

## The LSD notation 'for communicating systems' revisited

Over twenty five years have elapsed since the author proposed the LSD notation. This report revisits the ideas set out in the first research report on LSD in 1986 [1]. In particular, it reappraises the relevance of LSD as a way of supporting the development of communicating systems. It is quite apparent that the original report has many limitations, and reflects both the immaturity of the ideas that were emerging at that time and the difficulty of relating them to the established literature on concurrency.

From one perspective, it might seem that the impact of the intervening years has been to change the whole perspective on LSD so significantly that the original report should simply be discarded. The very language in which LSD is described in [1] is at odds with the subsequent development of the framework of *Empirical Modelling* within which the notation is most appropriately interpreted. For instance: the term 'agent' has displaced 'process'; the emphasis has shifted from 'programming' to 'modelling' and from 'specifications' to 'descriptions' or 'accounts'; what is described as an agent's 'protocol' is often more appropriately viewed as a family of 'privileges'. And where the original report referred to "The LSD notation for communicating systems" with some evident hint of objectives in common with formal notations for specifying concurrent behaviour (cf. Robin Milner's *Calculus for Communicating Systems* [2]) it has become clear that the distinctive emphasis in the use of LSD is on the broader activity of developing the domain understanding that might eventually underpin systems.

There is an alternative perspective. Without doubt, it was never appropriate to view LSD as predominantly a notation suitable for discussing programming, systems, processes, specifications and protocols. But though it may be quite misleading to interpret LSD as primarily a notation for *concurrent programming*, subsequent research has put this interpretation into a more appropriate conceptual framework rather than discredited it entirely. Indeed, several subsequent publications linking Empirical Modelling to programming indicate that there is a strong motivation for interpreting programming activity as a specialised form of concurrent agent interaction. These include several papers (e.g. [Parallel Computation in Definitive Models, Formal Spec, Rethinking Programming, Realising Software Development as a Lived Experience]) and graduate theses (e.g. Mike Slade, Pi-Hwa Sun, Nick Pope). The LSD notation can play an important role in supporting the open-ended exploration of agent interaction that leads, through specialisation, to programs. In keeping with an Empirical Modelling perspective on behaviour, and in sharp contrast with the classical models of development, the transition from open-ended exploration of agent interaction to program-like behaviours is not traced by a succession of artefacts, diverse in character (cf. UML diagrams, statecharts, object models, specifications, programs) each of which is to be related to the next in a conceptually distinctive way (cf. elaborating an abstract model through special techniques, such as e.g. the consideration of problem frames, the use of Play-in Scenarios refining a specification, compiling a program). In contrast, there is conceptually but one interactive artefact from the beginning to the end of "a program development", and but one overriding principle that guides way in which this artefact evolves. The status of the artefact is moulded at all times by the modeller's moment-by-moment experience and interpretation of her interactions with it according to how these interactions can be correlated with interactions with the environment in which the programming application of the artefact emerge. The degree of freedom that the modeller enjoys in this process of correlation is determined by pragmatic and empirical considerations. What makes personal and objective sense is constrained in much the same way that the interactions and interpretations of the engineering designer and the experimental scientist are constrained.

premature in 1986 / still premature in 2012 - but not irrelevant, potentially even mo

prior to Bonnie Nardi's account of spreadsheet development practices 1992  
but functional abstractions: spreadsheet-style dependencies much more prominent in the  
XP fashionable train of thought  
encouragement from No Silver Bullet, from the kind of response given to this by David  
also Michael Jackson's "What can we expect of program verification?" highlighting the  
qualified success of the EDEN tools  
ADM as follow up to the LSD research report  
latterly Nick Pope's programming environment

## CONTENTS

- Introduction
- 1. Background
- 2. The programming notation LSD
- 3. An operational model for behaviour
- 4. Some simple illustrative examples
- Future directions
- Acknowledgements
- References

## Introduction

The LSD notation was conceived with a view to specifying and describing communicating systems of processes acting concurrently. It was originally developed with a semantic model for the CCITT standard 'specification and description language' SDL in mind [2]. Like SDL, LSD is primarily conceived as a medium for describing systems at the higher levels of abstraction.

A central idea of LSD is the integration of functional and procedural models for **synchronisation** mechanisms at different levels of abstraction. The integration is based upon the "generalised spreadsheet" or definitive notation paradigm for programming ([3],[4]), and the **operational** semantics of LSD is defined in terms of dialogue states over such a notation. *def scripts* This paper includes an informal introduction to the notation, together with several illustrative examples. A methodology for the top-down development of LSD specifications for communicating systems is proposed.

LSD is a procedure-oriented notation, in the sense of [1]. There are three primary characteristics of a notation for concurrent programming to be considered [1]: how concurrent execution is expressed, how processes communicate, and how processes synchronise.

