RAL-92-062

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-92-062

Process Algebra in the Specification of Graphics Standards

C Reade

September 1992

Process Algebra in the Specification of Graphics Standards

Chris Reade [*]

August 1992

Abstract

Some issues which relate to equivalence and conformance in formal specification arising from a case study are discussed. This work is a continuation of a case study looking at the use of the formal specification language LOTOS [LOTOS 1989] in the description of graphics software and standards. (Earlier work on this case study was reported in [Purvis 1990].) One of the objectives of the study is to discern areas where formal reasoning might be helpful in comparisons and tools to support formal reasoning can be applied to solve problems.

An example of simulating output devices is used to illustrate some problems with modelling devices and designing protocols for interfaces. The notion of *stable observation testing* is introduced to deal with problems of delay.

1 Introduction

One of the main purposes of formalising standards is to ensure precision in the description of the standard. However, such precision is only useful if an appropriate notion of conformance is also made precise and methods for establishing conformance exist. A small example concerning the simulation of output devices is described here using process algebra to illustrate some issues related to (observational) equivalence and conformance in the specification of graphics software.

[*]Formal Methods Group, SEG, Informatics Dept., Rutherford Appleton Laboratory and Computer Science Dept., Brunel University

1.1 Background

This work is the continuation of a case study using LOTOS [LOTOS 1989] for specifying standards for graphics software. A previous report [Purvis 1990] contained several detailed examples, focussing on the formalisation of GKS [GKS 1985] in LOTOS. The case study is part of the work carried out for the ERIL project (Equational Reasoning in LOTOS [ERIL]). Thus, one of the objectives of the case study work has been to discern areas where formal reasoning is helpful and where tools to support formal reasoning can be applied to solve problems.

LOTOS is an ISO standard formal specification language developed for the description of Open Systems Interconnection Services and Protocols. It combines both a process algebra derived from CCS and CSP for describing the behaviour of systems (processes) with an algebraic specification language (ACT ONE) for describing data types. As such it provides a useful vehicle for studying the relationships between process and data in the specification of graphical systems. Furthermore, ISO now recommend the use of standardised specification languages (like LOTOS) for use in specifying other standards, so it is an appropriate choice for formalising grpahics standards.

1.2 Related Work

Several papers have reported work on using data algebra and, more recently, process algebra to formalise standards for graphics software. Earlier work in formalising graphics standards mostly focussed on the data, operations and primitives associated with output and transformations in GKS (e.g. [Arnold et al 1987, Duce 1988, Duce 1989]). More recently, the problem of specifying input has been tackled using process algebras which are more appropriate for describing concurrent behaviour of multiple interactions. In [Duce et al 1989] CSP was used to specify behaviour of input devices. Several aspects of modelling graphics software are discussed in [Gnatz 1990], but a less abstract (more implementation oriented) notion of process is used.

In [Purvis 1990] specific parts of GKS were formalised with LOTOS. This work uses the data abstractions of LOTOS to specify data objects and state components in much the same way as in Duce's earlier work with OBJ [Duce 1989]. The behavioural part of LOTOS is then used to extend such specifications to include interface protocols and abstract models of devices (especially the *logical input devices* of GKS).

1.3 Combining Data and Process in Specifications

The formalisation of output primitives and components of the state in GKS using algebraic data type specifications is relatively straight forward. The only real problem is choosing appropriate levels of abstraction which avoid overspecification but which still allow appropriate distinctions to be made. On the other hand, the formalisation of behaviour with behavioural abstractions is less clear cut because there are far more choices to be made. Issues of distributed state, multiple access, sharing, views, preserving integrity and protocol design are all involved at this level.

The problem of where to draw the boundary between abstract data type descriptions and process descriptions is a critical one. In fact this distinction between data and event is a fundamental problem in software design generally and certainly not unique to specifications of graphics software. In general there is a spectrum of possibilities between the two and descriptions need to be designed to avoid (i) overspecification (for example, when process descriptions force unnecessary constraints on the ordering of events) and (ii) underspecification (E.g allowing distributed state access to give erroneous or inconsistent results in certain circumstances).

A particular problem which arises is how to feed-back responses to input when the intermediate calculation is distributed and may be interleaved with other input and other responses. It is clearly not sufficient to insist on one input/output cycle at a time when access is distributed, so more intricate protocols must be introduced. Formal descriptions of behaviour can be used to express such protocols pecisely and also enable further analysis to check, e.g. that integrity is preserved and whether two processes have equivalent observable behaviour. Another problem is that of how to model physical devices in a suitably abstract way. Both these problems are illustrated in the example in section 2.

1.4 Equivalence and Conformance

Of direct relevance to decisions about specification style are the notions of equivalence and the relationship of satisfaction or conformance between implementation and specification. To a certain extent, the problem of where to draw the boundary between process and data descriptions can be aleviated by the ability to show equivalences. If two specifications can be shown to be equivalent despite differences in the style (e.g. in the separation between process and data), it does not really matter which is chosen as the preferred specification because they will both have the same posssible implementations (for a suitable notion of equivalence). However, it is important to take into account how easy the proofs of equivalence and satisfaction are. This is where further investigation is needed both in theory (methods for showing equivalences) and in practice (criterea for good choices of specification style).

For algebraic specifications of data types, there are well established notions of observational equivalence and sufficient completeness. For Process algebras there are many different notions of equivalence.

The LOTOS standard gives the semantics of behaviour expressions with labelled transition systems and desribes some equivalence relations in an annex. In particular: testing equivalence (with an associated congruence) is a weak equivalence which may identify too many processes for some applications, while weak bisimulation equivalence (with the associated congruence called observational equivalence) can be regarded as a minimal equivalence to be subsumed by all others.

