Specifying component and context specification using Promotion

John Derrick and Eerke Boiten
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 how the specification of components may be separated from the description of the context in which they are used. There are a number of ways in which this might be possible and here we show how to use the technique of promotion in Object-Z to combine components which are specified using process algebras.

We discuss two approaches, the first is to separate out the specification into two distinct viewpoints written in different languages. These viewpoints are then combined by a process of translation and unification. The second approach will be to use hybrid languages composed of a combination of CSP and Object-Z. We also consider how to refine such component based descriptions and consider issues of compositionality.

Keywords: Components; Viewpoints; Object-Z; LOTOS; CSP; Refinement.

1 Introduction

In this paper we discuss how the specification of components may be separated from the description of the context in which they are used.

The specification of a large and complex system often involves multiple instances of the same component. These components are combined together, perhaps in a number of different contexts, in order to provide some overall functionality. This type of component based software engineering has become an important mechanism to support code reuse and has a useful separation of concerns. Components have become particularly important in object oriented and distributed systems, both in terms of coping with the migration of legacy systems and also as a means to provide distribution independent behaviour across a number of platforms.

For example in a distributed system, multiple copies of a particular component might be used to provide the overall functionality or service required. Failure recovery can then be supported by keeping multiple copies of a component with identical state. An effective way to describe such a scenario is to specify the components separately and then provide a description of how they might be combined. By doing so we can support code reuse and separate development, and also allow appropriate specification languages to be used as and when needed. For example, one language may be used to describe the components whilst a different one is used to describe the context in which the components are used. There are a number of ways in which this might be possible and in this paper we show how to use the technique of promotion in Object-Z to combine components which are specified using a process algebra.

Object-Z [10], an object-oriented extension of Z [27], is a state based language which encapsulates state, initialisation and a number of operations into a class. Classes can be used as types, allowing object-instantiation to be specified. A class can thus contain objects, and its operations can be defined in terms of operations performed on the objects themselves. When such a global operation is defined in terms of a local operation upon an indexed component (e.g. object), the local operation is said to be *promoted* [29]. In Object-Z an operation will be promoted if we apply an operation to an object in a class, and this use of promotion is very common in both Object-Z and Z itself (although the mechanism needed to specify promotion is slightly more complex in Z due to the lack of encapsulation into classes).

We discuss two approaches to separating components from contexts. The first is separate out the specification into two distinct viewpoints (i.e. partial specifications) written in different languages (here LOTOS [3] and Object-Z). These viewpoints are then combined by a process of translation and unification. If we conform to certain promotion templates the resulting unification will be consistent (i.e. the unification has an implementation). The second approach will be to use hybrid languages composed of a process algebra part and a state-based part, for example, a combination of CSP [18] and Object-Z. Here we use CSP to describe the components, and Object-Z to describe how these components are combined. These approaches are illustrated with a specification of a transparency mechanism in a distributed system.

We also consider how to refine such component based descriptions and consider issues of compositionality.

The structure of the paper is as follows. In section 2 we introduce our example which shows how components can be used to specify a failure transparency mechanism for a distributed system. The subsequent section discusses how we can structure specifications to support the description and reuse of components separately from how they are used. In section 4 we look at the refinement of components, and we conclude in section 5.

2 Example: failure transparencies in ODP

We use the term component to mean an isolated part of a system which can be used in a number of different contexts to provide differing functionalities. Because of the natural encapsulation offered by object based languages, components can be thought of as one or more objects grouped together, however, in this paper we will think of each component as being encapsulated into one object*. Components have become particularly important in distributed systems, both in terms of coping with the migration of legacy systems and also as a means to provide distribution independent behaviour across a number of

*Some authors prefer to think of a component as a class, and then the use of components is phrased in terms of instances of components.

platforms.

To illustrate their use we consider an example of the latter. One of the central features of modern distributed system architectures is to hide certain aspects of actual distribution by providing a number of transparencies. The transparencies mean that the user, or indeed another part of the same system, is not concerned with the precise details of distribution and a seamless service is offered. A good example in practice is the use of *nfs* mountings of home directories to provide a location transparency to the user.

The *Open Distributed Processing* (ODP) architecture is a joint ITU/ISO standardisation framework for constructing distributed systems in a multi-vendor environment. ODP uses a number of viewpoints to specify a complete system. The architecture has reached a level of maturity, and the ODP Reference Model [19] has recently progressed to become an international standard. The reference model defines a number of transparencies, and the engineering viewpoint is concerned in particular with the provision of various transparencies needed to support distribution. For example, distribution transparencies defined in the ODP reference model include, amongst others:

- Access transparency: which masks differences in data representation to enable interworking.
- Location transparency: which masks the location of an object.
- Migration transparency: which masks from an object the ability of the system to change its location.
- Replication transparency: which masks the use of replicated objects.
- Failure transparency: which masks the failure and possible recovery of objects, and which might use replication transparency to do so.

