DEPARTMENT OF COMPUTER SCIENCE

Concrete Examples (using CSP) of Process Algebra Templates and Their Children

P.N. Taylor

Technical Report No. 280

April 1997

Concrete Examples (using CSP) of Process Algebra Templates and Their Children

P. N. Taylor.

Department of Computer Science, Faculty of Information Sciences,

University of Hertfordshire, College Lane, Hatfield, Herts. AL10 9AB. U.K.

Tel: 01707 284763, Fax: 01707 284303, Email: p.n.taylor@herts.ac.uk

April 4, 1997

Abstract

This paper describes the concepts of template, class, object and type between processes defined using the process algebra CSP.

We are primarily concerned with the issues of reuse and behavioural compatibility, particularly the stability of systems that incorporate object-oriented inheritance via the incremental modification of existing behaviour.

It is intended that this paper provide concrete process algebra examples for the main concepts of the object-oriented paradigm. The formal language of CSP is used to specify the object-oriented concepts defined in the ISO Reference Model for Open Distributed Processing (RM-ODP) 10746 (Part 2). In this paper CSP processes that are related via strict inheritance (i.e: incremental modification) are classified using their RM-ODP definitions. Conformance and extension testing are then used to prove the behavioural compatibility of simple examples which incorporate reuse.

In order to build an object-oriented concurrent system, components must be classified in terms of template, class, object and type and shown to exhibit behavioural compatibility where inheritance is used.

Introduction

The reuse of a specification's components via object-oriented inheritance is an efficient method of specifying a system, as behaviour need only be defined once and then used many times throughout the specification of a system.

Extending the defined behaviour of a process specialises that process. To ensure that one component is a valid implementation of an existing specification, conformance testing ($Q\ conf\ P$) and extension testing ($Q\ extends\ P$) are performed.

This paper extends a previous work [14] on the provision of natural number examples to cover the object-oriented concepts defined by the International Standardization Organization (ISO) Reference Model for Open Distributed Processing (RM-ODP) 10746 (Part 2) [8].

The RM-ODP document is intended to act as a central repository for the definition of terms used in the production of networks and distributed systems. In this paper we develop examples of simple buffers which incorporate reuse in their definitions.

The RM-ODP definitions for template, class, object and type are the foundation for the testing of each of the case study's components in order to identify behavioural compatibility.

An inheritance hierarchy is defined between the given examples in this paper. Inheritance is realised as an incremental modification technique using CSP external choice as the glue that binds new behaviour to old. The reader is referred to the work of Wegner [15] for an in-depth discussion on incremental modification. In this work three forms of inheritance are identified:

1. Strict Inheritance (i.e: incremental modification) which guarantees behavioural compatibility.

2. Casual Inheritance (i.e: the modification of existing behaviour) which does not guarantee behavioural compatibility.

3. Transitive Inheritance (e.g: $R \sqsubseteq_{inherits} Q \sqsubseteq_{inherits} P \Rightarrow R \sqsubseteq_{inherits} P$) which further defines the inheritance hierarchy.

The process algebra CSP [7] is used to define the behaviour of the example processes. No prior knowledge of CSP is expected. However, reference material, such as [7] and [5] are recommended for a deeper understanding of the concepts and proofs contained in this paper.

The descriptions of the ODP terms presented here are taken jointly from the ODP document itself [8] and its interpretation by both Rudkin [11] and Cusack [4, 3].

Section 1 of this paper introduces the ODP definitions of template, class, object and type as they appear in the work of the ODP [8], Rudkin [11] and Cusack [4, 3]. Subtype and subclass definitions are also given and are further related in section 6. Conformance and extension testing are used to provide evidence of compatibility between the example process definitions in view of the ODP definitions.

Section 2 presents concrete examples of the ODP terms introduced in section 1. Proofs of behavioural compatibility between the example processes are presented using conformance (Q *conf* P) and extension (Q *extends* P) testing.

Section 3 presents proofs between templates to show that one behaviour extends another.

Section 4 defines subtype and subclass relationships between the examples used throughout this paper.

Section 5 discusses inheritance realised as the incremental modification of templates. Contradiction of predicates, their substitution in an environment and examples of contravariance and covariance are also presented.

Section 6 looks at subtype and subclass relationships between templates and presents examples of when these two terms are not equivalent.

Section 7 presents the conclusions drawn from this work and suggests future work concerning the reuse of specifications using inheritance with incremental modification together with asynchronous communications to avoid 'deadlock'.