In the literature, there are other interesting equivalences, based on De Nicola and Hennessy Testing [DeN Henn 1984], Refusal Testing [Phillips 1987], Generalised Failures [Langerak 1990]. In [Abramsky 1987] a framework for studying many equivalences was developed and it was shown how observational equivalence was a form of testing equivalence with an extra-rich notion of test.

There is also the relationship of conformance discussed in [Brinksma et al 1986]. In process algebra, behaviour expressions are used both for describing specifications and implementations, thus the satisfaction of a specification by an implementation (the conformance of the implementation to the specification) is a relationship between behaviour expressions. One possible notion of conformance is shown in [Brinksma et al 1986] to be a combination of extension and reduction. Behaviour B extends C if it may perform additional actions without disrupting the actions required by C (it has no new failures involving actions required by C). Behaviour A reduces B if it is more deterministic (a reduction in the *don't care* choices without introducing new failures). A conforms to C if there is some B such that A reduces B and B extends C. (Formal definitions are given in section 3). This notion of conformance allows for arbitrary behaviour in an environment which does not behave properly and so does not say anything about robustness. It is clear from this that using processes to specify required behaviour involves more than just describing some ideal behaviour and hoping that equivalence will capture all the alternative descriptions of behaviour which would be allowable substitutes. It seems necessary to consider the environment and a possible range of alternative behaviours which might be acceptable but which are not equivalent when writing a specification. However, it may be possible to develop new equivalences (or forms of test) which capture classes of allowable behaviours to ease the burden of specification.

In the next section we describe the example; in section 3 formalisation problems are discussed and the formal definitions for stable observation testing are given.

2 Example: Simulation of an Output Device

This example is based on some abstract descriptions of graphics output devices (taken from [Gnatz 1990]) and couched here as LOTOS process specifications.It is chosen to illustrate several problems about observation as well as representation techniques.

2.1 Output Devices VECDEV and CURPOS

We assume an underlying type STROKE

```
type STROKE is INDEX POINT COLOUR
sorts   Stroke
opns    mty: -> Stroke
        stroke: Stroke Index Index Colour -> Stroke
        wndw: Index Index -> Bool
        apprnc: Stroke Point -> Colour
eqns ....
endtype
```

with a sort **Stroke** and an operation **stroke** to create **Stroke** values starting from an initial empty **Stroke** value (**mty**). The **wndw** function determines the area of interest and the **apprnc** (appearance) function is an observer function allowing the inspection of **Stroke** values at chosen points to produce a colour. We will not introduce the laws relating these here as they are not needed for the example. However we do note that **stroke(s,i,j,c)** is only regarded as properly defined when **wndw(i,j)** = **true**. In [Gnatz 1990] this restriction on the application of **stroke** is expressed in the laws for the type. In LOTOS specifications, functions are always total so we have put such constraints into process specifications which control use of the type. This type is common to two different abstract output devices which are pictured in figure 1.

The device **VECDEV** (vector device) has primitive access operations (event gates) **clr** (clear/reset operation) and **drwln** (draw-line operation with parameters of type **Index**, **Index** and **Colour**). There is also a gate for passing back a boolean value to indicate if the last **drwln** operation was successful.

This, apparently simple feedback turns out to be an interesting problem which allows us to illustrate a general problem about distributed states and communication. In an actual implementation, this feedback might well be implemented as a return value of a procedure or using an update to a shared variable. In this context, we need to express the feedback as a communication between processes (i.e. as an event), but there are different ways to do this:-

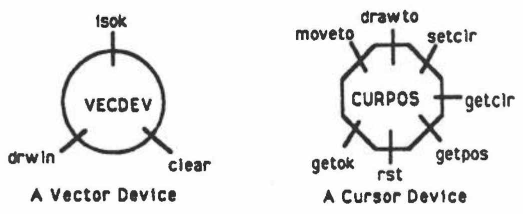

Figure 1: VECDEV and CURPOS

- A simple offering of the return event after a **drwln** action which requires the event to be accepted (suspending other offers until after the return value has been communicated)
- Buffering the return event through an intermediate process so that it can be ignored (i.e. so that the writing and reading of the value do not have to be synchronised).
- Making the return value part of the event **drwln** (i.e. making this a two-way communication with values passing both ways).

Each of these is reasonable, but we choose the second method in order to model the possibility of ignoring the result which we believe to be the more typical case in implemented software. The first method requires more constraints on external processes, while the third method would causes problems for events which take some time to complete. A fourth possibility where the return event is offered alongside alternative choices is ruled out as unreasonable. This is because a process waiting for the return event could be pre-empted by another process trying to proceed with further **drwln** requests causing the first process to deadlock or receive the wrong results.

We specify the process **VECDEV** using components running in parallel, where one process (**OK**) is just a buffer for the **isok** return value (written to by the main process on gate **setok** and able to be read independently by an external process (the application program) through **isok**:-

```
process OK[setok,isok](b:Bool)
  :=  (  setok?ok:Bool; OK[...]ok
      [] isok!b; OK[...]b
      )
endprocess
```

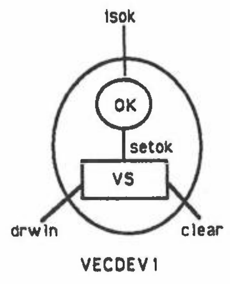

Figure 2: Construction of VECDEV1

(We use ... to abbreviate the repetition of the event gate names in the header). So process **OK** can either accept a new boolean value (**ok**) on gate **setok** or output its current value on gate **isok** and repeat. The other process **VS** is given by:-

```
process VS[clear,drwln,setok,observe](vs:Stroke)
  :=   (  clear; VS[...]mty
       [] drwln?u:Index,v:Index,c:Colour;
            (  [wndw(u,v)]     ->  setok!true;
                                   VS[...](stroke(vs,u,v,c))
            [] [not wndw(u,v)]  ->  setok!false;  VS[...]vs
            )
       [] observe?p:Point!apprnc(vs,p); VS[...]vs
       )
endprocess
```

VS is parameterised by a current stroke value (**vs**) and will engage in a **clear** request to return to the initial **mty** state. Alternatively it will engage in any **drwln** action with values for the two indexes (**u** and **v**) and colour (**c**); after which the appropriate boolean value is sent to the companion process by a **setok** action, depending on whether **wndw(u,v)** is **true** or **false**. In the former case, the parameter is changed to record the new state (**stroke(vs,u,v,c)**), otherwise it is left unchanged. The additional (**observe**) action is discussed in the next sub-section, and we ignore it for the moment.