In LSD, each process instance is associated with a process definition, and process instances can be dynamically created and terminated. The set of concurrently executing processes is then determined by which processes are active at any stage. The actions taken by a process may include invocations of other processes, to be executed concurrently.

Communications between processes is by means of shared observables. Typically the value of a observable is under the control of a single active process instance, and can be inspected by another. A observable known to a process is classified according to its role by its kind - oracle, handle or

derivate. In brief: an oracle for a process P is a observable which has an explicit value subject to change entirely beyond the control of P, a handle for P is a observable which has an explicit value conditionally under the control of P, and a derivate for P is a observable whose value is specified by a functional definition in terms of the handles and oracles of P. This classification of observables is significant in the semantics of LSD.

Synchronisation of processes is described either explicitly by means of suitable procedural protocols constraining the behaviours of communicating processes, or implicitly (in an idealised fashion) by specifying appropriate functional relationships between observables.

The paper is in four sections. In Section 1, the background to the LSD design is described, and the motivation for the process model is introduced. In particular, the three primitive kinds of observable (oracle, handle and derivate) are described with reference to the intended semantic model. In section 2, the formal notation is outlined, and used to describe a rudimentary telephone system. Section 3 deals with the operational semantics of LSD, and examines system behaviour in more detail. Section 4 comprises three simple examples to illustrate the main themes of the paper, and the final section discusses directions for further research.

## 1. Background

Spreadsheets have proved to be a very effective basis for human-computer interaction. The interactive use of *definitive notations*, which can be viewed as generalising spreadsheets in some respects, has been advocated by the author [3]. In using a definitive notation, the state of the interaction is represented by a family of definitions ("a definitive script"). Such a script combines observables with explicit values and observables which are implicitly defined by formulae. Viewed from a perspective that relates to programming, the state of an interaction is represented partly by procedural abstractions - observable whose values can be updated, and record "transient values" - and partly by functional abstractions - observables whose values are functionally specified, and record "persistent relationships". This offers a semantic basis for communication richer than that supplied by many other programming paradigms. In view of this, it is very natural to seek a medium for describing communicating processes founded upon a similar semantic model.

Accordingly, consider a communicating system of sequential processes, and suppose that the model of the system state as viewed by one of the participating processes takes the form of a dialogue state, as represented via a definitive notation. It will be assumed that processes within the system are in general created, live for some period of time, and terminate, and that there are observables associated with each active process. Pre-existent and immortal processes are also possible, and in particular, there may be a global environment process pre-existent and immortal, to which global observables are bound.

Within the view of one of these processes P, there will then be a set of known observables (not necessarily bound to P), some with explicit values, and some having known definitions in terms of other observables and constants. The values represented by these observables may be of several different sorts (eg boolean, integer or process identifier), but it will be convenient to suppose that the level of abstraction of the dialogue is such that a change in the value of a observable is an atomic action.

A observable whose value is implicitly defined by a formula will represent a functional abstraction known to P. Such a observable will be called a derivate of P. A derivate is not significant in determining the state of P, since its value is functionally determined by the values of explicitly defined observables. Amongst the observables with explicit values known to P, three different kinds



can be distinguished. There are those observables whose values are perceived by P as fixed; they can neither be changed by P, nor are they subject to change beyond the control of P. These are the constants of P, and form a subclass of the derivates. There are also observables which have explicit values subject to change entirely beyond the control of P: the oracles of P. Finally, there are observable which have explicit values under the control of P (perhaps subject to some preconditions being met): these are the handle observables for P. At this stage, the possibility that a observable behaves both as a handle observable and as an oracle for P (is a handle-and-oracle observable for P) can be admitted, though such features within a system can make formal analysis of behaviour difficult, or even lead to paradoxical behaviour. It should also be noted that the same observable will in general be categorised in different ways by different processes, and it would be more precise to speak of the perceived derivates, constants, oracles and handle of P.

This classification of observables will be clarified by appropriate examples in due course, but it will be helpful at this stage to give some informal illustrations drawn from common practical experience, which will help to motivate the definitions. These are based upon an anthropomorphic view of processes which is a very fruitful source of illustrations. Indeed, for the purpose of describing a particular interaction of a person and a system, it is often convenient to model a person's role by a process.

The archetypal oracle is "absolute time", which (in an appropriate system model) might be known by all persons, but is subject to the control of none. Such a concept can be represented by a observable time which is bound to the environment process. Other oracles may be bound to short lived processes, or relevant only to a specific role. The state of the traffic lights, the speed of the taxi, the length of a queue, the people present at a meeting are of this kind.

The simplest examples of handle observables are attributes which are conditionally under a person's control. The conditional nature of this control is illustrated by the different ways in which "being silent", "being drunk", "being rich", "being alive" can be seen as subject to a person's control. These examples indicate how handle observables can be used to model states of the process in a traditional sense, but the concept of handle observable is broader, and encompasses attributes of other processes over which control can be exercised. For instance, in a doctor's waiting room, whether or not a person is to be treated as a VIP may be up to the receptionist. Whether or not a car has its lights on is under the control of the driver.

