A constraint-oriented style for object-oriented formal specification

Tommaso Bolognesi* and John Derrick†

*CNR Istituto di Elaborazione dell'Informazione,
Pisa, Italy. (Email: t.bolognesi@iei.pi.cnr.it).

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

Short title: A constraint-oriented style for OO formal specification

Abstract

We propose a specification style which combines the features and advantages of object-oriented and constraint-oriented system decomposition. A system description is decomposed into data-handling objects, which usually reflect objects and individual operations in the real system, and temporal-ordering constraints, which capture aspects of functionality as behavioural sequences, with a possibility to introduce also entities which blurr the distinction between these two extreme cases. Composition is achieved via synchronisation on shared operations: different objects/constraints insisting on an operation express different views on the enabling conditions and effects of that operation.

Objects, constraints, and their composition can be formally specified in Object-Z, an object-oriented extension of the Z notation, with pure temporal ordering constraints equivalently expressed as transition graphs. However, expressing object/constraint compositions in Object-Z is cumbersome. We solve this problem by proposing a natural textual notation, called co-expression, which is a most direct description of an object/constraint interconnection graph, and we define a mapping from co-expressions to Object-Z. Thus, specifications in an object/constraint-oriented style can be conveniently written using transition graphs and interconnection diagrams mixed with Object-Z text, and then translated into this language.

1 Introduction

Object-oriented design is a well established discipline for the design of large software systems, based on a peculiar way to break the complexity of a system and decompose its description into parts. In [7] Meyer writes:

> Object-Oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs.

A central idea in O-O design is that it is not only natural, but also convenient, for maintainability and reusability reasons, to let the design reflect the categories of objects found in the real system, provided these are viewed at a sufficiently high level of abstraction.

An object is an entity that encapsulates data structures, and offers operations to its users for manipulating those structures. Thus, in O-O design a system is fundamentally conceived as a set of interacting objects, each encapsulating a fragment of the global state; interaction involves one object offering an operation capable of modifying the object's internal data, and another object triggering that operation (see Figure 1).

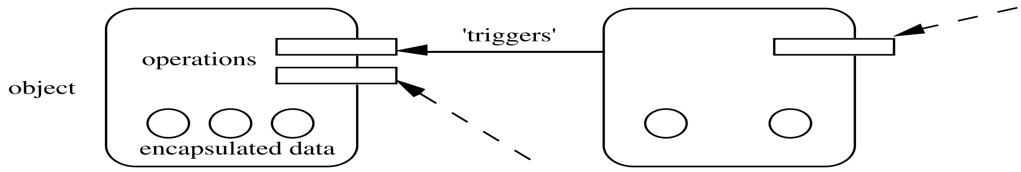

Figure 1 - Object-oriented system decomposition

We propose here to extend the O-O decomposition process to operations. Let us anticipate the essence of this idea with the support of a few abstract pictures. An operation shall be conceived as composed of parts, each one reflecting different concerns and affecting the encapsulated data of different objects (see Figure 2). Objects shall interact also by sharing operations, rather than by only triggering the operations of one another in a client-server fashion.

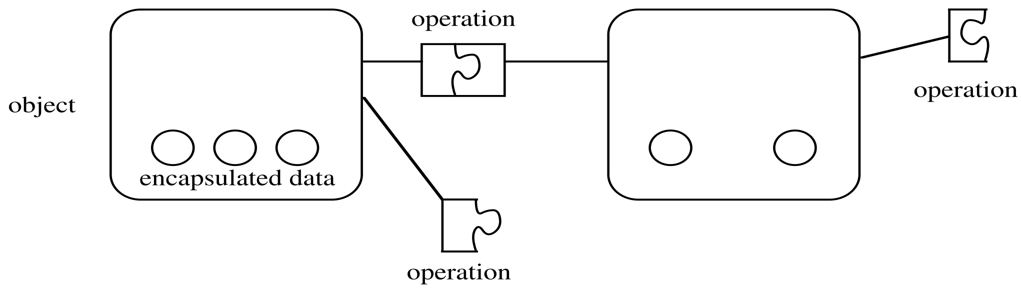

Figure 2 - Decomposing systems and operations

Under this enhanced decomposition criterion, which we view as complementary rather than alternative to classical O-O decomposition, the notion of object expands to a wider concept, that of constraint, and we shall talk about constraint- and object-oriented (C-O-O) decomposition, and C-O-O design.

A first, simple form of C-O-O design is illustrated by the abstract example of Figure 3. In this type of system one can draw a clear separation line between the traditional, data encapsulating objects, each one offering its own set of operations, and the operation-ordering constraint, which constrains all those operations according to a precisely defined temporal pattern. In Figure 3, the graphical elements filled by vertical lines refer to the handling of encapsulated data, that is, they represent the data items themselves (the circles) and those aspects of an operation that relate to how the data structures are handled (the puzzle pieces labelled 'how'); the graphical elements filled by horizontal lines refer to the ordering of operations in time, that is, they represent an operation just as an event related to other events in time (the puzzle pieces labelled 'when'), and the auxiliary, typically binary state variables used for enforcing the temporal ordering.

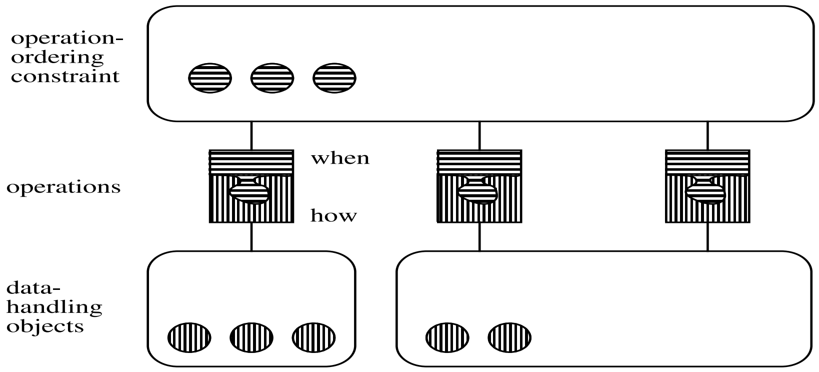

Figure 3 - Decomposition into data-handling objects and operation-ordering constraint

In a more general form of C-O-O design, the concerns about handling data structures and ordering operations are intertwined, so that we may have pure data-handling objects, pure operation-ordering constraints, and objects/constraints dealing with both aspects (see Figure 4). Since an operation-ordering constraint can be viewed as an object, and a data-handling object can be viewed as a constraint relating the encapsulated data values and the operation parameters, in the new C-O-O setting we shall frequently use the terms 'object' and 'constraint' as synonyms. Note that an object may impose on the same operation both temporal ordering and data-related constraints: in this case the relevant puzzle piece is painted both with horizontal and with vertical lines.