1 ODP Definitions of Object-Oriented Concepts

The following definitions are taken directly from the ODP Reference Model document [8] and are supplemented with extra observations from the work of Cusack [4, 3] and Rudkin [11].

- X **Template**: The specification of the common features of a collection of Xs in sufficient detail that an X can be instantiated using it. X can be anything that has a type.

  Also, a template is an abstraction of that collection. Templates may be combined according to some calculus. The precise nature of the combination is determined by the specification language used (in this paper CSP external choice).

- **Class (of Xs)**: The set of all Xs satisfying a type. The elements of the set are referred to as members of the class.

  Also, the set of all objects obtained from a given template (known as the class template) by a process of instantiation. Each object is an instance of the class template.

Note: A class may have no members. The size of the class may vary over time, depending upon the definition of the type.

- **Object (instance)**: An object is an instance of a class when it is related to the class template via some chosen membership relation (i.e: it conforms to the predicate of the class template).

- **Type (of an X)**: A predicate characterising a collection of Xs. An X is of the type (or satisfies the type) if the predicate holds for that X. A specification defines which of the terms it uses have types (i.e: are actually Xs). In RM-ODP types are needed for, at least, objects, interfaces and actions (these last two are relevant to this study using CSP).

- **Subtype/Supertype**: A type A is a subtype of a type B, and B is a supertype of A, if every X which satisfies A also satisfies B.

- **Subclass/Superclass**: One class A is a subclass of another class B, and B is a superclass of A, precisely when the type associated with A is a subtype of the type associated with B.

The target process algebra CSP can now be briefly described prior to illustrating the ODP terms introduced in this section.

2 Concrete Examples of Object-Oriented Concepts

The main aim of this paper is to provide the reader with concrete examples of the ODP terms described in section 1 using a process algebra. The language of CSP can be used to describe such terms.

2.1 Basic Laws of CSP

The process algebra Communicating Sequential Processes (CSP) was developed by Tony Hoare and others at Oxford University towards the end of the 1970's [6].

CSP permits the formal specification of sequential processes and the composition of those processes to describe concurrent communicating systems.

The central core of CSP is built around a process which is governed by its alphabet, consisting of those externally visible events with which the process may engage.

αP denotes the *alphabet* of a process P which may only engage in actions that are specified in its alphabet.

$\mathit{traces}(P)$ denotes the sequence of observable actions with which P has previously engaged.

$\mathit{failures}(P)$ denotes the failures of P which is a set of (s, X) tuples. Each (s, X) consists of a trace s and a refusal set X of P, denoting events that P refuses after s has occurred.

$\mathit{divergences}(P)$ denotes the divergences of P. The traces after which P behaves like CHAOS [1] (i.e: behaves like any other process; being completely non-deterministic).

$(P \parallel Q)$ denotes the parallel composition of processes P and Q which then communicate with each other on events shared between their alphabets.

$(P \,\Box\, Q)$ denotes an external (deterministic) choice between process P or Q that is offered to the environment. The choice being influenced by the environment.

$(P \sqcap Q)$ denotes internal (non-deterministic) choice between process P or Q, where the processes themselves make the choice. The choice not being influenced by the environment.

$(a \rightarrow P)$ denotes event a followed by behaviour P.

$(a \rightarrow P \mid b \rightarrow Q)$ denotes choice between the initial events a or b, followed by the corresponding behaviour. Note that $(a \rightarrow P \mid b \rightarrow Q) \equiv (a \rightarrow P \,\Box\, b \rightarrow Q)$ if $a \neq b$ [7, p.107].

Now that the core components of CSP have been presented the ODP terms of the previous section can be illustrated.

2.2 ODP Classification in CSP

A simple buffer example is introduced and expanded using an incremental modification technique.

