems goes back to He[10] and Josephs[15], who have developed refinement relations for state-based transition systems which are complete and sound with respect to CSP refinement. Woodcock and Morgan[27] have produced similar results in the context of action systems and weakest precondition formulae. In this section we adapt the work of Josephs to the Object-Z setting. This work is directly applicable to this context because it uses the failures semantics (as opposed to the failures-divergences model) and places the same restrictions on hiding that we have adopted. We produce two refinement relations, called upward and downward simulation, which together are sound and complete with respect to CSP refinement. Using these rules we can refine the Object-Z components of an integrated Object-Z / CSP specification such that the entire specification is also refined.

Josephs considers a state-based system P to be defined by a tuple $(A, S, \longrightarrow, R)$ where A is its alphabet, S its states, $\longrightarrow$ its transition relation and R its initial states ($R \subseteq S, R \neq \varnothing$). As usual we will denote a transition under event e from state σ_1 to σ_2 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 $next_P(\sigma)$, i.e.

$$next_P(\sigma) = \{e \in A \mid \exists\, \sigma' \in S \bullet \sigma \xrightarrow{e} \sigma'\}$$

Refinement in state-based systems is based on the concept of simulations. For example, simulation forms the basis of the refinement rules in Z as they are usually presented[25]. Josephs uses two versions called downward and upward simulation (sometimes called forward and backward simulations respectively) defined 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. $\forall\, \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 D \sigma_2 \implies next_{P_1}(\sigma_1) = next_{P_2}(\sigma_2)$
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. $\forall\, \sigma_2 \in R_2 \bullet \exists\, \sigma_1 \in R_1 \bullet \sigma_1 D \sigma_2$

Definition 2 *Upward simulation*
P_2 is an upward simulation of P_1 if there is a relation $U \subseteq S_1 \times S_2$ such that

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

Josephs then proves that these two relations are sound and complete with respect to CSP refinement.

To use these results, we first adapt the definitions to the Object-Z setting. The translation is straightforward, and the relations D and U between the state spaces are re-cast as retrieve relations (denoted Abs) between the abstract state ($Astate$) and the concrete state ($Cstate$).

To translate the rules involving $next_P(\sigma)$ we introduce a new precondition operator Pre. This is necessary because when we model Object-Z classes as processes we relate operations to events by removing the decorations ? and !. Therefore the simulation rules presented above will treat outputs in the same way as inputs. This is in contrast to standard Z refinement where the constraints on inputs can be weakened and those on outputs strengthened[25]. Doing this in our notation would mean that we could reduce the events that occur under a refinement, and hence restrict possible synchronisation with other processes. Compositionality would then be lost.

So in order to reflect the above simulation rules accurately and maintain compositionality in the Object-Z setting, we define Pre to hide the post-state of an operation, but not its outputs, i.e. Pre $Op \mathrel{\widehat=} \exists State' \bullet Op$. The event corresponding to an Object-Z operation Op is in $next_P(\sigma)$ iff Pre Op is true in the state representing σ. This is because the interpretation of operations in Object-Z differs from that in Z in that an operation cannot occur when its precondition is not enabled[7]. We can now give the definition of downward and upward simulation in Object-Z.

Definition 3 *Downward simulation*
An Object-Z class C is a downward simulation of the class A if there is a retrieve relation Abs such that every abstract operation AOp is recast into a concrete operation COp and the following hold.

DS.1 $\forall Astate;\ Cstate \bullet Abs \implies (Pre\ AOp \iff Pre\ COp)$

DS.2 $\forall Astate;\ Cstate;\ Cstate' \bullet Abs \land COp \implies \exists Astate' \bullet Abs' \land AOp$

DS.3 $\forall Cinit \bullet \exists Ainit \bullet Abs$

Definition 4 *Upward simulation*
An Object-Z class C is an upward simulation of the class A if there is a retrieve relation Abs such that every abstract operation AOp is recast into a concrete operation COp and the following hold.