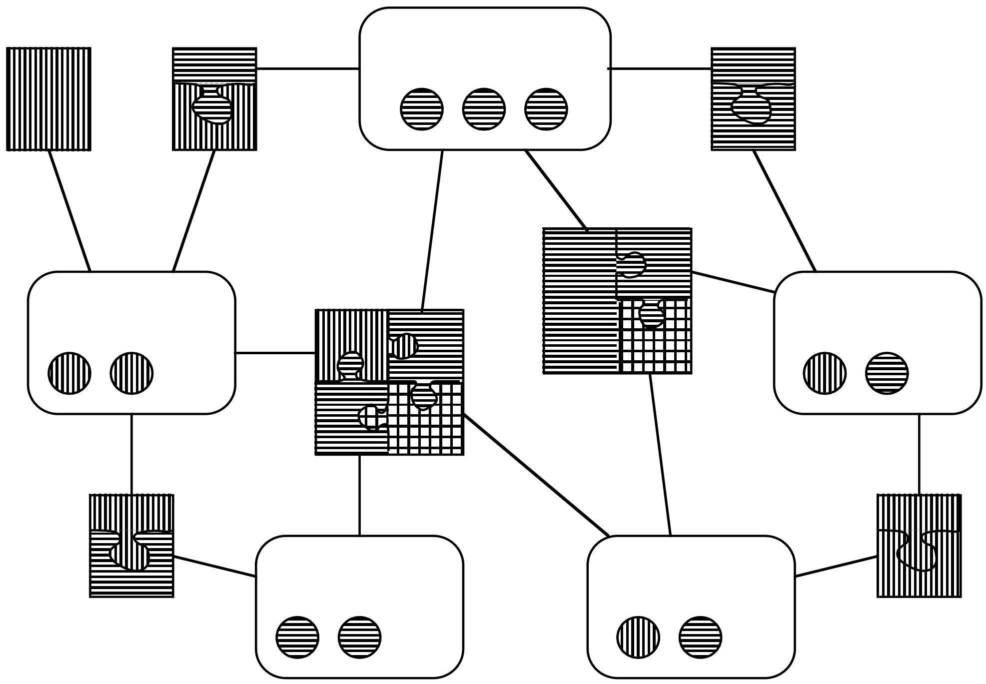

Figure 4 - General C-O-O decomposition

The interest in formal methods for (hardware and) software development has grown quite substantially in the last few years, with industrial environments paying more and more attention to various formal specification and verification techniques, and to a growing body of supporting tools. The C-O-O design technique that we propose here is fully formal, in that it consists of a special, constraint-oriented usage of the Object-Z formal specification language [3], and of an associated, convenient syntactic extension; Object-Z is, in turn, a well known object-oriented extension of the Z notation [11].

We expect some important advantages from our C-O-O design technique, that derive from its three-faceted nature. Being object-oriented, it supports specifications that reflect the (essential features of) actual objects in the 'real' world, with the usual benefits in terms of understandability, maintainability and reusability. Being formal (mapped into Object-Z), it provides support for analysis and verification. Further advantages are expected from the third aspect of C-O-O design, namely constraint-oriented specification.

At a very general level, we may define the 'constraint-oriented specification style' as a way of structuring the description of virtually anything as the conjunction of constraints on a set of atomic elements, each constraint representing a partial view of the global picture. A system of inequalities is a simple example of constraint-oriented description, with variables playing the part of the atomic elements, individual inequalities representing constraints, and the whole space of solutions corresponding to the global picture.

The advantage offered by the constraint-oriented specification style is, again, understandability; in this respect, while structuring a specification into objects is convenient because it reflects the concrete objects found in the 'real' system, structuring it into constraints is convenient because it partitions the system behaviour into sufficiently simple and separated functional aspects. Furthermore, composing constraints is natural, almost as natural as putting in sequence some sentences in natural language. Interestingly, while O-O decomposition has been originally proposed as an alternative to functional-decomposition, as apparent from the introductory quote from Meyer, with constraints we are re-integrating a flexible form of functional decomposition into O-O design.

The structure of the paper is as follows. In Section 2 we introduce the constraint/object-oriented formal specification style by using an example, the jewellers system. In Section 3 we show how a composition of constraints can be concisely described by a 'co-expression', and in Section 4 we formally define the expansion of co-expressions into Object-Z text. In Section 5 we summarise our contribution and relate it to existing work.

The paper assumes familiarity with the Z notation [11], and basic notions of object-oriented programming.

2 Specifying the jewellers system in Object-Z

In this section we specify our running example, the Jewellers shop, in Object-Z (this example was first used in [1] for illustrating the flexibility of other specification languages). The specification illustrates that Object-Z supports a constraint-oriented specification style, and provides for a nice separation of concerns, in that component objects are used to either specify ordering constraints or used to specify data handling. The specification is completed by describing the interaction among these components, which is given in a top level system object, whose purpose is to specify the system-wide constraints.

We begin by providing an informal description of the Jewellers shop; within the description, italics will be used to identify keywords used in the formal specification.

1. A jeweller shop is run by three friends: *Mike*, *Mary* and *Jane*. Together, they open the shop every morning and close it every evening (*openShop*, *closeShop*), perhaps via a three-key lock. Every night Mike and Mary go to the theatre (*toTheatre*), while Jane goes to a discotheque (*toDisco*). The following is a description of what may happen during the day in the shop.

2. *Mike* takes care of accepting new precious pieces (*pieceIn*) from external producers that need not be specified, and of storing them in the *safe*. At any time he may also update a table (*ValTable*) that indicates the value of each precious metal, namely *platinum*, *gold*, and *silver*, per weight unit (*unitVal*). An update consists of a new (material, unitary-value) pair (*matValPair*). The input pieces are characterised by their weight, material, and value (*wei, mat, val*), the third parameter depending exclusively on the first and second ones, via the table of unitary values.

3. *Jane* is the window dresser; she may move pieces between the *Safe* and the shop *Window* (*pieceToSafe* and *pieceToWindow*).

4. *Mary* is in charge of the cash deposit (*Cash*), which she opens (*cashUnlock*) right after entering the shop, and closes (*cashLock*) before leaving. She only sells the pieces in the *window* (*pieceOut)*, at a *price* which is *1.3* times their current value according to their material and weight. This amount of money is put in the *Cash* deposit. Note that the value of a piece when leaving the shop may differ from its value when entering it, in case the table had been updated between these two events.

   Two different customer behaviours are also described.