Simple derivates of a person are his/her "date of birth", a constant observable bound to the process, and his/her "age", which is implicitly defined as "time - date\_of\_birth". in general, derivates will depend both upon oracle and handle observables: so that for instance, "belonging to a club" may be equivalent to "being over a minimum age" and "having paid the entrance fee".

These simple examples illustrate a number of underlying principles. The convenience of using a single process definition to describe a generic mode of behaviour, and allowing many instances of a particular type of process to participate in a system is evident. It is also clear that the "same" concept can be invoked in different ways, depending upon the context. Over an extended period of time, as in a school record, age will be seen as a derivate, but not as a constant: at any particular time, as in registering for a competition, age may be viewed as a constant: in a play, the age of a character might be seen as a handle observable under the control of the actor: to a barman, the age of a customer is an oracle.

A full discussion of how system behaviour is interpreted within the above model will be given at a later stage, but an informal outline of the principal features is appropriate.



The primary model for communication in this context is via shared observables. A handle or derivate observable for one process and an oracle for another is used for communicating a value. Bearing in mind that the LSD notation is intended for describing systems at a high-level of abstraction, it is natural that the interpretation of system behaviour incorporate some idealisation. The idealised view of the system behaviour will be based on the assumption that the values of a single observable as perceived by different processes are always consistent, so that, in effect, communication of the value of a shared observable is instantaneous, and all derivatives are instantaneously updated as appropriate. Of course, these will not in general be realistic suppositions. In particular, the oracles of a process are not necessarily reliable, and may lead to behaviour of the system outside the intended scope, in much the same way that the action taken by a barman in admitting a young person to the bar can be inconsistent with the idealised view of how the licencing system should operate. In a similar spirit, it may be that the oracle time is better represented by "time by a personal clock", which may be fast, slow, stopped or erratic. For purposes of implementation, it will usually be necessary to replace the functional abstractions in the idealised description by suitable explicit synchronisation and communication protocols at a lower level of abstraction.

As explained in more detail later, the classification of observables which most naturally suggests itself in an informal description of a system will not necessarily be suitable for a formal specification. For instance, though it may be natural to regard an attribute of a shared resource as a handle-and-oracle observable for two or more processes, this may lead to paradoxical situations if both processes can exercise simultaneous independent control over its value. In some cases, refinement of the model may offer an alternative to allowing a observable to behave as a handle-and-oracle observable. For instance, the liveness of a person is to an extent under his/her control (suicide), under the control of other persons (murder), and under environmental control (accident), but "being alive" could be viewed as a derivate "not having killed oneself and not having been killed and not having died natural death", in which the handle and oracle components are separated.

## 2. The programming notation LSD

The formal notation to describe processes will reflect the ideas illustrated above. To define a process type, the handle, oracles and derivatives are given, together with a protocol which describes the nature of the control exercised over handle observables. The binding of observable names to observables will be determined from the context when a process instance is created. The skeleton of each process type definition has the form:

```
process process_name ( parameter_list ) {
    oracle          list of oracle names
    state          list of state names
    handle         list of handle names
    derivate       list of derivate names
    protocol       list of guarded commands of the form: guard → action
}
```

A typical guard takes the form of a boolean condition on the handle and oracle observables, and a typical action consists of a sequence comprising assignments to handle observables and invocations of process instances. The semantics of actions is such that at most one assignment to a particular handle observable within each action can be assumed. Termination of a process instance is achieved in two ways. At any point, setting a private boolean handle observable LIVE to false is an action which leads to termination: alternatively, a process may terminate when a special private boolean derivate LIVE becomes false. The semantics of termination can be expressed formally by adjoining the clause :LIVE and ..." to each guard. Informally, the specification of the behaviour of a process instance has a declarative component associated with the functional definitions of its derivatives, and a

procedural component associated with the actions in its protocol.

An extended example of the use of the LSD notation will help to clarify the concepts. This will be developed in a systematic fashion, to illustrate the principles which may be used to derive an LSD account.

*instantiate user (X, M) user (Y, N)*

2

Consider a simple telephone system, in which the role of a user, and of a telephone is represented by a process. (For simplicity, the descriptions below have been expressed in terms of two users  $user(X)$  and  $user(Y)$  whose telephones are  $phone(M)$  and  $phone(N)$  respectively.) An appropriate model of the user's role is a process which does not terminate, together with a protocol which reflects the many different conditions in which a user may find the telephone. A telephone process is likewise - for the purpose of this application - non-terminating, and acts in response both to the user and to signals generated by calling processes.

