Applying formal object-oriented design principles to Smalltalk—80

J P Gibson and J A Lynch

Object-oriented design comprises a set of principles that can be useful when structuring and implementing systems, particularly in software. Although opinions differ on the details it is widely accepted that the techniques may soon offer a sound basis for understanding complex design and programming problems. A brief introduction and suggestions for further reading are given.

This paper attempts to show how a powerful commercial software environment (Smalltalk-80)™, designed to support object-oriented programming, can be used as suggested by recent research into fundamental aspects of object-oriented design undertaken in British Telecom. The research established that any specification language satisfying some simple criteria will allow a relation between objects (components of a design) that captures some properties of inheritance (an important concept in object-oriented design which is explained below); here the relation is called refinement. It is shown that when simple programming controls are enforced objects in Smalltalk-80 exhibit similar inheritance relationships to those formally described by a definition of refinement in the formal language CSP. This is shown to be helpful for system development and understanding reuse of objects and classes. The refinement relationship is explained intuitively and the necessary controls are prescribed.

Brief introductions to CSP and Smalltalk-80 are included.

1. Introduction

Interest in the field of object-oriented design has grown dramatically over the last few years, particularly in the programming community. The technique involves decomposition of systems into manageable components whose interaction with the rest of the system is well defined. This modularity is usually combined with abstraction (the ability to focus on pertinent issues without worrying about irrelevant detail) and inheritance (allowing features defined in one place to be used elsewhere). Different languages may employ these features in different ways, the merits of which are often debated [1,2,3]. Therefore there is, as yet, no widely accepted generic theory of objects. A more detailed summary of object languages is given in Annex 1 and a wider discussion of their use is found in BYTE [4].

The work described by Cusack [5,6] was the first stage in a project aimed at developing a formal definition of object-oriented design that could be applied to the ISO (International Organisation for Standardisation) standard formal description language LOTOS [7] (CSP is similar to basic LOTOS and was used as an intellectual tool because it is more mature and better understood). This showed that any specification language satisfying certain simple criteria has some useful properties associated with object-oriented programming languages. However, these languages are fundamentally different to object-oriented programming languages.When used, they are at different stages of the software life-cycle or often in conflicting regimes of software production. The work leading to this report aimed to:

- exercise the principles of OOCSP (see Section 1.1) and see how well they could be applied to an established object-oriented programming language,
- identify areas where the model could be brought closer to that of other object-oriented languages.

An improved formal model has now been developed and is currently being applied to LOTOS. LOTOS comprises two component languages; the process algebra (basic LOTOS) is similar to CSP but the data-typing (ACT-ONE) element needs more research.

1.1 CSP

Communicating sequential processes (CSP) is one of a family of formal languages known as process algebras whose full definition is given by Hoare [8]. In CSP systems are specified by constraining behaviour (events) to occur in certain orders. For any two distinct events one

™ Smalltalk—80 is a trademark of Xerox Corporation.

must occur before the other, events do not overlap. A CSP process is a sentence describing such behaviour; constructs exist for combining several processes into one larger process. In general any CSP process can be viewed as an arbitrary number of interacting (communicating) component processes.

Cusack [5] describes a way of incorporating a formal (mathematical) inheritance model, based on refinement, in CSP. Any CSP process can be an object, so every object is formally defined. Class-subclass and class-instance relationships are indistinguishable, defined by a notion of refinement. Here refinements only remove non-determinism or add behaviour (new events) to an object, the temporal ordering of existing events is unchanged. Use of CSP in this way will be referred to as OOCSP. The research by Cusack [5,6] has now been further developed to differentiate between class-subclass and class-instance relationships. It forms the basis of some work on modelling open distributed processing systems within the international standards organisation, ISO.

OOCSP would be expected to follow a logical hierarchy (rather than implementation) in the sense of Zdonik and Wegner [9].

1.2 Smalltalk-80

Smalltalk was one of the early object-oriented languages and is used in this paper because of its wide acceptance. It was developed in the 1970s and upgraded to its present form in 1980. All references to Smalltalk in this paper refer to Smalltalk-80. Other object-oriented languages are often described in comparison to Smalltalk.

Throughout this paper a knowledge of the class-subclass structure and message passing construct in Smalltalk is assumed. The first three chapters in Robson & Goldberg [10] should provide enough background material if the reader is unfamiliar with the language. Familiarity with Smalltalk syntax is not necessary.

Smalltalk is discussed in some detail in later sections.

2. Objectives and scope

The aim of ths paper is to show that an intuitive definition of the formal refinement relation in OOCSP can be realised in a programming language. An analogous relation, replacement, is introduced for Smalltalk.

First, correspondences are found between fundamentals in the two languages. These are used to reflect the rules for OOCSP refinement in terms of Smalltalk. They are then developed using class-subclass and class-instance relations in Smalltalk until replacement satisfies the intuitive understanding of refinement. Replacement rules take the form of controls on how classes can be modified. Of course Smalltalk was an arbitrary choice; the essentials of refinement could have been investigated for other languages.