5. *Peter* has a *bag* which he can fill with pieces that he buys during a shopping session. The current weight of the bag (*currWei*) cannot exceed a maximum weight capacity (*MaxWei*), but Peter is not happy until the money spent in the current shopping session (*currExp*) has reached a given threshold (*TargetExp*); at that point he empties his bag, say, at home (*reset*) and is ready for a new shopping session.

6. *Paul* buys a piece only if he has enough *funds*. However, he can *refill* his funds by adding some amount of money (*delta*), from time to time.

2.1 The formal specification

Our formal specification is structured into objects/constraints. In the informal specification above the system appears as partitioned into 'persons', and we shall preserve this aspect in the formal description by creating a constraint for each person (*Mike, Mary*, ...). However, to achieving an effective separation of concerns each person-constraint shall only deal with the temporal ordering of the person's operations (temporal ordering constraints), while the individual components of the data space (*ValTable, Cash*, ...) and the way these are affected by those operations shall be specified by means of other objects (data handling objects).

We will specify the example in Object-Z, which is a typical formal object-oriented specification language. Object-Z [3] is an object-oriented extension of the specification language Z [11] which has been developed over a number of years and is perhaps the most mature of all the proposals to extend Z in an object-oriented fashion.

Object-Z uses a *class schema* to encapsulate a state schema together with the operations acting upon that state. It is represented as a named box with zero or more generic parameters. The class schema may include local type or constant definitions, at most one state schema and initial state schema together with zero or more operation schemas.

Let us start by specifying the temporal ordering objects. A most direct and concise representation of the operation orderings for *Mike, Jane* and *Mary* is provided by the labelled transition graphs of Figure 5.

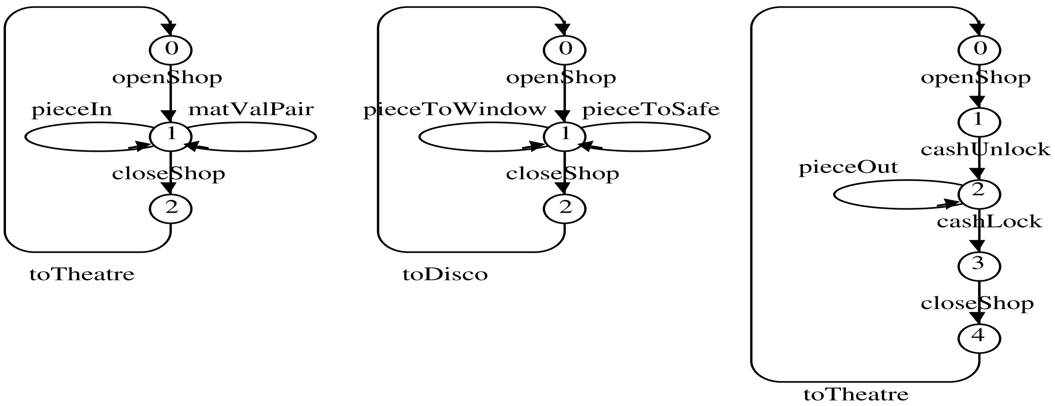

Figure 5 - Labelled transition graphs for the operations of (a) *Mike*, (b) *Jane* and (c) *Mary*.

To formally specify the object *Mike*, we identify the actions the object can perform (e.g. *openShop, closeShop* etc) together with their ordering. The specification will need a number of type declarations:

$$\begin{aligned}&\mathit{material} ::= \mathit{platinum} \mid \mathit{gold} \mid \mathit{silver}\\ &\mathit{weight} == \mathbb{N}\\ &\mathit{value} == \mathbb{N}\end{aligned}$$

$$\begin{array}{|l}\mathit{TargetExp} : \mathbb{N}\\ \mathit{MaxWei} : \mathbb{N}\end{array}$$

The object *Mike* can then be given as follows.

Mike

$$s : \{0, 1, 2\}$$

INIT

$$s = 0$$

$$\begin{aligned}&\mathit{openShop} \mathrel{\widehat=} [\Delta(s) \mid s = 0 \land s' = 1]\\ &\mathit{closeShop} \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 2]\\ &\mathit{pieceIn} \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 1]\\ &\mathit{matValPair} \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 1]\\ &\mathit{toTheatre} \mathrel{\widehat=} [\Delta(s) \mid s = 2 \land s' = 0]\end{aligned}$$

The variable s declared in the (unnamed) state schema is local to the class. The initial state schema *INIT* defines the initial values of the variables in the state schema. The class specified above has five operations: *openShop*, *closeShop*, etc. Each operation has a Δ-list which contains those state variables which may change when the operation is applied to an object of that class. As in Z, primed variables are used for denoting the effect of the change. An operation does not change the state variables that are not listed in its Δ-list.

In this specification the state variable s is used solely to prescribe an order in which the operations can be performed. Initially its value is set to zero. The operations change the state space, for example, the effect of *closeShop* is to give the value 2 to s. The preconditions and postconditions of the operations force them to be invoked in a particular order *. For example, the postcondition of *openShop* is that $s = 1$, which is also the precondition of *pieceIn*. Thus these two operations may occur in sequence.

Having specified *Mike* in this fashion, we can provide similar descriptions of the objects *Jane* and *Mary* as follows.

Jane

$$s : \{0, 1, 2\}$$

INIT

$$s = 0$$

$$\begin{array}{l}
openShop \mathrel{\widehat=} [\Delta(s) \mid s = 0 \land s' = 1] \\
closeShop \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 2] \\
pieceToWindow \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 1] \\
pieceToSafe \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 1] \\
toDisco \mathrel{\widehat=} [\Delta(s) \mid s = 2 \land s' = 0]
\end{array}$$

Mary

$$s : \{0, 1, 2, 3, 4\}$$

INIT

$$s = 0$$

$$\begin{array}{l}
openShop \mathrel{\widehat=} [\Delta(s) \mid s = 0 \land s' = 1] \\
closeShop \mathrel{\widehat=} [\Delta(s) \mid s = 3 \land s' = 4] \\
cashUnlock \mathrel{\widehat=} [\Delta(s) \mid s = 1 \land s' = 2] \\
cashLock \mathrel{\widehat=} [\Delta(s) \mid s = 2 \land s' = 3] \\
pieceOut \mathrel{\widehat=} [\Delta(s) \mid s = 2 \land s' = 2] \\
toTheatre \mathrel{\widehat=} [\Delta(s) \mid s = 4 \land s' = 0]
\end{array}$$

We move now to the description of the data handling objects of the system. We can identify three such components, namely *ValTable*, *Cash* and *SafeWindow*. The object *ValTable* represents the table of values per unit weight of each metal, and the operations which access or alter that table. The object *Cash* simply represents the cash deposit and the operation *pieceOut* that changes it. Finally *SafeWindow* represents the pieces in the safe and window display, together with the collection of operations that alter them.