As an example of the use of components we will consider the outline specification of a computational object together with engineering mechanisms which support failure transparency. To do so a single computational object is first specified (here we just give a template) as an Object-Z class:

CompObj

```
state declarations come here
```

INIT

```
initialisation predicate comes here
```

$put \mathrel{\widehat{=}} \ldots$

$get \mathrel{\widehat{=}} \ldots$

```
other operation definitions come here
```

$\vdots$

which contains a number of operations (not fully specified): *put*, *get*, ... The engineering mechanism consists of a number of *Nodes*, where each such *Node* consists of a computational object c, an identifier and a boolean representing whether that node is in service. It has an operation *start* to bring it into service and a *put* operation which performs a *put* on the computational object c.

$$
\begin{array}{|l}
\hline
\textit{Node} \\
\quad c : \textit{CompObj} \\
\quad id : \mathbb{N} \\
\quad l : \textit{bool} \\
\hline
\quad \textit{INIT} \\
\quad c.\textit{INIT} \wedge l = \textit{false} \\
\hline
put \mathrel{\widehat=} c.put \\
start \mathrel{\widehat=} [\Delta(l) \mid l'] \\
\hline
\end{array}
$$

This is a very simple example of promotion where the local *put* on c is promoted to a *put* on the *Node*. The failure transparency mechanism is achieved by using a collection of nodes indexed by a function over some index set I. Distinct nodes have distinct identifiers. There is an operation *start* to bring a node into service, and an *alarm* when the number of nodes in service is less than two (and therefore failure transparency might fail!). The failure transparency itself is ensured by replicating the computational operations across all nodes, so the *put* operation is promoted to an operation in this class and is performed simultaneously on all nodes in service. The replication ensures that if one node fails then there is at least one more node containing a correct copy of the state which can be used in its place.

$$
\begin{array}{|l}
\hline
\textit{FTransparency} \\
\quad f : I \rightarrowtail\!\!\!\!\!+ \textit{Node} \\
\quad \forall\, i, j : \operatorname{dom} f \bullet i \neq j \Rightarrow f(i).id \neq f(j).id \\
\hline
\quad \textit{INIT} \\
\quad \#\operatorname{dom} f = 2 \wedge \forall\, i : \operatorname{dom} f \bullet f(i).\textit{INIT} \\
\hline
put \mathrel{\widehat=} \|_{(i : \operatorname{dom} f \bullet f(i).l)}\, f(i).put \\
alarm \mathrel{\widehat=} [report! : R \mid \#\{i : \operatorname{dom} f \mid f(i).l\} \leq 1 \\
\qquad\qquad\qquad\qquad \wedge\ report! = \text{`}start\ new\ node\text{'}] \\
start \mathrel{\widehat=} [id? : \mathbb{N} \mid \forall\, i \bullet f(i).id = id? \Rightarrow f(i).start] \\
\hline
\end{array}
$$

The use of a number of nodes indexed by f has allowed us to separate out the global behaviour (e.g. the *alarm* operation) from the behaviour at each node.

This is a typical use of promotion. We have used Object-Z here, however, the same promotion facility is available in Z (see [29, 1] for comprehensive accounts), although because Z does not have object encapsulation, the exact mechanism is syntactically more complex than in Object-Z.

3 Structuring Specifications

We have specified the example above entirely in Object-Z, however, it is now well recognised that it is sometimes necessary or desirable to use different languages to specify different parts of a system. This is particularly true in a large complex distributed system which might encompass many concerns, and the ODP reference model acknowledges this by splitting a single specification into a number of partial specifications called *viewpoints*, and recognising that different languages might be applicable in different viewpoints.

Viewpoints provide a basic separation of concerns, enabling different participants to observe the system from suitable perspectives and at suitable levels of abstraction. It is a central device for structuring and managing the complexity inherent in describing systems. ODP uses five predefined viewpoints (enterprise, information, computational, engineering and technology), but is not prescriptive about the choice of specification language to be adopted with particular viewpoints. However, it does advocate that the chosen languages should be formal [5]. Because of the perspectives the viewpoints offer, these languages will typically include behavioural techniques such as process algebras (e.g. LOTOS, CSP etc) and state based techniques such as Z and Object-Z.

With issues such as these in mind there have been a number of proposals to combine or integrate Z and Object-Z with process algebras [25, 12, 14, 26, 13, 22, 28, 15], and we are interested here in supporting component based specification by using such methodologies. The key idea, as shown in the above example, is to use promotion to separate out the component from how components are combined and used globally. There are two possible approaches to this which we discuss in turn.

The first approach will be to separate out the specification into two distinct partial specifications written in different languages (here we will use LOTOS and Object-Z). Each partial specification will be largely independent and self contained, but can be combined by a process of translation and unification. If we conform to certain templates the resulting unification is guaranteed to be consistent by construction.