In developing the LSD description, the first step is to identify the handles, derivatives and oracles for the principal processes. For the user, the oracles are the signals which he can receive from the telephone: in this case, the boolean observable ringing: whether the telephone bell is ringing, and a observable (say integer-valued) tone: what tone is emitted (as appropriate) in the earpiece. The user's handle observables comprise a boolean onhook: whether the telephone receive is onhook, and an integer dialled\_number to represent the current contents of the dialling register within the telephone. Note that all these observables are bound to the telephone with which the user is associated - this is indicated by preceding these observable names by a '#' symbol. (A simplified "instantaneous dialling" facility is assumed, in so far as assigning a value to dialled\_number is interpreted as an atomic action in the model.) For the telephone process, there are two oracles under the control of the user: whether the phone is onhook, and the value of the dialled\_number, and other oracles to reflect signals received from the telephone network. The observables ringing and tone must also be defined within the telephone process, and may most appropriately be conceived as derivatives defined in terms of other processes yet to be specified. The condition of ringing, for example, requires that the telephone is onhook, and that the telephone number is presently being called. Similarly, the tone emitted by the telephone will reflect the status of a process invoked when an attempt is made to make a call, and is functionally determined in this manner.

Having decided upon the principal processes and the nature of their associated observables, it is appropriate to develop protocols for the user and telephone processes. The relevant ideas are very simple: examine the handle observables and consider under what preconditions the handle observables can be changed, and under what preconditions other processes are invoked. For the user, the handle observables are onhook and dialled-number. When the phone is offhook, the receiver can be put down at any stage: this has the effect of re-initialising the telephone by clearing the dialling register. When the phone is onhook and ringing, it can be picked up and answered: when it is onhook and not ringing, it can be picked up and a process for setting up a call invoked. (The process  $initcall()$  used at this point might be more appropriately extended to a more detailed dialling process.) For the telephone process, in the absence of handle observables, there is a simple protocol whereby a valid number  $N$  in the dialling register at an appropriate time will invoke a calling process  $call(M,N)$  which attempts to make a connection.

The manner in which the auxiliary processes  $init\ call()$  and  $call(M,N)$  are defined resembles that described above; the details will be left to the reader to infer from the complete description below. (The notation  $|...|$  indicates evaluation of the enclosed expression.) A significant feature of the final description is the attempted use of derivatives in the  $exchange()$  process to resolve the problems which concurrent access to a single phone can present. The  $exchange()$  process is intended to ensure that at

any one time at most one call(\*,N) instance can be responsible for phone N ringing; its limitations are discussed in §4.

```

agent user(U,S) {
oracle (int) tone[S], (bool) ringing[S]
handle (bool) onhook[S], (int) dialled_number[S]
protocol
    not onhook[S] -> onhook[S]= true; dialled_number[S]=@,
    not onhook[S] & (tone[S] == D) -> dialled_number[S]=N,
    not onhook[S] & (tone[S] == C) -> <speaking>,
    onhook[S] and not ringing[S] -> onhook[S] = false ; dial(S),
    onhook[S] and ringing[S] -> onhook[S] = false ; <speaking>
}

agent telephone(S) {
oracle (bool) onhook[S], connecting[S,*], dialling[S],
(int) dialled_number[S],
(bool) connected[S,*], engaged[S,*]
state (bool) onhook[S] = true,
(int) dialled_number[S],
(bool) connecting[S,*] = false,
(bool) dialling[S] = false,
(char) tone[S]
derivate
(char) tone[S] = D if dialling[S]:
    E if connecting[S, dialled_number[S]]
    and engaged[S, dialled_number[S]]:
    R if connecting[S, dialled_number[S]]
    and not engaged[S, dialled_number[S]]:
    C if connected[S,?] and connected[?,S]:
    @ otherwise
}

agent exchange() {
oracle (bool) onhook[*], connected[*,*], connecting[*,*], answered[*]
state (bool) onhook[*], connected[*,*] = false, answered[*] = false
(bool) ringing[*], engaged[S,*],
(time) Tdial , Tcall
derivate
(bool) ringing[*] = connected[?,*] and onhook[*] and not answered[*],
(bool) engaged[S,*] = connecting[S,*] and (ringing[*] and not onhook[*]),
(time) Tdial = <timeout for dialling>,
(time) Tcall = <timeout for calling>
}

agent dial(S) {
oracle
(int) dialled_number[S],
(time) Tdial, time,
(bool) onhook[S],
(bool) connecting[S,*] = false
handle (time) tstart
state (time) tstart = |time|,
(bool) valid = <dialled_number[S] is valid>
derivate
(bool) dialling[S] = not onhook[S] and ((time - tstart) < Tdial),
(bool) LIVE = dialling[S] and not connecting[S, dialled_number[S]]
protocol
dialling[S] and valid -> connect(S, dialled_number[S])
}

agent connect(S,D) {
oracle (bool) onhook[S], onhook[D], ringing[D], engaged[S,D],