We begin by specifying *ValTable* as follows.

*The interpretation of operations in an Object-Z class differs from that in Z, in that an Object-Z operation cannot occur outside its precondition

$$
\begin{array}{l}
\textbf{ValTable} \\
\quad valtable : material \rightarrow\!\!\!\!\!\!+\; value \\
\quad \textbf{INIT} \\
\quad valtable = \{(silver, 80), (gold, 100), (platinum, 120)\} \\
\quad \textbf{pieceIn} \\
\quad wei? : weight, mat? : material, val? : value \\
\quad \hline \\
\quad val? = wei? \times valtable(mat?) \\
\quad \textbf{matValPair} \\
\quad \Delta(valtable) \\
\quad mat? : material \\
\quad unitVal? : value \\
\quad \hline \\
\quad valtable' = valtable \oplus (mat?, unitVal?) \\
\quad \textbf{pieceOut} \\
\quad wei! : weight, mat! : material, price! : value \\
\quad \hline \\
\quad price! = 1.3 \times (wei! \times valtable(mat!))
\end{array}
$$

The table is represented as a partial function from *material* to *value*. The initialisation gives an initial value for each precious metal. The operations *pieceIn* and *pieceOut* use information contained in the table, which can be updated by the operation *matValPair*. The operation *pieceIn* accepts three inputs (by convention input variables are decorated by a question mark) and specifies that an invariant must be preserved between the value, weight and material of the incoming piece. *pieceOut* specifies that the price of a sold piece must be 1.3 times its value (outputs are used so that we can synchronise with an operation in another object later). Notice that the operations only specify the data handling, and in this object no constraints are placed upon their temporal ordering.

In a similar manner we can specify the objects *Cash* and *SafeWindow*. The former contains a description of how *pieceOut* effects the cash deposit, whilst the latter describes how operations alter the contents of the safe and window display. Because the safe and the window may contain multiple copies of pieces, we model them as bags (i.e. multisets).

$$
\begin{array}{l}
\textbf{Cash} \\
\quad cash : value \\
\quad \textbf{INIT} \\
\quad cash = 0 \\
pieceOut \mathrel{\widehat=} [\Delta(cash), price! : value \mid cash' = cash + price!]
\end{array}
$$

SafeWindow

$safe, window : \mathrm{bag}(weight \times material)$

INIT

$safe = [\![\,]\!], window = [\![\,]\!]$

pieceIn

$\Delta(safe)$
$wei? : weight, mat? : material$

$safe' = safe \uplus [\![(wei?, mat?)]\!]$

pieceToSafe

$\Delta(safe, window)$
$piece? : weight \times material$

$piece? \text{ in } window$
$safe' = safe \uplus [\![piece?]\!]$
$window' = window \setminus [\![piece?]\!]$

pieceToWindow

$\Delta(safe, window)$
$piece? : weight \times material$

$piece? \text{ in } safe$
$window' = window \uplus [\![piece?]\!]$
$safe' = safe \setminus [\![piece?]\!]$

pieceOut

$\Delta(window)$
$wei! : weight, mat! : material$

$(wei!, mat!) \text{ in } window$
$window' = window \setminus [\![(wei!, mat!)]\!]$

Having specified the component objects we now describe their interaction by specifying the Jewellers shop as a class of interacting objects. This can be achieved in Object-Z because a class can also include instances, i.e. objects, of other classes as state variables. This allows the concise specification of the interaction between components of a system. For example, the Jewellers class specified below comprises six component objects - one for each of the components specified above. Initially these objects are all in their own initial states. In the class we specify all the operations that can be performed, and each operation is described in terms of its effect on one or more of the component objects, that is, in terms of their respective operations.

The objects have operations applied to them using the dot notation. For example, the *cashUnlock* operation only effects the object *mary*, so this is specified simply as $cashUnlock \mathrel{\widehat=} mary.cashUnlock$. This has the effect of changing the state of the object *mary* according to its definition of *cashUnlock*. We are in effect promoting the local operation defined in *mary* to an operation available in this class.

The description of *openShop* is more complex, as it requires cooperation of three of the components. Each of these components contains a partial description of the *openShop* operation; the complete specification requires that the *openShop* operation of *mike, mary* and *jane* are performed in synchronisation. This can be specified in Object-Z using the parallel composition operator $\|$. The parallel operator $\|$ is an additional schema operator provided in Object-Z, and it enables communication between objects to be specified. It behaves like conjunction but also equates inputs and outputs with the same basename [3]. In the case of *openShop* there is no communication between the component operations, so the effect is a simple synchronisation of the components[†]. In the case of operation *pieceIn* the synchronisation of the three interacting objects also involves an agreement on the values of some input variables, since the operation *pieceIn* of object *ValTable* and that of object *SafeWindow* share variables *wei*? and *mat*?. The composition of operations that share some input and/or output variables is at the heart of constraint-oriented reasoning: each object participating in a composed operation expresses a different, partial constraint on that operation, by affecting and being affected by some of its parameters.

Jewellers

$$
\begin{array}{l}
mike : Mike, \\
jane : Jane, \\
mary : Mary, \\
cash : Cash, \\
safeWindow : SafeWindow, \\
valTable : ValTable
\end{array}
$$

INIT

$$mike.init \land jane.init \land mary.init \land cash.init \land safeWindow.init \land valTable.init$$

$$
\begin{array}{l}
openShop \mathrel{\widehat=} mike.openShop \parallel jane.openShop \parallel mary.openShop \\
closeShop \mathrel{\widehat=} mike.closeShop \parallel jane.closeShop \parallel mary.closeShop \\
cashUnlock \mathrel{\widehat=} mary.cashUnlock \\
cashLock \mathrel{\widehat=} mary.cashLock \\
pieceIn \mathrel{\widehat=} mike.pieceIn \parallel valTable.pieceIn \parallel safeWindow.pieceIn \\
pieceOut \mathrel{\widehat=} mary.pieceOut \parallel valTable.pieceOut \parallel cash.pieceOut \parallel safeWindow.pieceOut \\
pieceToWindow \mathrel{\widehat=} jane.pieceToWindow \parallel safeWindow.pieceToWindow \\
pieceToSafe \mathrel{\widehat=} jane.pieceToSafe \parallel safeWindow.pieceToSafe \\
matValPair \mathrel{\widehat=} mike.matValPair \parallel valTable.matValPair \\
toTheatre \mathrel{\widehat=} mike.toTheatre \parallel mary.toTheatre \\
toDisco \mathrel{\widehat=} jane.toDisco
\end{array}
$$

This provides a description of the Jewellers shop, and the specification is completed by describing the behaviour of the customers *Peter* and *Paul* and their interaction with the shop. *Peter* collects his pieces in a bag, and because the current weight of the bag cannot exceed a maximum weight capacity a state invariant must be preserved; this is specified in the state schema. In the state invariant of *Peter* we use the *count* function for bags [11], which counts the number of elements in a bag. The state invariant consists of four conjunctions. The first specifies that *currExp* is the money spent in the current shopping trip, whilst the third conjunct specifies that *currWei* is the current weight of the bag.