The second approach will be to use hybrid languages composed of a process algebra part and a state-based part, for example, a combination of CSP and Object-Z. Here a complete specification consists of one language being used to combine elements described in another. The hybrids described in [25, 12] use Object-Z to specify the components together with CSP to describe the component interaction. Here we reuse this mechanism to enable us to use promotion as a global way of gluing the components together. Although these hybrid languages are clearly applicable to viewpoint architectures such as ODP, they in fact provide a complimentary approach by describing a single specification composed of two languages as opposed to two partial specifications.

3.1 Viewpoints and promotion

One approach to using promotion to specify components is to specify the component in one viewpoint, with the description of how components will be used in another viewpoint. These viewpoints are thus partial specifications of the complete system specification. The viewpoints are linked by correspondences which describe the relationship between the viewpoints.

One of the problems of using partial specifications in development is that descriptions of the same or related entities can appear in different viewpoints and must co-exist. Thus, different viewpoints can impose contradictory requirements on the system under development and therefore the *consistency* of specifications across the viewpoints becomes important. Two viewpoints are said to be consistent if we can find a single implementation satisfying both viewpoints (i.e. the implementation must be a refinement of both viewpoints). The problem is complicated by the fact that we can expect viewpoint specifications to be written in different languages.

Given one viewpoint specification written in, say, Object-Z and another viewpoint written in LOTOS, how can we reconcile these two viewpoints for both consistency and further development? One way to do this is by translating the LOTOS viewpoint into an observationally equivalent Object-Z specification. We can then check the consistency of the two viewpoints now both expressed in Object-Z. The constructive method used for this results in a common refinement of the two Object-Z viewpoints, whose existence demonstrates consistency of the original viewpoints [2].

3.1.1 Specifying the viewpoints

For example, we could use ODP viewpoints to specify the failure transparency mechanism described above. Because we have separated the description into two partial specifications we can use different specification languages in each of them. The computational viewpoint specifies a single computational object *COMPOBJ*, and we might choose to specify this in LOTOS. The second viewpoint, an engineering view, describes how a number of *COMPOBJ* components are used to provide the overall failure transparency functionality.

The specification of the computational viewpoint is simply the single computational object *COMPOBJ* given as a LOTOS process.

process
$\quad COMPOBJ[put, get] : \mathbf{noexit} := put?x : nat;\ get!x;\ COMPOBJ[put, get]$
endproc

The specification of the engineering viewpoint consists of a number of *Node*s, the overall functionality being specified in the class *FTransparency* as before. However, since the *CompObj* class was defined in another viewpoint in order to use it here we have to include it in this viewpoint, however, we only define its signature and do not prescribe any behaviour. That is, this viewpoint does not make any assumptions about a *CompObj* and the effect of the operations, apart from declaring their existence[†].

Our viewpoints are thus partial in the sense that the functionality of a viewpoint might be extended by another viewpoint, but they must be complete in the sense that they need to type check and every item (e.g., class) must at minimum be declared even if it isn't given any behaviour. The engineering viewpoint is thus given by:

CompObj

INIT

$put \mathrel{\widehat{=}} [\ false\ ]$

$get \mathrel{\widehat{=}} [\ false\ ]$

```
other operation definitions come here
```

$\vdots$

together with *Node* and *FTransparency* exactly as before. Here in *CompObj* the state and initialisation really do have completely empty signature and predicate regardless of what is contained in the computational viewpoint.

These viewpoints overlap in the parts of the system that they describe, therefore we need to describe the relationship between the viewpoints. In simple examples such as this one, these parts will be linked implicitly by having the same name and type in both viewpoints. However, in general we may need more complicated descriptions for relating common aspects of the viewpoints. The correspondence here links the two viewpoints and simply identifies the *COMPOBJ* class/process with its use as a component in the engineering viewpoint, and can then be documented as a relation which says that *COMPOBJ* corresponds to *CompObj* and the two *put* events coincide, etc:

$$\{(COMPOBJ, CompObj), (put, put), (get, get)\}$$

3.1.2 Combining the viewpoints

Comparing viewpoints written in LOTOS and Object-Z requires that we bridge a gap between completely different specification paradigms. Although both languages can be viewed as dealing with states and behaviour, the emphasis differs between them. To support consistency checking between these two languages we exploit a behavioural interpretation of Object-Z.

[†]In fact for technical reasons we declare all operations to have no behaviour, and this is done by specifying that they have *false* predicate. This allows us to construct the combined behaviour correctly.

Object-based languages have a natural behavioural interpretation, and there is a strong correlation between classes in object-oriented languages and processes in concurrent systems (see for example [31, 12, 24]). We have used this correlation as the basis of a translation between the two languages, which has been verified by defining a common semantics for LOTOS and Object-Z.

The translation is given in [9], where it is verified against a common semantic model of the two languages. This model is based upon the semantics for Object-Z described in [24], which effectively defines a state transition system for each Object-Z specification. This model is used as a common semantic basis by embedding the standard labelled transition system semantics for LOTOS into it in an obvious manner.