US.1 $\forall Cstate \bullet \exists Astate \bullet Abs \land Pre\ AOp \implies Pre\ COp$

US.2 $\forall Astate';\ Cstate;\ Cstate' \bullet COp \land Abs' \implies \exists Astate \bullet Abs \land AOp$

US.3 $\forall Astate;\ Cinit \bullet Abs \implies Ainit$

Using these rules we can show that the Kurbel class is an upward simulation, and hence a refinement, of the Marlowe class without having to calculate the failures. To do so we first record the relationship between the two classes as a retrieve relation given by

Ret

$Kurbel.\textsc{State}$
$Marlowe.\textsc{State}$

$bkd = \operatorname{dom} tkt$
$kpool = mpool \cup \operatorname{ran} tkt$
$mpool \cap \operatorname{ran} tkt = \varnothing$

[7] In Z when operations occur outside their preconditions, the post-state is undefined.

$Kurbel.\textsc{State}$ denotes the state schema in the class $Kurbel$, etc.

Firstly, to prove the initialisation correct (US.3) we must prove the following:

$$\forall\, Marlowe.\textsc{State};\ Kurbel.\textsc{Init} \bullet Ret \implies Marlowe.\textsc{Init}$$

To do so we must show the following holds (which it clearly does).

$$\forall\, mpool : \mathbb{P}\, Ticket;\ tkt : Name \rightarrowtail Ticket;\ kpool : \mathbb{P}\, Ticket;\ bkd : \mathbb{P}\, Name \mid bkd = \varnothing \bullet \\ bkd = \operatorname{dom} tkt \land kpool = mpool \cup \operatorname{ran} tkt \land mpool \cap \operatorname{ran} tkt = \varnothing \implies tkt = \varnothing$$

Next, we must show that US.1 holds for the operations *Book* and *Arrive*. For the *Book* operation, this requires us to show that

$$\forall\, Kurbel.\textsc{State} \bullet \exists\, Marlowe.\textsc{State} \bullet Ret \land \operatorname{Pre} Marlowe.Book \implies \operatorname{Pre} Kurbel.Book$$

This amounts to showing that

$$\forall\, kpool : \mathbb{P}\, Ticket;\ bkd : \mathbb{P}\, Name \bullet \exists\, mpool : \mathbb{P}\, Ticket;\ tkt : Name \rightarrowtail Ticket \bullet \\ (bkd = \operatorname{dom} tkt \land kpool = mpool \cup \operatorname{ran} tkt \land mpool \cap \operatorname{ran} tkt = \varnothing) \land \\ (name? \notin \operatorname{dom} tkt \land mpool \neq \varnothing) \implies \\ (name? \notin bkd \land \# bkd < \# kpool).$$

Given the declarations and the constraints in Ret, we proceed as follows.

$$\begin{array}{ll} & name? \notin \operatorname{dom} tkt \land mpool \neq \varnothing \\ \implies & name? \notin \operatorname{dom} tkt \land \# mpool > 0 \\ \implies & name? \notin \operatorname{dom} tkt \land \# \operatorname{ran} tkt < \#(mpool \cup \operatorname{ran} tkt) \\ \implies & name? \notin \operatorname{dom} tkt \land \# \operatorname{dom} tkt < \#(mpool \cup \operatorname{ran} tkt) \qquad [\text{since } \# \operatorname{dom} tkt = \# \operatorname{ran} tkt] \\ \implies & name? \notin bkd \land \# bkd < \# kpool \qquad [\text{By } Ret] \end{array}$$

A similar proof can be given for the operation *Arrive*.

