Combining component specifications in Object-Z and CSP

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 [13, 32], an object-oriented extension of Z [35], 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 [38]. 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 [4] 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 [22] 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 sections discusses how we can structure specifications to support the description and reuse of components separately from how they are used. In section 5 we look at the refinement of components, and we conclude in section 6.

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. Although 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,
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 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 \( n fs \) mountings of home directories to provide a location transparency to the user.

The \textit{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 [23] 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 inter-working
- 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. We initially give the specification entirely in Object-Z. A single computational object is given by the class \textit{CompObj} (given here just as a template), which perhaps represents some sort of watchdog timer:
\[\text{CompObj}\]

\[
\text{state declarations come here}
\]

\[\text{INIT}\]

\[
\text{initialisation predicate comes here}
\]

\[
\begin{align*}
enable &\equiv \ldots \\
reset &\equiv \ldots \\
timeout &\equiv \ldots \\
halt &\equiv \ldots
\end{align*}
\]

which contains a number of operations (not fully specified): \textit{enable}, \textit{reset} et cetera.

The engineering mechanism consists of a number of nodes, where each \textit{Node} consists of a computational object \(c\), an identifier, a boolean representing whether that node is in service, and a variable to \textit{count} the number of resets. It has an operation \textit{start} to bring it into service together with operations \textit{enable}, \textit{reset} etc performed on the computational object \(c\). These latter operations are simple examples of promotion, where for example, the local \textit{reset} operation in \(c\) is promoted to a \textit{reset} on the \textit{Node}.

\[\text{Node}\]

\[
\begin{align*}
id &\colon \mathbb{N} \\
c &\colon \text{CompObj} \\
l &\colon \mathbb{B} \\
count &\colon \mathbb{N}
\end{align*}
\]

\[\text{INIT}\]

\[
c.\text{INIT} \land \neg l \land count = 0
\]

\[
\begin{align*}
\text{start} &\equiv [\Delta(l); r! : \mathbb{B} \mid \neg l \land l' \land r! = l] \\
enable &\equiv [l] \bullet c.\text{enable} \\
reset &\equiv [\Delta(count) \mid count' = count + 1 \land l] \bullet c.\text{reset} \\
timeout &\equiv [l] \bullet c.\text{timeout} \\
halt &\equiv [l] \bullet c.\text{halt}
\end{align*}
\]

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 \textit{start} to bring a node into service, and an \textit{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 \textit{enable} 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.

\[
\begin{align*}
\text{FTransparency} & \\
& \quad f : I \rightarrow \text{Node} \\
& \quad \forall i, j : \text{dom } f \cdot i \neq j \rightarrow f(i).id \neq f(j).id \\
\text{INIT} & \\
& \quad \| \text{dom } f \geq 2 \land \forall i : \text{dom } f \cdot f(i).\text{INIT} \\
\text{enable} & \equiv \| i : \text{dom } f 
\cdot f(i).\text{enable} \\
\text{alarm} & \equiv \left[ \text{report}! : R \mid \# \{ i : \text{dom } f \mid f(i).l \leq 1 \land \text{report}! = \text{"start new node"} \} \right] \\
\text{Select} & \\
& \quad i? : I \\
& \quad i? \in \text{dom } f \\
\text{start} & \equiv \text{Select} \cdot f(i?).\text{start} \\
\text{inservice} & \equiv \text{Select} \cdot \left[ r! : \mathbb{B} \mid f(i?).l = r! \right]
\end{align*}
\]

The use of a number of nodes indexed by \( f \) has allowed us to separate out the global behaviour (e.g. the \text{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 [38,1] for comprehensive accounts), although because Z does not have object encapsulation, the exact mechanism is syntactically more complex than in Object-Z.

2.1 Structuring Specifications

The above example was specified 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 [6]. 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 [31, 15, 18, 33, 16, 26, 36, 19], 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 [31, 15] 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 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 [3].

3.1 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

\[ \text{COMPOBJ}[\text{reset, timeout, halt}] : \text{noexit} := \text{reset}; \text{COMPOBJ}[\text{reset, timeout, halt}] \]

\[ \text{timeout?} : \text{nat}; \text{halt!}t; \text{stop} \]

endproc

The specification of the engineering viewpoint consists of a number of Nodes, 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. The signature of an Object-Z class includes boolean state variables, one for each operation, to account for the fact that these operations may be enabled in some (unspecified) circumstances, and disabled in others. Any operation requires its enabling boolean to be true, and may change all of these booleans. This allows us to construct the combined behaviour correctly.