Since memory and processing power are becoming more abundant and less expensive it is reasonable to assume that programming in the future will look to enhancing capability rather than tuning and economising. Reuse of existing software is even more attractive in this type of environment. The ideas in OOCSP are directed towards adding behaviour to existing specifications without causing change in the original model. This is done in a controlled way. A discussion of how the concept of replacement could be useful in the building and maintenance of object-oriented systems from CSP specifications is given below.

It is worth noting that CSP and Smalltalk are very different languages. They were designed from different backgrounds for different purposes. This is not an exercise in translation between the two languages. It is an attempt to use the very basic but powerful concepts from OOCSP in a programming language (Smalltalk). Smalltalk implementations of simple CSP specifications are included to show how the ideas may be used in a programming environment.

3. Summary

When certain programming controls are enforced a Smalltalk object can be replaced by another whilst preserving original system behaviour exactly and adding extra behaviour without side effects. This replacement is analogous to refinement in OOCSP but is not (of course) rigorous or formally defined.

Updating and maintaining systems or finding successive stages in an evolutionary design cycle depend upon the ability to enhance existing systems or designs. Replacement allows Smalltalk systems to be enhanced in the sense that:

- an existing object is replaced by one that offers the system additional behaviour;
- a designer's understanding of the original object is kept as the basis for understanding the replacement;
- the additional behaviour is incorporated explicitly in one place;
- there are no unwanted knock-on effects to the original system.

Replacement can also be of practical use when building a system of objects from library components. Components can be taken 'off the shelf' provided they are valid replacements for the required objects.

Applications programming could involve controlling the interaction between objects rather than creating the objects themselves. The classes of which these objects are instances may be created elsewhere by systems programmers.

For example, consider two CSP specifications, S1 refining S2 (see Fig 1). S2 will be more detailed (less abstract) or more deterministic than S1 but when that extra detail is ignored the two are indistinguishable to an observer. By analogy, it is possible to find a replacement (I2) for an implementation of S1 (I1) such that I2 implements S2. Implementations are fully determined[1] but the behaviour of I2 would include all the behaviour of I1.

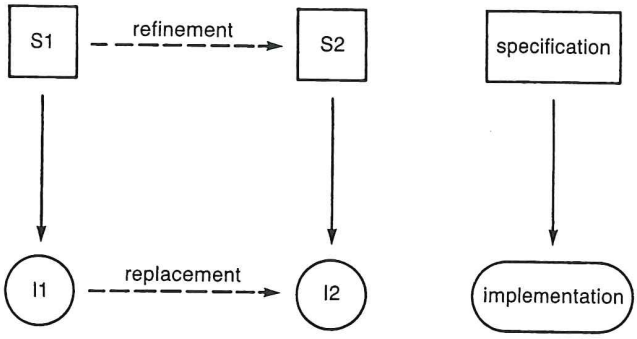

Fig 1 OOCSP refinement and Smalltalk replacement.

The following suggest ways in which replacement may be of practical use:

- If an object is required and replacement of it is available (e.g. from a catalogue) then the replacement can be used immediately. Extra behaviour can be ignored and will not impact on the rest of the system (i.e. I2 can be used wherever I1 can).
- Where an object is required that is a replacement for an existing component it is desirable to evolve the component rather than produce the object from scratch (i.e. I2 can be developed from I1).
- In a design process where a system goes through a number of intermediate prototypes, making replacements for objects in successive stages would help in understanding the increasingly complex system.

Of course programming I1 and I2 separately and from their respective specifications would normally yield very different programs although their external behaviour should exhibit the requirement of the replacement relation (i.e. extended behaviour without side-effects). The rules for replacement in sections 5 and 6 (summarised in section 7) are a list of guidelines a programmer must follow to guarantee a new object replaces the original. This is not the only way to capture the replacement relation and in many cases the rules only tie down informal mental checks a designer would employ anyway. They do not deal specifically with implementing from CSP specifications.

The rules for replacement are defined and explained below. There are found to be two cases, one for instances (object replacement), the other for classes (class replacement) which requires some extra syntactic constraints.

4. Language comparisons

One thing that CSP and Smalltalk have in common is that they are both languages used to describe behaviour.

- "A CSP process is the behaviour pattern of an object, insofar as it can be described in terms of the set of events in its alphabet" [8].
- "Objects are more than just data, more than just containers of information. They are also more than just their message protocols and object visibilities. An object is a complex combination of these things; it is a behaviour" [1].

Therefore, behaviour is probably a good basis from which to build an intutitive relationship between the two languages. Objects can be treated as communicating processes [5,11].

4.1 Features

When trying to capture a common property in any two languages it is useful to begin by finding any similarities between constructs in those languages. Table 1 below shows the parts of Smalltalk and CSP that appear to take similar roles. Details are given in Annex 2; of course the relations are informal and based only on observation.