Finally, we must show that US.2 holds for the operations *Book* and *Arrive*. For the *Arrive* operation, this requires us to show that

$$\forall\, Marlowe.\textsc{State}', Kurbel.\textsc{State}, Kurbel.\textsc{State}' \bullet \\ Kurbel.Arrive \land Ret' \implies \exists\, Marlowe.\textsc{State} \bullet Ret \land Marlowe.Arrive.$$

That is, given the declarations we need to show that

$$(name? \in bkd \land bkd' = bkd \setminus \{name?\} \land t! \in kpool \land kpool' = kpool \setminus \{t!\} \land \\ bkd' = \operatorname{dom} tkt' \land kpool' = mpool' \cup \operatorname{ran} tkt' \land \varnothing = mpool' \cap \operatorname{ran} tkt') \implies \\ \exists\, mpool : \mathbb{P}\, Ticket;\ tkt : Name \rightarrowtail Ticket \bullet \\ (bkd = \operatorname{dom} tkt \land kpool = mpool \cup \operatorname{ran} tkt \land mpool \cap \operatorname{ran} tkt = \varnothing \land \\ name? \in \operatorname{dom} tkt \land mpool = mpool' \land tkt' = \{name?\} \lhd tkt \land t! = tkt(name?)).$$

This can be seen to be true if we take $mpool = mpool'$ and $tkt = tkt' \cup \{name? \mapsto t!\}$. We only need to prove the first three conjuncts of the consequent, the rest follow trivially from our choice of $mpool$, etc. For example, with these choices we can then make the following deductions.

$$\begin{array}{ll} \operatorname{dom} tkt & = \operatorname{dom}(tkt' \cup \{name? \mapsto t!\}) = \operatorname{dom} tkt' \cup \{name?\} \\ & = bkd' \cup \{name?\} = (bkd \setminus \{name?\}) \cup \{name?\} \\ & = bkd \end{array}$$

$$mpool \cup \operatorname{ran} tkt = mpool' \cup \operatorname{ran} tkt' \cup \{t!\} = kpool' \cup \{t!\} = kpool$$

Finally, to show that $mpool \cap \operatorname{ran} tkt = \varnothing$ we note that (since $\operatorname{ran} tkt = \operatorname{ran} tkt' \cup \{t!\}$)

$$mpool \cap \operatorname{ran} tkt = (mpool \cap \operatorname{ran} tkt') \cup mpool \cap \{t!\} = \varnothing \cup (mpool \cap \{t!\})$$

Now from $t! \in kpool \land t! \notin kpool'$ we deduce that $t! \notin mpool' = mpool$. Therefore $mpool \cap \operatorname{ran} tkt = \varnothing$.

This concludes the proof that *Kurbel* is an upward simulation of *Marlowe*, and therefore a CSP refinement. As with the failures approach, from this we can conclude that $BookingSystem_K$ is indeed a refinement of *BookingSystem*.

4 Verifying Object-Z and CSP specifications

This section presents a method of verification for the integrated notation. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes. It comprises three phases.

- The first phase involves reasoning about the CSP part of the specification. System properties are stated and transformed to properties of the component Object-Z classes using the notation and laws for CSP operators of [12].
- The properties of the Object-Z classes derived in the first phase will often include terms not readily reasoned about in Object-Z. The second phase involves extending the Object-Z classes with auxiliary variables to model these terms. This is achieved using Object-Z inheritance which allows the addition of variables and predicates to the state schema, initial state schema and operations of a class. Reasoning can then be carried out using the logic for Object-Z presented in [19].
- The final phase involves showing that the classes extended with the auxiliary variables are refined by the original Object-Z classes and hence the original classes also satisfy the desired properties.

To illustrate the approach, we will verify the property of *BookingSystem* stated at the end of Section 2.

4.1 Reasoning about the CSP processes

Properties about CSP processes can be stated in term of their failures. Given a process P with failures F, the property $\forall (tr, ref) \in F \bullet S(tr, ref)$ can be expressed using the notation of [12] as P **sat** $S(tr, ref)$. For example, the following property of the process *BookingSystem* states that the number of bookings made is greater than or equal to the number of tickets allocated to arriving customers.

$$BookingSystem \textbf{ sat } \#tr \downarrow Book \geqslant \#tr \downarrow Arrive$$

To prove such a property in CSP, we would use the laws for the various CSP operators given in [12]. Therefore, we re-express the property in terms of CSP operators by replacing *BookingSystem* with its definition in terms of component processes.

$$(|||_{n:Name}\ Customer_n) \parallel Marlowe\ \mathbf{sat}\ \#tr \downarrow Book \geqslant \#tr \downarrow Arrive$$

In this form, we can apply the following law for the parallel composition operator[8].