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:
\[
\begin{align*}
\text{CompObj} & \quad \text{prer, pret, preh : } \mathbb{S} \\
\text{INIT} & \quad \text{reset} \equiv [\Delta(\text{prer, pret, preh}) \mid \text{prer}] \\
& \quad \text{timeout} \equiv [\Delta(\text{prer, pret, preh}) \mid \text{pret}] \\
& \quad \text{halt} \equiv [\Delta(\text{prer, pret, preh}) \mid \text{preh}]
\end{align*}
\]

together with \textit{Node} and \textit{FTransparency} exactly as before.

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 \textit{COMPOBJ} class/process with its use as a component in the engineering viewpoint, and can then be documented as a relation which says that \textit{COMPOBJ} corresponds to \textit{CompObj} and the two \textit{reset} events coincide, etc:

\[
\{(\text{COMPOBJ, CompObj}), (\text{reset, reset}), (\text{timeout, timeout}), (\text{halt, halt})\}
\]

### 3.2 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.

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 [40, 15, 30]). 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 [12], 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 [30], 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. This provides a basis by which we can verify that the translation is correct, that is, that the meaning of a term in one language is bisimulation equivalent to the meaning of that term after translation. [12] verifies the translation in detail.
The ADT component of a LOTOS specification is translated directly into the Object-Z type system. 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. Adopting this approach allows a natural mapping to be identified between many of the behavioural constructs in the two languages, for example, we find that process instantiation in LOTOS corresponds naturally to object instantiation in Object-Z.

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 \textit{stop}, \textit{exit} or any process instantiations). The translation of a process definition begins with its terminals and successively applies the operator translation rules given in \cite{source} until each operator/terminal has been translated.

For example, to translate the behaviour of \textit{COMPOBJ}, we apply the translation algorithm to produce an Object-Z class called \textit{COMPOBJ} containing operations schemas \textit{reset}, \textit{timeout} and \textit{halt}. 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:

\begin{verbatim}
COMPOBJ
  s : N
  x : N

INIT
  s = 0

reset ≜ [s = 0]

_TIMEOUT
  △(s, x)  ch? : N
  x' = ch?
  (s = 0 ∧ s' = 1)

  △(s)
  ch! : N
  x = ch!
  (s = 1 ∧ s' = 2)

halt
\end{verbatim}

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 \textit{COMPOBJ} class to an operation in the \textit{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 [3]. 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. An example of such a situation is given in [2] where partial specifications are related by non-trivial correspondences that relate portions of behaviour. Situations like this, where aspects of behaviour are related (as opposed to merely operation names), can be due to complex relationships between the state spaces of the partial specifications, or even due to action refinement between the sets of operations in the partial specifications (see [2] for a discussion).

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 5.

4 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.

There are a number of possible approaches to combining state-based languages with process algebras. One approach is to combine Object-Z classes with CSP operators as described by Smith in [31] (see also the work of Smith and Derrick [34, 33] which consider refinement and verification in this notation). Alternative approaches to combining Z or Object-Z with CSP or CCS include the work of Fischer [15, 17], Mahony and Dong [27, 28], Galloway [19] and Taguchi and Araki [36]. A survey of some of these approaches is given in [16].

Fischer combines Object-Z with CSP by using a failure-divergence semantics as the basis for the integration in a manner similar to [31]. However, Object-Z classes are extended with channel definitions and a CSP process. In addition, both the precondition and
guard of an operation are defined and events can either be atomic or have duration (and therefore have a start and end). The resulting notation, known as CSP-OZ, is complex but expressive and like [31] refinement has been defined within the notation.

Malony and Dong [27, 28] use a combination of Object-Z and timed CSP in their TCOZ language. However, instead of identifying operations and events, operation parameters are mapped to events. Similar approaches, where the granularity of operations and events is different, but using Z and CCS include the work of Galloway [19] and Taguchi and Araki [36].

There has also been work on combining CSP with the B notation, representative examples include that of Butler [10] and Treharne and Schneider [37]. Butler uses CSP to control the temporal ordering of B operations in a manner similar to the approach discussed later in this paper. However, to define the meaning of the specification he defines a translation of CSP into B, and therefore uses a restricted subset of CSP.

In the approach of Treharne and Schneider, in addition to controlling the temporal ordering of events, the CSP acts as a control executive for the B machine. That is, internal inputs and outputs are also passed between the CSP process and the B machine parts of the combined specification.