A first approximation to the main process is **VECDEV1** (see figure 2) which consists of **VS** and **OK** in parallel, communicating via **setok** actions which are then hidden from outside:-

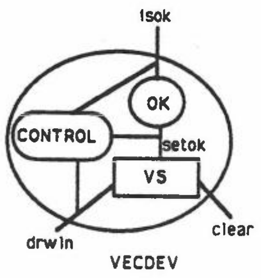

Figure 3: Construction of VECDEV

```
process VECDEV0[clear,drwln,isok,setok,observe]
  :=   VS[clear,drwln,observe,setok](mty)
          |[setok]|
       OK[setok,isok](true)
endprocess

process VECDEV1[clear,drwln,isok,observe]
  := hide setok in VECDEV0[clear,drwln,isok,setok,observe]
endprocess
```

This is referred to as a *first approximation* because it is *wrong*!

The problem is that the concurrentisation of the behaviour into two communicating processes allows an application program to do a **drwln** and then read the return value on **isok** too early (before it is set by the internal communication).

This illustrates a fundamental problem with distributed states. The problem can be avoided by writing **VECDEV** as a non-distributed (sequentialized) process. However, we prefer to seek a more general solution which allows us to keep the distributed state, avoiding the erroneous behaviour by adding some further control.

The solution given below, makes essential use of the multi-party synchronising communications which are available in LOTOS and CSP (but not CCS). We introduce a third, controlling / synchronising process which insists on following a **drwln** action by a **setok** action before allowing an **isok** action (or another **drwln**):-

```
process CONTROL[drwln,setok,isok]
  := (  drwln?u:Index,v:Index,c:Colour;
             setok?b; CONTROL[drwln,setok,isok]
     [] isok?b; CONTROL[drwln,setok,isok]
     )
endprocess
```

This CONTROL process is put in parallel with VECDEVO so that all drwln, setok and isok actions must synchronise (see figure 3):-

```
process VECDEV[clear,drwln,isok,observe]
  := hide setok in
       VECDEVO[clear,drwln,observe,setok]
          |[drwln,setok,isok]|
       CONTROL[drwln,setok,isok]
endprocess
```

This now has the correct behaviour. Furthermore, we claim that VECDEV is (observationally) equivalent to the following non-distributed (sequentialised) version (SVD):-

```
process SVD[clear,drwln,observe](vs:Stroke, b:Bool)
  :=    (  clear; SVD[...](mty,b)
        [] drwln?u:Index,v:Index,c:Colour;
              (  [wndw(u,v)]     -> SVD[...](stroke(vs,u,v,c),true)
              [] [not wndw(u,v)] -> SVD[...](vs,false)
              )
        [] observe?p:Point!apprnc(vs,p); SVD[...](vs,b)
        )
endprocess
```

An outline proof of this is given in the appendix. Although this is only a small example, it illustrates the potential for a formalism which allows such proofs.

The second device CURPOS is more like a device drawing dots and keeping track of the current cursor position and a current colour as well as a stroke value. Once again it comprises a main process CS in parallel with OK and a CONTROL'. CS is expressed as:-

```
process CS[rst,stclr,drwto,moveto,qc,qp,
           observe,setok] (vs:Stroke,vc:Colour,cp:Index)
  := (  rst; CS[...](mty,bg,dx(0,0))
     [] stclr?c:Colour; CS[...](vs,c,cp)
```

```
        [] drwto?u:Index ;
             (  [wndw(cp,u)] -> setok!true;
                                 CS[...](stroke(vs,cp,u,vc),vc,u)
             [] [not wndw(cp,u)]  ->  setok!false;
                                         CS[...](vs,vc,cp)
             )
        [] moveto?u:Index ;
             (  [wndw(cp,u)]   ->  setok!true;  CS[...](vs,vc,u)
             [] [not wndw(cp,u)]  ->  setok!false;
                                         CS[...](vs,vc,cp)
             )
        [] qc!vc ;   CS[...](vs,vc,cp)
        [] qp!cp ;   CS[...](vs,vc,cp)
        [] observe?p:Point!apprnc(vs,p); CS[...](vs,vc,cp)
        )
   endprocess
```

and has a **rst** action (similar to **clear**) which resets all the parameter values of type **Stroke Colour** and **Index** respectively. **CS** will also accept: **stclr** (set colour) with any colour (**c**) an action which is used to update the colour stored in the current state (parameter); **drwto** (draw to) with an index value (**u**) which is used along with the current position index value (**cp**) and current colour (**vc**) to draw a stroke (update the stroke value) and update the current position provided **wndw(cp,u)** = **true**; **moveto** with an **Index** value (**u**) which is used along with the current position index value (**cp**) to update the current position provided **wndw(cp,u)** = **true**. [For both **drwto** and **moveto**, **OK** is sent the appropriate boolean value (via **setok**) before any change of parameters]. The actions **qc** and **qv** are just to allow the parameters **vc** and **cp** (recording the current colour and current position, respectively) to be inspected. (Once again, we ignore the **observe** action until the next sub-section.)