Template

From the ODP definitions in section 1 we know that an entity X is said to be of a certain type if X satisfies the predicate for that type. So, the template P, from its CSP definition, has a

[1] $\mathit{CHAOS}_A = \mu X : A.X$

behaviour which forms the predicate and its type. Note, from their respective definitions, that predicate and type are closely related.

$$\begin{aligned} P(q) &= \mathit{in}?x \rightarrow \mathit{self}(\langle x\rangle \frown q) & (1)\\ &\quad \Box\ \mathit{out}!\mathit{hd}(q) \rightarrow \mathit{self}(\mathit{tl}(q)) \lhd (q \neq \langle\rangle) \rhd \mathit{self}(q) & \\ Q(q) &= P(q)\ \Box\ \mathit{flush} \rightarrow \mathit{self}(\mathit{empty}) & (2)\\ R(q) &= Q(q)\ \Box\ \mathit{delete} \rightarrow (\mathit{out}!\mathit{len}(q) \rightarrow \mathit{Stop}) & (3) \end{aligned}$$

Note that the original behaviour of P is extended by Q to offer more choice to the environment on initial actions, and likewise for R. We use Rudkin's *self* primitive [11, p.416] to solve the redirection problem identified by Cook regarding self-referential objects and inheritance [2, p.435]. Each occurrence of *self* is instantiated with the name of the process to which control is to be passed after the action sequence ends.

If explicit process naming is used in place of *self* then recursion is trapped at the level of the last process called, thereby failing to return control to the calling process and offer all possible choices of behaviour again.

Class

A class is defined as a set of all objects conforming to the predicate of the template. In the case of process algebras, the actual behaviour of the template. Both Q and R are templates defined by reusing the originally defined behaviour of P. Any instances of P, Q or R (e.q: p, q, r) are valid members of the class associated with template P. The class members for each template derived from P are shown as follows:

$$\begin{aligned} C_P &= \{p, q, r\} \text{ taken from template } P\\ C_Q &= \{q, r\} \text{ taken from template } Q\\ C_R &= \{r\} \text{ taken from template } R \end{aligned}$$

Note the convention used to describe templates and instances. Upper case initials denote a template. Lower case initials denote a class instance.

Object

The template classes C_P, C_Q and C_R contain object instances that are said to conform to their respective template's predicate. In a later proof (on page 9) we shall also attempt to establish that template R is transitively related to P, via Q.

To show the relationship between P, Q and R it is necessary to prove that each process is *at least* compatible with its parent (i.e: offers at least the same behaviour). In the case of our inheriting buffer examples, R is *at least* compatible with Q, which is *at least* compatible with P. For the proof of behavioural compatibility we shall adopt a two-stage approach. Firstly, proving that Q *conforms* to P and secondly, that R *conforms* to Q. It is noted that conformance is not necessarily transitive [3, p.131]. Formalisation of the conformance law is now presented [1, p.11]:

Definition 1 Conformance

Let Q and P be processes.

(Q conf P) iff

$$\begin{array}{l} \forall\, s \in \mathit{Traces}(P). \forall A \subseteq L(P) \\ \quad \text{if } \exists\, Q' \bullet \forall\, a \in A \bullet Q \xrightarrow{s} Q' \not\xrightarrow{a} \\ \quad \text{then } \exists\, P' \bullet \forall\, a \in A \bullet P \xrightarrow{s} P' \not\xrightarrow{a} \end{array}$$

If Q fails to offer an action a (after sequence s) then P must also fail to offer the same action a (after the same sequence s).

If the conformance law is satisfied then a new process can replace an original process in all places where the original process was expected; substitution can then take place [2].

The environment of the original process will be unaware of the exchange as each function offered by the old process will also be offered by the new process. Conformance guarantees *at least* the same behaviour as was originally present in the environment before the substitution took place.

Conformance can be further defined in CSP as follows [3, p.135]:

Law 1: $\alpha P \subseteq \alpha Q$.

Law 2: If (s, X) is a failure of Q with $s \in \mathit{traces}(P)$, then (s, X) is a failure of P.

Law 3: $(\mathit{divergences}(Q) \cap \mathit{traces}(P)) \subseteq \mathit{divergences}(P)$.

If the three CSP laws above hold for (Q *conf* P) and (R *conf* Q) then we have our compatibility proof. We shall also attempt to prove that a transitive relationship between R and P exists in order to say that (R *conf* P) which implies that $\{q, r\} \subseteq C_P$ is true.