Table 1 Comparison of Smalltalk and CSP.

CSP		Smalltalk
Process	↔	Object
Event	↔	Message send and reply returned
Alphabet	↔	Message Interface

4.2 Communication

At the level where objects co-exist to form a system language similarities are more tenuous. Both CSP and Smalltalk provide communication mechanisms as fundamental constructs. CSP communication occurs between processes and is used for event synchronisation (i.e. the occurrence of an event common to all processes participating in the communication). Smalltalk com-

[1] During implementation of a non-deterministic specification decisions must inevitably be made that restrict the system to a precise behaviour.

munication is used for message passing between two objects. Annex 3 makes some analogies between the communication mechansims and highlights the distinctions.

5. Object Replacement

5.1 Intuitive definition

The CSP concept of refinement [5] has an intuitive analogy in programming. This corresponding idea will be called replacement. The informal notion of replacement is as follows. An object is needed with some specified behaviour. There is an object available whose specified behaviour is a refinement of the behaviour required. This object can be used as a replacement.

This situation may be common if object-oriented programming is used to greatest effect. Building new applications should involve 'ordering components from catalogues and combining them, rather than re-inventing the wheel every time' [12]. Systems would be built from library components that are valid replacements of the actual components needed. However, it is dependent on components being paired with specifications from which refinements can be found.

Example — A Stack

A programmer requires an object that has the behaviour of a stack. This stack should have methods push and pop with last-in-first-out behaviour.

A different object is available from a library — a counting stack. This object has methods push, pop and count. The additional count operation returns the number of elements on the stack.

CSP specifications for these behaviours are given in Appendix A followed by Smalltalk implementations. Proof that the CSP specification of the counting stack is a refinement of the stack is given. Intuitively, the counting stack implementation should be a valid replacement of the stack.

This example relies on its simplicity to give the feel that a replacement is possible (i.e. the counting stack can always be used in place of the stack). In general it will not be obvious that a replacement relationship exists. The complex problem of deciding when two programs given the same behaviour needs to be avoided. A simple example of the possible difficulties follows.

Example — Complex Numbers

An object that has the behaviour of a complex number is required. It should have methods for addition, subtraction and multiplication. Different representations are available:

- cartesian co-ordinates (x, y)
- polar co-ordinates (r, θ).

Smalltalk implementations of these are given in Appendix B.

In this case the implementations should be valid replacements for each other but this is difficult to verify from the code alone. It is only mathematical knowledge that reassures the user that one can replace the other.

Guidelines are needed to help decide if a replacement is valid. Inheritance can be used to check differences in objects and show if one replaces the other (see section 5.3). Normally a subclass can be found such that it has an instance that is a replacement of an instance of its superclass.

5.2 Rules for refinement

This section states the rules for refinement [5] and deduces some preliminary guidelines for replacement in Smalltalk. Some requirements specific to programming languages are addressed in later sections.

The rules for refinement as defined in Cusack [5] are as follows.

Process Q is said to refine process P precisely when the following hold:

- $\alpha P \subseteq \alpha Q$,
- $tr(Q)\gamma\alpha P \subseteq tr(P)$,
- failure condition,
- divergence condition.

The first rule states that the set of names of events relevant to the description of P (i.e. αP) is a subset of the corresponding set for Q. A trace of a CSP process is a sequence of event names recording the events a process has engaged in up to some moment in time [8]. The second rule states that the set of all possible traces of Q, with those event names irrelevant to the description of P omitted, is a subset of all possible traces of P. The third and fourth rules are not detailed because rule 3 is only relevant for non-deterministic processes (Smalltalk programs are fully determined) and rule 4 relates to diverging CSP processes which are of little use for implementation.

Suppose Smalltalk object A is a valid replacement for object B. The following suggest issues corresponding to the above rules.

- Correspondence between alphabet and message interface was explained earlier. Object A must have methods for all the messages object B has methods for.
- Consider the trace of an object to be a recording of the messages that the object has received. Now there is nothing to check because A is always capable of receiving any message that is part of its interface.
- There is no correspondence for a CSP failure in Smalltalk; the step between specification and implementation involves removing non-determinism.
- Diverging processes are not considered because of their limited use.

One problem is that in CSP an event is uniquely identified by its name. Two events with the same name denote the occurrence of identical events or a single event occurring in two processes at the same time[2]. However when treating a Smalltalk message pass as an 'event' it cannot be assumed that because two methods have the same name they provide the same behaviour. The actual code inside the methods has to be considered. This is a fundamental problem which inheritance can be used to solve.

5.3 Inheritance and replacement

Class-subclass inheritance can be used to implement replacement:

> 'The new class should have the same properties as the old class, together with a few additional ones. Thus, an instance of the new class should be allowed at every place where an instance of the old class is allowed'. P America [13].