The main **CURPOS** is just the parallel composition of **CS** and **OK** communicating via **setok** with **CONTROL'** used to synchronise. It has the same events as **CS** except that **setok** is hidden from outside and the other gate from **OK** (renamed **getok**) is also visible:-

```
   process CURPOSO[rst,stclr,drwto,moveto,getok,qc,qp,observe]
     := CS[rst,stclr,drwto,moveto,
              qc,qp,observe,setok] (mty,bg,dx(0,0))
                 |[setok]|
        OK[setok,getok](true)
   endprocess
```

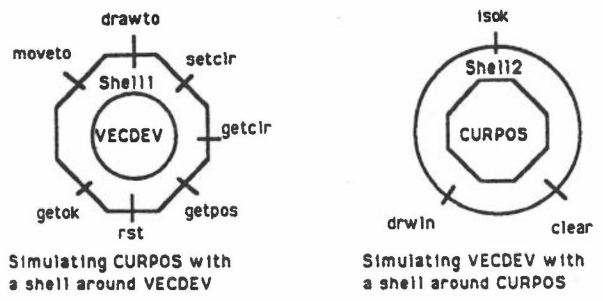

Figure 4: Simulation Shells

```
process CONTROL'[drwto,moveto,setok,getok]
  := (  drwto?u:Index ; setok?b;
            CONTROL'[drwto,moveto,setok,getok]
     [] moveto?u:Index ; setok!b;
            CONTROL'[drwto,moveto,setok,getok]
     [] getok?b; CONTROL'[drwto,moveto,setok,getok]
     )
endprocess

process CURPOS[rst,stclr,drwto,moveto,getok,qc,qp,observe]
  := hide setok in
       CURPOSO[rst,stclr,drwto,moveto,getok,qc,qp,observe]
            |[drwto,moveto,getok,setok]|
       CONTROL'[drwto,moveto,setok,getok]
endprocess
```

2.2 Simulations of the Devices

We are interested in the possibility of writing shells around VECDEV and CURPOS so that they appear to behave like the other device as far as applications programs are concerned and allow the same displays to result. Such shells are pictured in figure 4 and can be simply modelled as processes put in parallel with the underlying device with internal links hidden from the outside. However, we also need to deal with observation by the user. At the moment, we have only discussed the behaviour through the gates which are used by applications programs but we know that the effects on states, or at least the appearance of displays can be observed. In process descriptions, we need to model such observations explicitly as events in order to avoid identification of different states. (Processes can only be distinguished by their behaviour in terms of events at

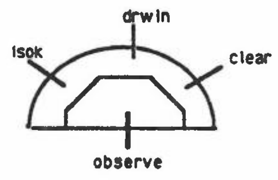

Figure 5: Allowing for External Observation of Appearance

gates so a user's observation gate is needed as well as the application program gates.) We have therefore added the observation gates through which the appearance of any point can be ascertained. This is modelled as a two way event with a point provided by an observing process which receives the appearance of the point (in one communication). [The use of a two-way communication is not significant, but is just one way around a restriction in LOTOS. Ideally, the entire appearance function from points to colours with its domain should be communicated in a single event, but passing such a functional value is not possible directly in LOTOS. Another way round this problem is to encode the function as a value via its (finite) graph.]

When the shells are added, they do not link with the observation gate (see figure 5). The shells (`SHELL1` and `SHELL2`) are given in figure 6 and the descriptions of the composed devices and shells are given in figure 7.

The shells are relatively straight forward, for example, when `SHELL2` receives a `drwln` event with values `(u,v,c)`, it sends a `setclr!c` event (to be received by `CURPOS`), then sends a `moveto!u`, then checks if this was ok (using a `getok` event). If it was ok, then it sends a `drawto!v` and checks to see if this was also ok, storing the result locally in its own parameter.

The combined process `SIMVECDEV` is supposed to simulate the behaviour of `VECDEV` and is built from `CURPOS` running in parallel with `SHELL2` communicating via the gates of `CURPOS` (except `observe`) which are then hidden from other processes. Similarly `SIMCURPOS` is supposed to simulate the behaviour of `CURPOS` and is built from `VECDEV` running in parallel with `SHELL1` communicating via the gates of `VECDEV` (except `observe`) which are then hidden from other processes.

Unfortunately, the relationship between these simulating devices and the original devices is not just one of observational or even testing equivalence.

```
process SHELL1[clear,drwln,isok,rst,stclr,drwto,moveto,
               getok,qc,qp] (vc:colour,cp:indx, b:bool)
 := (  rst; SHELL1[...](bg,dx(0,0),b)
    [] stclr?c:Colour; SHELL1[...](c,cp,b)
    [] drwto?u:Index ; drwln!cp,u,vc ; getok?ok:Bool ;
          (  [ok]     ->  SHELL1[...](vc,u,true)
          [] [not ok]  ->  SHELL1[...](vc,cp,false)
          )
    [] moveto?u:Index ;
          (  [wndw(cp,u)]    ->  SHELL1[...](vc,u,true)
          [] [not wndw(cp,u)]  ->  SHELL1[...](vc,cp,false)
          )
    [] qc!vc ;  SHELL1[...](vc,cp,b)
    [] qp!cp ;  SHELL1[...](vc,cp,b)
    [] getok!b ;  SHELL1[...](vc,cp,b)
    )
endprocess

process SHELL2[rst,stclr,drwto,moveto,getok,
               clear,drwln,isok](b:Bool)
:=  (  clear; rst;  SHELL2[...]b
    [] drwln?u:Index,v:Index,c:Colour;
          stclr!c;  moveto!u;  getok?ok1:Bool;
          (  [ok1]     ->  drawto!v;  getok?ok2:Bool;
                           SHELL2[...]ok2
          [] [not ok1]  ->  SHELL2[...]false
          )
    [] isok!b;  SHELL2[...]b
    )
endprocess
```