If P **sat** $S(tr)$
and Q **sat** $T(tr)$
then $(P \parallel Q)$ **sat** $(S(tr \upharpoonright \alpha P) \wedge T(tr \upharpoonright \alpha Q))$.

Let $S(tr \upharpoonright \alpha(|||_{n:Name}\ Customer_n)) = true$ and, since the alphabet of *Marlowe* is identical to that of *BookingSystem*, let $T(tr \upharpoonright \alpha Marlowe) = \#tr \downarrow Book \geqslant \#tr \downarrow Arrive$. Using the law for the parallel composition operator, the above property is true whenever the following is true.

$$Marlowe\ \mathbf{sat}\ \#tr \downarrow Book \geqslant \#tr \downarrow Arrive$$

This property is now in terms of a process corresponding to an Object-Z class and we can no longer use the laws for CSP operators. To complete the proof, we require a method for showing the above property is true for the Object-Z class *Marlowe*.

4.2 Reasoning about the Object-Z classes

Building on the work in [26], a logic for reasoning about Object-Z classes is presented in [19]. Properties of classes are expressed as sequents of the form

$$A :: \quad d \mid \Psi \vdash \Phi$$

where A is a class name, d is a list of declarations and Ψ and Φ are lists of predicates. The sequent is valid, i.e. the stated property is true, whenever given the declarations d and predicates Ψ at least one of the predicates in Φ is true in class A. For example, the following is a valid sequent ($\textsc{Init}$ denotes the declarations and predicates of the $\textsc{Init}$ schema of *Marlowe*).

$$Marlowe :: \quad \textsc{Init} \vdash tkt = \varnothing$$

The predicates in Ψ and Φ are only defined in terms of variables and constants which are accessible in the class or declared in d. Hence, it is not possible to state properties about sequences of events such as those we would like to prove about the CSP process corresponding to a class. Therefore, we need to introduce auxiliary variables to capture such properties. For example, an auxiliary variable $bks : \mathbb{N}$ could be added to the class *Marlowe* to model the CSP term $\#tr \downarrow Book$. Initially *bks* would be zero, it would be incremented each time *Book* occurs and remain unchanged each time *Arrive* occurs. Similarly, an auxiliary variable $arrs : \mathbb{N}$ could be added to model the CSP term $\#tr \downarrow Arrive$.

The addition of such variables to a class is possible using Object-Z inheritance (see [7]). When a class inherits another, schemas from the inherited class are implicitly conjoined with common-named schemas in the inheriting class. For example, consider the following class *auxMarlowe* which inherits *Marlowe*.

[8] As mentioned in [12], this law is valid provided S and T do not mention refusal sets.

auxMarlowe

Marlowe

$bks, arrs : \mathbb{N}$

$\#tkt = bks - arrs$

INIT

$bks = arrs = 0$

Book

$\Delta(bks)$

$bks' = bks + 1$

Arrive

$\Delta(arrs)$

$arrs' = arrs + 1$

The class *auxMarlowe* includes all the definitions of class *Marlowe* and extends them as follows. The state schema has the additional state variables *bks* and *arrs* and the additional predicate $\#tkt = bks - arrs$. This predicate isn't strictly necessary but aids the proof of the refinement relation between *Marlowe* and *auxMarlowe* as shown in Section 4.3. The initial state schema includes the additional constraint that *bks* and *arrs* are equal to zero and the operations *Book* and *Arrive* increment the variables *bks* and *arrs* respectively.