Smalltalk is a very flexible programming language. It permits a subclass to override any, or all, methods that are inherited from its superclass. So subclasses can be very different to their superclasses. Here, instances of a subclass should be able to replace an instance of its superclass so this is an undesirable property of the language. When method overriding is forbidden all facilities in a class are unchanged in its subclasses. This is known as strict inheritance.

When strict inheritance is enforced all objects may be valid replacements for instances of their superclasses (see Annex 4). But enforcing strict inheritance in a language may be too severe a control; programmers prefer more freedom. It is obvious that a common ground must be found between the extremes of total flexibility and strict inheritance. Ideally inheritance should be as strict as possible, allowing replacement with minimum modifications made to existing code.

The following shows how the class-subclass replacement relation is compromised when strict inheritance is not enforced. In each case controls are suggested to preserve replacement whilst using a more flexible inheritance model.

First consider how a Smalltalk subclass may differ from its superclass. In general subclasses are used for:

- horizontal extension — altering existing behaviour,
- vertical extension — adding new behaviour,
- specialisation — a combination of the above.

Horizontal extension involves altering the methods inherited from the superclass. It is used purely for code sharing. Although it is commonly found there is some agreement that this is bad programming practice which can lead to complex code. Annex 5 gives more detail. It is suggested that objects with similar behaviour should be made subclasses of the same (abstract) superclass.

Vertical extension involves adding new methods and possibly new state (variables). The first counting stack in Appendix C is a good example. In the same appendix a second example is used to illustrate method overriding. The replacement property is still valid but this is clear only because of the simplicity of the example.

Adding new variables can create other problems since behaviour of most objects is state dependent (state is viewed as the value of all instance variables). It is important that new variables are not used in deciding how the object reacts to any message that was part of the original method set. When methods have to be overridden to take into account the new state variables it is recommended that this is done only to alter the values of these variables and not for testing purposes. The method overriding in Appendix C is of this type. It is beyond the scope of this paper to examine more complex cases.

The final problem to be considered is how new behaviour may affect the state of some other object. This is an important consideration when dealing with systems. In Smalltalk it is possible for the state of an object to be altered indirectly using shared variables, this is undesirable. Recommended practice is to use shared variables to contain only constant information [14]. This may be overkill but it does ensure no problems can arise from their use.

[2] From Hoare [8] processes are just different components of a system and events are occurrences that a designer wishes to record. Naturally, some occurrences are important to several components and should be recorded in more than one place. However, when the components are viewed together each single event must only be recorded once. Synchronisation is therefore the recording of one event in more than one process.

Annex 6 examines the message passing construct in detail to show how the state of an object may be changed. Simple controls on the use of shared variables are given to ensure that the state of an object can only ever be changed by one other object. Therefore, any new behaviour that is added cannot affect other objects in the system.

This section showed that instances of a Smalltalk class can be replaced by instances of its subclass provided certain conditions are satisfied. The conditions are summarised in section 7.

Note that if object A is a valid replacement for object B then some change of symbol may be necessary before the replacement can occur. For example, a stack with operations add and remove may be valid replacement for a stack with operations push and pop. However, before the replacement can take place the obvious change of method name has to occur. This simple syntactic change is taken for granted in these discussions.

6. Class replacement

Programming systems in Smalltalk involves the creation of classes and class hierarchies. Often, when systems are built they are being used and adapted at the same time. The ability to change behaviour or add new behaviour without disrupting the overall system is important. By using the class structure it is hoped that change can be controlled through replacement of classes.

Again class replacement tackles only the problem of adding new behaviour. In a class system it might be necessary for the old behaviour to be available even though the new behaviour has been added. In this case, any instance of the old class must have an object replacement as a possible instance of the new class. However, this is not the only requirement for valid class replacement.

In Smalltalk classes are themselves objects. It was hoped that class replacement would be the same as object replacement but this is not the case. Smalltalk inheritance produces dependencies between classes that prevent true encapsulation [15].

Behaviour can be added to the system by replacing class B with class C say (Fig 2). However, existing behaviour is not to be altered, only new behaviour added. It is necessary to consider effects on:

- instances of B,
- superclasses of B,
- subclasses of B.

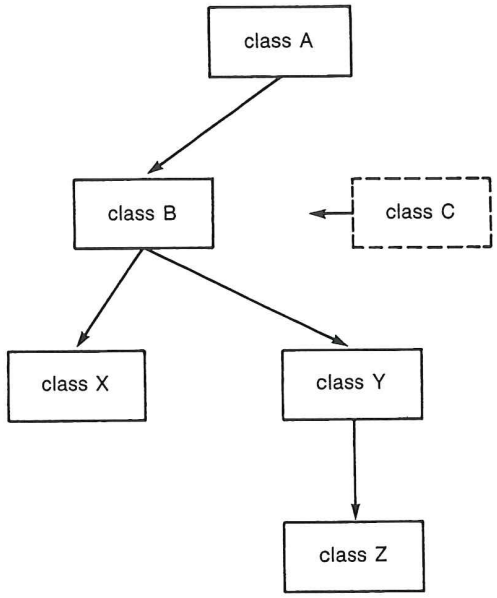

Fig 2 A typical branch of the Smalltalk class hierarchy tree.