```

```

        (time) Tcall , time
handle  (time) tcall ,
        (bool) connected[S,D], answered[D]
state   (time) tcall = |time|,
        (bool) connected[S,D] = false,
        (bool) answered[D] = false
derivate
        (bool) connecting[S,D] = not connected[S,D],
        (bool) LIVE = not onhook[S] and
            ( (connecting[S,D] and (time-tcall )<Tcall)
              and ringing[D] and
              (connected[S,D] and (not answered[D] and not onhook[D]))
            )
protocol
        not engaged[S,D] and not connected[S,D] -> connected[S,D] = true,
        not onhook[D] and not answered[S,D] and connected[S,D] -> answered[D] = true,
        engaged[S,D] -> delete connect(S,D)
}

agent environment() {
handle time
state time = 0
protocol
    true -> time = |time| + 1
}

```

It should be noted that the above example is offered as an informal use of LSD in a descriptive role, rather than as a rigorous specification. The techniques required for a full formal analysis of the behaviour of an LSD system have yet to be developed, but the context for studying the behaviour will be described below. In practice, it is clear that principles for developing specifications with tractable behaviour must also be devised. In the example above, for instance, no observable appears as a handle or derivate in more than one process, thereby reducing the likelihood of interference, and, the use of the LIVE derivates in the `init_call()` and `call()` processes makes it easy to determine which process instances can operate concurrently.

The above example also illustrates several characteristic features of the LSD notation. The classification of observables provides a useful form of documentation, and describes the interface between processes in an unconventional but effective manner. The role of functional abstractions within the description is very significant, and illustrates different ways in which idealisations are used to simplify the model. For instance, within the model, "lifting of the receiver" and "initiation of the dialling tone", occur instantaneously. Similarly, no signal delays at the exchange are assumed. Refinement of the model would entail the replacement of derivates by protocols to define equivalent behaviour.

### 3. An operational model for behaviour

This section informally develops a broad framework within which many aspects of system behaviour can (in principle) be described and analysed. These include, for instance, deviation from the idealised behaviour, as in the event of unreliable oracles, and the possibility of singular behaviour in exceptional circumstances within an imperfectly specified system. A formal treatment of behaviour is beyond the scope of this paper, and further work is required both on methods of analysis of specifications, and on the semantic rules which must be observed to avoid singular and ambiguous behaviour. It must be acknowledged that, if the form of specifications is unrestricted, an analysis of behaviour exclusively based on the operational semantics outlined here will generate the familiar combinatorial explosion of case analyses. On the other hand, there seem to be good prospects that the judicious use of functional abstractions will allow the top-down development of a formal system



specification in conjunction with an incremental analysis of the behaviour of the approximating specifications, and that an axiomatic approach [5] focussing on the guarded commands within protocols can be used to complement reasoning based upon the operational model.

The guiding principle behind the development of LSD is that the state of a system of concurrent processes should be representable by the state of a dialogue over a definitive notation. In some sense, this is an approximation to the truth; provision has only been made for the view of the system associated with each process to be representable by the state of a dialogue over a definitive notation. This is perhaps only reasonable; it is arguably the case that "the state of the system" has no meaning other than "the totality of system views of the participating processes". In general, it is to be expected that there will be some discrepancies between these views, and a naive attempt to synthesise the views of all the processes might easily result in inconsistency. For the system analyst, whose task is to reason about the behaviour of the entire system, the semantic problem is to reconcile an idealised model of the entire system behaviour with the actual behaviour as determined by the states of the participating processes.

Within the system model described above, the births and deaths of process instances which occur in the course of a period of the system history are constrained by some order relations. These "e from the fact that each process instance in general has an ancestry which must have been born before it, and may also necessarily be invoked only after other processes have died. Subject to these constraints, there will be certain families of process instances that can be alive simultaneously. The (idealised) state of the system at any time will be described in terms of such a family of process instances which are currently five. As suggested above, the system state will be synthesised from the views of the participating processes, each of which is a dialogue state within a definitive notation, and the idealised system state will itself be of this form. That is, each state of the system will be defined by means of a family of sorted observables, each of which has a current value which is either undefined, explicitly defined, or defined implicitly via a formula in terms of other observables. To describe the idealised state of the system it is necessary to specify the appropriate set of observables and defining formulae. The set of observables is simply obtained as the union of all observables bound to a live process instance, each of which will be an oracle, handle or derivate for that process type, as appropriate. The set of explicitly defined and undefined observables is then specified by the subset comprising oracles and handle observables, and the set of implicitly defined observables by the complementary subset consisting of derivatives. The values ascribed to oracle and handle observables within the idealised system state will coincide with their values as perceived by the process instances to which they are bound, which are interpreted as their authentic values.

The behaviour of the system is defined by the possible transitions to another state of the same form which can occur through the action of the live process instances. It will be important to distinguish between the actual behaviour, which is determined by the actual states of the participating processes, and will depend in general upon the potentially unreliable values ascribed to oracle and handle-and-oracle observables bound to other processes, and the idealised behaviour, which is the behaviour to be expected on the basis that all processes have reliable knowledge of observables bound to other processes. The latter is an essential fiction of the system analyst, whose role is that of an observer who knows the authentic values of all observables, but who has no part in determining the transitions which occur. This knowledge equips the system analyst to reason about the system behaviour in terms of idealised states, but not to predict the actual behaviour, since knowledge of the perceived values of all observables within the participating processes is not assumed.