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 [31]. 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 [34, 31], 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. In addition we will allow CSP processes to be used to define the temporal ordering of events within the Object-Z components. We do this by combining CSP processes and Object-Z classes in parallel composition, which acts as a conjunction between the ordering and behaviour defined in both components.

4.1 The semantics of hybrid languages

For a hybrid language to make sense it is necessary to give it a semantic model. For example, in [34, 31] 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 [8, 9]. 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 [8, 9].

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\(^1\) 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 due to the fact that 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 [33]. 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. One direction to extend this is to use CSP to determine the temporal ordering of events defined in another component, whilst another possibility is to use components specified in CSP within Object-Z components. The following sections briefly sketch some of the issues involved in these approaches.

4.2 Using CSP processes with Object-Z classes

The common semantic model of Object-Z and CSP described above allows an integration of CSP and Object-Z where Object-Z components are placed in parallel composition with CSP processes. As an example, consider the Node class given earlier. The purpose of the state variable $l$ was merely to control the correct sequencing of operations, and we can specify this required ordering via a CSP process $Order$.

\(^1\) The history model is the semantic model of Object-Z defined in [30].
\[\text{Order} = \text{start}\, ?\, r \rightarrow P \]
\[P = \text{enable} \rightarrow Q \]
\[Q = \text{reset} \rightarrow Q \]
\[\text{timeout} \rightarrow \text{halt} \rightarrow \text{skip} \]

The complete specification will be given by

\[\text{Order} \parallel \text{Node}_2\]

where \text{Node}_2 is the following Object-Z class.

\[
\text{Node}_2
\]

\[id : \mathbb{N}\]

\[c : \text{CompObj}\]
\[\text{count} : \mathbb{N}\]

\[
\text{INIT}
\]
\[c.\text{INIT} \land \text{count} = 0\]

\[\text{start} \triangleq [r! : \mathbb{B} \mid r! = \text{false}]\]
\[\text{enable} \triangleq c.\text{enable}\]
\[\text{reset} \triangleq [\Delta(\text{count}) \mid \text{count}' = \text{count} + 1] \bullet c.\text{reset}\]
\[\text{timeout} \triangleq c.\text{timeout}\]
\[\text{halt} \triangleq c.\text{halt}\]

The effect of this combined Object-Z/CSP specification is as follows. The CSP parallel composition causes the two components to run concurrently synchronising on common events, with all other events being interleaved. This gives us the required temporal ordering that was in the original \text{Node} class without the use of the variable \(t\).

Note the use of the input parameter on the \text{start} event in the CSP process specification. This is necessary because the CSP parallel composition operator requires synchronisation on identical events, and in the case of events with communication this requires channel name with communication value in addition to the event name. Thus if we wrote

\[\text{Order} = \text{start} \rightarrow P\]

for the CSP process then it would deadlock due to the \text{start} event not being compatible with the communicated value in the Object-Z \text{start} operation.

Since we do not want to constrain the value of the output \(r!\), we simply accept it as an input in the CSP part, this value is not hidden in the complete specification which thus has the same effect as the original \text{Node} class. We have not decorated the other events with inputs since the \text{CompObj} class was not specified in full. If there were inputs or outputs in the \text{CompObj} class, then events in \text{Order} would be similarly decorated with inputs in the same way that the \text{start} event is.
4.3 Using promotion in a hybrid language

We would also like to support combinations of components where more than just synchronous communication takes place. Consider, for example, the combination of components specified in the \textit{FTransparency} class. At this global level we have a number of operations to be specified including \textit{enable}, \textit{start} and \textit{alarm}.

The operation \textit{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 \textit{start} applies the \textit{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 operation \textit{enable} is a concurrent composition of the enable events 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.

We would like to use this flexibility in a hybrid language, but cannot as it stands. To rectify this 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 \textit{FTransparency}. To support this specification style using CSP and Object-Z 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 [31] 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 [30] 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 \( c.\textit{enable} \) is represented semantically by the history which states that the object \( c \) undergoes an event associated with the operation \textit{enable}.

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:

\[
\text{CompObj} = \text{enable} \rightarrow P \\
P = \quad \text{reset} \rightarrow P \\
\quad \square \\
\quad \text{timeout}\,?t: \mathbb{N} \rightarrow \text{halt}\,!t \rightarrow \text{skip}
\]

Together with \textit{Node} and \textit{FTransparency} exactly as before, viz
An 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 details here, it should be clear that this is feasible modulo some technicalities. The technicalities arise 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. 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. 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.