Conditions to ensure these are only changed in the correct, controlled way are given below.

6.1 *Instance condition*

The first requirement is that instances of B must have a corresponding valid object replacement as a possible instance of C. Additional behaviour of an instance of C is not important because reference to those internal details will not be made.

Problems arise from the way variables and methods are inherited from the superclasses. If class C introduces new variables or methods then it should not conflict with names in classes A, X, Y and Z. However, with some simple syntactic constraints a valid replacement should be possible provided the instance condition holds.

6.2 *Syntactic constraints*

Earlier it was mentioned that a change of symbol for method names may be required in making an object replacement. This is also the case for class replacement. New methods introduced can be ignored provided they do not conflict with method names in the old class. This is because:

- in the superclass (A) there is no reference to the methods in C,
- in the subclasses (X,Y,Z) new methods introduced by C will either be overridden or never used[3].

Subclasses inherit state as well as methods so syntactic checks are necessary on new variables introduced by class C.

- Class C should not introduce new class, instance or temporary variables that conflict with those inherited from class A.
- Class C should not introduce new class or instance variables which conflict with class, instance or temporary variables in subclasses X, Y, Z.

In a large class sytem all these syntactic constraints may be difficult to check. It is unfortunate that Smalltalk cannot cope with these problems at source rather than simply producing errors. There has been work done on a language, Common Objects [16], that combines inheritance with true encapsulation. In this language none of these syntactic constraints would be necessary.

The OOCSP work with refinement also had to introduce similar constraints for the configuration problem. In this case all the processes in the system had to be checked for conflicts. The Smalltalk problem is not as restricting as this; only classes in the branch of the inheritance tree containing the replacement class need to be checked.

7. Summary of replacement

Sections 5 and 6 described how object and class replacements can be produced in Smalltalk using programming controls. By analogy with refinements from Cusack [5], replacements preserve original behaviour and add new functionality without side-effects. Class replacement reduces to object replacement with a few syntactic constraints. The following is a summary of the controls.

Methods in object replacement

Using the inheritance mechanism all methods can be taken from a superclass into a subclass. Unless they are overridden they work in exactly the same way. Thus an instance of a subclass may replace an instance of its superclass provided:

- new methods that are added do not affect the original state variables (instance and temporary),
- when methods are over-ridden original behaviour is preserved, (e.g. only add lines that manipulate new state variables, etc).

Variables in object replacement

State is viewed as the value of all instance and temporary variables. Notice that using new methods that manipulate only new state variables does not guarantee no side-effects:

- shared variables should be used only to represent constant information,
- new variables should not be used to decide how the replacement reacts to messages in the original message interface.

Class replacement

Possible syntactic conflicts are limited to classes in the same branch of the hierarchy:

- each instance of the original class should have an object replacement of itself as a possible instance of the replacing class,
- new class, instance or temporary variables should not conflict with any from the superclass,
- new class or instance variables should not conflict with any in subclasses.

8. Conclusions and limitations

It has been shown that when certain simple constraints hold, a Smalltalk-80 object can replace another whilst preserving original behaviour, adding extra behaviour and avoiding undesirable side effects. Possible implications of this for system development and maintenance were discussed in section 3.

The concept of 'replacement' applies one definition of refinement for CSP processes [5]) to Smalltalk objects at an intuitive level. It depends on a number of programming controls (summarised in section 7). Replacements can be found for classes and instances; class replacement being subject to syntactic contraints unnecessary for object (instance) replacement. This is in keeping with Cusack [5] where configuration involves syntactic constraints on refinement. An object's replacement can always be used in its place. Conversely, extra behaviour can be incorporated by producing a replacement from that object.

[3] Smalltalk does provide a means for marking methods 'private'. This is not enforced and these 'internal' methods can be accessed from outside the object. If the private methods were fully protected in the inheritance mechanism then private methods could be ignored when considering possible syntactic conflicts.

It would be possible for the controls and syntactic constraints to be enforced as part of a programming language. Programming would then only be concerned with behavioural dependencies when considering refinement of objects. It is important to remember that Smalltalk is not a formal language in the sense that CSP is and cannot be subject to mathematical proofs. Replacement can only be demonstrated to hold in certain circumstances, not proven for all cases.

Some correspondence has been found between the fundamental constructs of Smalltalk and CSP. This was restricted to sequential systems although both CSP and Smalltalk can model concurrency. It is not clear how concurrency would affect encapsulation and inheritance properties of implementations. This needs investigation before replacement can be generalised for other languages.

Refinement controls addition of behaviour in one particular branch of the class hierarchy. Further work will be necessary to address changes to behaviour of different branches. For example, at the moment it is still possible to duplicate extra behaviour in classes from different parts of the system. This is part of the distinction between adding only extra behaviour in a single place and changing the system as a whole.