To prove the property that the number of bookings is greater than or equal to the number of tickets allocated to arriving customers for the class *auxMarlowe*, i.e. *auxMarlowe* **sat** $\#tr \downarrow Book \geqslant \#tr \downarrow Arrive$, we need to show that the following sequents are valid.

$$
\begin{array}{ll}
auxMarlowe :: & INIT \vdash bks = 0 \land arrs = 0 \\
auxMarlowe :: & Book \vdash bks' = bks + 1 \land arrs' = arrs \\
auxMarlowe :: & Arrive \vdash bks' = bks \land arrs' = arrs + 1 \\
auxMarlowe :: & \vdash bks \geqslant arrs
\end{array}
$$

The first three sequents ensure that *bks* and *arrs* model the number of occurrences of the operations *Book* and *Arrive* respectively. They can easily be proved using the logic for Object-Z (see [20] for examples of proofs in the logic). The final sequent states the desired property. It can be proved by structural induction, i.e. by proving the following sequents.

$$
\begin{array}{ll}
auxMarlowe :: & INIT \vdash bks \geqslant arrs \\
auxMarlowe :: & Book \vdash bks \geqslant arrs \Rightarrow bks' \geqslant arrs' \\
auxMarlowe :: & Arrive \vdash bks \geqslant arrs \Rightarrow bks' \geqslant arrs'
\end{array}
$$

These sequents can also be easily proved using the logic for Object-Z.

The above can be generalised as follows. A property P of a process corresponding to a class C in terms of the number of occurrences of particular events $Op_1, \ldots, Op_n$,

$$C \textbf{ sat } P(\#tr \downarrow Op_1, \ldots, \#tr \downarrow Op_n)$$

is true when the following sequents are valid. (The set of operations of the class are $Op_1, \ldots, Op_m$ where $m \geqslant n$.)

$$
\begin{array}{ll}
C :: & \mathit{INIT} \vdash a_1 = 0 \land \ldots \land a_n = 0 \\
C :: & \mathit{Op}_1 \vdash a_1' = a_1 + 1 \land a_2' = a_2 \land \ldots \land a_n' = a_n \\
\vdots & \\
C :: & \mathit{Op}_n \vdash a_1' = a_1 \land \ldots a_{n-1}' = a_{n-1} \land a_n' = a_n + 1 \\
C :: & \mathit{Op}_{n+1} \vdash a_1' = a_1 \land \ldots \land a_n' = a_n \\
\vdots & \\
C :: & \mathit{Op}_m \vdash a_1' = a_1 \land \ldots \land a_n' = a_n \\
C :: & \vdash P(a_1, \ldots, a_n)
\end{array}
$$

Similarly, we can develop rules for proving other types of properties. For example, a CSP predicate in terms of $\mathit{Op} \in \mathit{ref}$ can be replaced by an Object-Z predicate in terms of $\neg$ pre Op where pre Op denotes the precondition of Op. Such rules need to be proved sound. This can be done with respect to the failures semantics of classes presented in Section 2.

4.3 Proving the refinement relations

To show that the property proved for *auxMarlowe* also holds for *Marlowe*, we need to prove the refinement relation $\mathit{auxMarlowe} \sqsubseteq \mathit{Marlowe}$. This can be done using the notion of downward simulation defined in Section 3. To do so we first note that the retrieve relation between *auxMarlowe* and *Marlowe* is simply the identity (which we denote *Id*). Therefore to prove the refinement we have to show that

DS.1 $\forall\, \mathit{auxMarlowe}.\mathit{STATE};\ \mathit{Marlowe}.\mathit{STATE} \bullet (\text{Pre}\ \mathit{auxMarlowe.Book} \iff \text{Pre}\ \mathit{Marlowe.Book})$

DS.2 $\forall\, \mathit{auxMarlowe}.\mathit{STATE};\ \mathit{Marlowe}.\mathit{STATE};\ \mathit{Marlowe}.\mathit{STATE}' \bullet$
$\mathit{Marlowe.Book} \implies \exists\, \mathit{auxMarlowe}.\mathit{STATE}' \bullet \mathit{auxMarlowe.Book}$

DS.3 $\forall\, \mathit{Marlowe}.\mathit{INIT} \bullet \exists\, \mathit{auxMarlowe}.\mathit{INIT} \bullet \mathit{Id}$

together with similar conditions for the operation *Arrive*. Because we have simply added new state variables under the refinement, these conditions are easily discharged.