[†]Given there is no communication, we could simply have used conjunction here instead of parallel composition. However, we use $\parallel$ for uniformity throughout the specification.

$$
\begin{array}{l}
\textit{Peter} \\
\hline
\textit{currExp} : \textit{value} \\
\textit{peterBag} : \mathrm{bag}(\textit{value} \times \textit{weight}) \\
\textit{currWei} : \textit{weight} \\
\hline
\textit{currExp} = \textit{SUM}\{\textit{Value} : \textit{value}, \textit{Weight} : \textit{weight} \mid \\
\qquad (\textit{Value}, \textit{Weight}) \mathrm{\,in\,} \textit{peterBag} \bullet \textit{Value} \times \textit{count}\, \textit{peterBag}(\textit{Value}, \textit{Weight})\} \\
\textit{currExp} < \textit{TargetExp} \\
\textit{currWei} = \textit{SUM}\{\textit{Value} : \textit{value}, \textit{Weight} : \textit{weight} \mid \\
\qquad (\textit{Value}, \textit{Weight}) \mathrm{\,in\,} \textit{peterBag} \bullet \textit{Weight} \times \textit{count}\, \textit{peterBag}(\textit{Value}, \textit{Weight})\} \\
\textit{currWei} \leq \textit{MaxWei} \\
\hline
\textsc{Init} \\
\textit{peterBag} = [\![\,]\!] \\
\hline
\textit{reset} \mathrel{\widehat{=}} [\Delta(\textit{peterBag}, \textit{currExp}, \textit{currWei}) \mid \textit{peterBag}' = [\![\,]\!]] \\
\hline
\textit{pieceOut} \\
\Delta(\textit{peterBag}, \textit{currExp}, \textit{currWei}) \\
\textit{wei}? : \textit{weight} \\
\textit{price}? : \textit{value} \\
\hline
\textit{peterBag}' = \textit{peterBag} \uplus [\![(\textit{wei}?, \textit{price}?)]\!]
\end{array}
$$

$$
\begin{array}{l}
\textit{Paul} \\
\hline
\textit{funds} : \textit{value} \\
\hline
\textsc{Init} \\
\textit{funds} = 0 \\
\hline
\textit{refill} \mathrel{\widehat{=}} [\Delta(\textit{funds}), \textit{delta}? : \textit{value} \mid \textit{funds}' = \textit{funds} + \textit{delta}?] \\
\textit{pieceOut} \mathrel{\widehat{=}} [\Delta(\textit{funds}), \textit{price}? : \textit{value} \mid (\textit{funds} \geq \textit{price}?) \wedge (\textit{funds}' = \textit{funds} - \textit{price}?)]
\end{array}
$$

Finally the interaction between the two customers and the shop is specified in the *System* class, which includes a *jewellers* object together with objects *peter* and *paul*. In a similar manner to the Jewellers class, we describe each operation in terms of how it effects the component objects. For all but one operation this involves simply promoting each operation to this class. The remaining operation describes how *pieceOut* changes (simultaneously) the jewellers shop and the customer. Note that this is the only case of parallel composition in the specification where matching between input variables (*wei*? and *price*?) and output variables (*wei*! and *price*!) takes place.

The disjunction ([] is the choice operator which behaves as disjunction here) of *peter.pieceOut* and *paul.pieceOut* in the definition of *pieceOut* in this class represents the fact that a piece is sold by the Jewellers to either Peter or Paul, thus *jewellers.pieceOut* synchronises with either *peter.pieceOut* or *paul.pieceOut*.

System

$jewellers : Jewellers, peter : Peter, paul : Paul$

INIT

$jewellers.init,$
$peter.init,$
$paul.init$

$openShop \mathrel{\widehat=} jewellers.openShop$
$closeShop \mathrel{\widehat=} jewellers.closeShop$
$cashUnlock \mathrel{\widehat=} jewellers.cashUnlock$
$cashLock \mathrel{\widehat=} jewellers.cashLock$
$pieceIn \mathrel{\widehat=} jewellers.pieceIn$
$pieceOut \mathrel{\widehat=} jewellers.pieceOut \parallel (peter.pieceOut [] paul.pieceOut)$
$pieceToWindow \mathrel{\widehat=} jewellers.pieceToWindow$
$pieceToSafe \mathrel{\widehat=} jewellers.pieceToSafe$
$matValPair \mathrel{\widehat=} jewellers.matValPair$
$toTheatre \mathrel{\widehat=} jewellers.toTheatre$
$toDisco \mathrel{\widehat=} jewellers.toDisco$
$reset \mathrel{\widehat=} peter.reset$
$refill \mathrel{\widehat=} paul.refill$

2.2 Assessment

The specification we have given above illustrates that Object-Z supports a constraint-oriented specification style: constraints are modelled as objects, and their cooperation is described by composing (which implies synchronising) their individual operations, thus obtaining new operations at a higher specification level. This enables one to separate concerns in a clean way. The description of temporal ordering of events was specified separately in *Mike*, *Mary* and *Jane*, and the data manipulation was contained elsewhere, in *ValTable, Cash* and *SafeWindow*. The constraints were then easily composed in the top level classes *Jewellers* and *System*.

The specification, however, can be criticised on the grounds of length and redundancy. The composition of constraints is not really described as a concise composition of entities, but is rather fragmented into a long, flat list of operation definitions, which is redundant. For example, we have to explicitly promote all the non synchronised actions of one entity, and there is no shorthand way of specifying this in Object-Z. Again the temporal composition could also be better expressed in a graphical format (transition graph), or at least in a more compact textual form.

3 Specifying constraint composition by co-expressions

Figure 6 describes the interconnection of objects implicit in the Object-Z specification of the Jewellers system.