In any state, the behaviour of the system is determined by the set of possible transitions. Each transition is defined by selecting a set of processes, and within each process a guarded command

whose guard is perceived to be true by that process. (This guard might indeed be false or undefined in the idealised model, which accounts for the unpredictability of the behaviour in general witnessed by the system analyst.) In a single transition, all the selected processes simultaneously execute the appropriate command atomically. If a system is to have consistent and non-singular behaviour, some non-interference constraints, to be described below, have to be satisfied. The effect of a transition (in general) is

- to change the values of handle observables
- to create new process instances, thereby introducing new observables and derivate relationships to eliminate some existing processes, thereby possibly deleting some observables and their definitions from the idealised state.

There are a variety of respects in which the above description of system behaviour has to be qualified. The functional relationships defined by the derivates must be consistent at any time: in particular, they will always be such as to define the values of certain observables implicitly in terms of others in such a way that there is no recursive reference. It will also be necessary to ensure that there is no possibility of the same observable being simultaneously assigned different values by two or more processes in a transition. These conditions will be enough to guarantee that the state reached through a transition is a dialogue state of the appropriate form (Even without these restrictions, it may still be useful to work within the same semantic model, and regard invalid transitions as singularities.) There is one further possible source of interference between actions in a transition: if an action includes the assignment of an expression involving an evaluated derivate or oracle, and the value of this expression can be affected by assignments within another action, then the dialogue state reached after the transition will depend upon the manner of synchronisation. It is clear that such interference will in general complicate reasoning about a specification; nevertheless, the importance of (for example) recording the time of initiation of a process instance, as in the `init-call()` and `call()` processes in the telephone example, indicates that such ambiguity can sometimes be accommodates

The idealised system view is that of the system analyst who knows the authentic values of all explicitly defined observables, and can compute the authentic values of all derivates. The least the analyst can ask of the system is that behaviour which properly reflects the values of observables within the idealised model should be acceptable, but it may be necessary in practice to consider a broader notion of good behaviour which reflects the unreliability of oracles. Good behaviour would have to mean that the integrity of the system state is guaranteed, and that actual behaviour would approximate to idealised behaviour. For example, the idealised behaviour might be stable in the sense that the system converged to this if handles and oracles took time to acquire their authentic values through signal delays.

It may be observed that the approximate behaviour of a system can be realised as the idealised behaviour of another system in which explicit value sharing processes have been introduced. Formally, all oracles which refer to a observable `v` bound to another process can be renamed, using distinct identifiers, and a signal process whose role is to update the value of the oracle as appropriate introduced. For instance, if the observable `v` is referenced as an oracle by the process `A`, but owned by process `B`, then the oracle observable `v` can be replaced by a observable `xv`, and the signal process:

```
process signal() {
    oracle v;
    handle xv;
    protocol true → xv=v
}
```

introduced. The idealised behaviour of the resulting system will then be the same as the behaviour of the original system under the assumption that A's oracle for v is intermittently updated. A simple illustration of these principles is given below.

#### 4. Some simple illustrative examples

In this section, three simple examples are used to illustrate some of the issues raised by the abstract model of system behaviour described in 3.

##### (a) The telephone example re-examined

Interpretation of the telephone example in the light of the operational model of system behaviour in 3 reveals some subtle points, and illustrates in particular some aspects of translating functional into procedural abstractions.

Within the telephone system, a critical aspect of the behaviour is ensuring that two or more calls cannot be simultaneously connected to the same destination phone. The `exchange()` process, in conjunction with the `call(M,N)` process, is intended to guarantee this. Certainly the given specification is such that initiation of the process `call(M,N)` "instantaneously" sets the boolean observable `isringing[N]` to true if phone N is not engaged, causing a subsequent initiation of `call(M',N)` to set the boolean observable `engaged[M']` to true. On the other hand, if two process calls `call(M,N)` and `call(M',N)` are initiated in the same transition (i.e. are truly concurrent), it will be seen that both M and M' will become connected to N.

Remedying this deficiency in the specification - an exercise to the reader - suggests further considerations. A natural approach to take is to regard `isringing[*]` as a handle observable rather than a derivate, to eliminate the functional definition of `isringing[*]`, and introduce appropriate actions to manipulate its value within a protocol for the `exchange()` process. When the possibility of two or more simultaneous invocations of `call(*,N)` attempting to set `isringing[N]` to true then arises, the choice of which is successful can be made by selection between the guarded commands in the `exchange()` protocol. Devising a protocol to guarantee such "mutual exclusion" naturally suggests a role for handle-and oracle observables, notwithstanding the potential complications. Procedural manipulation of the status of telephone N also entails a mechanism for resetting `isringing[N]` on termination of a `call(*,N)` instance, and can result in counter-intuitive behaviour if lazy execution of guarded commands within a protocol is assumed. For this reason, it may be convenient to introduce some other prioritising mechanism on actions in a protocol - stipulating for instance that particular guarded commands are to be greedily executed. The semantic distinction between a functionally defined derivate  $d=D$  and a greedy execution of the guarded command

$$d!=D \rightarrow d=|D|$$

is then of interest.