The translation of the behaviour of a LOTOS specification produces a number of Object-Z classes, each one representing a behaviour expression (e.g. process definition) of the LOTOS specification. The heart of the translation consists of a number of translation rules, one for each of the LOTOS operators or terminals (i.e. occurrences of *stop*, *exit* or any process instantiations). The translation of a process definition begins with its terminals and successively applies the operator translation rules given in [9] until each operator/terminal has been translated.

For example, to translate the behaviour of *COMPOBJ*, we apply the translation algorithm to produce an Object-Z class called *COMPOBJ* containing operations schemas *put* and *get*. Inputs and outputs of the operations perform the value passing, and predicates in the operations ensure the temporal ordering explicit in the process algebra specification is preserved in the implicit behaviour of the Object-Z class. The result of the translation is a class as follows:

COMPOBJ

$$s : \mathbb{N}$$
$$x : \mathbb{N}$$

INIT

$$s = 0$$

put

$$\Delta(s, x)$$
$$ch? : \mathbb{N}$$

$$x' = ch?$$
$$(s = 0 \land s' = 1)$$

get

$$\Delta(s)$$
$$ch! : \mathbb{N}$$

$$x = ch!$$
$$(s = 1 \land s' = 0)$$

In fact the details of the mechanics of the translation are immaterial here, we could instantiate this approach with any translation into Object-Z.

What is interesting about the use of components illustrated in this example is that it provides support for a change of granularity. This was the use of a single component in the computational viewpoint, and the use of promotion in

Object-Z when we promoted the operations defined in the skeleton *COMPOBJ* class to an operation in the *Node* class. To perform this promotion all we needed to know was the signature of the component. The behaviour of the component was defined in a separate viewpoint and the correspondence relation was trivial (it just linked up names). The advantage of this style is that it automatically guarantees the consistency of the two viewpoints, and to unify them all that is needed is the renaming of the signatures as specified in the correspondence.

Normally to check the consistency of two Z or Object-Z partial specifications we have to construct a least refined unification of the two viewpoints, in two phases [2]. In the first phase ("state unification"), a unified state space (i.e., a state schema) for the two viewpoints has to be constructed. The essential components of this unified state space are the correspondences between the types in the viewpoint state spaces. The viewpoint operations are then adapted to operate on this unified state. At this stage we have to check that a condition called *state consistency* is satisfied. In the second phase, called *operation unification*, pairs of adapted operations from the viewpoints which are linked by a correspondence have to be combined into single operations on the unified state. This also involves a consistency condition (*operation consistency*) which ensures that the unified operation is a refinement of the viewpoint operations.

For non-trivial behaviour checking this overlap can be complex. The beauty of using components and promotion is that the separation of concerns that this enforces is precisely one that reduces the complexity of recombining by unification and the resultant consistency checking. It even allows the viewpoints to be further developed in parallel, an issue we discuss later in Section 4.

3.2 Hybrid languages and promotion

The previous section considered how to combine two separate specifications written in different languages. An alternative approach to integrating different formal methods which we consider now is to define a hybrid language which consists of one or more differing techniques. These hybrid languages have typically used a state based technique together with a process algebra. In [13] Fischer provides a survey of some of the available techniques for combining Z and Object-Z with process algebras such as CCS and CSP. Examples of these approaches include [15, 28], which both offer combinations of Z and CCS and also combinations of Object-Z and CSP discussed in [25, 12, 14, 26].

Because the different languages used in the hybrid have different roles (e.g. Z to define the state space, CCS to define the communication), subsets of the languages are occasionally used. For example, there is no object instantiation in the Object-Z part of the hybrid language defined by Smith in [25]. How the hybrid language is used is thus defined in part by what is, and what is not, included in the components of the hybrid. In this section we will discuss the hybrid language defined in [25], and consider how we might use promotion within it to combine together a number of components. To do so we will amend the subset of Object-Z used within the hybrid, and allow components to be specified in CSP.

3.2.1 *The semantics of hybrid languages*

For a hybrid language to make sense it is necessary to give it a semantic model. For example, in [25] classes are given a failures-divergences semantics, and this allows classes defined in the Object-Z part of the specification to be used directly in the CSP part and hence for these two languages to be combined.

The failures-divergences semantics is the standard semantics of CSP [7, 8]. A process is modelled by the triple (A, F, D) where A is its *alphabet*, F is its *failures* and D is its *divergences*. The failures of a process are pairs (s, X) where s 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 s. 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. The semantics is well-formed if the failures and divergences satisfy a number of axioms [7, 8].

To define the semantics of the hybrid language, Smith models a class C by a process. The alphabet is taken to be the set of events of the class, and the traces are sequences of events corresponding to sequences of operations. The failures are derived from the histories[‡] of a class as follows: (t, X) is a failure if

- there exists a finite history of C satisfying the initial state,
- 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 the event.