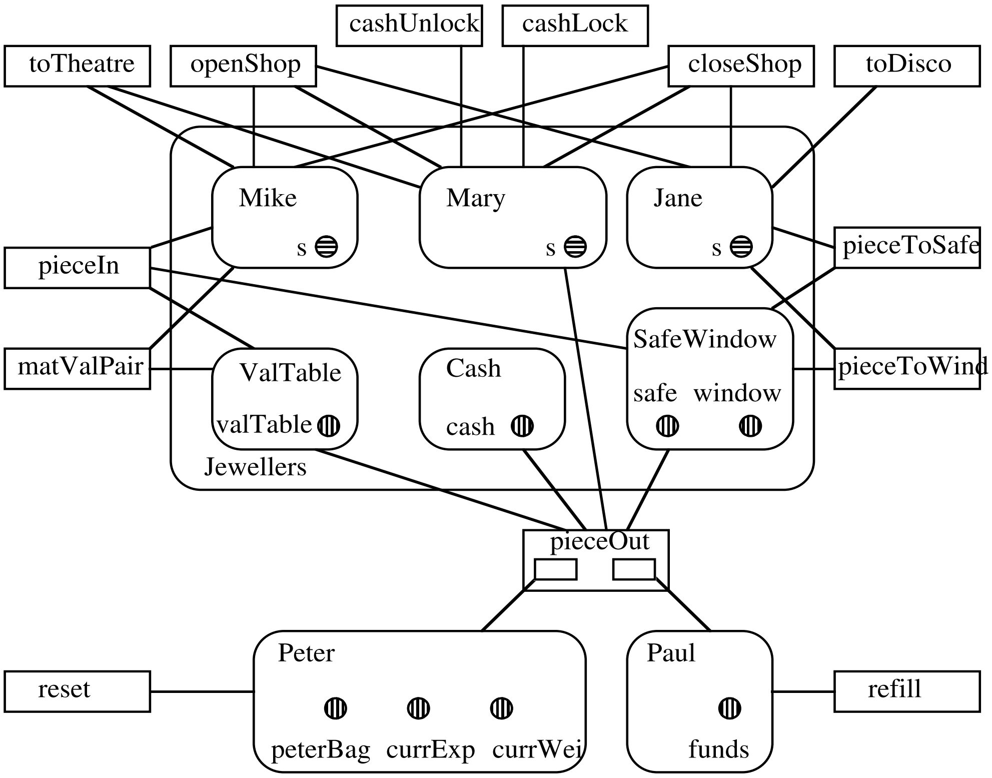

Figure 6 - Interconnected objects in the Jewellers system

Objects are represented as rounded rectangles, with encapsulated state variables and operations depicted, respectively, as circles and boxes. Again, state variables filled with horizontal lines are concerned solely with ordering constraints while those filled with vertical lines represent the actual data items of the system. Since in the Object-Z specification we have preserved names when composing and promoting operations, in Figure 6 we can concisely represent a composite operation just as a labelled box connected to the objects that provide its component operations; this graphical convention is also convenient because it explicitly suggests synchronisation among objects. A special notation is used for operation *pieceOut*. Here, the two component operations contributed by objects *Peter* and *Paul* are combined by disjoint composition (see the class *System* in Section 2.1), thus we have represented them by small, disjoint boxes inside the composite operation box.

A simple textual representation of the Jewellers component in Figure 6 is:

$$
\begin{aligned}
&Mike[openShop, closeShop, pieceIn, matValPair, toTheatre],\\
&Jane[openShop, closeShop, pieceToWindow, pieceToSafe, toDisco],\\
&Mary[openShop, closeShop, cashUnlock, cashLock, pieceOut, toTheatre],\\
&ValTable[pieceIn, matValPair, pieceOut],\\
&Cash[pieceOut],\\
&SafeWindow[pieceIn, pieceToWindow, pieceToSafe, pieceOut]
\end{aligned}
$$

In this expression objects are listed with their respective operations, and are separated by commas. We may think of the comma as a conjunction operator for objects, which achieves composition of all operations with equal names, and their promotion to a higher level operation with that name.

Looking now at the whole interconnection graph in Figure 6, we could write:

$Jewellers[openShop, closeShop, pieceIn, pieceOut, pieceToWindow, pieceToSafe,$
$matValPair, toTheatre, toDisco]$
$,$
$(Peter[pieceOut, reset]$
$;\ Paul[pieceOut, refill]$
$)$

where we still use the comma for expressing the conjunction of the *Jewellers* component with the rest of the system, and introduce the semicolon for describing the disjunction between components *Peter* and *Paul*, which are not expected to synchronise on the operation *pieceOut* that they share.

We believe that the interconnection diagram of Figure 6, and its textual counterpart, are substantially more clear and intuitive than the lengthy definitions of class schemata *Jewellers* and *System* in Section 2.1. The system is conceived as a composition of objects, one describing the shop and two describing different customers; the shop is in turn conceived as a composition of three temporal ordering constraints and three data handling constraints (recall that the terms 'object' and 'constraint' correspond to slightly different ways of looking at, basically, the same thing). A specification is more readable if this conceptual structure is directly reflected in its syntax, via the graphical or textual forms just introduced.

Therefore we propose to extend Object-Z by including the textual form for object composition, which we call 'co-expression'. Below we show how we can directly plug the two co-expressions above into the definitions of, respectively, classes *Jewellers* and *System* (the other class definitions of the specification are left unchanged).

Jewellers

$mike : Mike,$
$jane : Jane,$
$mary : Mary,$
$cash : Cash,$
$safeWindow : SafeWindow,$
$valTable : ValTable$

INIT

$mike.init \wedge jane.init \wedge mary.init \wedge cash.init \wedge safeWindow.init \wedge valTable.init$

$\langle mike[openShop, closeShop, pieceIn, matValPair, toTheatre],$
$jane[openShop, closeShop, pieceToWindow, pieceToSafe, toDisco],$
$mary[openShop, closeShop, cashUnlock, cashLock, pieceOut, toTheatre],$
$valTable[pieceIn, matValPair, pieceOut],$
$cash[pieceOut],$
$safeWindow[pieceIn, pieceToWindow, pieceToSafe, pieceOut]\rangle$

$$
\begin{array}{|l}
\hline
\textit{System} \\
\begin{array}{|l}
\hline
jewellers : Jewellers, peter : Peter, paul : Paul \\
\hline
\end{array} \\
\begin{array}{|l}
\hline
\textit{INIT} \\
jewellers.init, \\
peter.init, \\
paul.init \\
\hline
\end{array} \\
\langle jewellers \\
[openShop, closeShop, pieceIn, pieceOut, pieceToWindow, pieceToSafe, \\
matValPair, toTheatre, toDisco] \\
, \\
(peter[pieceOut, reset] \\
;\ paul[pieceOut, refill]) \\
\rangle \\
\hline
\end{array}
$$

Co-expressions appear in angle brackets, and are shorthands for the long lists of operation definitions found in the original Object-Z text. In co-expressions objects appear with their full list of operation names. We are not changing the semantics of the language, but simply offering convenient syntax that matches constraint-oriented decomposition. The fully general form of co-expression and its expansion into Object-Z are formally defined in the next section.