DS.1: This amounts to showing that

$$
\begin{array}{l}
(\mathit{name}? \notin \operatorname{dom} \mathit{tkt} \land \mathit{mpool} \neq \varnothing \land \#\mathit{tkt} = \mathit{bks} - \mathit{arrs} \land \\
\exists\, \mathit{tkt}' : \mathit{Name} \rightarrowtail \mathit{Ticket};\ \mathit{mpool}' : \mathbb{P}\, \mathit{Ticket};\ \mathit{bks}', \mathit{arrs}' : \mathbb{N} \bullet \\
\quad (\exists\, t : \mathit{mpool} \bullet \mathit{tkt}' = \mathit{tkt} \cup \{\mathit{name}? \mapsto t\} \land \mathit{mpool}' = \mathit{mpool} \setminus \{t\}) \land \\
\quad \#\mathit{tkt}' = \mathit{bks}' - \mathit{arrs}' \land \mathit{bks}' = \mathit{bks} + 1 \land \mathit{arrs}' = \mathit{arrs}) \\
\iff \\
(\mathit{name}? \notin \operatorname{dom} \mathit{tkt} \land \mathit{mpool} \neq \varnothing \land \\
\exists\, \mathit{tkt}' : \mathit{Name} \rightarrowtail \mathit{Ticket};\ \mathit{mpool}' : \mathbb{P}\, \mathit{Ticket} \bullet \\
\quad \exists\, t : \mathit{mpool} \bullet \mathit{tkt}' = \mathit{tkt} \cup \{\mathit{name}? \mapsto t\} \land \mathit{mpool}' = \mathit{mpool} \setminus \{t\})
\end{array}
$$

which is easily shown to be true (for example, $\#\mathit{tkt}' = \#\mathit{tkt} + 1 = \mathit{bks} - \mathit{arrs} + 1 = \mathit{bks}' - \mathit{arrs} = \mathit{bks}' - \mathit{arrs}'$).

DS.2: This amounts to showing the following, which again can easily shown to be true.

$$
\begin{array}{l}
(\mathit{name}? \notin \operatorname{dom} \mathit{tkt} \land \mathit{mpool} \neq \varnothing \land \exists\, t : \mathit{mpool} \bullet \mathit{tkt}' = \mathit{tkt} \cup \{\mathit{name}? \mapsto t\} \land \mathit{mpool}' = \mathit{mpool} \setminus \{t\}) \\
\implies \\
(\exists\, \mathit{bks}', \mathit{arrs}' : \mathbb{N} \bullet \\
\quad \mathit{name}? \notin \operatorname{dom} \mathit{tkt} \land \mathit{mpool} \neq \varnothing \land \exists\, t : \mathit{Ticket} \bullet \mathit{tkt}' = \mathit{tkt} \cup \{\mathit{name}? \mapsto t\} \land \mathit{mpool}' = \mathit{mpool} \setminus \{t\} \land \\
\quad \#\mathit{tkt} = \mathit{bks} - \mathit{arrs} \land \#\mathit{tkt}' = \mathit{bks}' - \mathit{arrs}' \land \mathit{bks}' = \mathit{bks} + 1 \land \mathit{arrs}' = \mathit{arrs})
\end{array}
$$

DS.3: To prove this, it is sufficient to show the following, which is easily done.

$$\forall\, tkt : Name \pfun Ticket \mid tkt = \varnothing \bullet \exists\, bks, arrs : \mathbb{N} \mid \#tkt = bks - arrs \land bks = arrs = 0$$

The conditions for *Arrive* can be proved in a similar fashion. Hence, $auxMarlowe \sqsubseteq Marlowe$. Since we have shown that $auxMarlowe$ **sat** $\#tr \downarrow Book \geqslant \#tr \downarrow Arrive$ we can deduce that $Marlowe$ **sat** $\#tr \downarrow Book \geqslant \#tr \downarrow Arrive$, and hence conclude the proof that the booking system satisfies the desired property. Furthermore, since $Marlowe \sqsubseteq Kurbel$, we can also conclude that the Kurbel booking system satisfies the property.