Figure 6: (i) SHELL1 converts a VECDEV into a CURPOS (ii) SHELL2 converts a CURPOS into a VECDEV

```
process SIMVECDEV[clear,drwln,isok,observe]
 := hide rst,stclr,drwto,moveto,getok,qc,qp in
     CURPOS[rst,stclr,drwto,moveto,getok,qc,qp, observe]
         |[rst,stclr,drwto,moveto,getok]|
     SHELL2[rst,stclr,drwto,moveto,getok,
            clear,drwln,isok](true)
endprocess

process SIMCURPOS[rst,stclr,drwto,moveto,getok,qc,qp, observe]
 := hide clear,drwln,isok in
      VECDEV[clear,drwln,isok,observe]
          |[clear,drwln,isok]|
      SHELL1[clear,drwln,isok,rst,stclr,drwto,
             moveto,getok,qc,qp](bg,dx(0,0),b)
endprocess
```

Figure 7: (i) SIMVECDEV uses CURPOS and SHELL2 to simulate a VECDEV (ii) SIMCURPOS uses VECDEV and SHELL1 to simulate a CURPOS

The reason for the inequivalence is that delayed effects and unstable intermediate states of the simulating device allow (premature) observations of the display state to be used to distinguish between devices and their simulations. For example, after doing a **drawln** action, **SIMVECDEV** may need five internal actions before a stable state with the correct display can be observed.

The simulating devices do *conform* to the original devices but, as we show in the next section, this is not the end of the story.

3 Formalisation Problems

There are several inter-related problems illustrated by the example from the previous section. We separate these into two groups:-

Modelling problems: including choice of representation problems, observational equivalence and conformance problems.

Design problems, including the design of protocols for communication with devices.

Both groups are problems to do with creating graphics standards in general rather than just the formalisation of these standards, but we can explore and compare choices more easily using appropriate formalisms such as LOTOS.

The modelling problem of what to do about observing appearance is discussed in some detail below. Another problem concerns finding a good level of abstraction for appearance. Gnatz [Gnatz 1990] discusses the need for several different levels of abstraction to model geometric semantics and equivalences, graphical equivalences and physical/technological equivalences. We have not addressed these issues here which are mostly concerned with data algebra rather than process algebra. However process algebra enters into the problem as well because a single construction can often be broken up into several different sequences of primitive actions which reflect different ways to form the same construction. This means that data type equivalences may have to be studied through process algebra action sequences.

On the design side, the small protocol example concerning feedback of responses to requests given sub-section 2.1 was illustrative of a general problem in constructing standards for graphics software which involves distributed state. In this case the control to be added was obvious. However, for larger designs, the form of this control may not be so clear cut. This is where formalisation is particularly important to help sort out the various possibilities and compare for equivalences. Recent work on a Computer Graphics Reference Model [CGRM 1992] is an attempt to provide a framework for comparing graphics standards and to allow for the many ways in which the state can be distributed with pipelines between the operator and an application.

3.1 Observing Appearance

In the example, we saw the need to cater for recording effects which might be observed directly by the user on an output device, and did this by adding an observation gate. This is an essential step if we intend to model devices as processes and still have the ability to distinguish devices with observably different behaviour. However, we saw that a device designed to simulate another device may allow observations of appearance before the device has settled down to a stable state and thus may give erroneous results.

Technically, this could be seen as the same *synchronisation* problem discussed for the behaviour of VECDEV (section 2.1) where extra control was introduced to ensure synchronisation. For example, we could prevent observations at the wrong time in simulating devices like SIMCURPOS by extending the shell with an observation alternative (at the top level) of the form

```
[] observe?p:Point?c:Colour; SHELL1[...](vc,cp,b)
```

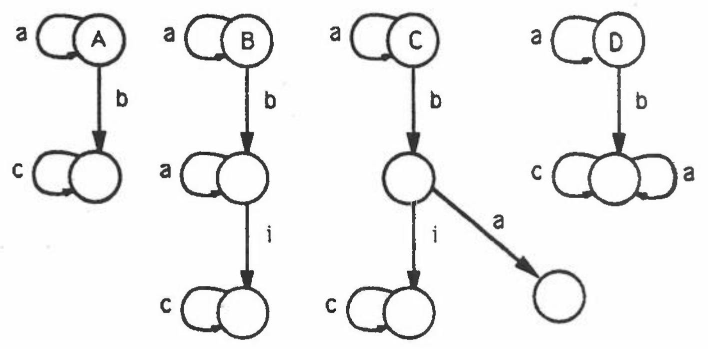

Figure 8: Processes A, B, C and D

This extra gate would be included in the list of synchronisation gates in the parallel composition of **SIMCURPOS** but not hidden in order to ensure that observations synchronise with both the shell and the underlying device (**CURPOS**) and exclude the possibility of premature observation. However, such a solution is *rejected* on the grounds that shells are intended only as interfaces between devices and applications programs and it is therefore unrealistic for a shell to have direct access to the user's observation gate.

We should accept that an output device may need time to stabilise before observations are made, and turn our attention to the way in which we make observations. That is, we need to look at the appropriate form of tests and notions of conformance for such devices rather than try to *correct* the simulation.

The stability problem is illustrated more abstractly in figure 8 with processes A and B. Here, actions a and c are to be thought of as (user) observations of the state while b is an ordinary action (e.g. an application program action like **drwln**). Process B corresponds to the process which might result from simulating A and behaves very much like A provided observations are made only at stable states. In both A and B, before doing b, only observation a can be made; after doing b only observation c can be made once the state is stable. This distinguishes an unacceptable implementation such as D (in figure 8) which does not have this required property. In D it is possible after a b event to first observe the (correct) c event and then afterwards observe the (incorrect) a event.

In fact B conforms to A according to the definition of conformance given below but, unfortunately, so too does the unnacceptable process D (showing that this notion of conformance is inadequate for our purpose).

Another problem is a process like C (in figure 8) which also conforms to A but which can clearly deadlock if a user observation is made at the unstable state. The behaviour of B when observed at the unstable state is benign, but that of C is not.

