Research report 87

THE LSD NOTATION FOR COMMUNICATING SYSTEMS

Meurig Beynon*

(RR87)

Abstract

The LSD notation is intended for the specification and description of communicating systems of sequential processes. It was originally developed with a semantic model for the CCITT standard 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. This 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. The 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.

* Department of Computer Science
University of Warwick
Coventry CV4 7AL, England

also:
Short Term Visiting Research Fellow
British Telecom Research Laboratories

Nov 1986

ABSTRACT

The LSD notation is intended for the specification and description of communicating systems of sequential processes. It was originally developed with a semantic model for the CCITT standard 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. This 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. The 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.

CONTENTS

Introduction

The LSD notation is intended for the specification and description of communicating systems of processes acting concurrently. It was originally developed with a semantic model for the CCITT standard 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. This 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. 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.

Communication between processes is by means of shared variables. Typically the value of a variable is under the control of a single active process instance, and can be inspected by another. A variable known to a process is classified according to its role by its *kind* - oracle, state or derivate. In brief: an **oracle** for a process P is a variable which has an explicit value subject to change entirely beyond the control of P, a **state** for P is a variable which has an explicit value conditionally under the control of P, and a **derivate** for P is a variable whose value is specified by a functional definition in terms of the states and oracles of P. This classification of variables 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 variables.

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 variable (**oracle**, **state** 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 programming notations which are essentially generalised spreadsheets ("definitive notations") has been advocated by the author [3]. In interaction using a definitive notation, the state of the dialogue is represented by a combination of variables with explicit values and variables which are implicitly defined by formulae. In effect, a dialogue state is represented partly by procedural abstractions - variables whose values can be updated, and record "transient values" - and partly by functional abstractions - variables 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 variables 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 variables are bound.

Within the view of one of these processes P, there will then be a set of known variables (not necessarily bound to P), some with explicit values, and some having known definitions in terms of other variables and constants. The values represented by these variables 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 variable is an atomic action.

A variable whose value is implicitly defined by a formula will represent a functional abstraction known to P. Such a variable 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 variables. Amongst the variables with explicit values known to P, three different kinds can be distinguished. There are those variables 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 variables which have explicit values subject to change entirely beyond the control of P: the **oracles** of P. Finally, there are variables which have explicit values under the control of P (perhaps subject to some preconditions being met): these are the **state** variables for P. At this stage, the possibility that a variable behaves both as a state variable and as an oracle for P (is a state-and-oracle variable 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 variable 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 states of P.

This classification of variables will be formally 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 variable **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 state variables 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 state variables can be used to model states of the process in a traditional sense, but the concept of state variable 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 variable 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 state variables: 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 state variable 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 variables. A variable which is a state or derivate variable 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 incorporates some idealisation. The idealised view of the system behaviour will be based on the assumption that the values of a single variable as perceived by different processes are always consistent, so that, in effect, communication of the value of a shared variable is instantaneous, and all derivates 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 variables 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 state-and-oracle variable 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 variable to behave as a state-and-oracle variable. For instance, the liveness of a person is to 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 state 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 states, oracles and derivates are given, together with a protocol which describes the exact nature of the control exercised over state. The binding of variable names to variables 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, possibly initialised*
    derivate *list of derivate definitions*
    protocol *list of guarded commands of the form: guard -> action*
}

A typical guard takes the form of a boolean condition on the state and oracle variables, and a typical action consists of a sequence comprising assignments to state variables and invocations of process instances. The semantics of actions is such that at most one assignment to a particular state variable within each action can be assumed. Termination of a process instance is achieved in two ways. At any point, setting a private boolean state variable 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 derivates, 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 description.