Another limitation arises when the joint behaviour of several Smalltalk objects is considered. In CSP several parallel processes can be viewed as one process so refinement applies directly. This is not the case for several interacting Smalltalk objects and replacement. Further investigation may be related to multiple inheritance.

Annexes

Annex 1 — Properties of objects and object-oriented languages

An object is a well-defined data structure coupled with a set of operations that describe how that data can be manipulated. It is a 'behaviour' protected from external manipulation by forcing all interaction with an object to be controlled by the object itself.

For a language to be considered object-oriented it must provide ways to create new objects and allow these objects to communicate with each other. Pure object-oriented programming languages usually contain the following (in some form):

- Modularity — No object should depend on the internal details of another object. In its purest form this is known as encapsulation — all access to the internal code/data of an object must be through the external interface of that object. Data abstraction and information hiding are forms of modularity.
- Inheritance — This is the ability to create an object with a new behaviour that is an extension of the behaviour of an already existing object. The class structure in Smalltalk provides this behaviour extension characteristic.
- Dynamic binding — allows the extension mentioned earlier without the need to modify existing code.

Other facilities are often associated with object-oriented languages (e.g. graphical workstation and interface) but are not pre-requisites. They usually derive from the culture that founded object-oriented programming and provide a user interface but are not basic and could be used with other paradigms.

Annex 2 — Basic correspondences between CSP and Smalltalk

Both CSP processes and Smalltalk objects are dynamic entities which may be history dependent (have state). CSP processes are constructed as ordered sequences of events. A CSP event is 'an action of interest', an occurrence of which should be regarded as 'an instantaneous or atomic action without duration' [8].

In Smalltalk all computations are achieved through message passing so a message send is obviously an action of interest. When one object sends another a message, flow of control passes to the receiver. The receiver then performs some internal computations (unseen by the sender) and passes an object back to the sender along with control. By ignoring the internal computations the message-send/object-returned pair can be treated as an atomic event. As explained earlier, this is not a rigorous translation exercise. However, provided a message-send/object-returned pair does not overlap with external messages it is a reasonable analogy.

Another fundamental in CSP is the alphabet of a process. 'The set of events which are considered relevant for a particular description of an object is called its alphabet. It is a permanent predefined property of an object' [8]. Alphabets in CSP are best reflected in Smalltalk by the message interface of an object.

Annex 3 — Comparison of communication mechanisms

In CSP communication between processes occurs when they synchronise and participate in the same event. Any number of processes may be involved. Smalltalk communication occurs when one object sends another a message. This communication is between the sender and the receiver. Therefore, in Smalltalk an 'event' occurs between exactly two objects with one of these in control — the sender. However, a CSP process can engage in events independently from any other process and when processes run in parallel there is no concept of one being

in control. These properties are different from those seen in the programming language. However a process is always running in parallel with its environment which itself is a process and may be thought of as being in control.

Annex 4 — Strict inheritance

It should be obvious that the use of strict inheritance will force an instance of a subclass to be a valid replacement for an instance of its superclass since:

- all methods in the superclass are in the subclass,
- the methods are unchanged because method overriding is forbidden.

The stack/counting stack example has been re-written in appendix C to illustrate strict inheritance. In this case the counting stack is implemented as a subclass of the stack. There is no method overriding so the counting stack is a valid replacement for a stack.

Annex 5 — Horizontal extension

Horizontal extension will involve altering the methods inherited from the superclass. An example of this is an object that counts. It can be intialised to zero and will add 1 to its count state when it receives the message increment. An extension to this may be to have a similar object which gets initialised when the count reaches a predefined limit.

This new class could be a subclass of the original counter. However, no new methods or state have to be added. In this case inheritance is used for code sharing rather than behaviour sharing. It should be obvious that the limited counter is not a valid replacement for the normal counter.

The following are sketched CSP specifications of the above behaviours.

```
(A) counter(x)      = increment→(count!x+1
                                → counter(x+1))
(B) count-to-ten(x) = if x<10→increment
                             →(count!x+1
                             →count-to-ten (x+1))
                      else →count!0
                           →count-to-ten(0)
```

When the alphabets of each process are only those events that can occur (B) is not a refinement of (A) since the alphabet of the counter is not a subset of the alphabet of the count-to-ten. The Smalltalk implementations of these specifications are given in appendix D. As expected the second implementation is not a valid replacement for the first.

Annex 6 — Smalltalk message passing

The object in Fig 3 has three instance variables X, Y, Z (themselves objects). It also has four methods in its method set, one of which has a temporary variable associated with it.

Constraints are required to enEach object in Smalltalk has a number of operations (methods) which it can perform. When an object receives a message it either has a corresponding method and performs a task or it returns a 'message not understood' signal. The message interface of an object is the set of messages for which that object has a corresponding method.