To aid in the identification of the failure sets, transition diagrams are presented in figure 1 for each process. The circles represent the different states of each process and the arcs are labelled

[2]Substitution is discussed further in section 5.2

to show an particular event leading to a new state. Recursion is shown using an unlabelled arc pointing back to the initial (top) state of the process. A terminal state is shown as a solid circle. Elementary sets are provided for each process in order to perform the conformance proofs. These

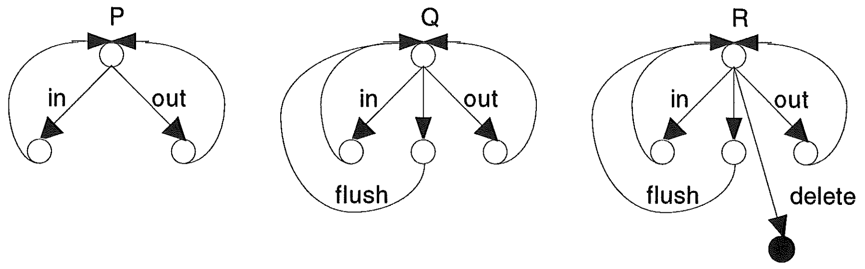

Figure 1: Transition Diagrams for buffer P and its children.

elementary sets are the alphabet, failures and divergences of each buffer process. The elementary sets for template P and its children can now be derived.

$$\begin{aligned}
\alpha P &= \{in, out\} \\
\alpha Q &= \alpha P \cup \{flush\} \\
\alpha R &= \alpha Q \cup \{delete\} \\
failures(P) &= \{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\})\} \\
failures(Q) &= \{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\})\} \\
failures(R) &= \{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\}), \\
&\quad (\langle delete\rangle, \{\{in\}, \{out\}, \{flush\}, \{delete\}\})\} \\
divergences(P) &= divergences(Q) = divergences(R) = \{\}
\end{aligned}$$

The CSP laws for conformance, from Cusack [3, p.134], can now be applied to $(Q \; conf \; P)$ and $(R \; conf \; Q)$.

Proof 1 *$(Q \; conf \; P)$*

Law 1: $\alpha P \subseteq \alpha Q$. In extension: $\{in, out\} \subseteq \{in, out, flush\}$.

Law 2: All (s, X) $(s \in traces(P))$ in $failures(Q)$ can also be found in $failures(P)$.

In extension: All elements of $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\})\}$ (common to $traces(P)$) can be found in $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\})\}$

Law 3: $(divergences(Q) \cap traces(P)) \subseteq divergences(P)$. In extension: $(\{\} \cap \{\}) \subseteq \{\}$.

All three laws for conformance hold for P and Q, therefore $(Q\ conf\ P)$. The same steps can now be performed to prove $(R\ conf\ Q)$.

Proof 2 *$(R\ conf\ Q)$*

Law 1: $\alpha Q \subseteq \alpha R$. In extension $\{in, out, flush\} \subseteq \{in, out, flush, delete\}$.

Law 2: All (s, X) $(s \in traces(Q))$ in $failures(R)$ can be found in $failures(Q)$.

In extension: All elements of $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\}), (\langle delete\rangle, \{\{in\}, \{out\}, \{flush\}, \{delete\}\})\}$ (common to $traces(Q)$) can be found in $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\})\}$

Law 3: $(divergences(R) \cap traces(Q)) \subseteq divergences(Q)$. In extension: $(\{\} \cap \{\}) \subseteq \{\}$.

Finally, we attempt to prove a transitive relationship between the templates R and P; prove that $(R\ conf\ P)$ is true. Although conformance is not necessarily transitive this final proof will show that a transitive conformance relationship exists in the inheritance hierarchy. Implicitly we suspect that R conforms to the behaviour of P because within R we can identify the exact behaviour of P.

Proof 3 *$(R\ conf\ P)$*

Law 1: $\alpha P \subseteq \alpha R$. In extension $\{in, out\} \subseteq \{in, out, flush, delete\}$.

Law 2: All (s, X) $(s \in traces(P))$ in $failures(R)$ can be found in $failures(P)$.

In extension: All elements of $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\}), (\langle flush\rangle, \{\{\}\}), (\langle delete\rangle, \{\{in\}, \{out\}, \{flush\}, \{delete\}\})\}$ (common to $traces(P)$) can be found in $\{(\langle\rangle, \{\{\}\}), (\langle in\rangle, \{\{\}\}), (\langle out\rangle, \{\{\}\})\}$

Law 3: $(divergences(R) \cap traces(P)) \subseteq divergences(P)$. In extension: $(\{\} \cap \{\}) \subseteq \{\}$.

All three laws for conformance hold for $(R\ conf\ P)$. Therefore, as we would expect from incremental modification, the behaviour of each child of P can be found within P, so conformance holds for both immediate and transitive relations.

The types of each template and class member can now be defined.

Type

A relationship is defined between template and type as both can be described in terms of a predicate. The predicate for a type characterises a collection of object instances.