Divergence is not possible since Object-Z does not allow hiding of operations nor recursive definitions of operations, therefore the divergences of a class are empty.

This approach enables classes specified in Object-Z to be used within the CSP part of the specification. Thus Object-Z is used to describe the individual objects and CSP is used to describe how these are combined and interact. The motivation for this decomposition is given as: "Object-Z provides a convenient way of modelling complex data structures needed to define the component processes of such systems, and CSP enables the concise specification of process interactions" [26]. This is particularly useful when the system under discussion consists of a number of distinct components viewed as processes running concurrently.

Although, this does allow a very nice separation of concerns between object specification and process interaction, we sometimes might wish to specify more than just concurrent synchronisation when we combine components. Consider, for example, the combination of components specified in the *FTransparency* class. At this global level we require three operations to be specified: *put*, *start* and *alarm*.

[‡]The history model is the semantic model of Object-Z defined in [24].

The operation *alarm* does not appear in any of the components and thus is a new operation specified in terms of the data structures in the class and the components. The operation *start* applies the *start* operation in one (and only one) component $f(i)$ by promoting this operation, but which component used depends upon an input that matches the components identity. The final operation *put* is a concurrent operation of *puts* in each of the active components. None of these operations are simple synchronisations and we have exploited the flexibility of promotion here in that it allows a more complex interaction to be specified when defining global operations in terms of component ones.

3.2.2 Using promotion in a hybrid language

Can we use this flexibility with a hybrid language? Not as it stands, but what we would like to do is to use a hybrid language where the components can be combined with the use of promotion as in the class *FTransparency*. To do so we would need a semantic model where, for example, components specified in CSP can be used within the Object-Z part of the specification. We briefly sketch how this might be achieved.

The existing semantic approach defined in [25] gives a failures semantics to the hybrid language by turning the history semantics for an Object-Z class into a failures semantics in the manner described above. One option therefore is to simply extend this to the range of Object-Z specified now, and this consists of allowing object instantiation in the Object-Z part of the specification. The history semantics defined in [24] gives a meaning to object instantiation in a class by allowing it to be used as a type. The dot notation for initialising and promoting operations is then defined in terms of the history model. The promotion $c.put$ is represented semantically by the history which states that the object c undergoes an event associated with the operation *put*.

Therefore the history semantics is sufficient to model the complete range of Object-Z facilities that we are now using. Since the mapping from histories to failures works for an arbitrary history we can give a failure semantics to Object-Z specifications when they also contain object instantiation. The complete hybrid specification can thus be given a failure semantics and this allows us to use the flexibility of object instantiation and to promote local operations to global ones. Such a specification might look something like:

$$CompObj \mathrel{\widehat=} put?x \longrightarrow get!x \longrightarrow CompObj$$

together with *Node* and *FTransparency* exactly as before, viz

$$
\begin{array}{|l}
\hline
\textit{Node} \\
\quad \begin{array}{|l}
\hline
c : \textit{CompObj} \\
\textit{id} : \mathbb{N} \\
l : \textit{bool} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
c.\textit{INIT} \land l = \textit{false} \\
\hline
\end{array} \\
\quad \textit{put} \mathrel{\widehat=} c.\textit{put} \\
\quad \textit{start} \mathrel{\widehat=} [\Delta(l) \mid l'] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{FTransparency} \\
\quad \begin{array}{|l}
\hline
f : I \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$\mapstochar$\hidewidth}} \textit{Node} \\
\hline
\forall\, i, j : \operatorname{dom} f \bullet i \neq j \Rightarrow f(i).\textit{id} \neq f(j).\textit{id} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
\#\operatorname{dom} f = 2 \land \forall\, i : \operatorname{dom} f \bullet f(i).\textit{INIT} \\
\hline
\end{array} \\
\quad \textit{put} \mathrel{\widehat=} \|_{(i:\operatorname{dom} f \bullet f(i).l)}\, f(i).\textit{put} \\
\quad \textit{alarm} \mathrel{\widehat=} [\textit{report}! : R \mid \#\{i : \operatorname{dom} f \mid f(i).l\} \leq 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ \textit{report}! = \text{``}\textit{start new node}''] \\
\quad \textit{start} \mathrel{\widehat=} [\textit{id}? : \mathbb{N} \mid \forall\, i \bullet f(i).\textit{id} = \textit{id}? \Rightarrow f(i).\textit{start}] \\
\hline
\end{array}
$$

If we wished to calculate the failures of, say, the *Node* class, then we first note that in the history model, $c.\textit{INIT}$ is identical to the schema which states that c has undergone no events, i.e. $c.\textit{events} = \langle\,\rangle$. We can then subsequently calculate failures of the class *Node*. The operation *start* can always be applied. Initially a *put* event can occur, however, once it has done so no further *put* events are possible until a *get* has been invoked on c. Since this is not available in the class *Node* all further events on c are refused. The failures of *Node* are thus[§]:

$$\{(s, \varnothing) \mid s \upharpoonright \{\textit{start}\} = s\}$$

$$\{(s \frown \textit{put} \frown t, X) \mid s \upharpoonright \{\textit{start}\} = s \land t \upharpoonright \{\textit{start}\} = t \land X \subseteq \{\textit{put}\}\}$$