In CSP a process can only participate in an event which is in its alphabet. Similarly, in Smalltalk an object can receive a message and return an object in reply only if that message is 'accepted' by the external interface. Notice that an event can occur simultaneously in two or more CSP processes but messages are sent by a Smalltalk object before they can be 'not understood'. Thus this is not a strong analogy.

sure a message send will affect only the state of the receiving object (i.e. any instance variables changed are only part of the state of the receiver). It can then be assumed that the state of an object in a system can be altered by only one other object.

Object A can send messages only from inside one of its methods. This message can be to:

- an instance variable (X,Y or Z),
- a temporary variable (if the method has one),
- a parameter object that was passed in with the message that invoked this method,
- a shared variable (global, class or pool).

It is undesirable for A to be able to change the state of any objects other than itself. It is able to change the state of its instance variables (X,Y,Z) but these are part of its own state. Changing the state of a temporary variable does not matter because the variable ceases to exist after the method is executed.

The following controls on the use of parameters and shared variables are required.

- Parameters should be used to carry information into a method. The state of a parameter should not be changed inside a method.

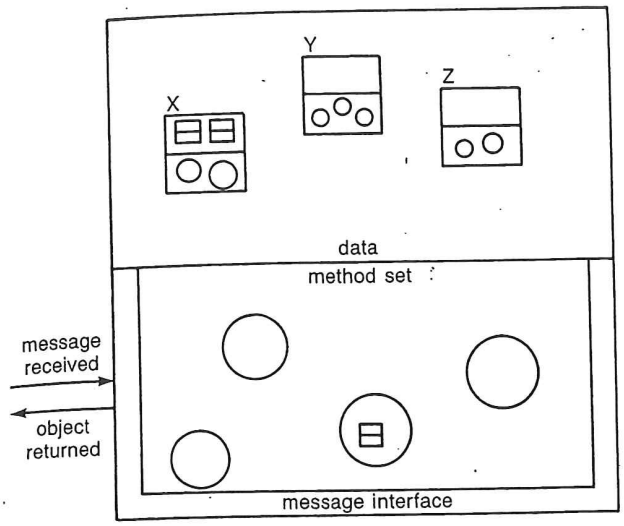

Fig 3 Representation of a typical Smalltalk object.

- Shared variables should be used to contain constant information for access by many classes. This information could be data or methods (classes are global variables used to provide methods for creation of instances).

By following these controls a message send can be seen as analogous to an event occurring in a single CSP process because the state of any other object in the system is not affected.

Appendix A

1 CSP stack

```
NewStack = in?x:data→(Stack(x);NewStack)
Stack(x) = in?y:data→(Stack(y);Stack(x) )
         | out!x→SKIP
```

α Newstack = {in, out}. Stack(x) is a specification of LIFO behaviour and all access to the internal details is controlled through in and out.

2 CSP counting stack

```
CountingStack = Counter(0) | | NewStack
Counter(n) = size!n→counter(n)
           | in?d:data→counter(n+1)
           | out!d:data→counter(n−1)
```

α CountingStack = {in, out, size}. CountingStack is a refinement of NewStack since

- α Newstack $\subseteq \alpha$ CountingStack.
- Without the size events the behaviours are identical.
- There are no refusals.
- There are no divergences

Note: Synchronisation between the counter and the newstack ensures that the 'counter state' (n) is always updated before another item is pushed on or popped off. Also, any number of 'size events' can occur between stack operations in the counting stack.

3 Smalltalk stack

```
class       :  Stack
superclass  :  OrderedCollection

instance methods
  pop
    ↑ super removeLast
  push: newObject
    ↑ super addLast: newObject

class methods
  new: anInteger
  "An initial value for the size of the stack must be given.
  The stack will grow if this value is exceeded.          "

    ↑(super new: anInteger ) setIndices
```

4 Smalltalk counting stack

Counting Stack Implementation

```
class              :  CountingStack
superclass         :  OrderedCollection
instance variables :  count

instance methods
  count
    ↑count
  push: newObject
    count←count+1.
    ↑super addLast: newObject
  pop
    count←count−1.
    ↑super removeLast

private methods
  initialise
    count←0

class methods
  new: anInteger
  |temp|
  temp←(super new: anInteger) setIndices.
  ↑temp initialise
```

Note: The methods which are inherited from the class Ordered Collection are not of interest here. The class methods are given only for completeness.