We have shown that $(Q\ conf\ P)$, $(R\ conf\ Q)$ and $(R\ conf\ P)$ are true. Therefore, the class sets $C_P = \{p, q, r\}$, $C_Q = \{q, r\}$ and $C_R = \{r\}$ are valid according to their types, which in turn conform to their respective predicates. Both p, q and r are valid instances of the template class described by the template P.

3 Further Classification of buffer P with Extension

One extra criteria separates the CSP rules for conformance $(Q\ conf\ P)$ from those of extension $(Q\ extends\ P)$. The traces of the original process must be present in the traces of the modified process [3, p.134]. Formally, we write this extra rule as:

Law 4 : $traces(P) \subseteq traces(Q)$

In order to prove that the three previous conformance proofs hold for extension we simply evaluate the elementary sets of each derivative of P in light of this new rule. Firstly, we prove that Q extends the behaviour of P.

Proof 4 *(Q extends P)*

Law 1,2,3: true, as per proof 1.

Law 4: $traces(P) \subseteq traces(Q)$. In extension: $\langle in, out, \ldots\rangle \subseteq \langle in, out, flush, \ldots\rangle$.

Next, we prove that R extends the behaviour of Q.

Proof 5 *(R extends Q)*

Law 1,2,3: true, as per proof 2.

Law 4: $traces(Q) \subseteq traces(R)$. In extension: $\langle in, out, flush, \ldots\rangle \subseteq \langle in, out, flush, delete\rangle$.

Finally, we attempt the transitive extension relationship which states that R extends the behaviour of P. Implicitly, we expect $(R\ extends\ P)$ to be true as, from the original CSP specification for R (see page 6), the behaviour of P is embedded within R. So, next we prove that R extends the behaviour of P.

Proof 6 *(R extends P)*

Law 1,2,3: true, as per proof 3.

Law 4: $traces(P) \subseteq traces(R)$. In extension: $\langle in, out, \ldots \rangle \subseteq \langle in, out, flush, delete \rangle$.

Not only can we say that the children of P *conform* to P but, after proof 6 above, we can also say that the children of P *extend* P.

4 Subtype/Supertype and Subclass/Superclass Relationships

According to the proofs in the previous section a buffer Q implies a buffer P because any object instance that conforms to the behaviour of Q must also conform to the behaviour of P (and likewise for *extends*). Therefore, by definition (from page 3), Q is a subtype of P (P is, by implication, a supertype of Q). From proofs 2 and 3 (page 9 onwards) we can also say that R is a subtype of Q and that R is also a subtype of P. This subtype relationship can be written formally as follows:

$$(Q \sqsubseteq_{subtype} P) \wedge (R \sqsubseteq_{subtype} Q) \Rightarrow (R \sqsubseteq_{subtype} P)$$

The subtype relationship can be extended to encompass the object instances drawn from the class sets for the two templates Q and P. Again, conformance is used to prove that behavioural compatibility exists between the object instances of both P and Q. The range of the traces of a subtype process, restricted to the alphabet of the supertype process can be compared to determine the validity of an object instance in a class set. For example, the traces of P and Q can be compared, as follows:

$$(\operatorname{ran} traces(R) \restriction \alpha P) \subseteq (\operatorname{ran} traces(Q) \restriction \alpha P) \subseteq \operatorname{ran} traces(P).$$

$$\text{In extension: } \{in, out, \ldots\} \subseteq \{in, out, \ldots\} \subseteq \{in, out, \ldots\}.$$

Extra behaviour of the subtype is ignored as conformance only guarantees that one process implements *at least* the behaviour of the other. Provided that Q acts like P any other behaviour that it offers to the environment is not relevant to the conformance proof. With *extends* more behaviour is assumed if *at least* the minimum behaviour of the parent is represented in the extended process.

5 Inheritance via Incremental Modification

The method of reuse employed in this paper is incremental modification (see [15]) and is referred to as strict inheritance. We do not imply any other form of inheritance here, only that of offering extra choice over the initial actions of a process.

Duplicate initial actions lead to non-determinism which we aim to avoid. Care must be taken when adding new behaviour to an existing process by avoiding the inclusion of initial actions that will duplicate any existing initial actions.