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 states, derivates 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 variable **ringing**: whether the telephone bell is ringing, and a variable (say integer-valued) **tone**: what tone is emitted (as appropriate) in the earpiece. The user's state variables comprise a boolean **onhook**: whether the telephone receiver is onhook, and an integer **dialled_number** to represent the current contents of the dialling register within the telephone. Note that all these variables are bound to the telephone with which the user is associated - this is indicated by preceding these variable 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 variables **ringing** and **tone** must also be defined within the telephone process, and may most appropriately be conceived as derivates 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 variables, it is appropriate to develop protocols for the user and telephone processes. The relevant ideas are very simple: examine the state variables and consider under what preconditions the state variables can be changed, and under what preconditions other processes are invoked. For the user, the state variables 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 **init_call()** used at this point might be more appropriately extended to a more detailed dialling process.) For the telephone process, in the absence of state variables, 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 derivates 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.

```
process user(X,M) {

    oracle   (int) tone[M], (bool) ringing[M];
    state    (bool) onhook[M], (int) dialled_number[M];

    protocol

        not onhook[M] -> onhook[M]=true; dialled_number[M]=@;
        not onhook[M] and tone[M] is D -> dialled_number[M] = N
        not onhook[M] and tone[M] is Q -> <speak>
        onhook[M] and not ringing[M] -> onhook[M]=false; init_call(M);
        onhook[M] and ringing[M] -> onhook[M] = false; <speak>
}

process phone(M) {

    oracle   (bool) #onhook[M],
             (int) #dialled_number[M],
             (bool) #active[M],
             (bool) #connected[M], #calling[M], #engaged[M], #dialling[M],
             (bool) isringing[M];

    derivate (int) #tone[M] = D if dialling[M]:
                              E if calling[M] and engaged[M]:
                              R if calling[M] and not engaged[M]:
                              Q if connected[M]:
                              @ otherwise,
                     (bool) #ringing[M] = onhook[M] and isringing[M];

    protocol
        not active [M] and < dialled_number = N is valid > -> call(M,N)

}
```

```
process init_call(M) {

    oracle   (int) dialled_number[M], (time) T_RTD, (bool) onhook[M];
    state    (time) #t_RTD = |time|;

    derivate (bool)  dialling[M] = time - t_RTD < T_RTD,
                     (bool) #LIVE = dialling[M] and not onhook[M]
                         and not dialled_number is valid;

}

process call(M,N) {

    oracle   (bool) onhook[M], onhook[N], isringing[N], (time) T_call;

    state    (bool)  engaged[M] = | not onhook[N] or isringing[N] |,
             (time) #t_call = |time|,
             (bool)  connected[M] = false;

    derivate
        (bool)  calling[M] = (time-t_call < T_call) and not connected[M],
        (bool) ring[M,N] = calling[M] and not engaged[M],
        (bool) #LIVE = not onhook[M] and ((connected[M] and not onhook[N])
            or (calling[M] and not connected[M])),
        (bool) active[M] = LIVE;

    protocol

        not engaged[M] and calling[M] and not onhook[N]
            -> connected[M] = true; < connect M and N >

}

process exchange() {

    oracle       #ring[*,*], onhook[*];
    derivate #isringing[*] = ring [?,*] and onhook[*],
                 (time) #T_RTD = < max dialling delay > ,
                 (time) #T_call = < max time for calling >;

}
```

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 variable appears as a state 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 variables 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 arise 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 live. 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 variables, each of which has a current value which is either undefined, explicitly defined, or defined implicitly via a formula in terms of other variables. To describe the idealised state of the system it is necessary to specify the appropriate set of variables and defining formulae. The set of variables is simply obtained as the union of all variables bound to a live process instance, each of which will be an oracle, state or derivate for that process type, as appropriate. The set of explicitly defined and undefined variables is then specified by the subset comprising oracles and state variables, and the set of implicitly defined variables by the complementary subset consisting of derivates. The values ascribed to oracle and state variables 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 state-and-oracle variables 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 variables 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 variables, 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 variables 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 state variables
- to create new process instances, thereby introducing new variables and derivate relationships
- to kill some existing processes, thereby possibly deleting some variables 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 variables 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 variable 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 accommodated.

The idealised system view is that of the system analyst who knows the authentic values of all explicitly defined variables, 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 variables 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 sytem 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 states 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 variable 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 variable v is referenced as an oracle by the process A, but owned by process B, then the oracle variable v can be replaced by a variable xv, and the signal process:

```
process signal() {
    oracle v;
    state 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 variable **isringing**[N] to true if phone N is not engaged, causing a subsequent initiation of **call**(M',N) to set the boolean variable **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 (ie 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 state variable 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 state-and oracle variables, 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 -> 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 state variable 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 variables, the system is intended to behave in such a way that each process terminates with the value of its state variable 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]
    state (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 (eg 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]
    state  #ov[i]=sv[i]
    protocol true -> ov[i]=sv[i]
}
```

where i=1, 2 and:

```
process update() {
    oracle ov[1], ov[2]
    state  #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]≥sv[i]>0 is a derivate, 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, 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]≥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) {
    state handler
    derivate LIVE = handler != Y
    protocol true -> handler=Y
}

process member(X) {
    state (bool) signed[X], 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, derivates 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, derivates and protocols enables the knowledge and manipulative scope of a process to be faithfully and subtly represented.

```
process member(X) {
    state (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(Y)
        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 of functional and procedural features which can be respectively used to model synchronisation mechanisms at higher and lower level levels of abstraction. The design methodology proposed has affinities both 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 which this work would not have been undertaken, and to Charles Jackson and Mathai Joseph for helpful advice.

References

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

[2] Meurig Beynon and Mark Norris, *A Formal Model for SDL* (submitted)

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

[4] Meurig Beynon, *A Programming Paradigm based on Definitions* University of Warwick, Computer Science Research Report #** (in preparation)

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