Appendix B

Complex numbers

1 Cartesian co-ordinates

```
class:                  Complex
superclass:             Number
instance variables:     real, imag

instance methods

  real: realVal imag: imagVal
    real←realVal.
    imag←imagVal

  real
    ↑real

  imag
    ↑imag

  + aComplex
  ↑ Complex new
    real: real + aComplex real
    imag: imag + aComplex imag

  — aComplex
    ↑Complex new
     real: real – aComplex real
     imag: imag – aComplex imag

  * aComplex
    ↑ Complex new
       real: (real * aComplex real)
             – (imag * aComplex imag)
       imag: (real * aComplex imag)
             – (imag * aComplex real)

class methods

"These are not important in the example"
```

2 Polar co-ordinates

```
class                   Complex
superclass              Number
instance variables      r, theta

instance methods

  real: realVal imag: imagVal
    self
      r: (rFromReal: realVal imag: imagVal)
      theta: (theta FromReal: realVal imag: imagVal)

  real
    ↑ r*(theta cos)

  imag
    ↑ r*(theta sin)

  + aComplex
    |realsum imagsum|
    realsum ← self real + aComplex real.
    imagsum ← self imag + aComplex imag.
    ↑ Complex new
        real: realsum
        imag: imagsum

  – aComplex
    |realsum imagsum|
    realsum ← self real – aComplex real.
    imagsum ← self imag – aComplex imag.
    ↑ Complex new
        real: realsum
imag: imagsum

  * aComplex
    ↑ Complex new
        r:r* aComplex r
        theta: theta + aComplex theta

private methods

  rFromReal: realVal imag: imagVal
    ↑ ((realVal ^ 2) + (imagVal ^ 2)) sqrt

    thetaFromReal: realVal imag: imagVal
      ↑(imagVal/realVal) arctan

    r: rVal theta: thetaVal
      r ← rVal.
      theta ← thetaVal
```

Appendix C Inheritance

1 Strict inheritance

```
class:          CountingStack
superclass:     Stack
instance methods

    count
    ↑super size
```

2 Non-strict inheritance

```
class:                  CountingStack
superclass:             Stack
instance variables:     count
instance methods

    count
      ↑ count
```

```
  push: newObject
    count← count + 1.
    super push: newObject

  pop
    count← count −1.
    super pop

private methods

  initialise
      count← 0

class methods

  new: anInteger
  |temp|
  temp← (super new: anInteger) setIndices.
  ↑ temp initialise
```

Appendix D Counter (code sharing)

1 Unlimited counter

```
class                Counter
superclass           Object
instance variables x

instance methods

  increment
    self output.
    x←x+1

private methods

  output
    "some method to output the count value"

  initialise
    x←0

class methods

  new
    (super new) initialise
```

2 Counter up to ten

```
class                CounterToTen
superclass           Counter

instance methods

  increment
  x<10  IfTrue: [super increment]
        IfFalse: [x←0. self output].

"All other methods are inherited from the superclass
counter"
```

References

1 Meyrowitz N: 'Object-oriented programming systems, languages and applications', ACM Press, Special issue of Sigplan notices, 22 , No 12 (October 1987).

2 Cointe P, Bezivin, J Hullot J M and Lieberman H', 'ECOOP'87 European Conference on Object-Oriented Programming (1987).

3 Nygard N and Gjessing S: 'ECCOP'88 European Conference on Object-Oriented Programming (1988).

4 Special issue on Smalltalk, BYTE 16 , No 8 (August 1981).

5 Cusack E: 'Formal object-oriented specification of distributed systems. In Specification and verification of concurrent systems', University of Stirling (July 1988).

6 Cusack E: 'Fundamental aspects of object-oriented specification', Br Telecom Technol J, 6 , No 3 (July 1988).

7 Information processing systems — open systems interconnection — LOTOS — A formal description technique based on the temporal ordering of observable behaviour. International Organisation for Standardisation IS—8807.

8 Hoare C A R: 'Communicating sequential processes', Prentice-Hall (1985).

9 Zdonik S B and Wegner P: 'Type similarity, inheritance and evolution, or what 'like' is and isn't like', Technical report, Brown University (1987).

10 Robson D and Goldberg A: 'Smalltalk-80: the language and its implementation', Addison-Wesley (1985).

11 Tokoro M and Yokote Y: 'Concurrent programming in concurrent Smalltalk.

12 Meyer B: 'Reusability: the case for object-oriented design', IEEE Software (March 1987).

13 America P: 'Object-oriented programming: a theoretician's introduction', Philips Research Laboratories, Eindhoven.

14 Rochet R: 'In search of good Smalltalk programming style', Technical report, CR-86-19, Tektronix Laboratories (September 1986).

15 Snyder A: 'Inheritance and the development of encapsulated software components', Proceedings of the twentieth annual Hawaii conference on system sciences (1987).

16 Snyder A: 'Common objects: an overview', ACM SigPlan Notices (October 1986).

Paul Gibson is currently completing his BSc(Hon) in computing science and mathematics at the University of Stirling. During summer 1988 he worked on the application of formal object-oriented design principles as a sponsored student with the formal methods group in British Telecom Research & Technology.

Jim Lynch joined the formal methods group in British Telecom Research & Technology after gaining his BSc(Hons) in physics from the University of Manchester in 1986. Since then he has worked on formal descriptions of communications protocols, tools support for formal methods and formal aspects of object-oriented design.