5 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, and discuss to what extent we can achieve compositionality.

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, for all contexts $C$, $C[X] \sqsubseteq_Z C[Y]$.
- if $C \sqsubseteq_Z D$ then, for all components $X$, $C[X] \sqsubseteq_Z D[X]$.

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 [14]. In Object-Z these take the following form [33]:

**Definition 1** Downward simulation

An Object-Z class $C$ is a downward simulation of the class $A$ if there is a retrieve relation $\text{Abs}$ such that every abstract operation $AOp$ is recast into a concrete operation $COp$ and the following hold.

\[ DS_1 \ \forall \text{Astate} \in C\text{state}; \ \text{Abs} \Rightarrow (\text{pre} AOp \iff \text{pre} COp) \]
\textbf{DS.2} $\forall \text{Astate}; \text{Cstate}; \text{Cstate}' \cdot \text{Abs} \land \text{AOp} \implies \exists \text{Astate}' \cdot \text{Abs}' \land \text{AOp}$

\textbf{DS.3} $\forall \text{Cinit} \implies \exists \text{Ainit} \cdot \text{Abs}$

\textbf{Definition 2} Upward simulation

An Object-Z class $C$ is an upward simulation of the class $A$ if there is a retrieve relation $\text{Abs}$ such that every abstract operation $\text{AOp}$ is recast into a concrete operation $\text{COp}$ and the following hold:

\textbf{US.1} $\forall \text{Cstate} \cdot \exists \text{Astate} \cdot \text{Abs} \land \text{pre AOp} \implies \text{pre COp}$

\textbf{US.2} $\forall \text{Astate}'; \text{Cstate}; \text{Cstate}' \cdot \text{COp} \land \text{Abs}' \implies \exists \text{Astate} \cdot \text{Abs} \land \text{AOp}$

\textbf{US.3} $\forall \text{Astate}; \text{Cinit} \cdot \text{Abs} \implies \exists \text{Ainit}$

We write $A \subseteq_Z C$ 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 [9], where we write $P \sqsubseteq_{FD} Q$ if

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

This is the standard notion of refinement in CSP and is closely related to the reduction refinement relation ($\text{red}$) in LOTOS [7]. In fact if we restrict ourselves to divergence free processes then $\text{red}$ 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).

The question of compositionality of components upon refinement is a general problem in the use of hybrid languages, and is discussed in [16, 29]. Clearly, answers to such questions depend on how the components are used within a context $C[\cdot]$.

\section{5.1 Compositionality within a process algebra}

Even in a single language such compositionality properties do not always hold. For example, consider LOTOS with the reduction refinement relation. Then the process $i; a; \text{stop}$ is a reduction of $a; \text{stop}$. However, if we place these two processes in the context of a choice we find that $b; \text{stop}||i; a; \text{stop}$ is not a reduction of $b; \text{stop}||a; \text{stop}$. 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, the absence of an explicit internal event in CSP at the specification level means that compositionality does hold in CSP with respect to failures-divergences refinement.
5.2 Compositionality in simple combinations of CSP and Object-Z

By simple combinations we mean specifications consisting of Object-Z classes combined using CSP parallel composition operators, or Object-Z classes composed with CSP processes, i.e. combinations as discussed by Smith and Derrick in [34] and those defined in Section 4.2.

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 [20, 39], and also for the Object-Z simulation rules (see [24] and the discussion in [33, 34]).

This, together with the compositionality of CSP, means that simple combinations of CSP and Object-Z are compositional. [33, 17] demonstrate this result 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 $C[\cdot]$ is the CSP part of the specification which describes the component interaction, then:

- if $X \subseteq Z Y$ then $C[X] \subseteq_{FD} C[Y]$.

with similar results hold for Object-Z classes composed with CSP processes.

5.3 Compositionality in a hybrid language

The situation in a hybrid language is slightly more complex. By hybrid we mean combinations of CSP and Object-Z where the CSP processes can be used in promotions as discussed earlier. However, the result really boils down to compositionality within Object-Z itself, and we discuss this now.