In describing devices as processes, we have had to add appearance observation as an event which is a two way communication or synchronisation. In reality, the passive observation of appearance is really only a one-way communication and should not influence the device being observed, so the possibility of deadlock is an artefact of the representation forced on us by the use of processes. Processes descriptions such as C do not correspond to any realistic process because of the way we interpret the (appearance) observation events.

This prompts us to define a set of *reasonable* processes with respect to (appearance) observations. This is formalised in the following sub-section.

3.2 Stable Observation Testing

A restriction on when appearance observations may be made will be introduced into tests. First we give formal definitions of the traces and failures of processes P for which we assume there is a labelled transition system with transitions of the form $P \xrightarrow{x} P'$, meaning P may do action x and become P'. Let $\alpha \in Act$ be any visible action and $x \in Act \cup \{i\}$ where i is the unique invisible action.

Notation

$P \overset{\varepsilon}{\Longrightarrow} P' =_{def} P = P'$ or $\exists n > 0\ P \xrightarrow{i^{(n)}} P'$

$P \overset{\alpha}{\Longrightarrow} P' =_{def} \exists P_1, P_2\ P \overset{\varepsilon}{\Longrightarrow} P_1 \xrightarrow{\alpha} P_2 \overset{\varepsilon}{\Longrightarrow} P'$

$P \overset{\sigma}{\Longrightarrow} P' =_{def} \exists P_1, ...P_n\ P \overset{\alpha_1}{\Longrightarrow} P_1 \overset{\alpha_2}{\Longrightarrow} P_2 ... \overset{\alpha_n}{\Longrightarrow} P_n$ where $\sigma = \alpha_1\alpha_2...\alpha_n$

$P \overset{\sigma}{\Longrightarrow}\ =_{def} \exists P'\ P \overset{\sigma}{\Longrightarrow} P'$

$Traces(P) =_{def} \{\sigma \in Act^* | P \overset{\sigma}{\Longrightarrow}\}$

$Failures(P) =_{def} \{(\sigma, X) \mid \exists P'\ P \overset{\sigma}{\Longrightarrow} P'$ and $\forall P'', \alpha \in X,$ *not* $P' \xrightarrow{a} P''\}$

Extension, reduction and conformance are defined by:-

Definition P **ext** $Q =_{def} Traces(P) \supseteq Traces(Q)$ and if $(\sigma, X) \in Failures(P)$ with $\sigma \in TR(Q)$ then $(\sigma, X) \in Failures(Q)$.

Definition P **red** $Q =_{def} Traces(P) \subseteq Traces(Q)$ and $Failures(P) \subseteq Failures(Q)$.

Definition P **conf** $Q =_{def}$ if $(\sigma, X) \in Failures(P)$ with $\sigma \in TR(Q)$ then $(\sigma, X) \in Failures(Q)$.

Both **ext** and **red** are pre-orders which generate the same equivalence (namely failures/testing equivalence). On the other hand, **conf** is not a pre-order as it is not transitive. It is easy to check that $P\,\mathbf{conf}\,Q$ if and only if $\exists R$, $P\,\mathbf{red}\,R$ and $R\,\mathbf{ext}\,Q$

To define an appropriate notion of equivalence for stable observations, we distinguish a new set (Obs) of actions which we wish to regard as special observations (only to be done at stable states). In the examples of the previous section, these would be events at the observe gate, but we generalise the concept for any new set of actions. In the formalisation below, we assume processes are defined by a labelled transition system. Let $a \in L$ be the normal visible actions (events) and $o \in Obs$ be (appearance) observation actions where $L \cap Obs = \emptyset$ and $Act = L \cup Obs$.

We define a process as being *reasonable* if doing an Obs observation cannot cause new deadlocks.

Definition P is reasonable if whenever $(\sigma, X) \in Failures(P)$ then $(\sigma \triangleright (Act - Obs), X) \in Failures(P)$.

where $\sigma \triangleright X$ means the projection of sequence σ with actions in X only.

In the following, we make use of the deadlock detection tests described in [Langerak 1990] and add special cases for the new set of actions. TLOTOS test T are given by

$$T \;::=\; pass \mid stop \mid a;T \mid i;T \mid T_1[\,]T_2 \mid \Sigma\mathcal{T} \mid \theta;T$$

where $a \in L$ and $\mathcal{T}$ is a countable set of tests. θ is Langerak's test for deadlock. The behaviour of test runs is given by first defining the behaviour of a test using a labelled transition system. We introduce $\surd$ as a new action indicating that a test has been passed:-

$$pass \xrightarrow{\surd} stop \qquad a;T \xrightarrow{a} T \qquad i;T \xrightarrow{i} T \qquad \theta;T \xrightarrow{\theta} T$$

$$\frac{T_i \xrightarrow{x} T}{T_1[\,]T_2 \xrightarrow{x} T} \qquad \frac{T \xrightarrow{x} T'}{\Sigma(\mathcal{T} \cup \{T\}) \xrightarrow{x} T'}$$

where $x \in Act \cup \{i, \theta, \surd\}$. The combined behaviour of a test and process in a test run is given by:-

$$\frac{P \xrightarrow{a} P' \quad T \xrightarrow{a} T'}{P \,||_\circ\, T \xrightarrow{a} P' \,||_\circ\, T'} \qquad \frac{P \xrightarrow{i} P'}{P \,||_\circ\, T \xrightarrow{i} P' \,||_\circ\, T} \qquad \frac{T \xrightarrow{i} T'}{P \,||_\circ\, T \xrightarrow{i} P \,||_\circ\, T'}$$

$$\frac{T \xrightarrow{\surd} T'}{P \,||_\circ\, T \xrightarrow{\surd} stop} \qquad \frac{\neg(\exists x \in Act \cup \{i, \surd\}\; P \,||_\circ\, T \xrightarrow{x}) \quad T \xrightarrow{\theta} T'}{P \,||_\circ\, T \xrightarrow{\theta} P \,||_\circ\, T'}$$

A completed test run of P and T is a derivation of $P \|_o T \overset{\sigma}{\Longrightarrow} stop$ for some $\sigma \in (Act \cup \{i, \surd, \theta\})^*$. A successful test run is a completed test run where σ ends in a $\surd$. We say that $P \,\mathtt{may}\, T$ if there is a successful run of P and T, and $P \,\mathtt{must}\, T$ if all completed runs of P and T are successful.