4 Formal expansion of co-expressions

In this section we introduce the general form of co-expression, and show how it is translated into Object-Z. The general form of a co-expression is given by the following abstract syntax:

$$
\begin{array}{lll}
co\_expression & ::= & < co\_exp > \\
co\_exp & ::= & item \mid (co\_exp) \mid co\_exp, co\_exp \mid co\_exp;\ co\_exp \\
item & ::= & obj\_item \mid op\_item \\
obj\_item & ::= & obj\_id[arg\_list] \\
arg\_list & ::= & arg \mid arg, arg\_list \\
arg & ::= & op\_id \mid op\_id / op\_id \\
op\_item & ::= & op\_id \mathrel{\widehat{=}} schema
\end{array}
$$

From the abstract syntax we can see that a co-expression is a list of items in angle brackets, where items are either separated by commas or semi-colons. The comma denotes conjunction and yields synchronisation of operations; the semicolon represents disjunction and yields operation interleaving. Co-expressions can also be bracketed if necessary to enforce a particular expansion.

Items are either object items (obj_id) or operation items (op_item), and each of these describe the operations that are synchronised by the co-expression. The simplest form of an item is an operation item, which has the form $op\_id \mathrel{\widehat{=}} schema$, and this simply introduces an operation name and its definition as a schema in its horizontal form. There is no requirement on the name being unique: indeed such an operation definition allows an additional constraint to be placed on an existing operation. We shall see an example of this below.

An object item introduces the name of an existing object (obj_id), that is, one declared in the state schema, together with a list of operations (the arg_list). An operation in the list is either an operation name, or a substitution of one name for another.

The co-expression describes synchronisation between these operations and those defined by the *op_items*. In order to illustrate how this is achieved we consider an abstract example of using a co-expression and its translation into Object-Z.

In the example below four component objects (M, N, P and Q) are specified. Object M contains specifications of operations $opM1$, $opM2$ and c; N contains specifications of operations $opN1$, $opN2$ and c, et cetera. The class *Main* includes a state schema where four objects of classes M, N, P and Q are declared, and a co-expression which makes use of these objects.

M

$\ldots$

INIT

$\ldots$

$opM1 \mathrel{\widehat=} \ldots$
$opM2 \mathrel{\widehat=} \ldots$
$c \mathrel{\widehat=} \ldots$

N

$\ldots$

INIT

$\ldots$

$opN1 \mathrel{\widehat=} \ldots$
$opN2 \mathrel{\widehat=} \ldots$
$c \mathrel{\widehat=} \ldots$

P

$\ldots$

Q

$\ldots$

$Main$

$m : M, n : N, p : P, q : Q, s : \mathbb{N}$

INIT

$\ldots$

$\langle m[opM1, b/opM2, c],$
$\ n[b/opN1, opN2, c],$
$\ (p[opP1, b/opP2];$
$\ \ q[opQ1, b/opQ2]),$
$\ b \mathrel{\widehat=} [\Delta(s) \mid Pred(s, s')]$
$\rangle$

op_list

The class may also contain additional operations, specified as usual; these are simply denoted *op_list* in the above example.

The co-expression represents a list of operation definitions, where after translation into pure Object-Z, each operation will be a promotion of an operation from a component object or of several synchronised operations. When translating an arbitrary co-expression into a full description of its behaviour in Object-Z there are several cases to consider. These are expanded according to the following rules:

- operations which are mentioned in only one *obj_item* are simply promoted from the component object to the class containing the co-expression. For example, $opM1$ occurs in only the first *obj_item* in the above co-expression, thus the expanded form contains the operation definition $opM1 \mathrel{\widehat=} m.opM1$, and similarly for $opN2$, $opP1$ and $opQ1$.

- operations which are mentioned in more than one *obj_item* are promoted to the class containing the co-expression, and all occurrences of such operations with the same name are synchronised. For example, the operation c occurs in the first two object items, therefore the expanded form contains the operation definition $c \mathrel{\widehat=} m.c \| n.c$.

- operation names can be substituted; this is denoted by, for example, $b / opM2$. This will result in an operation with name b in the expanded form, but will be defined in terms of the original name, i.e. in terms of $m.opM2$.

- operations which are mentioned in more than one *obj_item*, but where the objects are separated by a semi-colon (;) are disjoined upon promotion. For example, the b occurs in the object items p and q which are separated by a semi-colon, therefore the expanded form contains the operation definition $b \mathrel{\widehat=} p.opP2 \lor q.opQ2$.

- an operation defined by an *op_item* (e.g. $b \mathrel{\widehat=} [\Delta(s) \mid Pred(s, s')]$) is used for refining or defining an upper level operation with the same name: the body of the former definition appears literally (e.g. $[\Delta(s) \mid Pred(s, s')]$) in the expression defining the latter in the expanded form. In case *op_item* introduces a new operation name not found elsewhere in the co-expression, the operation is simply added to the list of upper level operations, as if it appeared outside the co-expression, in *op_list*.

Adopting these conventions we can expand the class *Main* given above to its full definition as follows.

Main

$m : M, n : N, p : P, q : Q, s : \mathbb{N}$

INIT

$\ldots$

$$
\begin{aligned}
&opM1 \mathrel{\widehat=} m.opM1 \\
&b \mathrel{\widehat=} m.opM2 \parallel n.opN1 \parallel (p.opP2 \lor q.opQ2) \parallel [\Delta(s) \mid Pred(s, s')] \\
&c \mathrel{\widehat=} m.c \parallel n.c \\
&opN2 \mathrel{\widehat=} n.opN2 \\
&opP1 \mathrel{\widehat=} p.opP1 \\
&opQ1 \mathrel{\widehat=} q.opQ1
\end{aligned}
$$

op_list

With these expansion rules we can see that after translation the compact specifications of classes *Jewellers* and *System* in the previous section do represent the original lengthy descriptions given in Section 2.

5 Conclusions and related work

In this paper we have illustrated a constraint-oriented specification style for Object-Z and a corresponding syntactic extension. The aim of this style is to make object based specification languages such as Object-Z suitable for the incremental description of systems where concerns about the handling of data structures and the temporal ordering of events are intertwined. By using this design technique the usual benefits of object-oriented design decomposition, such as adherence to the objects in the 'real' world, and reusability, are combined with those of constraint-oriented decomposition, such as specification flexibility and adherence to informal, functional descriptions.