The alternative approach is not to use the failure semantics, but to use the history semantics as the common semantic model. That is, keep the history model of the Object-Z part of the specification and give a history interpretation to the failures model of the CSP part of the hybrid specification. To do so one would need to turn each failure into a history. Although we do not go into

[§]The notation $s \upharpoonright A$ is the trace s restricted to events from the set A.

details here, it should be clear that this is feasible modulo some technicalities. The technicalities arises because histories contain names of states etc, which do not appear in failures. However, it would be possible to define a canonical embedding of failures into histories that resolves this problem¶. Having achieved this we will have extended the history semantics to cover both Object-Z and CSP parts of a hybrid specification, and therefore we can use CSP to define the components and Object-Z to define the global behaviour in terms of promoted operations.

4 Developing and refining components

A key aspect to component based software engineering is the ability not only to specify the components separately, but also to develop them independently of how they are used. In this section we investigate how this may be done.

By develop here we mean refine, and therefore we are interested in mechanisms by which we can refine components in a compositional manner. To consider what we require, let X and Y be component specifications written in a language with refinement relation $\sqsubseteq_P$, with $X \sqsubseteq_P Y$. Let $C[.]$ and $D[.]$ denote contexts in which a component may be used, written in a language with refinement relation $\sqsubseteq_Z$. Then the requirements of separate development are:

- if $X \sqsubseteq_P Y$ then $C[X] \sqsubseteq_Z C[Y]$.
- if $C \sqsubseteq_Z D$ then $C[X] \sqsubseteq_Z D[X]$.

That is, if the refinement of components produces a refinement of the global specification; and if we refine how a component is used then the complete system is refined. The second of these is an issue of compositionality within a single language and using a single refinement relation. However, the first of these asserts a relation between differing notions of refinement. To answer this in our context of integrating particular formal languages, we will need to use results which compare refinement relations in Object-Z with those in a process algebra.

Refinement in Object-Z and CSP

Refinement between Object-Z classes is defined in terms of simulations. It is well known that any valid refinement between state-based specifications (e.g. those written in Z and Object-Z) can be verified as a sequence of upward and downward simulations [11]. In Object-Z these take the following form [26]:

Definition 1 *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.

¶The translation between LOTOS and Object-Z discussed above effectively uses this type of canonical embedding to define appropriate input and output parameters in the Object-Z operations.

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 2 *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 $\forall\, Astate;\ Cinit \bullet Abs \implies Ainit$

We write $A \sqsubseteq_Z C$ if the Object-Z class C is a refinement of the class A.

Refinement in process algebras is often defined in terms of failures and divergences [8], where we write $P \sqsubseteq_{FD} Q$ if

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

This is the standard notion of refinement in CSP and is closely related to the reduction refinement relation (**red**) in LOTOS [6]. In fact if we restrict ourselves to divergence free processes then **red** and $\sqsubseteq_{FD}$ coincide, so for the sake of uniformity in the following discussion we will consider all specifications to be free of divergence (Object-Z specifications are divergence free anyway since there are no internal operations nor any operation hiding).

In order to answer the compositionality issues raised above for our use of components we use the result that simulations are sound and jointly complete with respect to CSP (i.e. failures-divergences) refinement. That is any CSP refinement can be verified as a sequence of upward and downward simulations, and that any simulation induces a CSP refinement. This result has been proved for the simulation rules used in Z [16, 30], and also for the Object-Z simulation rules (see [20] and the discussion in [26]).

Refining components

The questions of compositionality of components upon refinement is a general problem in the use of hybrid languages, and is discussed in [13, 23]. Clearly, answers to such questions depend on how the components are used within a context $C[.]$. [26, 13] give positive answers for the hybrids consisting of Object-Z and CSP where Object-Z classes are used within a CSP specification, i.e., if X and Y are Object-Z classes, and $C[.]$ is the CSP part of the specification which describes the component interaction, then:

- if $X \sqsubseteq_Z Y$ then $C[X] \sqsubseteq_{FD} C[Y]$.

However, even in a single language such compositionality properties do not always hold. For example, consider LOTOS with the reduction refinement relation. Then the process $i;\ a;\ stop$ is a reduction of $a;\ stop$. However, if we place these two processes in the context of a choice we find that $b;\ stop[]i;\ a;\ stop$ is not a reduction of $b;\ stop[]a;\ stop$. Thus even with a single refinement relation we have lost compositionality by placing components in a particular context (the problem here is the initial internal action i).