As a footnote, it may be observed that problems with true concurrency - models of behaviour in which genuinely simultaneous action is possible - are perhaps more likely to arise in the context of functional abstraction such as is used to defined `isringing[*]` in the telephone example. It may be that procedural refinement will of itself introduce the sequentiality which underlies a semantic model based on interleaving actions.

##### (b) A simple algorithm for the greatest common divisor

In 3, the concept of regarding the approximate behaviour of a concurrent system as the idealised

behaviour of another was introduced. It is as yet unclear how far such a concept can be exploited in practice, but a very simple example of this principle in action may be helpful.

Consider a system involving two processes, each of which owns a positive integer handle observable known to the other as an oracle. Starting from an initial configuration in which each process knows the authentic values  $m$  and  $n$  of both observables, the system is intended to behave in such a way that each process terminates with the value of its handle observable equal to the greatest common divisor of  $m$  and  $n$ . The processes to be used are defined as follows:

```
process p(i) {
    oracle (int) v[i+1]
    handle (int) #v[i]
    protocol v[i]>v[i+1] -> v[i]=v[i]-v[i+1]
}
```

where  $i = 1, 2$ , and  $i+1$  is interpreted modulo 2. It is very easy to see that the idealised behaviour of this system of processes is acceptable: it corresponds to the elementary sequential GCD algorithm:

```
v[1]=m; v[2]=n;
do v[1]>v[2] → v[1]=v[1]-v[2]
[] v[2]>v[1] → v[2]=v[2]-v[1]
od
```

since there is no possibility that both guards can be simultaneously true.

Now consider the behaviour of the same system under the more realistic assumption that the oracles in the processes  $p(i)$  do not always have authentic values, but acquire the authentic value from time to time (e.g. as and when a signal is received). The behaviour of the system is then precisely described by the idealised behaviour of a system comprising three processes:

```
process signal(i) {
    oracle sv[i]
    handle #ov[i]=sv[i]
    protocol true → ov[i]=sv[i]
}
```

where  $i = 1, 2$  and:

```
process update() {
    oracle ov[1], ov[2]
    handle #sv[1]=m, #sv[2]=n
    protocol
        sv[1]>ov[2] -> sv[1]=sv[1]-ov[2]
        sv[2]>ov[1] -> sv[2]=sv[2]-ov[1]
}
```

Inspection of the protocols easily shows that  $ov[i] \geq sv[i] > 0$  is a derivative, and that at most one of the guards in the process `update()` can be true at any one time. It follows that any idealised behaviour of this system can be simulated by a sequence of atomic actions, comprising single and parallel assignments, each of the form:

```
ov[1]=sv[1]
ov[2]=sv[2]
sv[1]=sv[1]-ov[2]
sv[2]=sv[2]-ov[1]
ov[1]=sv[1] || sv[1]=sv[1]-ov[2]
ov[2]=sv[2] || sv[2]=sv[2]-ov[1].
```



To prove that the idealised behaviour of the system leads to  $sv[1]=sv[2]=gcd(m,n)$  on termination, it is enough to show the invariance of the relation

$$gcd(m,n)=gcd(ov[1], ov[2])=gcd(ov[1], sv[2])=gcd(sv[1], ov[2])=gcd(sv[1], sv[2])$$

under each of the assignments and parallel assignments above. This is straightforward once it is observed that  $ov[2]=sv[2]$  is a necessary precondition for an atomic action in which the assignment

$$sv[1]=sv[1]-ov[2]$$

occurs, and dually. To justify this, note that  $ov[2]>ov[1]\geq sv[1]$  is a postcondition of the action

$$sv[2]=sv[2]-ov[1]$$

in the update() protocol, and thus the action  $ov[2]=sv[2]$  must be performed before the guard

$$sv[1]>ov[2]$$

can be satisfied, and dually. In effect, action from the appropriate signal process is essential when execution of the update() process switches between the two guarded commands in its protocol.