In Object-Z context 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 [30]. For example, we have the following Node/./context:
\[\text{Node[]}\]
\[
\begin{array}{l}
\text{id} : \mathbb{N} \\
c : \Box \\
l : \mathbb{B} \\
count : \mathbb{N}
\end{array}
\]

\[\text{INIT}\]
\[
c.\text{INIT} \land \neg l \land \text{count} = 0
\]

\[\text{start} \equiv [\Delta(l); \ r! : \mathbb{B} \mid \neg l \land l' \land r! = l]
\]

\[\text{enable} \equiv [l] \cdot c.\text{enable}
\]

\[\text{reset} \equiv [\Delta(\text{count}) \mid \text{count'} = \text{count} + 1 \land l] \cdot c.\text{reset}
\]

\[\text{timeout} \equiv [l] \cdot c.\text{timeout}
\]

\[\text{halt} \equiv [l] \cdot c.\text{halt}
\]

where \(\Box\) represents the elided class. \(\text{Node}[\text{CompObj}]\) is thus the class \(\text{Node}\) specified above.

In this scenario we can exploit relevant results about refinements of promotions. A promotion is said to be \textit{free} if the global context does not make constraints upon the component variables in the local state [38], [25] 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 [25, 38]. 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 we have to ensure that the structure of the encapsulation into classes produces promotions that are free.

This does not always hold, in particular direct access to a component’s state variable constrains the promotion and compositionality can fail. As an example, consider the use of \(\text{Node}\) within \(F\text{Transparency}\). The \textit{inside} operation makes direct reference to the boolean state variable \(l\) of \(\text{Node}\), thus a simple data refinement of \(\text{Node}\) (say for example one that toggles the value of \(l\)) will not give rise to a refinement of \(F\text{Transparency}\).

However, we can state a general result that guarantees compositionality when promotions conform to a particular template (for a proof see [11]). In particular, we allow declarations which are arbitrary partial functions, indexed over some set \(I\), of the form \(f_D : I \rightarrow D\), where \(D\) is a class. Thus \(f_D(d)\) will be an object of type \(D\) whenever \(i \in \text{dom} f_D\). This allows objects to be used in fairly general ways within a class including single instances, sets, sequences and partial or total functions.

The operation promotions we shall consider will be of the form

\[Op \equiv [i? : I \mid i? \in \text{dom} f_D] \cdot f_D(i?). Op
\]
This promotes an operation in \( D \) to an operation in the main class by applying it to a selected component.

**Theorem 1.** Let \( A = (A.\text{STATE}, A.\text{INIT}, \{Op\}) \) and \( C = (C.\text{STATE}, C.\text{INIT}, \{Op\}) \) be Object-Z classes. Suppose that \( C \) is a downward simulation of \( A \), and let the context \( \Phi(D) \) be defined by

\[
\begin{align*}
\Phi(D) = & \quad f_D : I \rightarrow D \\
\text{INIT} = & \quad \forall i : \text{dom} \ f_D \cdot f_D(i).\text{INIT} \\
Op = & \quad [i? : I \mid i? \in \text{dom} \ f_D] \cdot f_D(i?).\text{Op}
\end{align*}
\]

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

This, together with the common semantic basis for Object-Z and CSP means that we have compositionality of components (whether given in CSP or Object-Z) when our promotions conform to the form given in Theorem 1. So for example if a CSP component \( X \) is refined to \( Y \) then, because simulations are sound and jointly complete with respect to failures-divergences refinement, there exist simulations to verify this refinement. Since the promotion of an Object-Z refinement is a refinement of a promotion when the component conforms to the template given in Theorem 1, we then have \( C[X] \subseteq_z C[Y] \).

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 \texttt{CompObj} component.

We can refine the component

\[
\text{CompObj} = \text{enable} \rightarrow P \\
P = \begin{cases} \\
\text{reset} \rightarrow P \\
\text{\[timeout?t : N \rightarrow halt!t \rightarrow skip\]} \\
\end{cases}
\]

to an implementation which uses \texttt{CompObj2} containing separate sender and receiver processes which communication via a channel \texttt{mid} and an acknowledgement channel \texttt{ack}:

\[
\text{send} \triangleq \text{timeout?t : N} \rightarrow \text{mid!t} \rightarrow \text{ack} \rightarrow \text{send}
\]
\[ \text{rec} \equiv \text{mid?l} \rightarrow \text{halt}!l \rightarrow \text{ack} \rightarrow \text{rec} \]

\[ \text{CompObj2} \equiv (\text{send} \parallel \text{rec}) \setminus \{\text{mid. ack}\} \]

The failure transparency mechanism which uses \text{CompObj2} within \text{CompObj} is then a refinement of the original failure transparency mechanism.

6 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 operations specified in CSP within Object-Z.

The 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

33. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In Hinchey and Liu [21], pages 293–302.