However, our contexts and components have a particular form, namely that our components are objects and the context is their use within a promotion. Formally, a context in Object-Z is an incomplete class schema, with all the occurrences of a class, used in declaring objects in the context, elided [24]. For example, we have the following *Node[.]* context:

$$
\begin{array}{|l}
\hline
\quad Node[.] \\
\hline
\begin{array}{|l}
\hline
c : \boxdot \\
id : \mathbb{N} \\
l : bool \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\quad \mathit{INIT} \\
\hline
c.\mathit{INIT} \land l = false \\
\hline
\end{array} \\
\\
put \mathrel{\widehat=} c.put \\
start \mathrel{\widehat=} [\Delta(l) \mid l'] \\
\hline
\end{array}
$$

where $\boxdot$ represents the elided class. *Node[CompObj]* is thus the class *Node* specified above.

In this scenario we can exploit relevant results about refinements of promotions. A promotion is said to be *free* if the global context does not make constraints upon the component variables in the local state [29]. [21] contains a detailed discussion about the conditions necessary for promotion to factor through downward simulations. In particular, it is known that in Z the free promotion of a refinement is a refinement of a free promotion [21, 29]. This is in the context of Z, however, it is also easily shown to hold for Object-Z where the local states are classes X and Y and a context $C[X]$ contains an object of type X and promotes its operations. Essentially the structure of the encapsulation into classes ensures that our promotions will be free, for example *Node[.]* contains no state predicates restricting or even refering to c, and we have the following result.

Theorem 1 *Let Object-Z class C be a downward simulation of the class A with operations COp_i and AOp_i respectively. Let the context $\Phi(D)$ be defined by*

$$
\begin{array}{|l}
\hline
\Phi(D) \\
\quad \begin{array}{|l}
\hline
f_D : I \rightarrowtail D \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\mathit{INIT} \\
\forall\, i : dom\, f_D \bullet f_D(i).\mathit{INIT} \\
\hline
\end{array} \\
\quad Op_1 \mathrel{\widehat=} [i? : I \mid i? \in dom\, f] \bullet f_D(i?).DOp_1 \\
\quad \vdots \\
\hline
\end{array}
$$

Then $\Phi(C)$ is a downward simulation of $\Phi(A)$.

Proof
Because C is a downward simulation of the class A, there exists a retrieve relation Ret which can be used to verify that refinement. To prove that $\Phi(C)$ is a downward simulation of $\Phi(A)$ one needs to define a retrieve relation between these two classes in terms of Ret. Suppose that Ret is defined by

$$
\begin{array}{|l}
\hline
Ret \\
A.\mathit{STATE} \\
C.\mathit{STATE} \\
\hline
pred \\
\hline
\end{array}
$$

where $A.\mathit{STATE}$ is the state space of the class A. We then define the promotion retrieve relation by

$$
\begin{array}{|l}
\hline
PromRet \\
f_A : I \rightarrowtail A \\
f_C : I \rightarrowtail C \\
\hline
\mathrm{dom}\, f_A = \mathrm{dom}\, f_C \\
\forall\, i : \mathrm{dom}\, f_A \bullet \exists\, Ret \bullet \theta A.\mathit{STATE} = f_A(i) \wedge \theta C.\mathit{STATE} = f_C(i) \\
\hline
\end{array}
$$

The conditions necessary for a downward simulation between $\Phi(C)$ and $\Phi(A)$ easily follow. $\square$

The specification of the operation Op_1 can be more complex than that given above in $\Phi(D)$, as long as the encapsulation of the components is preserved (as they are in our examples above). That is, we require the promotion to be free. With this in place we can state the following result.

Theorem 2 *Let X and Y be process definitions in LOTOS or CSP. Let $C[.]$ be a context where all occurrences of a class have been elided. Let $C[X]$be interpreted as a hybrid specification (CSP plus Object-Z). Then if $X \sqsubseteq_{FD} Y$ we have $C[X] \sqsubseteq_Z C[Y]$.*

Proof
Because simulations are sound and jointly complete with respect to failures-divergences refinement, if $X \sqsubseteq_{FD} Y$ then there exist simulations to verify this refinement. Because the promotion of an Object-Z refinement is a refinement of a promotion we then have $C[X] \sqsubseteq_Z C[Y]$. □

This result also holds when we consider the unification of two viewpoints written in the style discussed above using LOTOS and Object-Z. Because of the templates we used there, the unification process will result in the complete specification $C[X]$, and refinements can be undertaken incrementally.

The second requirement on compositionality, namely that $C[X] \sqsubseteq_Z D[X]$ whenever $C \sqsubseteq_Z D$ is trivially satisfied by the definition of refinement between classes.

The consequence of these results is that using our particular form of component composition with promotion, we can refine the components separately and still be left with a valid overall development.