In our buffer examples CSP external choice (e.g: $P \Box Q$) is used as the glue to attach any new behaviour to an existing process definition. As shown in the definitions for templates Q and R which inherit from P (see page 6). Consider another example template S which inherits from P and is a further restriction (sometimes called a 'specialisation') of P, resulting in a template offering the environment an extra initial action *flush*.

$$S(q) = P(q) \Box \mathit{flush} \rightarrow \mathit{self}(Q) \tag{4}$$

As more behaviour is added to a process it can become more difficult to satisfy the predicate defined by that new behaviour. As with our earlier paper (where natural number predicates were extended using logical conjunction [14, p.5]) when behavioural extension takes place the number of elements in the template class can be reduced. We note that if S implies P then S is a subtype of P.

$$C_S \subseteq C_P \Rightarrow \{q, r\} \subseteq \{p, q, r\}$$

The associated classes for S, namely elements that meet the predicate for S in C_S, are equivalent to the behaviour of Q (by definition of Q on page 6). Consequently, $(S \equiv Q)$ which allows us to substitute S in all places where Q was originally expected. The concept of substituting processes is discussed in more detail in section 5.2.

Further extension of a process denotes more external choice, remembering that extension which introduces duplicate initial actions must be avoided (see section 5.1). Consider the definition of another template which inherits from S:

$$T(q) = S(q) \Box \mathit{delete} \rightarrow (\mathit{out}!\mathit{len}(q) \rightarrow \mathit{Stop}) \Box \mathit{length} \rightarrow (\mathit{out}!\mathit{len}(q) \rightarrow \mathit{self}(q)) \tag{5}$$

The class set for T is $\{r\}$ as only instantiations of R can offer *at least* the behaviour of template T. To formalise the inheritance relationship between these new templates that we have derived from P we write the following:

$$T \sqsubseteq_{inherits} S \sqsubseteq_{inherits} P$$

Using the established inheritance hierarchy we identify the subtype relationship between the templates:

$$T \sqsubseteq_{subtype} S \sqsubseteq_{subtype} P$$

The subset operator on sets can then be used to define a relationship between instances of each type, as follows:

$$\{r\} \subseteq \{q, r\} \subseteq \{p, q, r\}$$

The cardinality of the template's class sets may reduce as the predicate to satisfy increases. Note that extending the predicates of a template will not necessarily specialise it further and may have no effect on the elements of the class set. For example:

$$U(q) = P(q) \,\Box\, \mathit{flush} \rightarrow \mathit{self}(\mathit{empty}) \tag{6}$$

The behaviour defined in template U does not alter the traces of Q (the behaviour of which is equivalent), although U does introduce non-determinism into the template definition. In this example, the new behaviour of U is the same as for an existing branch (as defined in P). External choice (the glue for our incremental modification) is idempotent if the behaviour is identical on both sides of the external choice operator. Therefore, $(P \,\Box\, P \equiv P)$ [7, p.107, Law 1], which allows us to remove the extra duplicate behaviour in U. The reader is left the task of proving that $U \equiv P$. (Hint: use associative, symmetric and idempotent laws on external choice [7, p.107, Laws 3,2,1]). Needless to say, the following is true:

$$C_U = C_Q = \{q, r\}$$

So, as you can see, it is not guaranteed that incremental modification will reduce the scope of the objects in a class template. What is guaranteed is that incremental modification will not increase the scope of the number of objects captured by a class template.

5.1 Contradiction of Predicates Leading to Non-determinism

A contradiction with a template's type predicate restricts the template's class set to an empty set. Contradiction within process specifications occurs when an initial action is duplicated within a process where external choice is present. From the previous section, U contains a contradiction which can be resolved by the idempotent law. With a slight change to the behaviour offered by U consider the following definition for buffer V:

$$\begin{aligned} V(q) &= P(q) \Box \mathit{flush} \rightarrow \mathit{Stop} \\ C_V &= \{\} \end{aligned} \tag{7}$$

No instances exist that match the new behaviour of template V that are derivable from P. We cannot predict which behaviour will result from choosing the initial action *flush*, hence V exhibits non-deterministic behaviour, one branch of which leads to a terminal state. Note that instances of V (i.e: elements of C_V) are not deemed stable as they can 'halt' execution of the process.

5.2 Substitution

The substitution of S for Q was achievable (see page 12) because they both shared the same external behaviour and could not be distinguished from each other by testing (e.g: $(S\ \mathit{conf}\ Q)$). As an observer we would accept one template in place of the other because each action required by the environment of Q could be serviced by either Q or S.