5 Conclusion

In this paper we have presented methods for refining and verifying specifications written using a combination of Object-Z and CSP. Because we have not modified either of the languages used in the combined notation, we have been able to use existing methods in our approach to refinement and verification in the combined notation. For example, by giving Object-Z classes a CSP semantics, we can use CSP refinement as the refinement relation for the integrated notation. A refinement can be verified by either calculating the failures semantics directly, or by applying standard state-based refinement relations to the Object-Z components.

To verify behavioural properties of the CSP system specification we use the Object-Z logic to prove subsidiary properties of the Object-Z component classes, these properties are then combined by application of CSP laws to deduce the desired behavioural properties of the overall system.

Some further areas of work remain. In particular, in addition to the state-based methods of refinement presented above, further methods of refinement need to be developed for specifications whose system structure changes under the refinement. For example, how can one verify the refinement of the Object-Z *Kurbel* class in the example presented above into two or more communicating Object-Z classes without having to resort to calculation of their semantics? Section 4.2 developed an approach for reasoning about the Object-Z classes in a combined specification, and presented rules for verifying certain properties. Further verification rules for a range of other types of properties need to be developed, and these need to be proved sound with respect to the Object-Z logic and the failures semantics developed in this paper.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[2] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[3] 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.

[4] J. Derrick, E.A.Boiten, H. Bowman, and M. Steen. Supporting ODP - translating LOTOS to Z. In *First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems*. Chapman & Hall, 1996.

[5] J. Dong, R. Duke, and G. Rose. An object-oriented approach to the semantics of programming languages. In G. Gupta, editor, *17th Annual Computer Science Conference (ACSC'17)*, pages 767–775, 1994.

[6] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90:VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 1990.

[7] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[8] M. Nielsen et al. The RAISE language, methods and tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[9] C. Fischer. Combining CSP and Z. Submitted to Formal Methods Europe (FME '97), 1997.

[10] J. He. Process refinement. In J. McDermid, editor, *The Theory and Practice of Refinement*. Butterworths, 1989.

[11] M. Heisel and C. Sühl. Formal specification of safety-critical software with Z and real-time CSP. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security*, pages 31–45. Springer, 1996.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[13] ITU Recommendation X.901-904. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

[14] C.B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, 1986.

[15] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[16] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

[17] A.W. Roscoe. An alternative order for the failures model. *Journal of Logic and Computation*, 3(2), 1993.

[18] A.W. Roscoe. Unbounded nondeterminism in CSP. *Journal of Logic and Computation*, 3(2), 1993.

[19] G. Smith. Extending $\mathcal{W}$ for Object-Z. In J. Bowen and M. Hinchey, editors, *9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 276–295. Springer-Verlag, 1995.

[20] G. Smith. Formal verification of Object-Z specifications. Technical Report 95-55, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1995.

[21] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[22] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. To appear in Formal Methods Europe (FME '97), 1997.

[23] J.M. Spivey. *The Z Notation: A Reference Manual (2nd Ed.)*. International Series in Computer Science. Prentice-Hall, 1992.

[24] M. Weber. Combining Statecharts and Z for the design of safety-critical systems. In M.-C. Gaudel and J.C.P. Woodcock, editors, *FME '96 – Industrial Benefits and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 307–326. Springer-Verlag, 1996.

[25] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, 1996.

[26] J.C.P. Woodcock and S.M. Brien. $\mathcal{W}$: A logic for Z. In J.E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 77–98. Springer-Verlag, 1992.

[27] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90:VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.