It is shown in [Langerak 1990] that only **may** tests are necessary with the addition of θ, and furthermore, only a restricted form of sequential tests are necessary to distinguish processes.

We will add new restricted forms of tests involving actions in *Obs*, ensuring that they can only be performed after a deadlock test. We will add these to the sequential tests defined below

Let $A \subseteq L$, $O \subseteq Obs$ and $X \subseteq Act$. We introduce $\theta_X T$ to abbreviate $\Sigma\{x; stop | x \in X\}[]\theta; T$. Sequential tests T are defined (with auxiliary restricted tests (R) and simple tests (S)) by:-

$$\begin{array}{lcl} T & ::= & S \mid \theta_A R \\ R & ::= & \theta_O \, pass \mid o; R \mid S \\ S & ::= & pass \mid a; T \end{array}$$

(Note that $\theta_\emptyset T = \theta; T$). The added *Obs* actions in restricted tests R can only be done following a deadlock test (in T) and hence will only be done starting in a stable state.

We need to add $o; S \xrightarrow{o} S$ to the transitions for tests ($o \in Obs$) and we can derive

$$\theta_X T \xrightarrow{\theta} T \qquad \frac{x \in X}{\theta_X T \xrightarrow{x} stop}$$

The combined behaviour of a test and process in a test run is modified to include:-

$$\frac{P \xrightarrow{o} P' \quad T \xrightarrow{o} T'}{P \|_o T \xrightarrow{o} P' \|_o T'}$$

This extension allows us to restrict testing of the special observations to start in stable states and therefore make fewer distinctions between processes. (We require that processes considered are reasonable with respect to *Obs* for this). Relative to this restricted form of testing, we claim that the devices described in section 2 are testing equivalent to the simulated versions.

In figure 8, A, B and C are (stable observation) testing equivalent, but D can be distinguished with the test $T = b; \theta; a; pass$ (recall that $Obs = \{a, c\}$ and $L = \{b\}$ in this example). C is not distinguishable, but as pointed out before, can be ruled out of consideration as it is not a *reasonable* process $((ba, \{c\})$ is a failure of C but $(b, \{c\})$ is not a failure of C. The fact that B, C and D all conform to A (in fact they are all extensions of A) shows a problem with the notion of conformance in this context. The notion of extension assumes extra behaviour as irrelevant, but this should not be so if the extra behaviour iinvolves erroneous displays of information.

4 Conclusions

The example of simulations of output devices illustrated several problems of design and modelling in the use of process algebra to formalise graphics standards and software. In the report, we hope also to have shown that a formal specification language like LOTOS can provide a basis for a much more detailed analysis and the potential for studying equivalence and conformance problems formally. In particular, we illustrated a proof of equivalence for two versions of a model for an output device. We also showed that an equivalence proof for a simulating device with the original version was not so straight forward. This was due to problems in modelling passive observations and also the possible delays in effects due to distributed processing (unstable states). A solution was formalised through considering notions of equivalence, testing and conformance.

This work is regarded as a first step. There is clearly much more to say about conformance, observation and equivalence in this context. There is a need to take the theory much further in order to establish whether or not there are any useful equational axiomatisations for new notions of conformance.

Finally, we would like to acknowledge that a tool for checking bisimulation equivalence - the concurrency workbench [Cleaveland et al 1990] was useful in this work. Although the bisimulation proof given in the appendix is simple enough to do by hand, a small use of the workbench allowed the discovery of some non-equivalences in earlier examples and highlighted the delay problem discussed in sub-section 3.1.

References

[Abramsky 1987] S. Abramsky *Observation Equivalence as a Testing Equivalence* Theoretical Computer Science No. 53 p.225-241 North Holland 1987

[Arnold et al 1987] D.B. Arnold, D.A. Duce, G.J. Reynolds *An Approach to the Formal Specification of configurable models of graphics Systems* EUROGRAPHICS '87, Proc Euro-

	pean Computer Graphics Conference and Exhibition, (eds G.Marechal) North Holland, pp. 439-463
[Baumann 1990]	P. Baumann *A Description Technique for the Computer Graphics Reference Model* (Unpublished? Draft communicated to D. Duce July 1990)
[Brinksma et al 1986]	E. Brinksma, G. Scollo, C. Steenbergen *LOTOS Specifications, Their Implementations and Their Tests* Proc 6th Int Symposium on Protocol Specification, Testing and Verification, Montreal, June 1986, North Holland
[Cleaveland et al 1990]	Cleaveland R., Parrow J., Steffan B. A Semantics-Based Verification Tool for finite state Systems (IFIP) Protocol Specification, Testing and Verification, IX, E.Brinksma, G.Scollo, C.A.Vissers (Eds) Elsevier Science Publishers B.V. North Holland 1990
[CGRM 1992]	ISO - Information Processing Systems - Computer Graphics - Computer Graphics Reference Model. ISO/IEC IS 11072
[DeN Henn 1984]	R. De Nicola, M.C.B. Hennessy *Testing Equivalence for Processes* Theoretical Computer Science 34 pp.83-133
[Duce 1988]	D.A. Duce *Formal Specification of Graphics Software* in Theoretical foundations of Computer Graphics and CAD, R.A. Ernshaw (Ed), Springer Verlag, London 1988
[Duce 1989]	D.A. Duce *GKS, Structures and Formal Specification* EUROGRAPHICS '89, Proc European Computer Graphics Conference and Exhibition, (eds W. Hansmann, F.R.A. Hopgood, W. Strasser) North Holland, pp 307-323.
[Duce et al 1989]	D.A. Duce, P.G.W. ten Hagen, van Liere *Components, Frameworks and GKS Input* EUROGRAPHICS '89, Proc European Computer Graphics Conference and Exhibition, (eds W. Hansmann, F.R.A. Hopgood, W. Strasser) North Holland.
[ERIL]	*Equational Reasoning in LOTOS* SERC/DTI funded project IEATP/SE/IED4/1/1477 (Verification Techniques for LOTOS).
[GKS 1985]	Information Processing Systems - Computer Graphics - Graphical Kernel System ISO 7942