If $(C_Q \subset C_P)$ then $(C_P \not\subset C_Q)$, by definition of the proper subset operator. Therefore, objects in C_P offer less behaviour than their C_Q counterparts; P is therefore contravariant in relation to Q. Whereas we might use Q in place of P we cannot reverse the substitution, only S can do so (as defined on page 12).

5.3 Contravariance and Covariance

This section introduces contravariance and covariance in the light of the subtype/subclass relationships defined over templates P, Q and R.

Definition 2 Contravariance*: Arguments of the subtype must be less-than-or-equal to arguments of the supertype.*

If P and Q are regarded as functions then each action is mapped to a state of the buffer's queue. The domain of the function being equal to the alphabet of the process.

$$f_P : \alpha P \rightarrow q$$

$$f_P = \{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q)\}$$

$$\begin{array}{|l}
f_Q : \alpha Q \to q \\
\hline
f_Q = \{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q), \mathit{flush} \mapsto empty\}
\end{array}$$

The possible set of states accessible from the full range of initial events can then be derived:

$$f_P \overbrace{(\{in, out\})}^{contravariance} \longrightarrow \{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q)\}$$
$$f_Q(\{in, out, \mathit{flush}\}) \longrightarrow \{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q), \mathit{flush} \mapsto empty\}$$

If Q is a subtype of P then the initial actions offered by P are said to be *contravariant* in relation to Q because $\{in, out\} \subset \{in, out, \mathit{flush}\}$ is true. Buffer P does not allow for the full range of actions provided by Q. For example, after an event *in* the buffer Q may accept an event *flush*, which is something that buffer P may never do.

Definition 3 Covariance: *Results of the application of the subtype must be less-than-or-equal to the results obtained from the supertype, given the same arguments.*

Using the same examples above, consider the case when P offers similar behaviour to that of Q, given the same initial event.

$$f_P(\{in, out\}) \longrightarrow \overbrace{\{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q)\}}^{covariance}$$
$$f_Q(\{in, out, \mathit{flush}\}) \longrightarrow \{in \mapsto \langle x \rangle \frown q, out \mapsto tl(q), \mathit{flush} \mapsto empty\}$$

Covariance is identified in P as it returns the same result as Q given the same inputs. From these simple examples we can see that an invalid subtype relationship would result from contravariance due to more restrictive arguments of one of the types (in this case P).

If we attempt type assignment of the templates for P and Q we see another example of subtype violation due to contravariance, as shown by the following example:

$$\begin{array}{lcl}
p & : & f_P \text{ // variable declaration} \\
q & : & f_Q \text{ // variable declaration} \\
 & \vdots & \\
q & := & p \text{ // type assignment}
\end{array}$$

Attempts to apply function $q(\mathit{flush})$ will fail as q is assigned p of type f_P, the only acceptable initial action for which is either *in* or *out*. Contravariance is responsible for the failure here as it restricts the range of inputs in f_P to those of type f_Q.

6 Defining Subtype and Subclass Relationships

The subtype relationship between P, Q and R has been defined. If $Q \subseteq_{subtype} P$ then we have shown that $C_Q \subseteq C_P$ (see page 11). This section explains the differences between the subtype and subclass relationships. It is not always the case that objects related by the subtype relationship

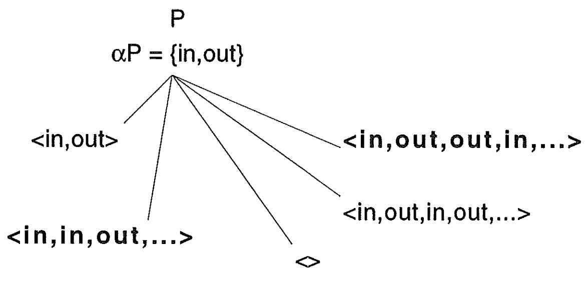

Figure 2: The Set of Traces Derivable from buffer P.

are necessarily related by the subclass relationship. From the template P all other templates that accept the initial events *in* or *out* can be derived. Two examples of templates that are type compatible with P but not class compatible are shown as bold traces in the diagram in figure 2.

Neither $\langle in, in, out, \ldots\rangle$ nor $\langle in, out, out, in, \ldots\rangle$ can be reached using the predicates that describe either P or its children. These bold traces represent subtypes but not subclasses of P. Note that the bottom ($\perp$) of the type-tree in figure 2 is the empty trace ($\langle\rangle$) and the top of the type-tree is template P itself.