### (c) The attendance form example

The LSD notation is primarily intended for high-level specification of systems, and may be useful in a wide variety of applications. In particular, LSD processes can be used to model systems such as office environments in which the use of administrative records and physical transfer of data is involved. For instance, a licence can be viewed as a process incorporating a derivate

```
(bool) valid = (time - purchase_date < period_of_validity).
```

By way of further illustration the following example describes the circulation of an attendance record at a meeting for signature by all members present

```
process attendance_record() {
    oracle (bool) signed[(member)* ]=false,
           (member) handler=secretary;
    derivate (bool) has_form[(member)*] = (handler==*);
}

process transfer_form(Y) {
    handle handler
    derivate LIVE = handler!= Y
    protocol true → handler=Y
}

process member(X) {
    handle (bool) signedm, requests[X];
    oracle (bool) signed[*\X], requests[*\X], has_form[X];
    derivate (bool) requests[X] = not has_form[X] and not signed[X]
    protocol
        has_form[X] and signed[X] and requests[Y] → transfer_form(Y)
        has_form[X] and not signed[X] → <X signs form> signed[X]=true
        has_form[X] and X != secretary → transfer_form(secretary)
}
```

Note that the derivate `has_form()` in the attendance record is strictly redundant, since it can be inferred from knowledge of the current handler, but it is useful to distinguish between "knowing when I have the form" from "knowing who currently has the form". In effect, derivatives can be used

in this manner for information hiding. The example can also be elaborated: not all members will continue to request the form as the derivate stipulates for instance, and some may not wish to sign the form unless obliged to do so. An alternative profile for the member() process, taking account of this possibility is given below. It is of particular interest to note the way in which an appropriate use of oracles, derivatives and protocols enables the knowledge and manipulative scope of a process to be faithfully and subtly represented.

```

process member(X) {
  handle (bool) signed[X], requests[X];
  oracle (bool) signed[*\X], requests[*\X], has-form[X];
  protocol
    has_form[X] and signed[X] and requests[Y] -> transfer_form(Y)
    has_form[X] and signed[X] and not signed[Y] -> transfer_form
    has_form[X] and not signed[X]
      -> <X signs form> signed[X]= true; requests[X] =false;
    has_form[X] and X != secretary -> transfer_form(secretary)
    not has_form[X] and not signed[X] -> requests [X] =true
}

```

## Future directions

The LSD notation was originally conceived in connection with semantic models for the CCITT standard language SDL (cf [2]). SDL has an enormous range of constructs intended to specify and describe systems at many different levels of abstraction, and a central problem which must be addressed in using the language is the lack of a satisfactory and consistent semantic framework. The solution proposed in this paper is based upon the idea of developing an LSD specification by successive refinement in such a way that an analysis of behaviour can be developed incrementally in parallel. Such an approach is plausible only because of the integration within LSD and procedural features which can be respectively used to model synchronisation mechanisms and lower-level levels of abstraction. The design methodology proposed has affinities with formal specification using a functional notation, and with systematic program development using the axiomatic method.

The design of LSD is still in its early stages, and more work is needed to consolidate the preliminary design sketched in this paper, and to develop further examples. There is a particular need to identify the semantic constraints which a specification must satisfy in order to make the operational behaviour consistent and free of singularities. It will also be important to develop a complementary axiomatic approach to the analysis of system behaviour, and consider the issues of ensuring deadlock freedom, mutual exclusion and fairness.

## Acknowledgements

The author is grateful to British Telecom for sponsorship on a Short Term Research Fellowship.

He is much indebted to Mark Norris for enthusiastic encouragement and motivating ideas without whom this work would not have been undertaken, and to Charles Jackson and Mathai Joseph for helpful advice.

## References

- W.M.Beynon. The LSD notation for communicating systems. Computer Science Research Report 87, University of Warwick, 1986.
- W.M.Beynon, R.C.Boyatt and S.B.Russ. Rethinking Programming. In Proceedings IEEE Third International Conference on Information Technology: New Generations (ITNG 2006), April 10-12, 2006, Las Vegas, Nevada, USA 2006, 149-154.
- Meurig Beynon, Realising Software Development as a Lived Experience. An essay in *Onward! '12* :

*Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, October 21-25, Tucson, Arizona, USA, 2012, ISBN 978-1-4503-1562-3, 229-244.

W.M.Beynon, J. Rungrattanaubol and J. Sinclair. Formal Specification from an Observation-Oriented Perspective. *Journal of Universal Computer Science*, Vol. 6 (4), 407-421, 2000.

W.M.Beynon, M.D.Slade and Y.W.Yung. Parallel computation in definitive models. CONPAR'88, British Computer Society Workshop Series CUP, 359-367, 1989.

[1] G R Andrews & F B Schneider, Concepts and Notations for Concurrent Programming, *ACM Computing Surveys* Vol 15 Number 1, March 1983, 3-43

[2] Meurig Beynon and Mark Norris, Comparison of SDL and LSD. *Proc. SDL'87*, ed R Saracco and P A J Tilanus, North-Holland, 201-209, 1987.

[3] Meurig Beynon, Definitive Notations for Interaction in *Proc. BCS Conference "People and Computers: Designing the Interface"* ed. Johnson and Cook, Cambridge University Press, 23-34, 1985.

[5] E W Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976

---