[Gnatz 1990] R.Gnatz *Graphic Data Types, Abstract Graphic Machines, Non-Deterministic Specifications and the Derivation of Interactions.* Draft Report, Technische Universitat Munchen

[Langerak 1990] R.Langerak *A Testing Theory for LOTOS using Deadlock detection* (IFIP) Protocol Specification, Testing and Verification, IX, E.Brinksma, G.Scollo, C.A.Vissers (Eds) Elsevier Science Publishers B.V. North Holland 1990

[LOTOS 1989] ISO - Information Processing Systems - Open Systems Interconnecion - *ISO-LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior* Int Standard ISO 8807, 1989

[Phillips 1987] I. Phillips *Refusal Testing* Theoretical Computer Science 50 pp.241-284

[Purvis 1990] J.B. Purvis *The Use of LOTOS in the Specification of Graphics Software* Brunel University, Computer Science Technical Report CSTR-90-5, July 1990

[Ruggles and Yee 1987] C.L.N. Ruggles and S.T. Yee *Notes on Attempting a Top Down Formal Specification of GKS in META-IV* Technical report No.3, Dept. of Comp. Sci., Univ. Leicester, Sept 1987

Appendix: Equivalence Proof

We prove that VECDEV is equivalent (*weak bisimulation congruent*) to the sequential alternative SVD after making some simplifications to avoid dealing with a large number of similar states. The main simplification is to make type `Stroke` trivial with one value (i.e. all stroke values are equivalent to `mty`). We can then simplify the other value passing actions as follows:-

> o stands for all `observe` actions (which are now identified)
> a stands for all `drwln!(u,v,c)` actions for which `wndw(u,v) = true`
> b stands for all `drwln!(u,v,c)` actions for which `wndw(u,v) = false`
> c stands for `clear`
> t stands for `isok!true`
> f stands for `isok!false`
> st stands for `setok!true`
> sf stands for `setok!false`

With these abbreviations, SVD and VECDEV can be expressed as

```
SVD  = t;SVD  [] c;SVD  [] a;SVD [] b;SVD'
   SVD' = f;SVD' [] c;SVD' [] a;SVD [] b;SVD'

VECDEV = hide st,sf in VD
   VD = VDO |[a,b,t,f,st,sf]| C
   VDO = VS |[st,sf]| OK               (=VECDEVO)
   VS = a;t;VS [] b;f;VS [] c;VS
   OK = t;OK [] st;OK [] sf;NO
   NO = f;NO [] sf;NO [] st;OK
   C = t;C [] f;C [] a;C' [] b;C'      (=CONTROL)
   C' = sf;C [] st;C
```

By expanding out VDO and then VD (see figure 9) we calculate new expressions for VD and VECDEV:-

```
VD  = t;VD  [] c;VD  [] a;st;VD [] b;sf;VD'
VD' = f;VD' [] c;VD' [] a;st;VD [] b;sf;VD'

VECDEV  = t;VECDEV  [] c;VECDEV  [] a;i;VECDEV [] b;i;VECDEV'
VECDEV' = f;VECDEV' [] c;VECDEV' [] a;i;VECDEV [] b;i;VECDEV'
```

where `i` is the silent (internal) action. It is now clear that VECDEV and SVD are equivalent using the law that `x;i;A = x;A`.

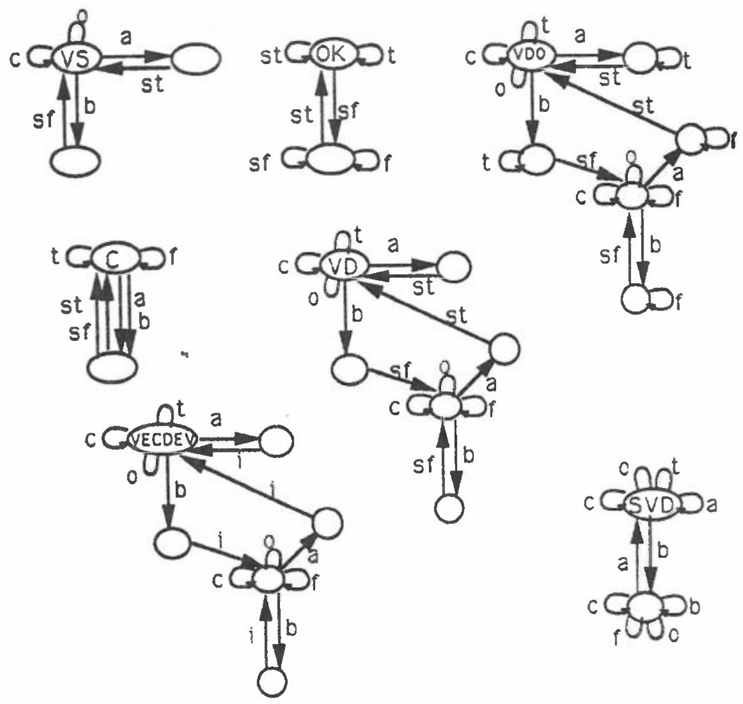

Figure 9: Transition diagrams for VS, OK, VD0, C, VD, VECDEV and SVD