7 Conclusions and Future Work

This paper has presented concrete examples of processes taken from simple examples in order to classify them according to the ODP object-oriented definitions that appear in [8].

The main focus of this paper was to prove the behavioural compatibility of templates that reuse existing behaviour via incremental modification (i.e: strict inheritance). Two other forms of inheritance are worth mentioning at this point; transitive and casual inheritance. Transitive inheritance has been discussed in this paper in relation to the successful conformance testing of templates R and P (see page 9). The discussion of casual inheritance (where behavioural compatibility is not guaranteed between templates) is left to a future paper.

Conformance and extension testing were used to prove that templates incorporating reuse were valid implementations of their parents. Proofs of subtype and subclass relationships between example processes were also supplied.

If a communicating system requires modification then certain processes are natural candidates for substitution. Certainly, any process that meet the laws of conformance will be capable of replacing existing behaviour. However, an issue for future discussion is the cost of the replacement if stability is compromised by the newcomer.

Most process algebras, including CSP, use synchronisation between processes in order to model communication. An inheriting replacement process could wait on some extra communication not present in the original process. The new process's failure to respond during synchronisation could then result in a chain reaction of 'deadlocked' processes which will eventually lead to deadlock of the entire system. We refer to this chain reaction of waiting processes as *domino waiting* [12]. Further work is required to see if asynchronous communications using unbounded buffers, such as the work of Josephs [10, 9] and our own work [13], can offer a solution to the synchronisation problem of inheriting processes. When using external (deterministic) choice as the glue to patch new behaviour onto existing process behaviour we need to be able to guarantee the stability of the resulting system.

Given that conformance must be present in order for inheritance and substitution to produce a stable system, the importance of establishing a valid inheritance, subtype and subclass hierarchy is made clear. The method by which this inheritance and substitution is to be achieved is still open to debate. Our work continues.

Acknowledgements

Many thanks go to David Smith for his insight into templates and their representation as natural numbers. Also, thanks to Jean Baillie for various discussions regarding CSP and non-determinism and finally, Mary Buchanan for contributing to my knowledge about types, contravariance and covariance.

References

[1] E. Brinksma and G. Scollo. Formal notations of implementation and conformance in LOTOS. Memorandum inf-86-13, Department of Informatics, University of Twente, The Netherlands, 1986.

[2] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA '89 Proceedings*. ACM Press, October 1989.

[3] E. Cusack. Refinement, Conformance and Inheritance. *BCS Formal Aspects of Computing*, 3(2):129–141, 1991.

[4] E. Cusack, S. Rudkin, and C. Smith. An object-oriented interpretation of LOTOS. In *The 2nd International Conference on Formal Description Techniques (FORTE89)*, December 1989.

[5] M.G. Hinchey and S.A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill, London, 1995.

[6] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[8] ISO/IEC. Information retrieval. transfer and management for OSI - committee draft - basic reference model of open distributed processing - part 2: Descriptive model. Technical Report ISO/IEC JTC1/SC21 N6079 8807, International Standardization Organisation, June 1991.

[9] M.B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[10] M.B. Josephs, C.A.R. Hoare, and H. Jifeng. A theory of asynchronous processes. Technical Report PRG-TR-6-89, Programming Research Group, University of Oxford, Oxford University Computing Laboratory, 11 Keble Road, Oxford. OX1 3QD, 1989.

[11] S. Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques*, volume VI, pages 409–423. Elsevier Science Publishers B.V: North Holland, 1992.

[12] P.N. Taylor. Research issues associated with Process Algebras in an Object-Oriented design framework. Unpublished lecture notes, Department of Computer Science, Faculty of Information Sciences, University of Hertfordshire, Hatfield, U.K. AL10 9AB, November 1996.

[13] P.N. Taylor. Resilient Process Theory (RsPT). Unpublished lecture notes, Department of Computer Science, Faculty of Information Sciences, University of Hertfordshire, Hatfield, U.K. AL10 9AB, November 1996.

[14] P.N. Taylor. Concrete examples of templates and their children. Unpublished lecture notes, Department of Computer Science, Faculty of Information Sciences, University of Hertfordshire, Hatfield, U.K. AL10 9AB, January 1997.

[15] P. Wegner and S.B. Zdonik. Inheritance as an Incremental Modification Technique or What Like is and Isn't Like. In *Proceedings of ECOOP'88*, pages 55–77, Oslo, Norway, August 1988. ECOOP.