We showed how a composition of constraints can be concisely described by a co-expression which closely resembles expressions used in LOTOS and in the co-notation for similar purposes. The use of this co-expression removes much of the redundancy found in a pure Object-Z specification, and allows one to use a more compact textual form that directly refers to the relevant constraints. Finally, we have shown how to translate co-expressions into pure Object-Z.

As mentioned in the introduction a constraint-oriented style is already supported by a number of formal techniques, although few in an object-based setting. The exceptions to this include VDM++ and Z++. VDM++ [6] is a concurrent, real-time and object-oriented extension of VDM-SL, Z++ [5] is similar proposal based upon Z which adds a class construct and mechanisms for specifying concurrency and real-time behaviour via real time logic.

The constraint-oriented specification style is an important methodological style used in the language LOTOS [2, 8, 12, 4]. To specify a system in this style its behaviour is conceived as a composition of constraints on the ordering and values of interaction events. To do so one first identifies the actions involved in the system, and then specifies the system behaviour by composing the constraints (which are represented by processes) using parallel composition.

A different use of constraints is found in the Z notation [11]. A Z specification typically describes a state space together with a collection of operations acting on that state. The state space and operations are presented as schemas, each containing a declaration together with a predicate, and the predicates enable constraints to be specified. For example, a state invariant (i.e. a predicate in the state space schema) constrains the values of the variables in the state space. In a similar way, predicates in an operation definition can be seen as constraints on a systems behaviour. The schema calculus provides a set of schema operations used to structure the presentation of the operations of the system.

However, Z does not offer facilities for placing global constraints on the temporal ordering of events, as the constraints specified in Z appear either as an individual state invariant or on an operation by operation basis. For example, if we compose two schemas A and B by conjunction to define a new operation C

$$C \mathrel{\widehat{=}} A \wedge B$$

the conjunction represents an individual constraint on the single operation C as opposed to a constraint on a number of events. In contrast, the LOTOS parallel composition operator supports the specification of increasingly complex constraints involving an increasing number of events.

One pleasing aspect of constraints in Z is that synchronisation does not require identical action structure. For example, operation $jane.pieceToWindow$ has no inputs/outputs whilst $safe.pieceToWindow$ does have an input. Forming the parallel composition of these two operations is well-formed in Object-Z, whereas in LOTOS successful interaction would require identical action structure in each of its components.

The co-notation [1] can be seen as an attempt to combine the constraint-oriented specification styles of both Z and LOTOS and is meant to offer facilities for handling, on one hand, actions, interaction, the ordering of their occurrences, and the composition of constraints on temporal patterns of events (as offered by LOTOS), and, on the other, state variables and the composition of constraints on their values (as offered by Z).

For example, in co-notation, the co-expression:

$$P[a, b](s, t), Q[b, c](s), R[c](t)$$

denotes the composition of three constraints, each insisting on some actions (in square brackets) and on some state variables (in round parentheses). Constraints P and Q share action b, and synchronise on it; similarly, constraints Q and R synchronise on action c. This is analogous to LOTOS process composition. Furthermore P and Q share state variable s, while P and R share state variable t. This is analogous to Z schema composition. Compound constraints are defined in terms of other constraints, and may encapsulate internal state variables: they are thus similar to objects, with actions playing the part of operations.

Although the constraint-oriented extension to Object-Z presented here has been largely inspired by the co-notation, some differences are to be reported: constraint composition in extended Object-Z is less powerful, since it only addresses shared actions, not shared variables; however, synchronisations in this language are more flexible, because they do not require identical action structures as in co-notation. Apart from these differences, the upper levels of a specification in co-notation and in extended Object-Z are similar, in that they express object/constraint compositions. But differences emerge when considering the leaves of the specification. In co-notation we find *active predicates*, and these are quite different from the objects we find in extended Object-Z: they do not encapsulate state variables, and must refer to precisely one action (operation).

There have also been a number of proposals which aim to integrate state based languages and process algebras. For example, in [9, 10] a detailed methodology is given for the use of Object-Z together with CSP to describe and refine complex systems. The methodology involves using Object-Z to specify the component processes, with all the interaction of system components being specified in CSP. Therefore a restricted subset of Object-Z is used which does not include instantiation of objects of a class. It also doesn't include polymorphism, class union or object containment which are only used in the context of object instantiation, and the parallel $||$ and enrichment $\bullet$ operators which are used to model object interaction.

A specification of the Jewellers shop might be thus described in this fashion as

$$Mike \mid\mid Jane \mid\mid Mary \mid\mid Cash \mid\mid SafeWindow \mid\mid ValTable$$

where *Mike* etc are the Object-Z classes given in Section 2, and $||$ is the CSP parallel composition which is being used to define the overall temporal pattern of the operations in this specification. The use of constraints here can be seen to be similar to the use in LOTOS discussed above, and does enable a reduction of redundancy in operation definition similar to that we achieved here by using co-expressions within classes. However, like in LOTOS, well formed synchronisation requires identical action structure in all the components. The classes *Mike* etc would have to be modified (possibly by using inheritance) before composition. This, we believe, is in contrast with the separation of concerns, since one is forced to consider the structure of all methods in all components simultaneously. In particular the separation into temporal and data handling concerns is now lost.

An additional degree of flexibility provided by the co-expressions used in this paper is that they can be defined on only some of the component objects, and that no constraints are made on the initial states of the component objects. This allows, for example, the initial state of the Jewellers shop to be different from just the conjunction of the initial states of its components. This is not possible in the method described in [9].

Further work on the use of co-expressions in Object-Z can be envisaged in a number of areas. For example, we have described a specification methodology here, and not a development methodology. This could be rectified by adding and integrating a notion of refinement for co-expressions which was sound with respect to refinement in Object-Z.

References

[1] T. Bolognesi. Expressive flexibility of a constraint-oriented notation. *The Computer Journal*, 40(5), 1997.

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

[3] 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.

[4] E. Brinksma (ed.). Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour, 1989. IS 8807.

[5] K. C. Lano. Z^{++}, an object-orientated extension to Z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 151–172. Springer-Verlag, 1991.

[6] K. C. Lano and S. Goldsack. Integrated formal and object-oriented methods: The VDM^{++} approach. In T. Bryant and L. Semmens, editors, *Methods Integration*, Electronic Workshops in Computing. Springer-Verlag, 1996.

[7] B. Meyer. Reusability - the case for object-oriented design. *IEEE Software*, 4(2):50–64, 1987.

[8] J. Quemada and A. Azcorra. A constraint oriented specification of al's node. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 83–88. North-Holland, 1989.

[9] 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.

[10] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302, Hiroshima, Japan, November 1997. IEEE Computer Society.

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

[12] C. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification, VIII*, pages 189–204, Atlantic City, USA, June 1988. North-Holland.