Example 1 *Refining the* CompObj *component.*

We can refine the component

$$CompObj \mathrel{\widehat{=}} put?x \longrightarrow get!x \longrightarrow CompObj$$

to an implementation *CompObj2* using separate sender and receiver processes which communication via a channel *mid* and an acknowledgement channel *ack*:

$$send \mathrel{\widehat{=}} put?x \longrightarrow mid!x \longrightarrow ack \longrightarrow send$$

$$rec \mathrel{\widehat{=}} mid?x \longrightarrow get!x \longrightarrow ack \longrightarrow rec$$

$$CompObj2 \mathrel{\widehat{=}} (send \parallel rec) \setminus \{mid, ack\}$$

The failure transparency mechanism which uses *CompObj2* in place of *CompObj* is then a refinement of the original failure transparency mechanism.

5 Conclusions

In this paper we have considered how to use components with Object-Z classes by promoting the component operations. We looked at how we could use viewpoints combined together by a process of translation and unification, and also looked at how we might use a hybrid language composed of Object-Z and CSP.

These two approaches could in fact be extended to other languages, all that is necessary is to provide a translation at either the syntactic or semantic level to allow components to be used within an Object-Z specification. In the first approach using viewpoints and partial specifications we utilised a syntactic translation between LOTOS and Object-Z that allowed us to use LOTOS processes in the Object-Z classes. In the second approach there was also a translation, but it was at the semantic level, i.e., we translated the history semantic model into a failures semantics (or in fact we could also go the other way round). This semantic translation allows us to build a hybrid language containing parts of both CSP and Object-Z. By including object instantiation in this hybrid we could use promotion of operations to allow reuse of components specified in CSP within Object-Z.

The nice interplay between failures-divergences refinement and state-based simulations together with the refinement properties of promotion means that we can develop components and contexts separately.

However, one issue we have not considered here is the use of more general components. We have restricted ourselves to considering a component to be encapsulated within a single object. Whilst this clearly is of some use, further research should look at components composed of more than one object.

Acknowledgements: We would like to thank Howard Bowman, Clemens Fischer and Graeme Smith for discussions of some of the ideas in this paper. This work was partially funded by the EPSRC under grant "ODP viewpoints in a Development Framework".

References

[1] R. Barden, S. Stepney, and D. Cooper. *Z in practice.* Prentice Hall, 1994.

[2] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, March 1996.

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

[4] J. P. Bowen, A. Fett, and M. G. Hinchey, editors. *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[5] H. Bowman, J. Derrick, P. Linington, and M. Steen. FDTs for ODP. *Computer Standards and Interfaces*, 17:457–479, September 1995.

[6] E. Brinksma, G. Scollo, and C. Steenbergen. Process specification, their implementation and their tests. In B. Sarikaya and G. v. Bochmann, editors, *Protocol Specification, Testing and Verification, VI*, pages 349–360, Montreal, Canada, June 1986. North-Holland.

[7] 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.

[8] 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.

[9] J. Derrick, E.A. Boiten, H. Bowman, and M. Steen. Translating LOTOS to Object-Z. In D.J.Duke and A.S.Evans, editors, *2nd BCS-FACS Northern Formal Methods Workshop*, Workshops in Computing. Springer-Verlag, July 1997.

[10] 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, September 1995.

[11] Kai Engelhardt and W-P de Roever. *Model-Oriented Data Refinement.* To appear.

[12] 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.

[13] C. Fischer. How to combine Z with a process algebra. In Bowen et al. [4], pages 5–23.

[14] C. Fischer and G. Smith. Combining CSP and Object-Z: Finite or infinite trace semantics. In T. Higashino and A. Togashi, editors, *FORTE/PSTV'97*, pages 503–518, Osaka, Japan, November 1997. Chapman & Hall.

[15] A. Galloway and W. Stoddart. An operational semantics for ZCCS. In Hinchey and Liu [17], pages 272–282.

[16] He Jifeng and C.A.R. Hoare. Prespecification and data refinement. In *Data Refinement in a Categorical Setting*, Technical Monograph, number PRG-90. Oxford University Computing Laboratory, November 1990.

[17] M. G. Hinchey and Shaoying Liu, editors. *Formal Engineering Methods: Proc. 1st International Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, 12–14 November 1997. IEEE Computer Society Press.

[18] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[19] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

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

[21] P. J. Lupton. Promoting forward simulation. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 27–49. Springer-Verlag, 1991.

[22] B. Mahony and J. S. Dong. Adding timed concurrent processes to Object-Z: A case study in TCOZ. In Bowen et al. [4], pages 308–327.

[23] E.-R. Olderog. Combining specification techniques for processes, data and time. In Bowen et al. [4], page 192. Abstract.

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

[25] 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.

[26] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In Hinchey and Liu [17], pages 293–302.

[27] J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[28] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In Hinchey and Liu [17], pages 283–292.

[29] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

[30] 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 - Formal Methods in Software Development*, LNCS 428, pages 340–351, Kiel, FRG, April 1990. Springer-Verlag.

[31] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, 1987.